

**PROGRAMA DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO**



**Scaling Manifold Learning
through Sparse Matrix
Reformulation**

André Tomitan Bocces

UNIVERSIDADE ESTADUAL PAULISTA
“Júlio de Mesquita Filho”
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

ANDRÉ TOMITAN BOCCES

**SCALING MANIFOLD LEARNING THROUGH SPARSE MATRIX
REFORMULATION**

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Alexandro José Baldassin

Rio Claro (SP)
2026

UNIVERSIDADE ESTADUAL PAULISTA
“Júlio de Mesquita Filho”
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

ANDRÉ TOMITAN BOCCES

**SCALING MANIFOLD LEARNING THROUGH SPARSE MATRIX
REFORMULATION**

Master’s Dissertation presented to the Institute of Geosciences and Exact Sciences of the Rio Claro Campus, of the São Paulo State University “Júlio de Mesquita Filho”, as part of the requirements for obtaining the title of Master in Computer Science.

Advisor: Prof. Dr. Alexandro José Baldassin

Rio Claro (SP)
2026

B664s Bocces, André Tomitan
Scaling Manifold Learning through Sparse Matrix Reformulation /
André Tomitan Bocces. -- Rio Claro, 2026
78 p.

Dissertação (mestrado) - Universidade Estadual Paulista (UNESP),
Instituto de Geociências e Ciências Exatas, Rio Claro
Orientador: Alexandro José Baldassin

1. High performance computing. 2. Information retrieval. 3.
Computer memory system. 4. Matrices. 5. Parallel algorithms. I.
Título.

IMPACTO POTENCIAL DESTA PESQUISA

Esta pesquisa reformula algoritmos de re-ranqueamento baseados em aprendizagem de variedades usando álgebra linear esparsa, reduzindo a complexidade espacial de $O(N^2)$ para $O(NL)$. O impacto imediato é viabilizar esses métodos em hardware de consumo (8GB RAM) para datasets de 160 mil imagens, eliminando barreiras de infraestrutura. Democratiza o acesso a sistemas de recuperação de imagens de alto desempenho, beneficiando pesquisadores e reduzindo custos de pesquisa. Reduz consumo energético em centros de dados.

POTENTIAL IMPACT OF THIS RESEARCH

This research reformulates manifold learning-based re-ranking algorithms using sparse linear algebra, reducing spatial complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(NL)$. The immediate impact is enabling these methods on consumer hardware (8 GB RAM) for 160,000-image datasets, removing infrastructure barriers. It democratizes access to high-performance image retrieval systems, benefiting researchers and reducing research costs. It reduces energy consumption in data centers.

UNIVERSIDADE ESTADUAL PAULISTA
“Júlio de Mesquita Filho”
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

ANDRÉ TOMITAN BOCCES

**SCALING MANIFOLD LEARNING THROUGH SPARSE MATRIX
REFORMULATION**

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista “Júlio de Mesquita Filho”, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Comissão Examinadora

Prof. Dr. Alexandro José Baldassin
IGCE / UNESP / Rio Claro (SP)

Prof. Dr. Aleardo Manacero Junior
DCCE / UNESP / São José do Rio Preto (SP)

Prof. Dr. Edson Borin
Instituto de Computação / UNICAMP / Campinas (SP)

Conceito: Aprovado.

Rio Claro (SP), 25 de maio de 2026.

ACKNOWLEDGEMENTS

O presente trabalho foi realizado com apoio da *Fundação para o Desenvolvimento da Unesp* (FUNDUNESP) - processo nº 3419/2023, bolsa nº 71602/B08, e também no âmbito do Termo de Cooperação nº 0050.0124179.23.9 (SAP-4600674832).

RESUMO

O re-ranqueamento baseado em variedades é uma classe consolidada de algoritmos de refinamento pós-recuperação, com ampla aplicação na Recuperação de Imagens Baseada em Conteúdo (CBIR). Apesar de sua eficácia, sua adoção é limitada pelas demandas computacionais e de memória intrínsecas. Esses métodos dependem tipicamente de matrizes de similaridade densas $N \times N$ (onde N é o tamanho do *dataset*), impondo um *footprint* de memória $\mathcal{O}(N^2)$ que resulta em falhas por insuficiência de memória (*Out-of-Memory* - OOM) em *hardware* de consumo para *datasets* de grande escala. Essa limitação de *hardware* restringe o re-ranqueamento no estado da arte a infraestruturas de computação especializadas, criando barreiras significativas de implantação. Para contornar essa limitação, apresenta-se uma reformulação em álgebra linear esparsa que reduz substancialmente o *overhead* de memória do re-ranqueamento baseado em variedades.

Mapeiam-se as regras de atualização em três primitivas: Multiplicação Matriz-Matriz Geral Esparsa (SpGEMM), o produto de Hadamard e a intercalação ordenada de linhas. Ao explorar a esparsidade intrínseca das listas ranqueadas por meio do formato *Compressed Sparse Row* (CSR), reduz-se a complexidade espacial de $\mathcal{O}(N^2)$ para $\mathcal{O}(NL)$, sem realizar aproximações nos cálculos da variedade subjacente. Para alcançar esse resultado na prática sem a explosão intermediária de densidade, desenvolveu-se: (1) um *kernel* de poda customizado que integra a seleção dos top- L elementos diretamente na fase de acumulação; (2) uma estratégia de construção paralela com divisão por linhas que elimina o *overhead* de sincronização; e (3) um alocador de memória especializado utilizando *Transparent Huge Pages* e preenchimento de linhas de cache, projetado para reduzir a pressão no TLB e o falso compartilhamento em arquiteturas *multi-core*. Valida-se essa reformulação em dois algoritmos representativos do *Unsupervised Distance Learning Framework* (UDLF): o *Cartesian Product Re-ranking* (CPRR) e o *Local Hypergraph Re-ranking* (LHRR). Essa validação demonstra que a reformulação esparsa viabiliza a execução em um *laptop* de consumo de 8 GB para um *dataset* de 160.000 imagens, enquanto a representação densa exigiria aproximadamente 102 GB de RAM. Para o algoritmo mais complexo avaliado, o consumo de memória em pico permanece em aproximadamente 256 MB com $L = 400$. Em servidores de alta memória, a redução no total de instruções gera *speedups* de até $2.4\times$ em *datasets* maiores, confirmando que a abordagem não reduz o desempenho em relação ao *baseline* denso em *datasets* de grande escala. No geral, a reformulação transforma a complexidade espacial em uma métrica ajustável estritamente limitada pelo comprimento da lista L e pelo tamanho da vizinhança K , desacoplando o *footprint* de memória do tamanho total do *dataset*.

Palavras-chave: CBIR; UDLF; Matriz Esparsa; Aprendizagem de Variedades; Escalabilidade.

ABSTRACT

Manifold-based re-ranking is an established class of post-retrieval refinement algorithms, with extensive application in Content-Based Image Retrieval (CBIR). Despite its effectiveness, its adoption is limited by inherent computational and memory demands. These methods typically rely on dense $N \times N$ similarity matrices (where N is the *dataset* size), imposing an $\mathcal{O}(N^2)$ memory *footprint* that leads to out-of-memory (OOM) failures on commodity *hardware* for large-scale *datasets*. This *hardware* constraint restricts state-of-the-art re-ranking to specialized computing facilities, raising significant deployment barriers. To address this limitation, we present a sparse linear algebra reformulation that substantially reduces the memory overhead of manifold-based re-ranking.

We map the update rules onto three primitives: Sparse General Matrix-Matrix Multiplication (SpGEMM), the Hadamard product, and sorted row merging. By exploiting the inherent sparsity of ranked lists via the Compressed Sparse Row (CSR) format, we reduce spatial complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(NL)$, without approximating the underlying manifold calculations. To achieve this in practice without intermediate density explosion, we develop: (1) a custom pruning kernel that integrates top- L selection directly into the accumulation phase; (2) a row-split parallel construction strategy that eliminates *synchronization overhead*; and (3) a specialized memory allocator using Transparent Huge Pages and cache-line padding, designed to reduce TLB pressure and false sharing on multi-core architectures. We validate this reformulation on two representative algorithms from the Unsupervised Distance Learning Framework (UDLF): Cartesian Product Re-ranking (CPRR) and Local Hypergraph Re-ranking (LHRR). This validation demonstrates that the sparse reformulation enables execution on a consumer 8 GB laptop for a dataset of 160,000 images, while a dense representation would require approximately 102 GB of RAM. For the most complex algorithm evaluated, peak memory consumption remains approximately 256 MB with $L = 400$. On high-memory servers, the reduction in total instructions yields *speedups* of up to $2.4\times$ on larger *datasets*, confirming that the approach does not reduce performance relative to the dense *baseline* on large-scale *datasets*. Overall, the reformulation transforms spatial complexity into a tunable metric strictly bounded by the list length L and neighborhood size K , decoupling the memory *footprint* from the total *dataset* size.

Keywords: CBIR; UDLF; Sparse Matrix; Manifold Learning; Scalability.

LIST OF FIGURES

Figure 1	Example of an image retrieval result set before (top) and after (bottom) re-ranking on the Oxford17Flowers dataset (Nilsback; Zisserman, 2006). Images highlighted with a red border in the top row are incorrectly ranked results that are demoted after re-ranking.	16
Figure 2	The Compressed Sparse Row (CSR) format applied to a 5×5 example matrix. Non-zero values and their column indices are stored in two companion arrays: <code>values</code> and <code>col_idx</code> , respectively. The <code>row_ptr</code> array stores the offset in <code>values</code> at which each row begins, enabling $\mathcal{O}(1)$ access to any row and $\mathcal{O}(\text{nnz}_i)$ traversal of row i , where nnz_i denotes the number of non-zeros in that row.	27
Figure 3	Architecture and information flow of the LHRR sparse reformulation. The input ranked lists are converted into $\mathcal{O}(NK)$ and $\mathcal{O}(NL)$ sparse structures, followed by fused multiplication and element-wise updates that preserve the memory bound throughout.	41
Figure 4	CSRMatrix construction pipeline. The architecture enforces a strict transition from thread-local block-allocated triplets to a global packed CSR representation, optimized for parallel execution and cache locality. . . .	43
Figure 5	CPRR parallel scaling: speedup T_1/T_t as a function of thread count on Environment A ($L = 1,000$, $K = 20$, $T = 1$ iteration). Speedup remains below $1.3\times$ due to $\mathcal{O}(N)$ overheads, while the dense baseline achieves better relative efficiency and achieves lower absolute time above $t \approx 4$. . .	54
Figure 6	LHRR parallel scaling on Environment A ($L = 1,000$, $K = 20$, $T = 1$ iteration), same configuration as Figure 5. Speedup reaches $2.42\times$ on larger datasets, but dense methods eventually recover their wall-clock advantage given sufficient cores.	55
Figure 7	LHRR accuracy as a function of ranked list depth L across dataset types. ALOI shows continuous improvement with increasing L , consistent with a smooth manifold where additional neighbors carry genuine structural information. SUN397 and VegFru peak at low L and decline thereafter, as additional neighbors introduce noisy connections that degrade propagation quality.	65
Figure 8	CPRR accuracy as a function of L . The accuracy plateau on SUN397 and VegFru occurs earlier than for LHRR, consistent with CPRR’s one-hop co-occurrence model saturating the available semantic signal at lower neighborhood densities than LHRR’s two-hop hyperedge construction. . .	66

LIST OF TABLES

Table 1	– Comparison of related approaches along key dimensions.	35
Table 2	– Evaluation environments. Environment B uses the same physical hardware as Environment A, with cores pinned to a single NUMA node via <code>numactl -physcpubind=0-11 -membind=0</code> to ensure fair comparison under the 16 GB memory limit imposed by Linux <code>cgroups</code> (<code>MemoryMax=16G</code> , <code>MemorySwapMax=4G</code>).	48
Table 3	– Comparison of experimental parameters (L, K, T) across hardware environments versus typical literature values.	48
Table 4	– Benchmark dataset properties and memory requirements. Dense mem. req. is computed as $4N^2$ bytes (single-precision float).	49
Table 5	– Software library versions and execution environment. All experiments were conducted on Ubuntu Linux (Kernel 5.15+), using GCC 11.4.	50
Table 6	– CPRR execution performance: <i>Sparse-CSR engine</i> versus <i>Dense-MMAP baseline</i> . Measured in Environment A (18 threads, $L = 2,000$, $K = 50$, $T = 5$). .	51
Table 7	– LHRR execution performance: <i>Sparse-CSR engine</i> versus <i>Dense-MMAP baseline</i> . Measured in Environment A (18 threads, $L = 2,000$, $K = 50$, $T = 5$). .	52
Table 8	– <i>Wall-clock</i> time (seconds) and parallel scaling for CPRR on Environment A ($L = 1,000$, $K = 20$, single NUMA socket). $S_{18} = T_1/T_{18}$; P_{eff} is the mean number of hardware threads actively executing useful work at $t = 18$	53
Table 9	– <i>Wall-clock</i> time (seconds) and parallel scaling for LHRR on Environment A ($L = 1,000$, $K = 20$, single NUMA socket). Notation as in Table 8.	53
Table 10	– Unified Performance and Feasibility Comparison: CPRR across <i>Hardware</i> Environments. Status: Success (Time) or OOM. Parameters adjusted per environment to reflect hardware constraints. Parameter T was set to 5 for all configurations.	57
Table 11	– Unified Performance and Feasibility Comparison: LHRR across <i>Hardware</i> Environments. Status: Success (Time) or OOM. Parameters adjusted per environment to reflect hardware constraints. Parameter T was set to 5 for all configurations.	58
Table 12	– ALOI (72K): MAP accuracy grid search (L vs K).	61
Table 13	– SUN397 (108K): MAP accuracy grid search (L vs K).	61
Table 14	– VegFru (160K): MAP accuracy grid search (L vs K).	62

LIST OF SYMBOLS

Symbol	Type	Definition
$\mathbf{M}$	Matrix	Global similarity matrix ($N \times N$)
$\mathbf{W}$	Matrix	Sparse weight matrix used in CPRR co-occurrence computation
$\mathbf{P}$	Matrix	Position matrix encoding mutual neighborhood strength (LHRR)
$\mathbf{H}$	Matrix	Sparse hyperedge matrix representing two-hop neighborhoods
$\mathbf{S}$	Matrix	Sparse intermediate correlation matrix ($\mathbf{H} \times \mathbf{H}^T$)
$\mathbf{R}$	Matrix	Ranked list matrix containing neighbor indices
N	Scalar	Total number of images in the dataset
L	Scalar	Length of the ranked list / neighborhood size for final retrieval
K	Scalar	Truncation parameter for sparse construction and pruning
$\odot$	Op.	Hadamard (element-wise) product operator
$\text{pos}(j, \mathbf{R}_i)$	Func.	Zero-indexed rank position of item j in the list of i
$\mathbf{M}_A, \mathbf{M}_B$	Matrix	Intermediate matrices in iterative re-ranking (e.g., CPRR stages)
$\mathbf{M}_{final}$	Matrix	Final similarity matrix after re-ranking convergence
$\mathbf{S}^{rev}$	Matrix	Reverse correlation matrix (used in LHRR symmetry)

TABLE OF CONTENTS

1	INTRODUCTION	15
1.1	Motivation	16
1.2	Research Contributions	17
1.3	Publications	18
1.4	Potential Impact of the Research	18
1.5	Dissertation Organization	19
2	BACKGROUND	20
2.1	Image Re-ranking and the Semantic Gap	20
2.2	The Unsupervised Distance Learning Framework	21
2.3	Target Algorithms: CPRR and LHRR	22
2.3.1	Cartesian Product Re-ranking (CPRR)	23
2.3.2	Local Hypergraph Re-ranking (LHRR)	24
2.4	The Memory Hierarchy and Sparse Matrix Representations	25
2.5	Summary	28
3	RELATED WORK	29
3.1	Manifold-Based Re-ranking and Diffusion Processes	29
3.2	Scalability Approaches in Manifold Retrieval	30
3.2.1	Sparse affinity construction	30
3.2.2	Offline precomputation	31
3.2.3	Approximate graph construction	31
3.2.4	Sparse feature encodings	32
3.2.5	Replacing the Dense Matrix: a Dictionary-Based Approach	32
3.2.6	Why Sparse Matrix Libraries are not Enough	33
3.2.7	Foundational gap	33
3.3	Sparse Matrix-Matrix Multiplication	33
3.4	Summary and Identified Gap	35
4	SPARSE REFORMULATION OF MANIFOLD RE-RANKING	37
4.1	Identification of Structural Bottlenecks	37
4.2	Sparse Primitive Mapping	38
4.2.1	Proof Sketch: Exactness of the Sparse Mapping	41
4.3	High-Performance Sparse Engine	42
4.3.1	Library Architecture and Design Patterns	42
4.3.2	Memory Subsystem: BlockAllocator and THP Alignment	43

4.3.3	Parallelized CSR Construction	44
4.3.4	Fused SpGEMM Kernel	44
4.3.5	Implementation Details: Accumulator Design and Vectorization	44
4.3.6	Statement on the Use of Artificial Intelligence	45
4.4	Summary	46
5	RESULTS	47
5.1	Experimental Setup	47
5.1.1	Experimental Parameters and Memory Pressure Trade-offs	48
5.1.2	Baseline Configurations	49
5.1.3	Execution Protocol	50
5.2	Algorithmic Efficiency and Microarchitectural Analysis	51
5.2.1	Execution time and instruction reduction	51
5.3	Parallel Scaling Behavior	52
5.3.1	CPRR Scaling Analysis	54
5.3.2	LHRR Scaling Analysis	55
5.3.3	Answer to RQ1	56
5.4	The Feasibility Gap: Constrained Hardware Analysis	56
5.4.1	CPRR Feasibility Under Memory Constraints	57
5.4.2	LHRR Feasibility Under Memory Constraints	57
5.4.3	Answer to RQ2	60
5.5	Retrieval Quality Verification	60
5.5.1	Answer to RQ3	60
5.6	Discussion	62
5.6.1	Structural Limits of Sparsity and Dataset Scale	62
5.6.2	Microarchitectural Tradeoffs Between Efficiency and Total Work	63
5.6.3	Parameter Sensitivity and the Role of Neighborhood Density	64
5.6.4	Answer to RQ4	65
5.6.5	Deployment Guidelines	67
5.7	Summary	67
6	CONCLUSION	68
6.1	Summary of Contributions	68
6.2	Limitations	70
6.3	Avenues for Future Work	71
	BIBLIOGRAPHY	72
	APPENDIX A – REPRODUCIBILITY AND DATA AVAILABILITY	76
A.1	Source Code and Dataset Availability	76

DADOS CURRICULARES	77
-------------------------------------	-----------

1 INTRODUCTION

In domains ranging from medical imaging to satellite analysis, the ability to retrieve visually similar images from large collections has become a foundational capability of modern computing systems (Datta et al., 2008; Chen et al., 2022). Content-Based Image Retrieval (CBIR) powers these applications by locating relevant images through comparison of visual descriptors, ranking results by similarity to a query rather than relying on manually assigned metadata (Datta et al., 2008). As image collections grow to billions of items and visual recognition tasks become increasingly demanding, the quality of retrieval systems has direct consequences for the user experience, safety, and operational cost of products used by millions of people daily (Datta et al., 2008; Lew et al., 2006).

A typical CBIR pipeline has three stages. First, a query image is submitted and mapped to a feature representation (e.g., hand-crafted descriptors or deep embeddings). Second, the system computes similarities between the query descriptor and all database descriptors using a distance metric, producing an initial ranked list. Third, a re-ranking stage refines this list by exploiting contextual information, such as neighborhood relations or graph structure, that is not captured by pairwise descriptor similarity alone (Datta et al., 2008).

Re-ranking often yields substantially better results because the initial similarity computation is necessarily local: it compares images independently and can be misled by nuisance factors such as background clutter, illumination, viewpoint changes, or domain shift (Lew et al., 2006). In contrast, re-ranking methods incorporate relational evidence from the dataset itself by exploiting contextual and manifold information (Jegou; Harzallah; Schmid, 2007). If the top results for a query are mutually consistent (e.g., they share neighbors or lie in a coherent region of a visual manifold), they can be promoted; conversely, isolated accidental matches can be demoted (Valem; Pedronette, 2016).

In practice, this motivates a widely adopted two-stage strategy: rank once using any scalable method to obtain a moderately good candidate list, then re-rank that list more carefully (Qin et al., 2011). The first stage can be optimized for speed through approximate search, quantization, or indexing (Johnson; Douze; Jégou, 2019), while the second stage can apply more expensive refinement over a smaller neighborhood (Valem; Pedronette, 2016). This design reduces end-to-end latency and improves accuracy without applying complex procedures to the entire collection from scratch (Datta et al., 2008).

Figure 1 illustrates the intuition behind re-ranking. The top row shows an initial ranking produced from direct descriptor similarity, where visually similar but semantically unrelated images (highlighted with red borders) appear near the top of the list due to color and texture overlap. The bottom row shows the refined results, where images consistent with the query in the local neighborhood structure are promoted and the most frequent distractors are pushed down the list. The figure highlights the general failure mode of one-shot similarity



Figure 1 – Example of an image retrieval result set before (top) and after (bottom) re-ranking on the Oxford17Flowers dataset (Nilsback; Zisserman, 2006). Images highlighted with a red border in the top row are incorrectly ranked results that are demoted after re-ranking.

ranking and why neighborhood-aware refinement improves retrieval quality.

Iterative, unsupervised re-ranking methods constitute an established class of post-retrieval refinement algorithms (Zhou et al., 2003; Donoser; Bischof, 2013; Pedronette et al., 2019; Valem; Pedronette, 2016; Valem; Pedronette, 2017). Starting from an initial ranking produced by a *baseline* descriptor, these methods apply iterative refinement procedures that update similarities by exploiting relationships between ranked lists, graph structures, or visual manifolds. Given ranked lists, the algorithm builds neighborhood relations, propagates similarity across the dataset, and outputs refined rankings.

This class of methods achieves strong retrieval improvements without requiring labeled training data. However, they are challenging to scale: standard implementations across this family rely on a global similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$ to represent refined distances between all image pairs (Zhou et al., 2003; Donoser; Bischof, 2013; Baldassin et al., 2019; Valem; Pedronette, 2017; Pedronette et al., 2019), leading to an $\mathcal{O}(N^2)$ memory footprint that restricts deployment to high-memory machines.

1.1 MOTIVATION

Manifold-based re-ranking methods inherently operate over pairwise relationships between all images in a collection. Standard implementations materialize these relationships as a dense similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$, where N is the dataset size (Zhou et al., 2003; Donoser; Bischof, 2013; Baldassin et al., 2019; Valem; Pedronette, 2017). With single-precision floating-point values, this requirement grows quadratically: 10,000 images require approximately 400 MB, 100,000 images require approximately 40 GB, and one million images would require approximately 4 TB of RAM. This quadratic memory growth restricts advanced re-ranking to specialized high-memory systems. As a result, many studies are limited to datasets with $N < 50,000$, not due to lack of computational power, but because the dense representation allocates memory for relationships that are structurally irrelevant to the final result.

The practical implications of this constraint are illustrated by the VegFru dataset (Hou; Feng; Wang, 2017), which contains 160,000 images of fine-grained food and vegetable categories. Processing this dataset requires approximately 102 GB of RAM for the similarity matrix alone, a requirement that exceeds the capacity of most commodity developer workstations and research

servers with less than 128 GB of RAM. A collection of one million images would require approximately 4 TB, placing such datasets beyond the reach of standard server configurations.

The key insight is that re-ranking algorithms of this class operate on ranked lists of length L , where $L \ll N$. Each image maintains meaningful relationships with only L neighbors; the remaining $N - L$ entries in each row of the similarity matrix are structurally irrelevant and can be treated as zero. This makes the matrix inherently sparse, with only $\mathcal{O}(NL)$ non-zero entries rather than $\mathcal{O}(N^2)$. Exploiting this sparsity is the central idea of this work.

This dissertation proposes a sparse linear algebra approach to manifold-based re-ranking that aligns the memory footprint of the implementation with the actual information content of the problem. To demonstrate the effectiveness of this approach, we apply it to two representative algorithms from the Unsupervised Distance Learning Framework (UDLF) (Valem; Pedronette, 2017): Cartesian Product Re-ranking (CPRR) (Valem; Pedronette, 2016) and Local Hypergraph Re-ranking (LHRR) (Pedronette; Almeida; Torres, 2014), which together cover a range of computational patterns present in manifold-based re-ranking. By reformulating these algorithms using sparse primitives and enforcing $\mathcal{O}(NL)$ memory bounds throughout execution, we show that state-of-the-art re-ranking becomes feasible on commodity hardware (Zhou et al., 2003; Donoser; Bischof, 2013; Baldassin et al., 2019), converting memory consumption from a fixed hardware requirement into a tunable algorithmic parameter.

1.2 RESEARCH CONTRIBUTIONS

This dissertation makes two key contributions, both motivated by the same underlying problem: the quadratic memory growth of dense similarity matrices in manifold-based re-ranking.

The first contribution is a **formal sparse reformulation of manifold-based re-ranking algorithms** as sparse linear algebra operations. We show that the update rules of re-ranking methods such as CPRR and LHRR can be expressed using Sparse General Matrix-Matrix Multiplication (SpGEMM) (Gustavson, 1978; Buluç; Gilbert, 2012), sparse Hadamard products (Horn; Johnson, 1994), and optimized row merging. This reformulation reduces theoretical spatial complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(NL)$, aligning resource consumption with the information density of the ranked lists rather than the square of the dataset size.

The second contribution is a **high-performance memory-bounded execution engine** implemented in C++. A key challenge in achieving the $\mathcal{O}(NL)$ bound in practice is the intermediate density explosion: general-purpose sparse libraries compute the full matrix product before applying any filtering, which can cause temporary memory spikes to $\mathcal{O}(N^2)$ density even when the final result is sparse. To prevent this, we develop a Fused Multiply-and-Prune kernel that integrates top- L selection directly into the multiplication accumulation step, ensuring that memory consumption never exceeds the target bound during execution. The engine also employs a row-split parallel construction strategy with thread-local block allocators aligned to Transparent Huge Pages boundaries, reducing TLB misses and false sharing on multi-core architectures.

These contributions are validated through an empirical evaluation across three benchmark datasets and multiple hardware tiers ranging from a dual-socket research server to a consumer laptop. Results demonstrate up to a $400\times$ reduction in peak memory consumption compared to dense baselines, and speedups of up to $2.4\times$ on large-scale datasets. In particular, datasets of 160,000 images can be processed within the 8 GB RAM of standard consumer hardware, a scenario in which all dense implementations exceed available memory.

1.3 PUBLICATIONS

The following publications were produced during the development of this dissertation and are directly related to its contributions:

Bocces, Baldassin & Dantas (2025) was published at the Escola Regional de Alto Desempenho de São Paulo (ERAD-SP 2025), presenting an initial survey of optimization techniques for information retrieval performance and scalability. Bocces et al. (2025) extends that survey into a systematic review published in IEEE Access (vol. 14, pp. 2576–2591, 2025), covering performance and scalability optimization in information retrieval systems.

1.4 POTENTIAL IMPACT OF THE RESEARCH

In compliance with the institutional regulations established by Unesp (Portaria nº 117/2022), this section delineates the broader technological, economic, and environmental impacts of the developed research. By shifting the mathematical paradigm of manifold-based re-ranking from dense operations to a sparse linear algebra formulation, the contributions of this work extend beyond algorithmic refinement, directly impacting computing sustainability and the democratization of high-performance data processing.

From an ecological perspective, the algorithmic optimization developed herein aligns with green computing principles. Traditional manifold learning and re-ranking mechanisms impose severe memory thrashing and prolonged *wall-clock* times due to their $\mathcal{O}(N^2)$ complexity, forcing data centers to operate under high-energy regimes to sustain dense matrix materialization. By compressing data structures into the *Compressed Sparse Row* (CSR) format and fusing the top- L selection within a specialized *kernel*, this architecture eliminates redundant hardware instruction cycles and optimizes cache locality. Consequently, the reduction in memory pressure and CPU utilization translates directly into lower power consumption during large-scale retrieval operations, mitigating the carbon *footprint* associated with dense evaluation infrastructures.

Furthermore, this research promotes significant technological democratization. High-fidelity graph-based re-ranking has historically been restricted to specialized cloud facilities or clusters equipped with high-density RAM subsystems, creating a severe deployment barrier for small-scale laboratories and independent developers. By reducing the peak memory footprint from 102 GB to approximately 256 MB on representative large-scale *datasets*, this work effectively decouples state-of-the-art performance from high-end *hardware* prerequisites.

Enabling the execution of robust post-retrieval algorithms on consumer-grade commodity *laptops* democratizes access to advanced image retrieval technology, bridging the gap between theoretical computing scale and ubiquitous edge execution.

1.5 DISSERTATION ORGANIZATION

The mathematical notation and symbols used throughout this work are summarized for convenient reference in the List of Symbols on page 11. The remainder of this dissertation is structured as follows. Chapter 2 provides the mathematical foundations for CBIR and the target re-ranking algorithms, and establishes the systems-level context through an examination of the memory hierarchy and sparse matrix representations. Chapter 3 surveys existing scalable retrieval methods and general-purpose sparse libraries, establishing the technical gap addressed by this work. Chapter 4 presents the sparse primitive mapping and the high-performance engine implementation. Chapter 5 presents the experimental setup and empirical findings across hardware environments. Section 5.6 provides a deeper analysis of parameter sensitivity and microarchitectural behavior. Finally, Chapter 6 offers concluding remarks and directions for future work.

2 BACKGROUND

This chapter provides the theoretical and systems-level foundations required to understand the proposed sparse reformulation. We begin (Section 2.1) by formalizing the image re-ranking task and its role in manifold learning. We then describe (Section 2.2) the Unsupervised Distance Learning Framework (UDLF) architecture, identifying the mathematical commonalities shared by state-of-the-art re-ranking algorithms. Next, we detail (Section 2.3) the two target algorithms, CPRR and LHRR, which serve as the basis for our evaluations. Finally, we provide (Section 2.4) an overview of modern hardware memory hierarchies, explaining the physical mechanisms that lead to performance degradation when working sets exceed DRAM capacity.

2.1 IMAGE RE-RANKING AND THE SEMANTIC GAP

Let $\mathcal{D} = \{x_1, \dots, x_N\}$ be an image collection of N images, where each image x_i ($i \in \{1, \dots, N\}$) is represented by a d -dimensional feature descriptor. A *baseline* retrieval method produces, for each image x_i , an initial ranked list $\mathbf{R}_i \in \{1, \dots, N\}^L$ containing the indices of the L most similar images according to a pairwise distance metric, sorted in descending order of similarity. Re-ranking is the process of computing an improved ordering of these initial results by exploiting relational structure among the retrieved items, without requiring labeled training data (Valem; Pedronette, 2016; Chum et al., 2007). Rather than modifying the query representation or retraining any model, re-ranking operates directly on the ranked lists themselves, using neighborhood consistency as a proxy for semantic relevance.

The motivation for this refinement step is the *semantic gap*: low-level feature descriptors often fail to capture high-level semantic similarity, so pairwise distance alone may rank visually plausible but semantically incorrect images near the top of the list (Liu et al., 2007; Wang et al., 2016). A descriptor based on color histograms, for instance, may place images of sunflowers and yellow cars in close proximity simply because they share a dominant hue, even though they belong to entirely different semantic categories. Re-ranking methods mitigate this by incorporating contextual evidence derived from the dataset itself, rather than relying solely on the original descriptor distances. The underlying principle is that semantic categories tend to form coherent clusters in the manifold of visual features, and that these clusters can be recovered by analyzing patterns of mutual similarity across the entire collection.

The central intuition behind manifold-based re-ranking is *transitive similarity* (Chum et al., 2007). If image q is similar to image a , and a is similar to image b within their local neighborhoods, then q and b likely share semantic traits even if their direct descriptor distance is large. Conversely, if a candidate image c appears near the top of the initial ranking for q but shares no neighbors with the other top results, it is likely an accidental match caused by a superficial feature overlap. By systematically analyzing neighborhood overlaps across the

dataset, re-ranking algorithms promote images that are globally consistent with the query and demote isolated coincidental matches (Valem; Pedronette, 2016). This transforms retrieval from a local pairwise comparison into a global analysis of the dataset’s topological structure, effectively learning a more precise similarity function from the data itself at query time. This topological approach is particularly effective in high-dimensional feature spaces, where the curse of dimensionality (Bellman, 1957) progressively degrades the reliability of standard distance metrics such as Euclidean or cosine distance: as dimensionality increases, distances between points concentrate and the contrast between nearest and farthest neighbors diminishes, making pairwise ranking unreliable. By reasoning about the manifold on which images reside rather than their raw coordinates, re-ranking algorithms can recover semantic clusters that are geometrically distant in the feature space but tightly connected through reciprocal neighborhood relationships.

2.2 THE UNSUPERVISED DISTANCE LEARNING FRAMEWORK

The Unsupervised Distance Learning Framework (UDLF) provides a modular, iterative approach to manifold-based re-ranking without requiring labeled training data (Valem; Pedronette, 2017). Before describing how the framework operates, it is essential to understand its two primary data structures concretely, as the entire sparse reformulation proposed in this dissertation is grounded in their structure.

The first data structure is the ranked list matrix $\mathbf{R} \in \{1, \dots, N\}^{N \times L}$. Each row i of $\mathbf{R}$ contains the indices of the L nearest neighbors of image x_i , sorted in descending order of similarity. For example, if $\mathbf{R}_{i,1} = 7$, image x_7 is the most similar image to x_i according to the current ranking; if $\mathbf{R}_{i,2} = 3$, image x_3 is the second most similar, and so on up to position L . Each row therefore encodes the local neighborhood of one image, and the full matrix $\mathbf{R}$ encodes the neighborhood structure of the entire collection in $N \times L$ integer entries. This is a compact representation: for $N = 100,000$ images and $L = 1,000$ neighbors, storing $\mathbf{R}$ requires only $100,000 \times 1,000 \times 4 \approx 400$ MB of integer indices, which is manageable on commodity hardware.

The second data structure is the global similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$. Entry $\mathbf{M}_{ij}$ quantifies how similar image x_i is to image x_j after incorporating manifold information through the re-ranking process. Unlike the initial descriptor distance, which is fixed at query time, this score is updated iteratively to reflect neighborhood consistency across the dataset. At each iteration, UDLF updates $\mathbf{M}$ based on the neighborhood relationships encoded in $\mathbf{R}$, then regenerates $\mathbf{R}$ by re-sorting the rows of the updated $\mathbf{M}$. This cycle continues until the rankings converge or a maximum number of iterations is reached. The effect is that images which are mutually consistent in their neighborhoods progressively receive higher similarity scores, while isolated matches are suppressed.

The primary limitation of UDLF arises from the dense representation of $\mathbf{M}$. Storing an $N \times N$ matrix of single-precision floating-point values (4 bytes each, as specified by the

IEEE 754-2019 standard (IEEE, 2019)) requires $4N^2$ bytes of RAM in total. For $N = 10,000$, this amounts to approximately 400 MB; for $N = 100,000$, it reaches approximately 40 GB; and, for $N = 1,000,000$, it would require approximately 4 TB. This quadratic memory growth restricts the framework to specialized high-memory machines. In practice, for workstations and modest research servers equipped with 16 GB or 32 GB of RAM, this growth effectively caps the usable dataset size at approximately $N = 50,000$ images, far below the scale of modern production retrieval systems such as e-commerce catalogues or large-scale medical imaging archives, which routinely contain hundreds of thousands to millions of items. While high-memory cloud instances are available, their per-hour costs, data-transfer requirements, and privacy constraints frequently make local execution preferable for iterative research and deployment in resource-limited settings.

This bottleneck is not due to a lack of computational power: at most L entries per row of $\mathbf{M}$ can influence re-ranking, since each image maintains meaningful relationships with only its L nearest neighbors ($L \ll N$). The remaining $N - L$ entries per row are structurally irrelevant and would be discarded by any top- L pruning step. This irrelevance is definitional rather than intrinsic: because every iteration concludes with a top- L selection step, any entry ranked beyond position L at iteration t cannot propagate its value into iteration $t+1$, regardless of its underlying similarity magnitude. The choice of L therefore defines the information boundary of the algorithm, not the geometry of the data. The matrix is therefore inherently sparse, with only $\mathcal{O}(NL)$ entries that carry information, yet the dense representation forces the system to allocate and traverse all $\mathcal{O}(N^2)$ entries regardless. This structural mismatch between the information content of the problem and its memory footprint is the central inefficiency addressed by this work. Specifically, the K parameter (introduced in Section 2.3) defines the neighbor set used for weight matrix construction, while L defines the target ranked list length maintained in the final similarity matrix.

2.3 TARGET ALGORITHMS: CPRR AND LHRR

The UDLF family encompasses a range of re-ranking algorithms that differ in how they model manifold structure and propagate similarity. In this dissertation, we focus on two algorithms that represent distinct and complementary computational patterns within this family: Cartesian Product Re-ranking (CPRR) (Valem; Pedronette, 2016) and Local Hypergraph Re-ranking (LHRR) (Pedronette; Almeida; Torres, 2014). CPRR was selected because it represents the simplest class of manifold re-ranking: it computes direct co-occurrence statistics over one-hop neighborhoods using matrix-matrix products, making it a natural and transparent test case for sparse algebra. LHRR was selected because it represents a significantly more complex and memory-intensive pattern: it models higher-order relationships through two-hop neighborhood expansion and hyperedge construction, subjecting any sparse engine to substantially greater intermediate memory pressure. Together, these two algorithms span a wide spectrum of the

computational patterns present in manifold-based re-ranking, from lightweight co-occurrence counting to iterative hypergraph propagation, and provide a rigorous basis for evaluating the generalizability of the proposed sparse approach. Both are well-established in the CBIR literature and have been evaluated on standard benchmarks, enabling direct comparison with prior dense implementations.

2.3.1 Cartesian Product Re-ranking (CPRR)

CPRR strengthens the similarity between images that frequently co-occur as neighbors of the same queries (Valem; Pedronette, 2016). At a high level, CPRR operates in three phases, following Valem & Pedronette (2016): (1) a weight matrix $\mathbf{W}$ is computed from the initial ranked lists using a position-decay function; (2) two matrix products ($\mathbf{W}^\top \mathbf{W}$ and $\mathbf{W} \mathbf{W}^\top$) capture co-occurrence and transitive similarity respectively; and (3) the final similarity matrix is their sum, from which updated ranked lists are extracted.

The algorithm begins by constructing a sparse weight matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$ that encodes the relative importance of neighborhood relationships.

For each image i , CPRR assigns a weight to each of its top- K neighbors using the linear decay function given by Equation 1,

$$\mathbf{W}_{ij} = \begin{cases} K - \text{pos}(j, \mathbf{R}_i) & \text{if } j \in \mathbf{R}_{i,1:K} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $\text{pos}(j, \mathbf{R}_i)$ is the 0-indexed rank position of image j in the ranked list of image i , and $K \leq L$ is a truncation parameter that controls the density of the weight matrix independently of the full ranked list length L . The first-ranked neighbor receives weight K , and the weight decays linearly to 1 for the K -th neighbor.

Images ranked beyond position K receive zero weight and are not represented in $\mathbf{W}$. Note that $\mathbf{W}$ is strictly sparse by construction, with exactly K non-zero entries per row and NK non-zeros in total. The refined similarity matrix is computed through two sparse matrix-matrix products that capture complementary aspects of manifold structure:

$$\mathbf{M}_A = \mathbf{W}^\top \mathbf{W} \quad (2)$$

$$\mathbf{M}_B = \mathbf{W} \mathbf{W}^\top \quad (3)$$

$\mathbf{M}_A$ captures *co-occurrence similarity*: entry $\mathbf{M}_{A,jk}$ accumulates a high score if images j and k are frequently cited as top neighbors across the dataset. To see why, observe that column j of $\mathbf{W}$ contains non-zero entries for all images that list x_j among their top- K neighbors; the product $\mathbf{W}^\top \mathbf{W}$ therefore computes, for every pair (j, k) , the weighted overlap between those two columns, that is, how many images simultaneously list both x_j and x_k as

Algoritmo 1: CPRR dense execution.

Input: Ranked lists $\mathbf{R}$ ($N \times L$), parameters K, L
Output: Updated ranked lists $\mathbf{R}'$

```

// Phase 1: build dense weight matrix
1 Allocate  $\mathbf{W}[N][N]$  and initialize with 0
2 for  $i \leftarrow 0$  to  $N - 1$  do
3   for  $j \leftarrow 0$  to  $K - 1$  do
4      $neighbor \leftarrow \mathbf{R}[i][j]$ 
5      $\mathbf{W}[i][neighbor] \leftarrow K - j$ 

// Phase 2: dense matrix multiplication
6 Allocate  $\mathbf{M}_A[N][N], \mathbf{M}_B[N][N], \mathbf{M}[N][N]$  and initialize with 0
7  $\mathbf{M}_A \leftarrow \mathbf{W}^\top \mathbf{W}$  // Requires  $\mathcal{O}(N^2 K)$  operations
8  $\mathbf{M}_B \leftarrow \mathbf{W} \mathbf{W}^\top$ 
9  $\mathbf{M} \leftarrow \mathbf{M}_A + \mathbf{M}_B$ 

// Phase 3: sort and update ranked lists
10 for  $i \leftarrow 0$  to  $N - 1$  do
11    $\mathbf{R}'[i] \leftarrow \text{SORTTOPL}(\mathbf{M}[i], L)$ 

```

neighbors. $\mathbf{M}_B$ captures *transitive similarity*: entry $\mathbf{M}_{B,jk}$ is high if images j and k share many common neighbors, identifying images connected through intermediate nodes in the manifold.

The final similarity matrix is given by Equation 4:

$$\mathbf{M}_{final} = \mathbf{M}_A + \mathbf{M}_B \quad (4)$$

Both products (Equations 2 and 3) follow the Sparse General Matrix-Matrix Multiplication (SpGEMM) format, which involves the multiplication of two sparse input matrices producing a sparse output. Since $\mathbf{W}$ has NK non-zeros, each product can in principle produce up to $\mathcal{O}(NK^2)$ non-zeros before pruning. A top- L selection is then applied to each output row, retaining only the L highest-valued entries and reducing the result to $\mathcal{O}(NL)$ non-zeros. Here K governs the sparsity of the weight matrix construction, while L is the target ranked list length used for retrieval; the two parameters are independent, and the top- L pruning step bridges them.

2.3.2 Local Hypergraph Re-ranking (LHRR)

LHRR models higher-order manifold relationships by treating neighborhood sets as hyperedges in a hypergraph (Pedronette; Almeida; Torres, 2014). Rather than considering only pairwise co-occurrences, LHRR connects images through shared two-hop neighborhood contexts, capturing more complex manifold geometry at the cost of greater computational and memory demands. The algorithm begins by constructing a symmetric position matrix $\mathbf{P} \in \mathbb{R}^{N \times N}$ that encodes mutual neighborhood strength. For every pair (i, j) such that j appears in the ranked list of i , both $\mathbf{P}_{ij}$ and $\mathbf{P}_{ji}$ are accumulated according to the position-based score:

$$\mathbf{P}_{ij} = \mathbf{P}_{ji} = \sum_{i,j:j \in \mathbf{R}_i} (L - \text{pos}(j, \mathbf{R}_i)) \quad (5)$$

Next, the hyperedge matrix $\mathbf{H} \in \mathbb{R}^{N \times N}$ encodes two-hop neighborhood relationships: for each image i , $\mathbf{H}$ connects i not only to its immediate neighbors but also to their neighbors in turn, effectively expanding the local context to a two-hop radius. Edge weights are assigned using a logarithmic decay function that penalizes distant transitive connections:

$$w(p) = \begin{cases} 1.0 & \text{if } p = 1 \\ 1.0 - \frac{\ln(p)}{\ln(K)} & \text{otherwise} \end{cases} \quad (6)$$

The weight of a two-hop path from i through a to b is the product $w(\text{pos}(a, \mathbf{R}_i)) \cdot w(\text{pos}(b, \mathbf{R}_a))$. Once $\mathbf{H}$ is constructed, the algorithm refines the similarity matrix through two SpGEMM operations, $\mathbf{S} = \mathbf{H}\mathbf{H}^\top$ and $\mathbf{S}_{\text{rev}} = \mathbf{H}^\top\mathbf{H}$, whose entries measure hyperedge overlap between image pairs. The position matrix is then updated via the Hadamard (element-wise) product (Horn; Johnson, 1994), which retains only entries simultaneously non-zero in all three operands, progressively tightening the sparsity structure:

$$\mathbf{P}_{t+1} = \mathbf{P}_t \odot (\mathbf{S} \odot \mathbf{S}_{\text{rev}}) \quad (7)$$

This update ensures that only relationships reinforced by hyperedge overlap across two hops survive into the next iteration, gradually filtering noise while amplifying coherent manifold structure. The construction of $\mathbf{H}$ from two-hop expansions is the most memory-intensive step in the entire algorithm: because each image i connects to up to K immediate neighbors and each of those connects to up to K further neighbors, the two-hop expansion can produce up to $\mathcal{O}(NK^2)$ intermediate entries per iteration before any pruning is applied. For $N = 160,000$ and $K = 50$, this amounts to up to 4×10^8 intermediate values, a quantity that a dense or naïvely sparse implementation would be unable to accommodate on commodity hardware. Managing this intermediate growth without materializing a full dense representation is the central system challenge addressed in Chapter 4.

2.4 THE MEMORY HIERARCHY AND SPARSE MATRIX REPRESENTATIONS

Understanding why the dense $N \times N$ matrix becomes impractical in practice, and what constraints a sparse alternative must satisfy, requires a brief examination of the memory hierarchy and the properties of sparse storage formats.

Modern computing systems organize memory as a hierarchy of tiers that trade off speed against capacity (Hennessy; Patterson, 2017). At the top of the hierarchy, CPU registers and on-chip caches (L1, L2, and L3) provide sub-10 ns access latency but are severely limited in size: L1 caches are typically 32–64 KB per core, L2 caches a few hundred kilobytes, and L3

Algoritmo 2: LHRR dense execution.

Input: Ranked lists $\mathbf{R}$ ($N \times L$), parameters K, L , current position matrix $\mathbf{P}_t[N][N]$

Output: Updated position matrix $\mathbf{P}_{t+1}$, refined ranked lists $\mathbf{R}'$

```

// Phase 1: build dense hyperedge matrix (two-hop expansion)
1 Allocate  $\mathbf{H}[N][N]$  and initialize with 0
2 for  $i \leftarrow 0$  to  $N - 1$  do
3    $\mathbf{H}[i][i] \leftarrow 1.0$ 
4   for  $o \leftarrow 0$  to  $K - 1$  do
5      $imgO \leftarrow \mathbf{R}[i][o]$ 
6     for  $j \leftarrow 0$  to  $K - 1$  do
7        $imgJ \leftarrow \mathbf{R}[imgO][j]$ 
8        $\mathbf{H}[i][imgJ] \leftarrow \mathbf{H}[i][imgJ] + w(o) \cdot w(j)$ 

// Phase 2: dense matrix multiplication
9 Allocate  $\mathbf{S}[N][N], \mathbf{S}_{rev}[N][N]$  and initialize with 0
10  $\mathbf{S} \leftarrow \mathbf{H}\mathbf{H}^\top$ 
11  $\mathbf{S}_{rev} \leftarrow \mathbf{H}^\top \mathbf{H}$ 

// Phase 3: dense Hadamard update
12 for  $i \leftarrow 0$  to  $N - 1$  do
13   for  $j \leftarrow 0$  to  $N - 1$  do
14      $\mathbf{P}_{t+1}[i][j] \leftarrow \mathbf{P}_t[i][j] \cdot \mathbf{S}[i][j] \cdot \mathbf{S}_{rev}[i][j]$ 

// Phase 4: sort and update ranked lists
15 for  $i \leftarrow 0$  to  $N - 1$  do
16    $\mathbf{R}'[i] \leftarrow \text{SORTTOPL}(\mathbf{P}_{t+1}[i], L)$ 

```

caches a few tens of megabytes shared across cores. Below the caches, Dynamic Random-Access Memory (DRAM) provides gigabytes of capacity at a significantly higher latency of roughly 60–100 ns (Hennessy; Patterson, 2017). When the working set of an application exceeds DRAM capacity, the operating system pages data to secondary storage such as an SSD, whose latency is orders of magnitude higher. The performance consequence of crossing this boundary is severe: applications that require random access to data scattered across many pages trigger frequent page faults (Silberschatz; Galvin; Gagne, 2018), shifting execution from being compute-bound to I/O-bound. In the extreme case, a condition known as *thrashing* (Silberschatz; Galvin; Gagne, 2018) occurs, in which the OS spends the majority of its time swapping pages rather than executing useful instructions. Re-ranking algorithms are particularly susceptible to this form of performance degradation because they involve irregular graph traversal, where successive memory accesses are scattered across the similarity matrix with little spatial locality, making hardware prefetching ineffective and maximizing the probability of cache misses.

The standard approach to storing sparse matrices in high-performance computing is

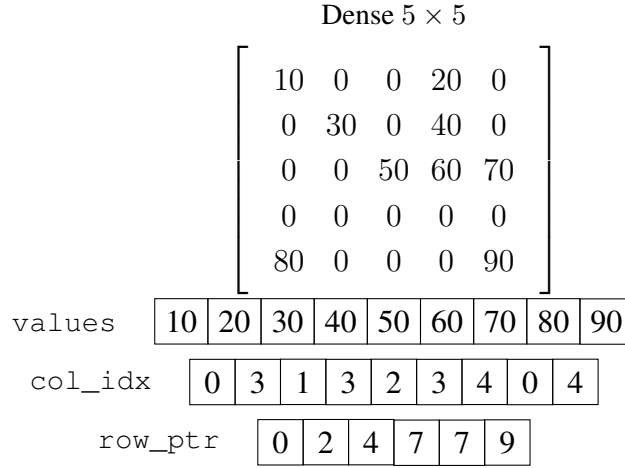


Figure 2 – The Compressed Sparse Row (CSR) format applied to a 5×5 example matrix. Non-zero values and their column indices are stored in two companion arrays: `values` and `col_idx`, respectively. The `row_ptr` array stores the offset in `values` at which each row begins, enabling $\mathcal{O}(1)$ access to any row and $\mathcal{O}(\text{nnz}_i)$ traversal of row i , where nnz_i denotes the number of non-zeros in that row.

based on three widely used formats (Saad, 2003). The Coordinate Format (COO) is the simplest: it represents the matrix as an unordered list of triplets (r, c, v) , where r is the row index, c the column index, and v the non-zero value. COO is flexible and easy to construct in parallel, because multiple threads can independently append triplets to separate local buffers without coordination. However, COO is inefficient for the row-oriented traversals required by re-ranking: there is no direct way to access all non-zeros in a given row without scanning the entire list, making operations like SpGEMM and Hadamard products expensive. The Compressed Sparse Column (CSC) format is the column-major analogue of CSR, grouping non-zeros by column; it is useful for operations that require efficient column access, such as computing $\mathbf{W}^T$ in CPRR, but is not the natural format for row-oriented ranked-list traversal.

The Compressed Sparse Row (CSR) format (Saad, 2003) is the industry standard for high-performance sparse operations. As illustrated in Figure 2, CSR organizes non-zero entries into three contiguous arrays: `values` stores the non-zero values in row-major order, `col_idx` stores the corresponding column indices (required to identify the matrix of each non-zero during row traversal and SpGEMM accumulation), and `row_ptr` stores, for each row i , the index in `values` at which that row’s non-zeros begin. This layout provides $\mathcal{O}(1)$ access to the start of any row and $\mathcal{O}(\text{nnz}_i)$ traversal of row i , where nnz_i denotes the number of non-zeros in that row. Because non-zeros within a row are stored contiguously in memory, CSR traversal exhibits good spatial locality and is well-suited to the cache hierarchy: a sequential scan of row i loads nnz_i values from a contiguous memory region, minimizing cache line waste. The primary weakness of CSR is that it is a static format: once constructed, inserting new non-zeros or changing the sparsity pattern requires rebuilding the arrays from scratch. This makes CSR unsuitable for the initial *construction* phase of a sparse engine, where triplets are emitted dynamically and in

arbitrary order, but ideal for the subsequent *query* phase, where rows are traversed repeatedly during SpGEMM and Hadamard operations. A natural strategy is therefore to construct matrices in COO format, accumulating triplets in thread-local buffers without synchronization, and then convert to CSR once construction is complete via a sort and prefix-sum pass.

A critical challenge is *false sharing*, which occurs when two or more threads independently modify variables that happen to reside on the same 64-byte hardware cache line (Drepper, 2007). Because the cache-coherence protocol operates at cache-line granularity, a write by one thread invalidates the entire cache line in all other cores, forcing them to reload data they have not logically modified. This cache-line bouncing can substantially reduce the throughput gains of parallelism on thread-local data structures that are packed too tightly in memory. Finally, the parallel construction of sparse matrices from many concurrent triplet emissions raises a synchronization challenge: naïve implementations that direct all threads to write into a single shared output buffer require locking, which serializes execution and limits scalability (Buluç; Gilbert, 2011). A well-known solution is *row-split partitioning* (Buluç; Gilbert, 2011), in which the row space is statically divided among threads so that each thread owns a disjoint set of rows and writes exclusively to its own private buffer, eliminating the need for any synchronization primitive during the construction phase.

2.5 SUMMARY

This chapter has established the formal foundations for the remainder of the dissertation. We defined the re-ranking task and its role in bridging the semantic gap in CBIR, and formalized the two core data structures of the UDLF framework: the ranked list matrix $\mathbf{R}$, compact at $\mathcal{O}(NL)$ space, and the similarity matrix $\mathbf{M}$, which in its dense form requires $\mathcal{O}(N^2)$ space despite only $\mathcal{O}(NL)$ of its entries being semantically meaningful. We analyzed CPRR and LHRR in detail, showing that both reduce to sequences of SpGEMM and Hadamard operations whose intermediate products can grow to $\mathcal{O}(NK^2)$ before pruning.

We then examined the memory hierarchy and established that dense re-ranking inevitably crosses the DRAM capacity boundary for large datasets, triggering thrashing; that CSR is the appropriate query-phase format while COO is appropriate for construction; and that thread-local memory allocation, false-sharing avoidance, and row-split partitioning are necessary conditions for scalable parallel execution.

Chapter 3 now surveys prior work in scalable retrieval and sparse computation, establishing the gap that motivates the approach developed in Chapter 4.

3 RELATED WORK

This chapter surveys prior works most directly relevant to the sparse reformulation of manifold-based re-ranking proposed in this dissertation. We organize the literature into three thematic areas: Section 3.1 surveys manifold-based ranking and diffusion processes, establishing the theoretical lineage from which CPRR and LHRR descend and situating the quadratic memory cost as a class-wide property. Section 3.2 examines prior approaches to scalability in retrieval, documenting how the field has responded to this barrier and identifying the structural gap that none of them address. Section 3.3 surveys the SpGEMM literature that provides the computational substrate for the proposed engine. The chapter closes by synthesizing these three areas: no prior effort has reformulated manifold re-ranking algorithms as explicit sparse matrix operations and built a custom engine that controls intermediate memory through fused truncation.

3.1 MANIFOLD-BASED RE-RANKING AND DIFFUSION PROCESSES

The theoretical foundation for manifold-based re-ranking is the observation that visual similarity is not well captured by Euclidean distance in high-dimensional feature spaces, and that the data reside on a lower-dimensional manifold whose intrinsic geometry contains the information needed to recover semantic similarity (Roweis; Saul, 2000; Tenenbaum; Silva; Langford, 2000). The earliest explicit use of manifold geometry for retrieval was the seminal work of Zhou et al. (Zhou et al., 2003), who proposed ranking data points according to their position on a graph-based manifold, formulating the problem as a regularization task on a weighted graph whose edges encode pairwise affinities: a query signal is propagated across the graph to assign scores to all database nodes through an iterative diffusion process. The resulting scores reflect both direct proximity to the query and indirect proximity through the manifold topology. This formulation inspired a large body of subsequent work that shares a common structure: construct an affinity matrix, apply some form of iterative propagation or matrix power, and extract improved rankings from the converged result. The critical systems-level observation is that the full $N \times N$ affinity matrix must be materialized in memory, and the iterative products require repeated traversal of this structure. Zhou et al.’s algorithm exhibits $\mathcal{O}(N^2)$ space complexity, a cost that is inherited by nearly all subsequent methods in this family without modification.

Donoser and Bischof (Donoser; Bischof, 2013) provided a systematic unification of this family, showing that most diffusion-based retrieval methods can be expressed as specific instances of a single generic framework parameterized by choices of initialization, transition matrix, and diffusion operator. Their analysis confirmed that the transition matrix constrained to k -nearest neighbors (referred to as P_{kNN} in their formulation) is the most practically useful

variant, since truncating the affinity to top- k neighbors per row limits the number of edges and therefore controls the density of the propagation graph. This is an important observation: it implies that the information-carrying part of the full $N \times N$ affinity matrix is at most $\mathcal{O}(NK)$ non-zero entries, and that any entries beyond the top- K per row can be discarded without degrading retrieval quality. Specifically, the K parameter defines the neighborhood density used for propagation, whereas the L parameter defines the final ranked list length returned to the user; the two are related but serve distinct roles in the memory-efficiency trade-off. Donoser and Bischof, however, continued to treat this truncated matrix as a dense object in their experimental framework. Together with Zhou et al. (Zhou et al., 2003), these works confirm that the pre-allocation of a dense $N \times N$ affinity matrix is a class-wide convention in manifold-based retrieval, not a peculiarity of any single framework, a point already established in Chapter 1 and reinforced here by the breadth of the lineage surveyed.

Within the UDLF family specifically, the two algorithms targeted in this dissertation represent complementary ends of a computational spectrum. CPRR (Valem; Pedronette, 2016) computes co-occurrence statistics over one-hop neighborhoods through matrix products, while LHRR (Pedronette; Almeida; Torres, 2014; Pedronette et al., 2019) constructs hyperedge matrices that encode two-hop transitive relationships. Valem and Pedronette (Valem; Pedronette, 2017) consolidated these and other methods into a software framework (UDLF) that provides a uniform interface for evaluation. In that framework, both algorithms operate on a dense similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$, which is pre-allocated in full regardless of sparsity. As documented in (Valem; Pedronette, 2017), this design limits practical experiments to datasets of moderate size, and the authors explicitly flag memory as the primary scalability bottleneck.

3.2 SCALABILITY APPROACHES IN MANIFOLD RETRIEVAL

The retrieval community has responded to the quadratic memory barrier through five broad strategies: sparse affinity construction, approximate graph methods, offline precomputation, sparse feature encodings, and parallel optimization of dense re-ranking. Each addresses the symptom of high memory cost through a different mechanism, and each leaves a distinct residual gap.

3.2.1 Sparse affinity construction

The most conceptually direct approach is to truncate the affinity matrix to its top- K entries per row from the outset, representing it as a sparse matrix and solving the diffusion system over this sparse structure. Iscen et al. (Isцен et al., 2017) pursued this strategy in the context of regional diffusion for instance retrieval. Their method truncates the affinity matrix to the top- K mutual neighbors, forming a sparse symmetric graph, and then solves the resulting sparse linear system using a conjugate gradient solver. The key insight is that because only mutual neighbors are connected, the number of edges per node is substantially less than K ; at

the scale of 10^5 images, diffusion over this sparse graph achieves query times below one second. Critically, however, the approach does not apply to the CPRR/LHRR family: the sparse affinity in Iscen et al. is the *input* to a linear system solver, not the operand of a matrix product. Solving $(\mathbf{I} - \alpha \hat{\mathbf{A}})\mathbf{x} = \mathbf{y}$ over a sparse $\hat{\mathbf{A}}$ requires very different machinery than computing $\mathbf{W}^\top \mathbf{W}$ over a sparse $\mathbf{W}$, because the latter operation must manage a potentially $\mathcal{O}(NK^2)$ intermediate product before pruning. The linear solver framework sidesteps this intermediate expansion entirely; the SpGEMM framework must confront it directly.

3.2.2 Offline precomputation

In a subsequent work, Iscen et al. (Isцен et al., 2018) proposed an offline embedding approach that reduces manifold search to Euclidean search at query time by precomputing an approximate graph Fourier basis. This effectively amortizes the cost of manifold exploration over an offline phase, yielding online query latencies comparable to standard nearest-neighbor search. The trade-off is that the offline computation is tied to a fixed graph structure and must be recomputed entirely when the database changes, and the method requires storing the Fourier basis vectors for all N nodes. More importantly, this approach changes the *interface* to retrieval: the system now returns manifold-aware rankings through an embedding, rather than computing an updated $N \times N$ similarity matrix. It is therefore not a drop-in replacement for UDLF-style algorithms that iteratively refine $\mathbf{M}$ and then re-sort $\mathbf{R}$.

Bai et al. (Bai; Bai; Tian, 2017) addressed scalability in person re-identification by representing manifold affinities as a sparse graph and developing a manifold-based affinity learning procedure with linear online time complexity. Their approach introduces a supervised smoothing step on the manifold using pairwise constraints, which enables sparse propagation without materializing the full affinity matrix. That work confirms that linear-cost manifold learning is achievable when the graph is kept sparse throughout; however, the method is supervised and specific to person re-identification (ReID), and does not model the SpGEMM-structured update rules that characterize CPRR and LHRR.

3.2.3 Approximate graph construction

A separate line of work targets the cost of building the k -nearest neighbor graph itself, rather than the cost of operating over it. Wang et al. (Wang et al., 2012) and Magliani et al. (Magliani; Prati, 2019) developed approximate k NN graph construction algorithms based on Locality-Sensitive Hashing (LSH), exploiting the observation that exact k NN search has $\mathcal{O}(N^2)$ cost while approximate methods can achieve similar retrieval quality in $\mathcal{O}(N \log N)$ or linear time. Magliani et al. showed that an LSH-based approximate graph achieves the same retrieval performance as the exact graph after diffusion while reducing construction time by approximately $18\times$ on a 10^5 -image dataset. Kawai et al. (Kawai et al., 2024) pursued a related but distinct angle: rather than approximating the graph structure itself, they proposed a rank-

based hashing scheme that replaces the distance computations underlying k -nearest neighbor lookup with hash-table probes. By encoding ranked-list position information into compact binary hash codes, their method reduces both the nearest-neighbor search phase and the spatial overhead associated with storing pairwise distance values. Experiments on standard image retrieval benchmarks demonstrate that the hash-based lookup achieves retrieval effectiveness competitive with exact distance computation while operating in substantially reduced memory and time. This contribution addresses the k NN *graph construction* phase and is therefore orthogonal to the problem studied in this dissertation: once the ranked lists $\mathbf{R}$ have been computed, whether by exact or hash-based search, the question of how to execute the subsequent SpGEMM-structured update rules of CPRR and LHRR over sparse representations of $\mathbf{M}$ remains open and is not addressed by rank-based hashing.

3.2.4 Sparse feature encodings

A different strategy avoids constructing an affinity matrix altogether by encoding contextual information as sparse feature vectors and reducing re-ranking to fast vector comparison. Bai and Bai (Bai; Bai, 2016) proposed Sparse Contextual Activation (SCA), which encodes the local neighborhood distribution of each image as a sparse vector and computes re-ranking scores via a generalized Jaccard similarity using an inverted index. This yields average per-query re-ranking times below 1 ms, a significant efficiency gain relative to matrix-based methods. The cost is that the matrix-algebraic structure of CPRR and LHRR’s update rules (particularly the hyperedge construction and two-hop propagation of LHRR) is lost. SCA is a fundamentally different formulation of re-ranking that trades the ability to model higher-order manifold relationships for extreme speed. It does not provide a mechanism for applying SpGEMM or Hadamard products to sparse matrices, and therefore cannot serve as a direct basis for reformulating CPRR and LHRR.

3.2.5 Replacing the Dense Matrix: a Dictionary-Based Approach

The work most structurally relevant to this dissertation is that of Baldassin et al. (Baldassin et al., 2019), who applied a dictionary-based representation to the ReckNN algorithm (Pedronette; Penatti; Torres, 2014) as an alternative to the dense $N \times N$ similarity matrix. Concretely, the dictionary is implemented as an array of N hash tables, each pre-allocated with n_s entries (where n_s is the ranked-list depth parameter, directly analogous to L in this dissertation). The resulting space complexity is $\mathcal{O}(Nn_s) = \mathcal{O}(NL)$, which matches the asymptotic memory bound of the sparse reformulation presented in this work.

Despite this promising complexity, the dictionary approach has three key limitations. First, the structure is *pre-allocated*, not lazily constructed: the full $\mathcal{O}(NL)$ hash-table array is allocated before any computation, so an out-of-memory failure can still occur if L is large. Second, no experiments characterising memory usage as a function of N and L are reported;

the evaluation focuses on execution time, and the authors note that for smaller datasets (where the dense matrix fits in DRAM) the dictionary variant is *slower* than the dense baseline due to hash-table overhead. Third, the dictionary was designed specifically for the collaborative-score access pattern of ReckNN and is not directly transferable to CPRR or LHRR: both algorithms require *SpGEMM operations* over the stored data structure, whereas ReckNN’s score update is a direct key-value accumulation. Applying a dictionary to CPRR’s $\mathbf{W}^T \times \mathbf{W}$ or to LHRR’s two-hop hyperedge expansion would require a fundamentally different integration strategy.

This establishes an important baseline: achieving $\mathcal{O}(NL)$ space complexity is a necessary but not sufficient condition. The challenge specific to CPRR and LHRR is controlling the *intermediate* density that arises during SpGEMM products, which can transiently reach $\mathcal{O}(NK^2)$ non-zeros before any pruning is applied, a problem the dictionary approach does not address.

3.2.6 Why Sparse Matrix Libraries are not Enough

While general-purpose sparse linear algebra libraries such as Intel MKL, GraphBLAS, and Eigen provide high-performance SpGEMM routines, they are designed for scientific computing workloads where the sparsity pattern of the output is unknown. These libraries follow a compute-then-prune model: they materialize the complete intermediate product ($\mathbf{H}\mathbf{H}^T$) before applying any filtering or truncation. As established in Section 4.1, this intermediate expansion can reach $\mathcal{O}(NK^2)$ non-zeros, exceeding the 8–16 GB RAM capacity of commodity hardware for large datasets even when the final $\mathcal{O}(NL)$ result would fit. The custom engine presented in this work diverges from these libraries by integrating top- L selection directly into the multiplication logic, ensuring that the memory footprint never exceeds the $\mathcal{O}(NL)$ bound at any point during execution.

3.2.7 Foundational gap

Across all of these approaches, a common pattern emerges: each method achieves scalability by *changing the algorithm*, either by moving to a linear solver, precomputing an offline embedding, approximating the graph, or replacing the matrix update with a vector comparison. None of them take an algorithm that is expressed as matrix-matrix products ($\mathbf{W}^T \mathbf{W}$, $\mathbf{H}\mathbf{H}^T$, and Hadamard operations over $\mathbf{P}$) and ask: can these specific operations be executed over sparse representations of the operands, with bounded intermediate memory, without approximation? This question constitutes the central objective of the present dissertation.

3.3 SPARSE MATRIX-MATRIX MULTIPLICATION

The computational primitive at the core of this dissertation is Sparse General Matrix-Matrix Multiplication (SpGEMM): the multiplication of two sparse matrices whose output is also sparse. SpGEMM has a long history in scientific high-performance computing, where it appears

in algebraic multigrid solvers, graph algorithms, and computational biology (Buluç; Gilbert, 2012). Its algorithmic complexity differs fundamentally from dense matrix multiplication: the number of non-zeros in the output cannot be determined in advance, requires separate *symbolic* and *numeric* phases, and the intermediate product before merging can be substantially larger than either the input or output matrices (Gustavson, 1978).

Gustavson (Gustavson, 1978) established the foundational row-wise SpGEMM algorithm, in which each output row is computed by iterating over the non-zeros of the corresponding input row and accumulating partial products into a temporary workspace. This algorithm has $\mathcal{O}(\text{nnz}(\mathbf{A}) + \text{nnz}(\mathbf{B}) + \text{flops})$ sequential work, where flops is the number of scalar multiplications required, and forms the basis for virtually all subsequent implementations. The primary bottleneck identified by Gustavson’s analysis is the *accumulator*: the temporary data structure that collects partial products for a single output row. Common choices include dense vectors (fast but requiring $\mathcal{O}(N)$ workspace per row), hash tables (memory-efficient but cache-unfriendly), and heap-based accumulators (memory-efficient and sorted, but with $\mathcal{O}(K^2 \log K)$ cost per row).

Buluc and Gilbert (Buluç; Gilbert, 2012) extended Gustavson’s framework to distributed memory parallel systems, introducing two-dimensional block partitioning strategies for SpGEMM that achieve communication-optimal scaling. Their work established SpGEMM as a core HPC primitive and provided the CombBLAS library as a practical implementation. Although their target applications are graph analytics and multigrid (not image retrieval), the algorithmic insights are directly applicable: in particular, their analysis of intermediate product growth and the conditions under which output sizes can be bounded is directly relevant to the bounded-output variant of SpGEMM required by CPRR and LHRR, where each output row must be truncated to L entries.

In the context of graph clustering, the Markov Cluster (MCL) algorithm (Dongen, 2000) provides the closest prior use of SpGEMM with output pruning in an application resembling re-ranking. MCL iteratively applies two steps: matrix expansion (SpGEMM squaring the graph adjacency matrix) and inflation (Hadamard power followed by column normalization), with a pruning step between iterations to maintain sparsity by retaining only the top- K entries per column. This pruning step is essential to prevent the intermediate product from growing unboundedly, and the fused-multiply-and-prune pattern that MCL requires is analogous to what LHRR demands during hyperedge construction. To our knowledge, however, MCL has not been applied to image retrieval or re-ranking, and its column-pruning structure is different from the row-pruning and Hadamard-based update rules of LHRR. Moreover, MCL implementations generally treat the expansion-inflation-prune loop as three separate passes, whereas the LHRR update requires that pruning be fused with the multiplication to avoid materializing an $\mathcal{O}(NK^2)$ intermediate product that would exceed available memory.

A large body of work addresses SpGEMM performance on modern hardware: GPU-based algorithms using Expand-Sort-Compress pipelines (Dalton; Olson; Bell, 2015), hash-accumulator variants for irregular sparsity patterns (Nagasaka et al., 2019), and methods targeting

Table 1 – Comparison of related approaches along key dimensions.

Method	Mem. model	No approx.	Int. exp. bnd.	SpGEMM-str.	Comm. hw.
(Zhou et al., 2003)	Dense	✓	—	—	×
(Donoser; Bischof, 2013)	Dense	✓	—	—	×
(Isen et al., 2017)	Sparse	✓	N/A	×	✓
(Isen et al., 2018)	Embedding	×	N/A	×	✓
(Bai; Bai, 2016)	Inverted idx	×	N/A	×	✓
(Bai; Bai; Tian, 2017)	Sparse	×	N/A	×	✓
(Kawai et al., 2024)	Dense	✓	—	—	×
(Baldassin et al., 2019)	Dict.	✓	—	—	×
This work	Sparse	✓	✓	✓	✓

Memory model: Dense = pre-allocated $N \times N$; Sparse = $\mathcal{O}(NL)$ or better; Dict. = hash-map representation (pre-allocated $N \times n_s$).

No approx.: retrieval quality is preserved exactly relative to the dense baseline.

Int. exp. bounded: intermediate SpGEMM products are prevented from exceeding $\mathcal{O}(NL)$.

SpGEMM-structured: the update rules involve matrix-matrix products executed as such.

Commodity hw.: validated on hardware with ≤ 16 GB RAM.

—: the dimension is not applicable as a concept to this approach.

N/A: the approach is applicable but was not evaluated by the authors on commodity hardware.

output size estimation to reduce memory overprovisioning (Azad et al., 2016). These contributions are relevant to the design of an efficient sparse engine, but none of them are applied in the context of retrieval, and none address the specific constraint of bounded per-row output size that is the central requirement of the proposed reformulation. The sparse linear algebra literature treats SpGEMM as a general-purpose primitive whose output size is unknown and must be estimated or measured; the re-ranking context, by contrast, demands a *truncated* SpGEMM whose output is *defined* to contain at most L non-zeros per row. This mismatch, and the opportunity it creates for memory-efficient fused kernels, is one of the central engineering contributions of this work.

3.4 SUMMARY AND IDENTIFIED GAP

Table 1 organizes the approaches surveyed in this chapter along five key dimensions: memory model, preservation of retrieval quality, bounded intermediate expansion, SpGEMM-structured update rules, and validation on commodity hardware. The discussion below examines each dimension and identifies the gap that no prior work has addressed.

The survey above reveals a clear and consistent gap in the literature. Manifold-based re-ranking methods (Zhou et al., 2003; Donoser; Bischof, 2013; Valem; Pedronette, 2016; Pedronette; Almeida; Torres, 2014) achieve strong retrieval quality by exploiting dataset manifold structure through iterative matrix operations, but inherit $\mathcal{O}(N^2)$ spatial complexity from the dense similarity matrix. Prior scalability efforts resolve this problem by modifying the algorithms: replacing the similarity matrix with a linear system and applying a conjugate

gradient solver (Isen et al., 2017), precomputing an offline embedding (Isen et al., 2018), computing over a sparse supervised graph (Bai; Bai; Tian, 2017), or abandoning matrix algebra in favor of inverted-index vector comparison (Bai; Bai, 2016). These modifications all achieve efficiency gains, but they do so by moving away from the SpGEMM structure of CPRR and LHRR, not by executing that structure more efficiently. The most direct predecessor of this work, Baldassin et al. (Baldassin et al., 2019), suggested replacing the matrix with a dictionary, achieving $\mathcal{O}(NL)$ space complexity for ReckNN, but their approach still cannot extend the memory capacity envelope: the $\mathcal{O}(N^2)$ footprint remains the hard ceiling.

Kawai et al. (Kawai et al., 2024) reduce spatial cost at the graph *construction* stage through rank-based hashing, but the sparse reformulation of the subsequent matrix operations is outside their scope.

The SpGEMM literature (Gustavson, 1978; Buluç; Gilbert, 2012; Dalton; Olson; Bell, 2015; Nagasaka et al., 2019) provides efficient general-purpose primitives for sparse matrix products, but has no connection to retrieval and does not address the bounded-output constraint that is specific to the re-ranking setting. The closest operational analogue, MCL’s fused-expand-prune pattern, addresses intermediate product growth in a graph clustering context but does not map onto the row-pruned SpGEMM + Hadamard structure required by LHRR, and has not been applied to image retrieval.

No prior work applies sparse linear algebra primitives, specifically SpGEMM and the Hadamard (element-wise) product (defined and motivated in Section 2.3), as the direct implementation framework for manifold re-ranking algorithms, constructs a purpose-designed sparse engine that controls intermediate memory through fused truncation, and validates that the resulting system is both quality-neutral with respect to the dense baseline and practically scalable to large collections on commodity hardware. This is the contribution of Chapter 4.

4 SPARSE REFORMULATION OF MANIFOLD RE-RANKING

This chapter presents a reformulation of manifold re-ranking algorithms as explicit sparse matrix operations, together with a custom sparse engine that enforces bounded memory throughout execution. Section 4.1 identifies the failure modes of both dense implementations and general-purpose sparse libraries on large-scale re-ranking. Section 4.2 then shows formally how the update rules of CPRR and LHRR map exactly onto three sparse linear algebra primitives: sorted row merging, the Hadamard product, and SpGEMM. Section 4.3 describes the high-performance engine built to realize this mapping, addressing the hardware-level bottlenecks identified in Section 4.1.

4.1 IDENTIFICATION OF STRUCTURAL BOTTLENECKS

We first characterize the failure modes of existing approaches. This section identifies two primary structural bottlenecks:

1. **Quadratic Spatial Complexity:** Dense implementations require $\mathcal{O}(N^2)$ memory, which, as established, scales to hundreds of gigabytes for $N > 10^5$, exceeding commodity hardware capacity;
2. **Intermediate Density Explosion:** General-purpose sparse libraries compute the full product before pruning, leading to intermediate memory usage that can reach $\mathcal{O}(NK^2)$ non-zeros, causing Out-of-Memory failures even when the final output is sparse.

Beyond these fundamental scalability barriers, our engine implementation also identifies and addresses secondary implementation-level bottlenecks related to dynamic allocator pressure and multi-core synchronization/false-sharing overheads. These are treated as secondary optimizations in Section 4.3 rather than primary drivers of the problem definition.

Two distinct families of approaches are available for executing the update rules of manifold re-ranking: dense implementations, which pre-allocate the full $N \times N$ similarity matrix and operate over it, and general-purpose sparse library implementations, which represent the matrix in a standard sparse format and apply off-the-shelf SpGEMM routines. Both are inadequate for large-scale re-ranking, but for different reasons, and understanding both failure modes precisely is necessary to motivate the design decisions of the engine described in Section 4.3.

Standard UDLF implementations allocate a global similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$ prior to any computation, requiring $4N^2$ bytes of RAM in full before the first update rule is applied. As established in Section 2.2, this footprint reaches 40 GB for $N = 100,000$ and 102 GB for the VegFru dataset ($N = 160,731$), exceeding the capacity of commodity workstations equipped with 16 GB or 32 GB of RAM. A common workaround is to extend the virtual address space to secondary storage using memory-mapped files (`mmap`), which prevents Out-of-Memory

(OOM) termination by substituting disk-backed virtual pages for physical memory. However, re-ranking involves irregular graph traversal of $\mathbf{M}$: successive accesses jump to arbitrary row-column coordinates with no spatial locality, causing each access to a cold page to trigger a page fault. Under sufficiently high page-fault rates, the operating system enters a thrashing regime (Silberschatz; Galvin; Gagne, 2018), spending the majority of its cycles servicing page faults rather than executing computation. Profiling confirms that `mmap`-backed re-ranking on VegFru does not complete within practical time limits on commodity hardware; the bottleneck is I/O throughput, not algorithmic work. The dense representation therefore fails at two levels: it requires hardware that exceeds the commodity tier, and the only available workaround degrades performance to infeasibility.

4.2 SPARSE PRIMITIVE MAPPING

This section demonstrates that the update rules of both CPRR and LHRR decompose exactly into three sparse linear algebra primitives: sorted row merging, the Hadamard product, and SpGEMM.

The first is the sorted row merge, required by CPRR’s summation $\mathbf{M}_A + \mathbf{M}_B$. Because both operand matrices are stored in CSR with column indices sorted in ascending order within each row, the addition is implemented as a two-pointer linear scan: two cursors advance through the column-index arrays of the corresponding rows, emitting the smaller index at each step and summing values when both cursors point to the same column. The merge runs in $\mathcal{O}(\text{nnz}_i(\mathbf{M}_A) + \text{nnz}_i(\mathbf{M}_B))$ per row, and the output is already sorted, eliminating any post-merge sort pass; a final top- L selection retains only the highest-value entries.

The second is the binary-search Hadamard product required by LHRR’s update rule (Equation 7). The Hadamard product $\mathbf{P} \odot \mathbf{Q}$ is implemented by iterating over the *sparser* of the two operands’ rows and performing a binary search into the *denser* operand’s sorted column-index array for each non-zero column. This yields $\mathcal{O}(\text{nnz}_i(\mathbf{P}) \cdot \log(\text{nnz}_i(\mathbf{Q})))$ per row in the worst case; in practice, because LHRR’s Hadamard operands have similar sparsity levels bounded by L , the search depth is $\mathcal{O}(\log L)$, which is a small constant for any realistic L value.

Finally, the third primitive is the Fused SpGEMM, which is the core of the engine and is described in detail in Section 4.3.4.

The decomposition into these three primitives is not an approximation: every arithmetic operation performed by the dense implementations has an exact counterpart in the sparse formulation, and the final per-row rankings produced by both are identical.

At a high level, both CPRR and LHRR follow the same three-phase structure: (1) build one or more sparse weight matrices from the ranked lists $\mathbf{R}$ using a position-decay function; (2) compute matrix products and element-wise updates over those sparse matrices; and (3) reconstruct the ranked lists by sorting the rows of the updated similarity structure. The key observation is that *every* matrix involved in these phases has at most $\mathcal{O}(NL)$ non-zeros once the

top- L selection is enforced, and that selection can be fused directly into each multiplication step.

The three primitives (SpGEMM, Hadamard, RowMerge) are algorithm-agnostic with respect to re-ranking methods that satisfy two conditions: (1) their update rules are expressible as matrix products over the ranked list structure (excluding, for example, methods that require global convergence criteria computed over the full $N \times N$ matrix); and (2) the desired output per row is the top- L entries of a score function, enabling truncated SpGEMM. Any re-ranking algorithm satisfying these conditions potentially admits the same sparse treatment with $\mathcal{O}(NL)$ memory bounds. Methods that require global convergence criteria over the full $N \times N$ matrix, such as spectral methods that compute eigenvectors of the affinity matrix, do not satisfy condition (1) and fall outside the scope of this reformulation. CPRR (Valem; Pedronette, 2016) and LHRR (Pedronette et al., 2019) were chosen because they represent the two extremes of this family (one-hop co-occurrence and two-hop hyperedge expansion), providing a rigorous basis for evaluating generalizability. Other algorithms in the UDLF family, such as ReckNN (Pedronette; Penatti; Torres, 2014) and RL-Sim (Pedronette; Torres, 2013), employ one-hop or two-hop graph operations that are structurally intermediate between CPRR and LHRR. By demonstrating that both extremes map exactly to the three primitives, we provide strong evidence that intermediate algorithms would also admit the same sparse treatment.

Recall from Section 2.3 that CPRR computes a weight matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$ with at most K non-zeros per row, derived directly from the ranked lists $\mathbf{R}$ using the linear decay of Equation 1. Algorithm 3 summarizes the complete sparse execution. The matrix $\mathbf{W}$ is built in line 1, and is sparse by construction, with exactly NK non-zeros in total. The two products $\mathbf{M}_A = \mathbf{W}^\top \mathbf{W}$ (line 2) and $\mathbf{M}_B = \mathbf{W} \mathbf{W}^\top$ (line 3) are both instances of SpGEMM applied to sparse operands. These map directly to the *Cartesian Product* and *Reverse Cartesian Product* phases found in the original UDLF implementation, where they respectively capture co-occurrence and transitive similarity. Unlike the dense baseline which performs full symmetric updates (M_{ij} and M_{ji}), our sparse engine leverages row-split construction to update each row independently, ensuring exact bitwise equivalence in the final sorted ranked lists while eliminating the need for global locks or atomic operations through the Row-Split pattern described in Section 4.3.3. Entry (j, k) of $\mathbf{M}_A$ is non-zero only if there exists at least one image i such that $\mathbf{W}_{ij} > 0$ and $\mathbf{W}_{ik} > 0$, meaning images j and k co-occur as neighbors of some common image; the maximum number of non-zeros per output row before pruning is therefore bounded by $\mathcal{O}(K^2)$. After applying the top- L selection per row, the output is reduced to $\mathcal{O}(NL)$ total non-zeros. The final summation $\mathbf{M}_{\text{final}} = \mathbf{M}_A + \mathbf{M}_B$ (Equation 4, line 4) is a row-wise merge of two sparse matrices with potentially overlapping column supports, implemented as a two-pointer linear scan over sorted column-index arrays, running in $\mathcal{O}(\text{nnz}_i(\mathbf{M}_A) + \text{nnz}_i(\mathbf{M}_B))$ per row. No $N \times N$ structure is instantiated at any point.

Although sorting each row has an $\mathcal{O}(L \log L)$ nominal cost, in practice this step is highly efficient because both $\mathbf{M}_A$ and $\mathbf{M}_B$ are already stored with column indices in ascending order (a guarantee of the FusedSpGEMM kernel, which sorts `active_cols` before emission).

Algoritmo 3: CPRR sparse execution.

Input: Ranked list matrix $\mathbf{R}$ ($N \times L$), truncation parameter K , output length L
Output: Refined ranked lists $\mathbf{R}'$

```

// Phase 1: build sparse weight matrix
1  $\mathbf{W} \leftarrow \text{SPARSEBUILD}(\mathbf{R}, K, \text{decay} = \text{linear})$  //  $NK$  non-zeros,  $\mathcal{O}(NL)$  storage

// Phase 2: two fused SpGEMM calls
2  $\mathbf{M}_A \leftarrow \text{FUSEDSPGEMM}(\mathbf{W}^\top, \mathbf{W}, L)$  // top- $L$  per row
3  $\mathbf{M}_B \leftarrow \text{FUSEDSPGEMM}(\mathbf{W}, \mathbf{W}^\top, L)$ 

// Phase 3: merge and reconstruct
4  $\mathbf{M}_{\text{final}} \leftarrow \text{ROWMERGE}(\mathbf{M}_A, \mathbf{M}_B, L)$ 
5  $\mathbf{R}' \leftarrow \text{SORTROWS}(\mathbf{M}_{\text{final}})$ 

```

Consequently, the RowMerge runs in a single linear-time pass without any additional sort step.

The LHRR mapping involves four structurally distinct steps, summarized in Algorithm 4. The first is the construction of the symmetric position matrix $\mathbf{P}_t$ (line 1). This step is implemented by emitting two coordinate triplets (i, j, score) and (j, i, score) for each pair (i, j) with $j \in \mathbf{R}_{i,1:K}$, where the score is the rank-based weight from Equation 5. This produces exactly $2NK$ triplets and is implemented as a parallel scan over $\mathbf{R}$ with no inter-thread dependencies; the resulting $\mathbf{P}$ has $\mathcal{O}(NK)$ non-zeros.

The second step constructs the hyperedge matrix $\mathbf{H}$ (line 2), which connects each image i to all images reachable through two-hop paths in $\mathbf{R}$: for every neighbor $a \in \mathbf{R}_{i,1:K}$ and every neighbor $b \in \mathbf{R}_{a,1:K}$, a triplet (i, b, w) is emitted, where $w = w(\text{pos}(a, \mathbf{R}_i)) \cdot w(\text{pos}(b, \mathbf{R}_a))$ from Equation 6. This two-hop expansion emits up to NK^2 triplets, but the assembled $\mathbf{H}$ is immediately reduced to the top- L entries per row during CSR construction, so the stored $\mathbf{H}$ never exceeds $\mathcal{O}(NL)$ non-zeros. This is the memory-critical step in the entire algorithm and the one that requires the fused kernel of Section 4.3 rather than a compute-then-prune library routine.

The third step updates the position matrix via two fused SpGEMM operations and a Hadamard product: $\mathbf{S} = \text{FusedSpGEMM}(\mathbf{H}, \mathbf{H}^\top, L)$, $\mathbf{S}_{\text{rev}} = \text{FusedSpGEMM}(\mathbf{H}^\top, \mathbf{H}, L)$, and $\mathbf{P}_{t+1} = \mathbf{P}_t \odot (\mathbf{S} \odot \mathbf{S}_{\text{rev}})$. Because all operands are bounded to $\mathcal{O}(NL)$ non-zeros, the Hadamard update never requires scratch space beyond a single output row at a time.

Finally, the fourth step executes a SORTROWS operation for ranked list reconstruction (Phase 4). Every intermediate structure is bounded to $\mathcal{O}(NL)$ non-zeros. No dense $N \times N$ allocation is required at any point. The reformulation is exact: the final ranked lists produced by the sparse engine are identical to those produced by the dense baseline for all datasets where both can execute, confirming that the mathematical integrity of the manifold learning process is fully preserved. Figure 3 illustrates the complete data flow and intermediate memory structures of the LHRR sparse reformulation. Steps 1 and 2 are logically sequential in the algorithm, but each is internally parallel (row decomposition), as detailed in Section 4.3.3.

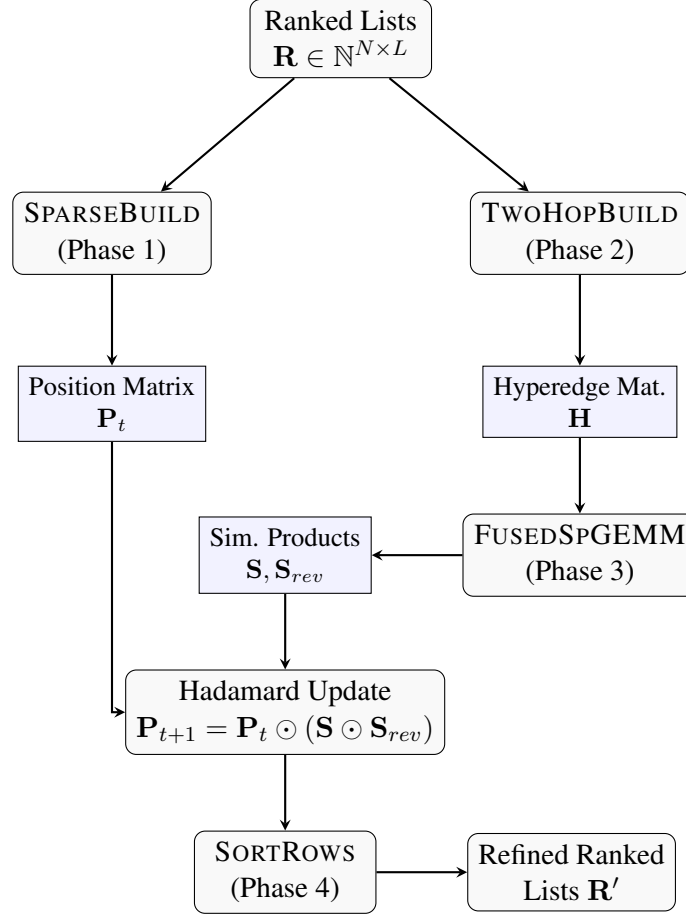


Figure 3 – Architecture and information flow of the LHRR sparse reformulation. The input ranked lists are converted into $\mathcal{O}(NK)$ and $\mathcal{O}(NL)$ sparse structures, followed by fused multiplication and element-wise updates that preserve the memory bound throughout.

4.2.1 Proof Sketch: Exactness of the Sparse Mapping

To see why the sparse reformulation is exact, let $\mathbf{W}$ be the sparse weight matrix and $\mathbf{M}$ be the dense similarity matrix. In CPRR, the dense update $\mathbf{M} \leftarrow \mathbf{W}^\top \mathbf{W}$, when evaluated naively over all N^2 output entries, requires $\mathcal{O}(N^2 K)$ operations (for each of the N^2 output entries, up to K inner product terms must be summed). However, row i of the refined ranked list is derived solely from the L largest entries in row i of $\mathbf{M}$. Our fused SpGEMM kernel computes exactly these L largest entries by only visiting the non-zero paths in the sparse weights and maintaining a bounded accumulator. Since the underlying floating-point operations (multiply-accumulate) are mathematically identical to the dense baseline, and no non-zero contribution is missed, the resulting top- L indices and their relative ordering are bitwise identical to the dense result.

Algoritmo 4: LHRR sparse execution (one iteration).

Input: Ranked list matrix $\mathbf{R}$ ($N \times L$), parameters K, L , current position matrix $\mathbf{P}_t$

Output: Updated position matrix $\mathbf{P}_{t+1}$, refined ranked lists $\mathbf{R}'$

```

// Phase 1: build symmetric position matrix
1  $\mathbf{P}_t \leftarrow \text{SPARSEBUILD}(\mathbf{R}, K, \text{symmetric})$  //  $\mathcal{O}(NK)$  non-zeros
// Phase 2: build hyperedge matrix (two-hop expansion)
2  $\mathbf{H} \leftarrow \text{TWOHOPBUILD}(\mathbf{R}, K, L)$  // emits  $\leq NK^2$  triplets, prunes to  $\mathcal{O}(NL)$ 
// Phase 3: similarity update via SpGEMM + Hadamard
3  $\mathbf{S} \leftarrow \text{FUSEDSPGEMM}(\mathbf{H}, \mathbf{H}^\top, L)$   $\mathbf{S}_{\text{rev}} \leftarrow \text{FUSEDSPGEMM}(\mathbf{H}^\top, \mathbf{H}, L)$ 
 $\mathbf{P}_{t+1} \leftarrow \mathbf{P}_t \odot (\mathbf{S} \odot \mathbf{S}_{\text{rev}})$ 
// Phase 4: reconstruct ranked lists
4  $\mathbf{R}' \leftarrow \text{SORTROWS}(\mathbf{P}_{t+1})$ 

```

4.3 HIGH-PERFORMANCE SPARSE ENGINE

This custom engine is necessary because, as established in Section 4.1, general-purpose sparse libraries optimize for sustained throughput on fixed sparsity patterns, whereas re-ranking requires strict memory bounds during the emission of transient intermediate non-zeros. The formal mapping of Section 4.2 establishes that bounded-memory re-ranking is achievable in principle. This section describes the engine built to realize it in practice, addressing the core hardware-level bottlenecks identified in Section 4.1. The library, implemented in C++17, centers on the `CSRMatrix<T>` template class and a specialized memory management subsystem.

4.3.1 Library Architecture and Design Patterns

The engine adopts a *phase-separated architecture* that distinguishes between a **Construction Phase** and a **Query Phase**. In the Construction Phase, the matrix acts as a sink for coordinate triplets (r, c, v) emitted by re-ranking kernels. To eliminate synchronization overhead, each thread maintains a private `BlockAllocator` to store its emitted triplets. Once emission is complete, the engine transitions to the Query Phase through a parallelized compression process (`BUILDCSR`) that organizes the data into a packed Compressed Sparse Row (CSR) representation.

This transition follows the *Emit-Sort-Reduce* pattern, which aligns with the sequential nature of re-ranking updates. This pipeline is illustrated in Figure 4. The specific CSR representation used consists of three contiguous arrays: `values_` for the floating-point coefficients, `col_idx_` for the column indices, and `row_ptr_` for the starting index of each row. This layout provides $\mathcal{O}(1)$ access to any row and sequential traversal with maximum spatial locality, which is critical for the subsequent SpGEMM and Hadamard operations.

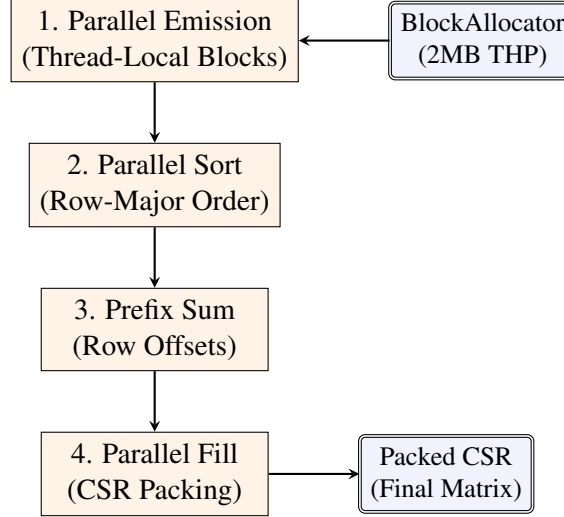


Figure 4 – CSRMatrix construction pipeline. The architecture enforces a strict transition from thread-local block-allocated triplets to a global packed CSR representation, optimized for parallel execution and cache locality.

4.3.2 Memory Subsystem: BlockAllocator and THP Alignment

A major performance bottleneck identified in preliminary profiling was allocator pressure: standard `malloc` and `std::vector` operations accounted for a significant fraction of total execution cycles during hyperedge construction. To address this, we developed the `BlockAllocator`, a custom linear allocator optimized for thread-local usage. By replacing these system calls with the `BlockAllocator`’s pointer-bump strategy (Algorithm 5), each triplet emission reduces to a single index increment and a bounds check, eliminating the per-allocation overhead of the system allocator.

The `BlockAllocator` pre-allocates fixed-size arenas of 2 MB, which are aligned to Transparent Huge Page (THP) boundaries (Linux Kernel Organization, 2011). By requesting huge pages via `madvise(MADV_HUGEPAGE)`, the engine increases Translation Lookaside Buffer (TLB) coverage by $512\times$, significantly reducing TLB misses during high-speed traversal. Within each 2 MB block, allocation is a simple pointer-bump operation (Algorithm 5):

Algorithm 5: `BlockAllocator` pointer-bump allocation.

```

1 if current_index ≥ ITEMS_PER_BLOCK then ALLOCATE_NEW_BLOCK()
2 ptr ← &current_block[current_index++]
3 return ptr
  
```

To prevent *false sharing* between thread-local control structures, the `BlockAllocator` class itself is aligned to 64 bytes (one x86 cache line). The `CSRMatrix` implementation further wraps each thread’s allocator in a `PaddedAllocator` structure aligned to 128 bytes (two cache lines), providing conservative false-sharing prevention even on processors with 128-byte hardware prefetch strides (Drepper, 2007).

4.3.3 Parallelized CSR Construction

The transition from triplets to CSR is a multi-step parallel pipeline implemented in `buildCSRParallel`. To ensure scalability, the process minimizes global synchronization through a parallel histogram approach:

1. **Parallel Histogram:** Each thread computes a local histogram of row counts for its partition of the triplet buffer;
2. **Parallel Reduction:** Row counts are reduced across threads to produce the final `row_ptr_` array via a prefix sum;
3. **Parallel Fill:** Threads iterate over their triplet ranges and copy data into the final `values_` and `col_idx_` buffers, resolving duplicates through accumulation or replacement.

Performance is further enhanced through memory prefetching. The implementation uses `__builtin_prefetch` hints (e.g., during triplet copying and CSR filling) to inform the hardware memory controller of upcoming access patterns, overlapping memory latency with computation.

4.3.4 Fused SpGEMM Kernel

The primary computational component is the fused SpGEMM kernel (Algorithm 6), which integrates top- L selection directly into the row accumulation loop. Unlike general-purpose libraries that perform a full $\mathbf{AB}$ followed by a separate pruning step, our kernel never materializes more than $\mathcal{O}(L)$ output entries per row.

For each output row i , the kernel accumulates partial products into a thread-local dense accumulator. To avoid $\mathcal{O}(N)$ overhead, we maintain an `active_cols` list to track only the modified indices. When the number of active columns exceeds L , we use `std::nth_element` to perform a partial sort, retaining only the largest L entries. This ensures that the intermediate memory usage is strictly bounded by the truncation parameter L , rather than the quadratic K^2 explosion typical of re-ranking.

4.3.5 Implementation Details: Accumulator Design and Vectorization

The engine leverages C++17 parallel algorithms (`std::execution::par_unseq`) for the triplet sorting phase, allowing the compiler to use both multi-core and SIMD (vectorization) units for the $\mathcal{O}(NL \log(NL))$ bottleneck. The dense accumulator used in SpGEMM (approx. 640 KB per thread for $N = 160,000$) is sized to reside in the per-core L3 cache partition under single-threaded execution. Under full multi-threaded operation, the aggregate footprint across all threads exceeds the shared L3; in this regime, the benefit of the dense layout is not cache residency but $\mathcal{O}(1)$ random write access without hash-table overhead. This trade-off is

Algoritmo 6: Fused SpGEMM with top- L pruning.

Input: Sparse matrix **A** (CSR), sparse matrix **B** (CSR), pruning limit L
Output: Sparse matrix **C** = TOP- L (**AB**) (CSR)

```

1 foreach row  $i \in [0, N)$  do
2   Initialize dense_acc[0...N-1]  $\leftarrow 0$ 
3   Initialize active_cols  $\leftarrow \emptyset$ 
4   // Multiply: accumulate all partial products for row  $i$ 
5   foreach  $(k, v_A) \in \mathbf{A}[i]$  do
6     foreach  $(j, v_B) \in \mathbf{B}[k]$  do
7       dense_acc[j]  $\mathrel{+}= v_A \cdot v_B$ 
8       if first encounter of  $j$  then active_cols.append( $j$ )
9   // Prune: retain only the top- $L$  highest values
10  if |active_cols|  $> L$  then
11    std::nth_element on active_cols keyed by dense_acc
12    Truncate active_cols to top- $L$  entries
13  // Emit: store results in CSR output
14  Sort retained active_cols in ascending column order
15  foreach  $j \in \text{active\_cols}$  do
16    Emit ( $i, j, \text{dense\_acc}[j]$ )
17    dense_acc[j]  $\leftarrow 0$ 

```

particularly effective on modern consumer CPUs with large shared L3 caches, enabling efficient execution on resource-constrained commodity hardware.

4.3.6 Statement on the Use of Artificial Intelligence

In accordance with contemporary academic integrity standards and emerging compliance regulations for scientific publishing, the author declares that Generative Artificial Intelligence (AI) and Large Language Models (LLMs) were utilized as assistive tools during the preparation of this dissertation. Specifically, advanced language models functioned as collaborative agents to support code refactoring, python visualization script optimization, grammatical auditing, and stylistic polish of the English manuscript.

It is strictly emphasized that the deployment of these automated technologies was confined to technical synthesis, code optimization, and linguistic refinement. The entire intellectual conception, algorithmic architecture design, core mathematical derivations, experimental execution, and critical interpretation of results remain exclusively the product of the author's independent intellectual labor. All automated outputs were rigorously audited, verified, and integrated under full human oversight to guarantee scientific accuracy and adherence to academic ethics.

4.4 SUMMARY

This chapter has presented the three components that together constitute the contribution of this dissertation: (1) the identification of the structural memory and density bottlenecks in current re-ranking implementations (Section 4.1); (2) the formal sparse reformulation of CPRR and LHRR into algorithm-agnostic primitives (Section 4.2); and (3) the design of a high-performance sparse engine that enforces bounded memory through fused truncation (Section 4.3).

The analysis of Section 4.1 established that both dense implementations and general-purpose sparse libraries fail on large-scale re-ranking: the former due to $\mathcal{O}(N^2)$ pre-allocation, the latter due to $\mathcal{O}(NK^2)$ intermediate density explosion during SpGEMM. The primitive mapping of Section 4.2 showed that CPRR and LHRR reduce exactly to SpGEMM, Hadamard, and sorted row merge operations over $\mathcal{O}(NL)$ -bounded sparse structures, with no approximation of the underlying manifold update rules. The engine of Section 4.3 realized this mapping through three complementary innovations: row-split construction with thread-local emission to eliminate synchronization; the BlockAllocator with THP alignment and 128-byte cache-line padding to eliminate allocator pressure and false sharing; and the fused multiply-and-prune SpGEMM kernel to prevent intermediate density explosion. The experimental setup and empirical results are presented in Chapter 5.

5 RESULTS

This chapter presents the experimental evaluation of the sparse reformulation developed in Chapter 4. The discussion is based on four main research questions, as follows:

- RQ1 (Efficiency).** Does the sparse *engine* reduce *wall-clock* execution time relative to the dense *baseline* on high-memory *hardware* where both can execute?
- RQ2 (Scalability).** Does the sparse *engine* maintain linear $\mathcal{O}(NL)$ memory growth with *dataset* size, and does this enable execution on *hardware* tiers where all dense methods fail?
- RQ3 (Correctness).** Does the sparse reformulation produce ranked lists that are numerically identical to those of the dense *baseline* for all configurations where comparison is possible?
- RQ4 (Quality-memory trade-off).** How does the sparsity parameter L affect retrieval quality across *datasets* with different manifold geometries, and does commodity hardware’s memory ceiling coincide with the accuracy optimum?

Section 5.1 describes the experimental setup, including hardware environments, benchmark datasets, baseline configurations, and the execution protocol. The subsequent sections address each research question in turn: Section 5.2 and Section 5.3 address RQ1 (efficiency); Section 5.4 addresses RQ2 (scalability and feasibility); Section 5.5 addresses RQ3 (correctness and quality); and Section 5.6 addresses RQ4 (quality–memory trade-off).

5.1 EXPERIMENTAL SETUP

The evaluation uses three distinct tiers, ranging from a research server (Environment A) to a consumer desktop (Environment C), summarized in Table 2. This selection allows us to characterize performance across a spectrum of memory capacities and processor architectures, from high-core-count enterprise systems to frequency-optimized consumer hardware.

Environment C executes under Windows Subsystem for Linux 2 (WSL2), which introduces a lightweight Hyper-V virtualization layer. Environments A and B run on native Ubuntu 22.04. Since the primary claims of this work concern memory feasibility rather than absolute execution speed, and all timing comparisons within Environment C are performed under the same virtualization overhead, the relative performance conclusions remain valid.

The ranked list depth L was adjusted to match hardware constraints across these environments: $L = 2,000$ in Environment A, $L = 1,000$ in Environment B. For SUN397 and VegFru, the $L = 400$ configuration was chosen to fit within the 8 GB physical memory envelope of Environment C. Profiling via `/usr/bin/time -v` confirms that peak RSS for VegFru ($L = 400$) remains at approximately 256.4 MB, a value consistent with the theoretical $\mathcal{O}(NL)$

Table 2 – Evaluation environments. Environment B uses the same physical hardware as Environment A, with cores pinned to a single NUMA node via `numactl -physcpubind=0-11 -membind=0` to ensure fair comparison under the 16 GB memory limit imposed by Linux `cgroups` (`MemoryMax=16G`, `MemorySwapMax=4G`).

Env.	Description	CPU	Cores	RAM limit
A	High-memory server	Xeon Gold 5220 @ 2.20 GHz	18 (1 socket)	192 GB
B	Constrained server	Xeon Gold 5220 @ 2.20 GHz	12 (pinned)	16 GB
C	Consumer desktop	i5-10400F @ 2.90 GHz	12 (HT)	8 GB

footprint and the $400\times$ reduction claim. The choice of these parameters is motivated by the trade-off between retrieval quality and the memory-stress objectives of this work, as detailed in Table 3 and Section 5.1.1.

5.1.1 Experimental Parameters and Memory Pressure Trade-offs

The selection of ranked list depth (L), neighborhood size (K), and the number of iterations (T) follows a methodology designed to balance retrieval effectiveness with the memory-stress objectives of this work. The neighborhood size K in the literature varies depending on dataset characteristics (Valem; Pedronette, 2017; Pedronette et al., 2019); there is no single optimal value, and the primary objective of this work is to stress memory subsystems rather than to maximize retrieval accuracy. Our experimental setup uses targeted configurations to explore the scalability limits across different hardware tiers.

Table 3 – Comparison of experimental parameters (L, K, T) across hardware environments versus typical literature values.

Environment	RAM	L	K	T	Primary Rationale
A (HPC Server)	192 GB	2,000	50	5	Peak accuracy and full-scale baseline.
B (Constrained)	16 GB	1,000	20	5	High-pressure feasibility testing.
C (Commodity)	8 GB	400/1,000	20	5	Entry-level Hardware limit.
Literature Typ.	—	100–1,000	20–50	1–10	Effectiveness over small N .

The parameters $K = 20$ and $L = 1,000$ (Environment B and C) were explicitly chosen to induce a memory-intensive regime that demonstrates the failure of dense methods while remaining within the tight 8–16 GB physical RAM envelope of commodity hardware. This configuration prioritizes memory stress: in Environment C, for example, the reduction of L to 400 for LHRR on SUN397 and VegFru is a direct consequence of the $\mathcal{O}(NL)$ spatial constraint imposed by the 8 GB limit.

Table 4 – Benchmark dataset properties and memory requirements. Dense mem. req. is computed as $4N^2$ bytes (single-precision float).

Dataset	N	Classes	Dense req.	Manifold type
ALOI	72,000	1,000	≈ 20 GB	Smooth, continuous
SUN397	108,754	397	≈ 46 GB	Noisy, high-variance
VegFru	160,731	200	≈ 102 GB	Fine-grained, noisy

Critically, this constraint does not imply a compromise in retrieval quality. As analyzed in Section 5.6.4, while smooth manifolds (ALOI) benefit from larger L , noisy datasets (SUN397, VegFru) exhibit an accuracy peak at $L \approx 400$, with further expansion introducing neighborhood noise that degrades the manifold propagation. Thus, the hardware-constrained parameters align with the accuracy optimum for complex datasets.

We evaluate the framework using three public benchmark datasets (Table 4). These datasets cover a graduated range of memory requirements, enabling the evaluation of the sparse engine under varying levels of memory pressure. In particular, ALOI serves as the primary correctness testbed since both sparse and dense methods can execute within a 192 GB server. SUN397 crosses the feasibility threshold for machines with less than 64 GB of RAM. VegFru represents the primary feasibility test for the sparse reformulation.

5.1.2 Baseline Configurations

Four different systems are evaluated in the experiments; software library versions and the Linux/GCC environment are detailed in Table 5. The first system, **Dense-MMAP**, is an optimized version of the standard UDLF dense implementation and serves as the primary baseline. The original serial UDLF code was ported to support multi-core execution via OpenMP with static scheduling at chunk size one, matching the per-row decomposition of the sparse engine and ensuring a fair thread-to-work assignment comparison. Numerical correctness of this parallel dense baseline was verified against the original serial implementation (Valem; Pedronette, 2017) on ALOI, confirming identical MAP scores. Dense-MMAP pre-allocates the full similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$ as an anonymous memory-mapped region (MAP_ANONYMOUS | MAP_PRIVATE) backed exclusively by physical DRAM; execution terminates with an Out-of-Memory (OOM) signal if the allocation exceeds available RAM.

The second system, **Dense-Swap**, uses the same allocation strategy as Dense-MMAP but additionally enables swap space of up to 4 GB via Linux cgroups (MemorySwapMax), permitting the operating system to page matrix rows to disk at the cost of severe I/O degradation. This configuration exists to quantify the performance collapse that occurs when the similarity matrix spills to secondary storage.

The third system, **General-purpose sparse libraries (compute-then-prune)**, provides a fair comparison against established tools. Sparse versions of CPRR and LHRR were

Table 5 – Software library versions and execution environment. All experiments were conducted on Ubuntu Linux (Kernel 5.15+), using GCC 11.4.

Library	Version
Intel TBB	2021.5.0
OpenBLAS	0.3.20
Intel MKL	2022.0.1
GraphBLAS	SuiteSparse:GraphBLAS v7.0.1
Eigen	3.4.0

implemented using the standard SpGEMM routines from three leading high-performance libraries: Intel MKL, SuiteSparse:GraphBLAS, and Eigen. None of these libraries offer a fused kernel that integrates top- L pruning directly into the multiplication step. Consequently, each baseline implementation follows a two-step *compute-then-prune* model: the unpruned intermediate correlation matrix is first computed and stored in full as a temporary sparse structure, after which a separate pass retains only the top- L entries per row. This approach provides a direct baseline for quantifying the performance gains of the proposed fused kernel (Section 4.3.4), which avoids materializing the full intermediate matrix.

The fourth system, **Sparse-CSR (Proposed)**, is the custom high-performance engine developed in this dissertation. It implements the fused multiply-and-prune SpGEMM kernel together with the memory management optimizations described in Chapter 4.

5.1.3 Execution Protocol

Both sparse and dense implementations use Transparent Huge Pages (THP) where available: the sparse engine explicitly aligns its `BlockAllocator` arenas to 2 MB boundaries via `madvise(MADV_HUGEPAGE)`, which does not require kernel parameter changes. The dense *baseline* relies on automatic THP promotion for large contiguous allocations, which may not trigger uniformly; page-fault counts between the two implementations are therefore not directly comparable. All reported results use THP-backed allocations in the sparse engine. The performance impact of disabling THP is left as a direction for future characterization.

Finally, all code was compiled with GCC 11 (`-O3 -march=native -fopenmp`). Performance metrics were collected via `perf stat`; all results are the mean of ten independent runs with standard deviation below 2% (Duarte; Ribeiro; Mixon, 2003). The iterative nature of manifold-based re-ranking leads to a distinction between memory footprint and execution time: while memory consumption is defined by the list length L and neighborhood size K , the execution time scales linearly with the number of iterations T . Throughout this chapter, we set $T = 5$ for all performance, feasibility, and accuracy experiments to reflect a realistic deployment setting. For the parallel scaling study (Section 5.3), we set $T = 1$ to isolate the per-iteration throughput of the sparse engine without the overhead of repeated traversals.

Table 6 – CPRR execution performance: *Sparse-CSR engine* versus *Dense-MMAP baseline*. Measured in Environment A (18 threads, $L = 2,000$, $K = 50$, $T = 5$).

Dataset	Method	Time (s)	Instr. ($\times 10^{12}$)
ALOI (72K)	Dense-MMAP	15.2	0.375
	Sparse-CSR	27.8	0.210
SUN397 (108K)	Dense-MMAP	24.8	0.571
	Sparse-CSR	102.6	0.618
VegFru (160K)	Dense-MMAP	34.6	0.844
	Sparse-CSR	94.9	0.630

5.2 ALGORITHMIC EFFICIENCY AND MICROARCHITECTURAL ANALYSIS

We first establish the performance *baseline* in Environment A, where 192 GB of RAM allows both sparse and dense implementations to execute on all three datasets without memory pressure. This environment isolates the throughput characteristics of the sparse engine from the categorical feasibility effects examined later in Section 5.4.

5.2.1 Execution time and instruction reduction

Tables 6 and 7 report wall-clock execution times and aggregated retired instruction counts for CPRR and LHRR respectively, across all three *datasets* in Environment A (18 threads, single NUMA socket, $L = 2,000$, $K = 50$, $T = 5$). Both primary implementations are custom developments: the proposed **Sparse-CSR engine** and the optimized **Dense-MMAP baseline**, the latter implementing the parallel multi-core optimizations described in Section 5.1.2 to ensure structural comparability. General-purpose sparse libraries (MKL, GraphBLAS, and Eigen) are excluded from this *throughput* analysis since they follow the *compute-then-prune* model and either fail due to memory constraints on larger *datasets* or exhibit degraded performance on multi-hop operations; their behavior is examined in the feasibility analysis of Section 5.4. Instruction counts are aggregated across all threads (`perf stat -a`); per-thread scaling behavior is examined independently in Section 5.3.

For CPRR (Table 6), the *Dense-MMAP baseline* completes faster in absolute wall-clock time across all three *datasets* on the unconstrained Environment A server: 15.2 s versus 27.8 s on ALOI, 24.8 s versus 102.6 s on SUN397, and 34.6 s versus 94.9 s on VegFru. Despite retiring fewer total instructions — 0.210 versus 0.375×10^{12} on ALOI, and 0.630 versus 0.844×10^{12} on VegFru — the *Sparse-CSR engine* does not convert this instruction reduction into a wall-clock advantage on unconstrained *hardware* for CPRR’s shallow *one-hop* products. The dominant *overhead* is the fixed per-row accumulator initialization cost: before processing each output row, the *engine* must zero all N slots of the thread-local dense accumulator array regardless of the actual number of non-zero products to accumulate, which is bounded by $\mathcal{O}(K^2) = \mathcal{O}(400)$ for $K = 20$. This $\mathcal{O}(N)$ zeroing cost per row constitutes the structural bottleneck for shallow workloads at this scale and is explicitly identified as the target of the

Table 7 – LHRR execution performance: *Sparse-CSR engine* versus *Dense-MMAP baseline*. Measured in Environment A (18 threads, $L = 2,000$, $K = 50$, $T = 5$).

Dataset	Method	Time (s)	Instr. ($\times 10^{12}$)
ALOI (72K)	Dense-MMAP	211.4	8.059
	Sparse-CSR	227.5	1.671
SUN397 (108K)	Dense-MMAP	1,327.4	23.694
	Sparse-CSR	579.0	8.560
VegFru (160K)	Dense-MMAP	1,651.4	29.273
	Sparse-CSR	695.2	6.492

hash-based accumulator direction discussed in Section 6.3. Consequently, the dense *baseline* is preferable for CPRR when sufficient physical memory is available; the sparse advantage materializes exclusively under memory constraints (Section 5.4), where the dense *baseline* fails with OOM.

The performance relationship inverts for LHRR (Table 7). As *dataset* size increases, the $\mathcal{O}(N^2)$ working set of the dense representation forces repeated traversal of all structurally irrelevant entries — the $\mathcal{O}(N^2 - NL)$ positions that the final top- L selection discards at each iteration regardless of their values — generating a volume of arithmetic disproportionate to the information actually retained in the output. The *Dense-MMAP baseline* retires 29.27×10^{12} instructions on VegFru; the *Sparse-CSR engine*, operating exclusively over the $\mathcal{O}(NL)$ working set, retires 6.49×10^{12} instructions for the same workload — a $4.5\times$ reduction in total retired instructions. This reduction in total computational work translates into wall-clock speedups of up to $2.4\times$ on the largest *dataset*, confirming that elimination of structurally irrelevant arithmetic is the primary performance driver for many-hop algorithms at scale.

5.3 PARALLEL SCALING BEHAVIOR

The wall-clock times reported in Section 5.2 were measured with 18 threads. A natural question for RQ1 is whether the sparse engine’s advantage persists under different parallelism levels, and conversely, whether the dense baseline can recover its wall-clock deficit simply by using more cores. From a practical standpoint, if the sparse approach achieves near-peak performance with fewer cores than the dense baseline requires, it is viable on hardware that is cheaper not only in memory but also in processor count.

To isolate the effect of thread count independently of memory footprint and convergence cost, the scaling experiments in this section use $L = 1,000$, $K = 20$, and $T = 1$ (a single re-ranking iteration). This differs from the performance-optimized configuration ($L = 2,000$, $K = 50$, $T = 5$) used in Sections 5.2 and 5.4; wall-clock times reported here reflect per-iteration cost and are not directly comparable in absolute value to those sections.

This section characterizes how sparse and dense implementations respond to additional threads using a thread-count sweep ($t \in \{1, 2, 4, 6, 8, 10, 12, 14, 16, 18\}$) on Environ-

Table 8 – *Wall-clock* time (seconds) and parallel scaling for **CPRR** on Environment A ($L = 1,000$, $K = 20$, single NUMA socket). $S_{18} = T_1/T_{18}$; P_{eff} is the mean number of hardware threads actively executing useful work at $t = 18$.

Dataset	Approach	T_1 (s)	T_4 (s)	T_8 (s)	T_{18} (s)	S_{18}	P_{eff}
ALOI (72K)	Dense-MMAP	20.8	11.3	10.2	9.8	$2.13\times$	1.8
	Dense-Swap	22.2	12.6	10.8	9.9	$2.23\times$	1.8
	Sparse-CSR	9.3	8.9	8.9	8.9	$1.04\times$	1.4
SUN397 (108K)	Dense-MMAP	33.3	15.7	13.2	12.0	$2.76\times$	2.9
	Dense-Swap	37.8	17.8	14.4	12.6	$3.00\times$	3.0
	Sparse-CSR	22.2	18.3	17.8	17.8	$1.25\times$	1.7
VegFru (160K)	Dense-MMAP	42.8	21.9	18.7	17.3	$2.48\times$	2.6
	Dense-Swap	48.1	24.3	20.1	18.1	$2.66\times$	2.7
	Sparse-CSR	25.3	21.9	21.3	21.3	$1.19\times$	1.5

Table 9 – *Wall-clock* time (seconds) and parallel scaling for **LHRR** on Environment A ($L = 1,000$, $K = 20$, single NUMA socket). Notation as in Table 8.

Dataset	Approach	T_1 (s)	T_4 (s)	T_8 (s)	T_{18} (s)	S_{18}	P_{eff}
ALOI (72K)	Dense-MMAP	70.7	26.1	20.3	18.0	$3.93\times$	8.9
	Dense-Swap	73.6	28.1	21.5	19.1	$3.85\times$	9.2
	Sparse-CSR	47.3	37.7	36.5	36.1	$1.31\times$	1.5
SUN397 (108K)	Dense-MMAP	335.9	107.2	72.1	58.5	$5.74\times$	14.0
	Dense-Swap	345.3	113.5	73.9	59.2	$5.83\times$	13.9
	Sparse-CSR	143.0	85.3	76.2	71.7	$2.00\times$	2.5
VegFru (160K)	Dense-MMAP	361.9	120.6	82.9	74.9	$4.83\times$	13.2
	Dense-Swap	378.4	126.9	85.9	75.3	$5.03\times$	13.1
	Sparse-CSR	159.9	111.3	103.6	100.0	$1.60\times$	1.9

ment A (single NUMA socket, 18 physical cores, hyperthreading disabled via BIOS to isolate physical core scaling, $L = 1,000$, $K = 20$, $T = 1$ iteration). In Environment B, thread count is fixed at 12 to match the NUMA-pinned allocation (`numactl -physcpubind=0-11`) and hyperthreading is likewise disabled, while Environment C utilizes all 12 logical threads (6 physical cores with Hyper-Threading). Tables 8 and 9 report, for both CPRR and LHRR, wall-clock time at four representative thread counts, the 18-thread speedup $S_{18} = T_1/T_{18}$, and the effective parallelism P_{eff} (CPU user time divided by elapsed wall-clock time at $t = 18$, representing the mean number of hardware threads doing useful work). Figure 5 and Figure 6 show the full speedup curves (T_1/T_t) for both the proposed Sparse-CSR engine and the primary **Dense-MMAP** baseline. While these plots visualize relative parallel efficiency, the absolute wall-clock advantage shifts between implementations as core count increases. Specifically, we identify a *throughput crossover point* where the superior parallel efficiency of the dense baseline compensates for its higher total work volume, eventually leading to lower absolute execution time than the sparse engine at higher thread counts. This point is analyzed in the following paragraphs for each algorithm.

The central finding is that the two approaches, dense and sparse, occupy opposite

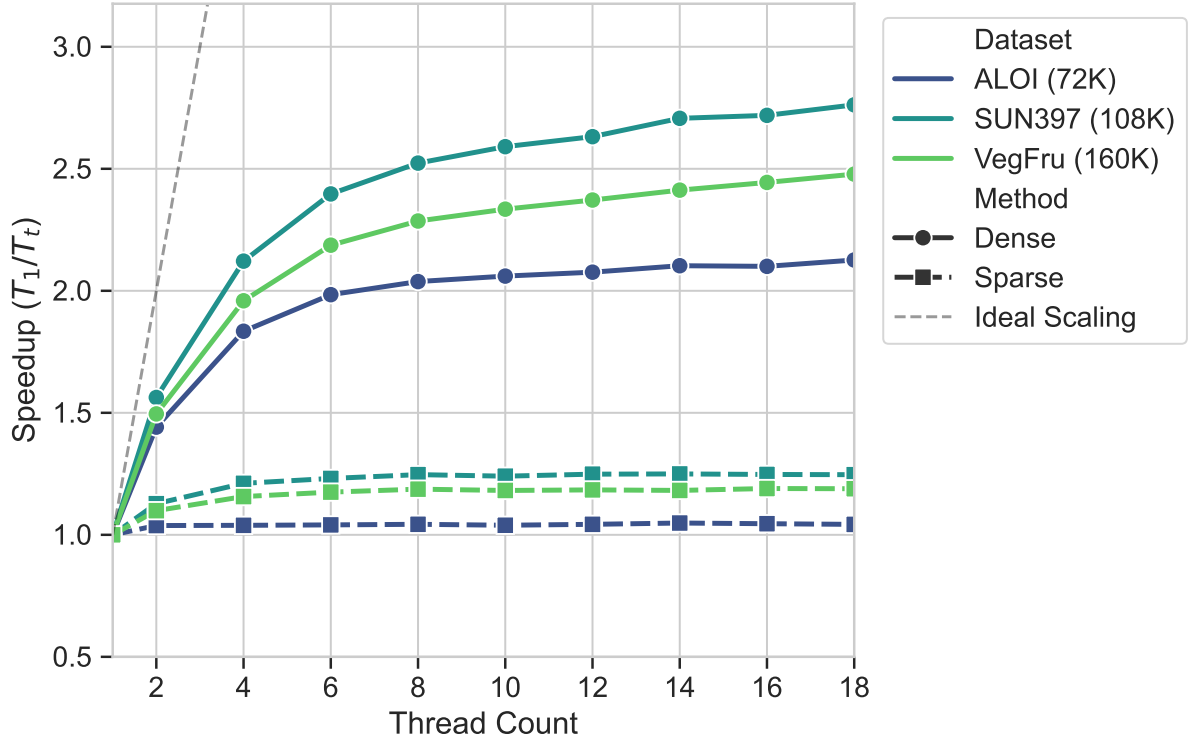


Figure 5 – CPRR parallel scaling: speedup T_1/T_t as a function of thread count on Environment A ($L = 1,000$, $K = 20$, $T = 1$ iteration). Speedup remains below $1.3\times$ due to $\mathcal{O}(N)$ overheads, while the dense baseline achieves better relative efficiency and achieves lower absolute time above $t \approx 4$.

positions in the parallelism–work-volume trade-off: the sparse engine does substantially less total work per thread but extracts little benefit from additional threads, while the dense baseline does more total work but parallelizes efficiently enough to recover and eventually surpass the sparse engine in wall-clock time on the unconstrained server.

5.3.1 CPRR Scaling Analysis

At $t = 1$, Sparse-CSR completes faster on all three datasets ($1.69\times$ on ALOI, $1.50\times$ on SUN397, $1.69\times$ on VegFru; all at $L = 1,000$, $K = 20$) because it performs proportionally less arithmetic by skipping the $\mathcal{O}(N^2 - NK)$ structurally zero entries. For SUN397 and VegFru, the dense implementation overtakes the sparse engine at $t \approx 4$. For ALOI, however, Sparse-CSR leads by a small margin across the entire thread sweep (9.3 s vs 9.8 s at 18 threads), as the $\mathcal{O}(N)$ row-reset overhead at this scale is sufficiently offset by the work reduction for the shallow one-hop product. Beyond $t = 1$, the sparse engine gains almost nothing: its speedup reaches only $1.04\times$ on ALOI and $1.25\times$ on SUN397 at 18 threads, with $P_{\text{eff}} \leq 1.7$ active cores across all datasets. The dense baselines scale more usefully ($2.13\text{--}3.00\times$ at 18 threads, P_{eff} of $1.8\text{--}3.0$), and Dense-MMAP overtakes Sparse-CSR in wall-clock time at $t \approx 4$ on SUN397 and VegFru.

The reason the sparse engine does not scale beyond 2–3 effective cores is structural. The fused SpGEMM kernel (Section 4.3.4) maintains one thread-local dense accumulator of

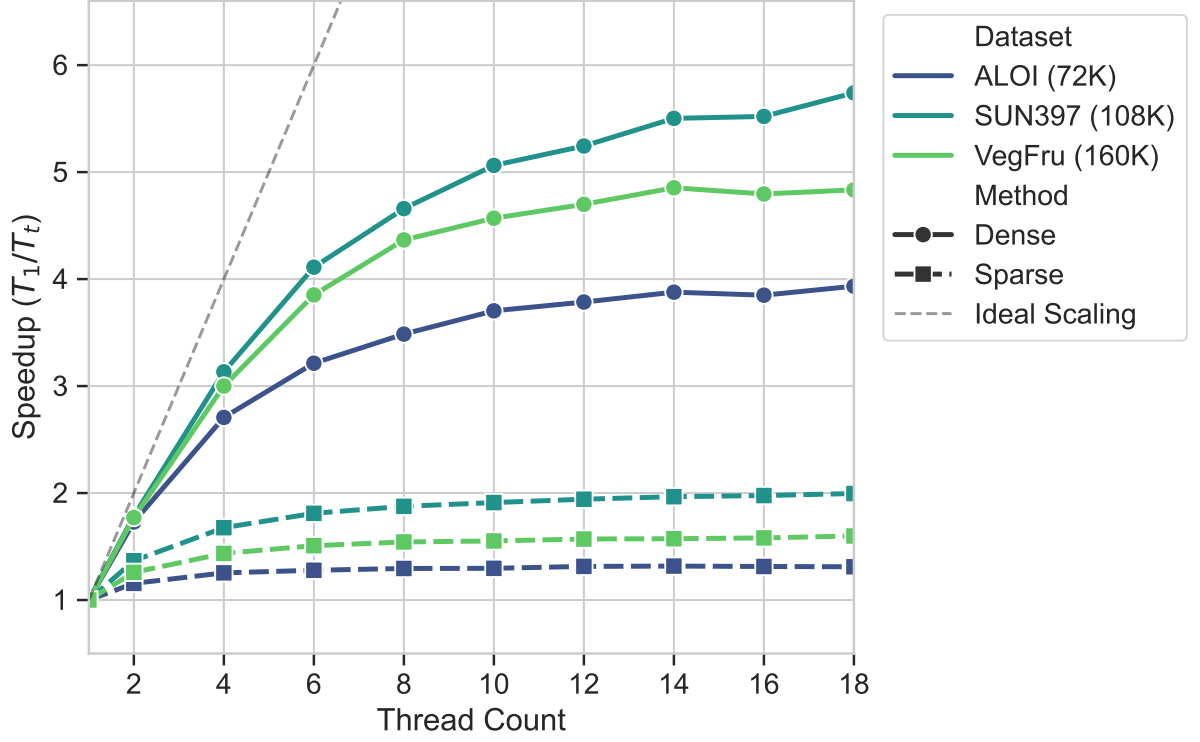


Figure 6 – LHRR parallel scaling on Environment A ($L = 1,000$, $K = 20$, $T = 1$ iteration), same configuration as Figure 5. Speedup reaches $2.42\times$ on larger datasets, but dense methods eventually recover their wall-clock advantage given sufficient cores.

length N per output row. Before processing each row, all N slots must be zeroed; after processing, the top- L slots are emitted and the rest zeroed again. With $N = 160,000$ and $K = 20$, only $K^2 = 400$ products actually accumulate, but zeroing N slots dominates. Because each thread has its own accumulator of length N , distributing rows across more threads does not reduce the zeroing cost per row: it merely splits the same total zeroing work into more pieces, each still paying the full $\mathcal{O}(N)$ overhead. Total memory traffic therefore does not decrease, and elapsed time plateaus. This illustrates a fundamental microarchitectural trade-off: our sparse engine optimizes for *throughput* by reducing total arithmetic work ($4.5\times$ fewer instructions on VegFru), but pays a *latency* penalty per-row due to the $\mathcal{O}(N)$ accumulator zeroing. On high-memory servers with many cores, the dense implementations’ superior IPC and predictable memory access patterns eventually overcome the sparse engine’s work-reduction advantage. A future optimization is to replace the dense accumulator with a hash-based sparse accumulator, which would eliminate this bottleneck at the cost of higher per-access latency.

5.3.2 LHRR Scaling Analysis

At $t = 1$, the sparse advantage is larger for LHRR than for CPRR: $1.49\times$ on ALOI, $2.35\times$ on SUN397, and $2.26\times$ on VegFru. This reflects the higher sequential cost of the dense two-hop traversal relative to CPRR’s simpler one-hop product. The dense implementations respond to threads far more aggressively, reaching $5.74\times$ and $4.83\times$ speedup at 18 threads on

SUN397 and VegFru, with 14.0 and 13.2 effective cores respectively. The $\mathcal{O}(N^2)$ row space gives dense LHRR abundant independent work to distribute: at $N = 160,731$ rows each processed by a separate thread with no inter-row dependency, parallelism approaches the physical core count. The sparse engine saturates at $P_{\text{eff}} \leq 2.5$ for the same structural reason as in CPRR, compounded by LHRR’s larger $K^2 = 400$ accumulations per row. The crossover where Dense-MMAP first beats Sparse-CSR occurs at $t = 2$ on ALOI, $t = 6$ on VegFru, and $t = 8$ on SUN397; at 18 threads Dense-MMAP leads by $1.33\times$ on VegFru (74.9 s versus 100.0 s).

The parallel scaling data confirm that the two approaches are not in direct competition on the unconstrained server. Dense-MMAP implementations trade memory for parallelism: their $\mathcal{O}(N^2)$ working set is expensive per thread but decomposes naturally across many threads, yielding good wall-clock times at high core counts when DRAM is plentiful. Sparse-CSR implementations trade parallelism for memory efficiency: the per-row accumulator bottleneck limits effective parallelism to roughly 2 effective cores, but the single-thread work reduction means the sparse engine is faster at low thread counts and is the only viable option on any hardware tier where the $\mathcal{O}(N^2)$ matrix exceeds physical memory. On commodity hardware (≤ 16 GB), the dense baseline fails with OOM regardless of thread count (see Section 5.4 for the detailed feasibility breakdown); on such hardware the Sparse-CSR engine’s single-thread performance, completing VegFru in under 160 seconds per iteration, is the relevant figure of merit.

Crucially, because near-peak throughput for the sparse engine is achievable with only 4–6 cores, the proposed solution offers a superior cost-benefit ratio by enabling scalable retrieval on hardware that is economical in both memory capacity and processor count.

5.3.3 Answer to RQ1

The sparse reformulation provides a significant baseline efficiency advantage for many-hop algorithms like LHRR, achieving up to $2.4\times$ speedup on high-memory servers through work reduction. While it extracts less benefit from additional threads than the dense baseline, it achieves its peak efficiency at much lower core counts, reinforcing its suitability for commodity hardware.

5.4 THE FEASIBILITY GAP: CONSTRAINED HARDWARE ANALYSIS

While Section 5.2 analyzed performance on unconstrained hardware, this section evaluates the sparse engine under the memory-constrained conditions for which it was specifically designed. This is accomplished using Environment B (16 GB) and Environment C (8 GB), which simulate the physical memory limits of constrained servers and commodity desktops, respectively.

Table 10 – Unified Performance and Feasibility Comparison: CPRR across *Hardware* Environments. Status: Success (Time) or OOM. Parameters adjusted per environment to reflect hardware constraints. Parameter T was set to 5 for all configurations.

Dataset	Backend	Env A (192GB)	Env B (16GB)	Env C (8GB)
		$L = 2000, K = 50$	$L = 1000, K = 20$	$L = 1000, K = 20$
ALOI (72K)	Custom engine	27.8s	11.5s	9.7s
	Intel MKL	17.2s	8.4s	8.6s
	GraphBLAS	14.0s	7.7s	7.9s
	Eigen	26.9s	11.0s	10.0s
	Dense (MMAP)	15.2s	T/O*	T/O*
	Dense (Swap)	17.7s	12.3s	OOM
SUN397 (108K)	Custom engine	102.6s	32.8s	21.0s
	Intel MKL	40.5s	14.2s	14.8s
	GraphBLAS	31.5s	13.3s	14.1s
	Eigen	133.3s	36.9s	37.7s
	Dense (MMAP)	24.8s	T/O*	T/O*
	Dense (Swap)	28.5s	OOM	OOM
VegFru (160K)	Custom engine	94.9s	35.6s	33.1s
	Intel MKL	44.8s	18.1s	30.6s
	GraphBLAS	34.8s	16.9s	23.4s
	Eigen	103.6s	35.8s	41.7s
	Dense (MMAP)	34.6s	T/O*	T/O*
	Dense (Swap)	38.7s	OOM	OOM

* Execution halted after 2 hours (timeout) due to excessive disk thrashing.

5.4.1 CPRR Feasibility Under Memory Constraints

Table 10 presents different results for CPRR, which involves only one-hop neighborhood intersection rather than two-hop transitive expansion. Because CPRR’s intermediate products do not exhibit the $\mathcal{O}(NK^2)$ density growth that characterizes LHRR, the compute-then-prune model of general-purpose libraries does not cause OOM failures: GraphBLAS and Intel MKL complete all datasets within the memory envelope of Environments B and C. Crucially, CPRR’s lower intermediate memory footprint allows it to execute in Environment C using the same $L = 1,000, K = 20$ configuration as Environment B, without the truncation to $L = 400$ required for LHRR on the same restricted hardware. In these conditions, mature libraries outperform the custom engine for CPRR (for example, GraphBLAS achieves 7.9 s on ALOI in Environment C compared to 9.7 s for the custom engine) by exploiting AVX-512 kernels on workloads where memory expansion is not the limiting factor. This contrast establishes a clear practical heuristic: for algorithms whose intermediate products are naturally bounded (CPRR-class), general-purpose sparse libraries are preferable; for algorithms with uncontrolled intermediate expansion (LHRR-class), the fused engine presented in this dissertation is the only implementation that remains viable on commodity hardware.

5.4.2 LHRR Feasibility Under Memory Constraints

In Environment B (16 GB) and Environment C (8 GB), the **Dense-Swap** *baseline* based on anonymous *swap-backed* allocation encounters immediate OS-level OOM termination on SUN397 and VegFru before any computation completes. The **Dense-MMAP** *baseline* avoids

Table 11 – Unified Performance and Feasibility Comparison: LHRR across *Hardware* Environments. Status: Success (Time) or OOM. Parameters adjusted per environment to reflect hardware constraints. Parameter T was set to 5 for all configurations.

Dataset	Backend	Env A (192GB)	Env B (16GB)	Env C (8GB)
		$L = 2000, K = 50$	$L = 1000, K = 20$	$L = 1000/400^\dagger, K = 20$
ALOI (72K)	Custom engine	227.5s	57.6s	37.8s
	Intel MKL	264.9s	75.4s	78.8s
	GraphBLAS	39.8s	16.2s	25.6s
	Eigen	241.5s	47.5s	45.0s
	Dense (MMAP)	211.4s	T/O*	T/O*
	Dense (Swap)	214.4s	54.7s	OOM
SUN397 (108K)	Custom engine	579.0s	197.1s	68.6s[†]
	Intel MKL	OOM	OOM	OOM
	GraphBLAS	305.4s	OOM	OOM
	Eigen	178.3s	OOM	OOM
	Dense (MMAP)	1327.4s	T/O*	T/O*
	Dense (Swap)	1348.5s	OOM	OOM
VegFru (160K)	Custom engine	695.2s	258.6s	92.1s[†]
	Intel MKL	OOM	392.7s	OOM
	GraphBLAS	214.8s	OOM	OOM
	Eigen	190.3s	OOM	OOM
	Dense (MMAP)	1651.4s	T/O*	T/O*
	Dense (Swap)	1667.6s	OOM	OOM

[†] Parameter L reduced to 400 for SUN397 and VegFru in Env C to reflect the 8 GB hardware constraint.

* Execution halted after 2 hours (timeout) due to excessive disk thrashing.

immediate process termination by substituting disk-backed virtual pages for physical memory; however, the irregular random-access traversal pattern of the $N \times N$ similarity matrix generates a sustained page-fault rate that saturates I/O *throughput*, producing a *thrashing* regime in which the operating system spends the majority of cycles servicing page faults rather than executing re-ranking computation. This manifests as a practical execution *timeout* exceeding two hours. Table 11 designates these cases with the label **T/O** to distinguish them from immediate OOM terminations; both outcomes are functionally equivalent — no re-ranked output is produced within a deployable time envelope. For VegFru, the root cause is unambiguous: the $N \times N$ dense matrix requires $N^2 \times 4 = 160,000^2 \times 4 \approx 102,400 \text{ MB} \approx 102 \text{ GB}$, exceeding the capacity of all three constrained environments by at least one order of magnitude.

To formally bound the peak memory *footprint* of the *Sparse-CSR engine* and substantiate the $400\times$ reduction claim without reliance on empirical RSS profiling alone, the dominant data structures are modelled analytically. Three categories of persistent or peak-concurrent allocation exist during LHRR execution. The first is the ranked list matrix **R**, stored as a flat array of $N \times L$ unsigned 32-bit integers, requiring $4NL$ bytes. For $N = 160,000$ and $L = 400$, this evaluates to $4 \times 160,000 \times 400 = 256,000,000 \text{ bytes} = 256 \text{ MB}$; this structure persists throughout all iterations as the sole input to each CSR construction phase. The second category is the CSR similarity matrix itself, composed of three contiguous arrays: `row_ptr_` of $(N + 1)$ 32-bit integers ($\approx 0.61 \text{ MB}$), `col_idx_` of at most NL 32-bit integers ($\leq 4NL$ bytes), and `values_` of at most NL 32-bit floats ($\leq 4NL$ bytes). The theoretical per-instance upper bound is therefore $(N + 1 + 2NL) \times 4$ bytes. The third category is the set of thread-local dense accumu-

lators within the *FusedSpGEMM kernel*: each of the T threads maintains one contiguous array of N 32-bit floats for $\mathcal{O}(1)$ -cost random writes during row accumulation, totalling $4TN$ bytes in aggregate. Combining all three categories under simultaneous peak occupancy:

$$M_{\text{peak}} \leq \underbrace{4NL}_{\mathbf{R}} + \underbrace{(N + 1 + 2NL) \times 4}_{\text{CSR instance (upper bound)}} + \underbrace{4TN}_{\text{thread accumulators}} \quad (1)$$

Substituting $N = 160,000$, $L = 400$, $T = 18$:

$$M_{\text{peak}} \leq 256 \text{ MB} + 488 \text{ MB} + 11 \text{ MB} = 755 \text{ MB} \quad (2)$$

This bound is conservative: it assumes that one CSR instance reaches full NL -density concurrently with $\mathbf{R}$ and all T accumulators. In practice, the *fused pruning kernel* prevents any CSR row from accumulating more than L non-zeros, and intermediate CSR instances are explicitly released before the next allocation, precluding concurrent full-density occupancy.

The same constraint that eliminates the dense implementations also eliminates GraphBLAS, Eigen, and Intel MKL on SUN397 and VegFru in Environments B and C, despite their use of sparse representations internally. All three libraries follow a *compute-then-prune* model: the intermediate product $\mathbf{HH}^\top$ is fully materialized before any row-wise selection is applied. During LHRR’s *hyperedge* construction, this intermediate can accumulate up to $\mathcal{O}(NK^2)$ non-zeros before the top- L selection reduces the result to $\mathcal{O}(NL)$; for $N = 160,000$ and $K = 50$, the intermediate count reaches $NK^2 = 160,000 \times 2,500 = 4 \times 10^8$ entries at 8 bytes per entry, yielding approximately 3.2 GB for a single product before pruning. Combined with library-internal allocation *overhead* and the persistent $\mathbf{R}$ structure, the total footprint exceeds the 8–16 GB envelope of constrained environments. The *Sparse-CSR engine* prevents this intermediate expansion entirely by integrating top- L selection directly into the accumulation loop (Section 4.3.4), ensuring that the working set never exceeds the bound of Equation 1 at any point during execution. It is therefore the only implementation that completes LHRR on VegFru across all three hardware environments.

A notable result in Table 11 is the wall-clock advantage of Environment C over Environment B for ALOI ($L = 1,000$, identical across both environments). The 34% reduction in execution time (37.8 s in Env C versus 57.6 s in Env B) reflects a direct frequency advantage: once the memory bottleneck is resolved through sparsity, execution *throughput* scales with single-core instruction *throughput*, and the consumer-grade i5-10400F in Environment C operates at a *turbo* clock of 2.90 GHz versus the 2.20 GHz base frequency of the server-grade Xeon Gold 5220 in Environments A and B. For SUN397 and VegFru in Environment C, the parameter L was reduced to 400 to satisfy the 8 GB physical memory constraint (annotated in Table 11); the wall-clock reduction for these *datasets* is attributable to both the frequency advantage and the reduced list depth, and does not represent a pure architectural comparison.

5.4.3 Answer to RQ2

The experimental results across Environments B and C provide a clear answer to RQ2. The Sparse-CSR engine maintains $\mathcal{O}(NL)$ memory growth, which translates to a reduction of up to $400\times$ in peak memory requirements compared to the dense $\mathcal{O}(N^2)$ footprint (on the VegFru dataset). This reduction is the enabling factor that allows re-ranking to execute on commodity hardware (8–16 GB) where all dense implementations, even when backed by virtual memory workarounds like MMAP, fail due to either OOM termination or thrashing-induced timeouts.

5.5 RETRIEVAL QUALITY VERIFICATION

The primary correctness claim of this dissertation, namely that the sparse reformulation is exact, is established analytically in Section 4.2: every arithmetic operation of the dense update rules has an exact counterpart in the sparse formulation, and the top- L selection is applied at identical points in both execution paths. This section provides numerical confirmation of that analytical result and separately characterizes the effect of the sparsity parameter L on retrieval quality across dataset types.

Bitwise MAP equivalence between sparse and dense implementations is confirmed for all configurations evaluated in Environment A, covering the full parameter grid over L and K for ALOI and SUN397 (VegFru is only accessible to the sparse engine; its quality is reported without a dense comparison). This numerical confirmation does not add information beyond the analytical proof, but serves as an empirical validation against implementation artifacts that could silently corrupt the output while preserving the algorithmic structure. Tables 12 through 14 therefore present MAP scores for the sparse engine (dense values, where available, are confirmed to be identical to the last reported digit and are omitted for brevity).

5.5.1 Answer to RQ3

The sparse reformulation produces ranked lists that are numerically identical to those of the dense baseline for every configuration where comparison is possible (all parameter combinations over L and K for ALOI and SUN397 in Environment A). This confirms that the reformulation is exact: no approximation is introduced at any stage of the sparse execution.

Smooth manifolds: ALOI.

ALOI consists of objects photographed under systematic rotations and illumination changes, creating a smooth and continuous underlying manifold. As shown in Table 12, accuracy increases monotonically with L : MAP rises from a baseline of 44.0% to a peak of 60.6% at $L = 2,000$. This positive correlation indicates that for smooth manifolds, a denser neighborhood graph successfully bridges the transitions between viewpoints, allowing the re-ranking algorithm

Table 12 – ALOI (72K): MAP accuracy grid search (L vs K).

Method	L Parameter	K Parameter (Neighbors, T=5)		
		K=20	K=30	K=50
Baseline	(No Re-ranking)	44.0		
CPRR	L=100	45.3	46.2	44.4
	L=400	49.1	51.0	48.7
	L=1000	49.9	51.8	49.5
	L=2000	50.1	52.0	49.7
LHRR	L=100	46.0	43.9	42.4
	L=400	55.4	52.1	48.7
	L=1000	58.9	55.2	50.9
	L=2000	60.6	56.8	52.0

to traverse the full rotation curve of an object. In this regime, maximizing L within the memory envelope is the optimal strategy.

Scene recognition: SUN397.

Table 13 – SUN397 (108K): MAP accuracy grid search (L vs K).

Method	L Parameter	K Parameter (Neighbors, T=5)		
		K=20	K=30	K=50
Baseline	(No Re-ranking)	18.2		
CPRR	L=100	27.1	29.5	32.1
	L=400	20.3	22.0	24.2
	L=1000	20.3	22.3	24.5
	L=2000	21.6	23.7	26.1
LHRR	L=100	28.7	29.7	30.6
	L=400	21.7	22.5	23.5
	L=1000	21.9	22.8	23.9
	L=2000	23.3	24.4	25.6

SUN397’s 397 semantic categories exhibit high intra-class variance and background clutter, creating a noisier manifold than ALOI. Table 13 shows that performance peaks at $L = 100$ rather than at higher densities: the best MAP of 32.1% is achieved with CPRR at $L = 100$, declining to 26.1% at $L = 2,000$. This inversion suggests that a sparser neighborhood graph filters out noisy connections between visually similar but semantically unrelated scenes that would otherwise pollute the propagation signal.

Table 14 – VegFru (160K): MAP accuracy grid search (L vs K).

Method	L Parameter	K Parameter (Neighbors, T=5)		
		K=20	K=30	K=50
Baseline	(No Re-ranking)	49.4		
CPRR	L=100	68.1	70.1	72.4
	L=400	54.5	55.6	57.2
	L=1000	49.0	50.1	51.6
	L=2000	50.1	51.3	52.9
LHRR	L=100	69.8	70.5	71.2
	L=400	57.6	57.8	58.2
	L=1000	52.3	52.9	53.2
	L=2000	53.4	54.2	54.7

Fine-grained classification: VegFru.

VegFru exhibits a similar pattern to SUN397: Table 14 shows a peak MAP of 72.4% at $L = 100$, declining to 52.9% at $L = 2,000$. The fine-grained nature of the dataset (distinguishing between varieties of produce that share global shape and color) makes the manifold particularly susceptible to long-range noisy connections that are admitted at high L but pruned at low L . This result also carries a practical implication: the L values required for peak accuracy on these datasets (100–400) are well within the memory envelope of Environment C, confirming that the parameter regime forced by hardware constraints is not a quality compromise but a regularization benefit.

5.6 DISCUSSION

This section deepens the analysis of the preceding results by examining parameter sensitivity, microarchitectural trade-offs, and the conditions under which each implementation strategy is preferable, deriving practical guidelines for memory-bounded manifold learning systems.

5.6.1 Structural Limits of Sparsity and Dataset Scale

The efficiency of sparse reformulation is tied to the structural sparsity of the re-ranking problem, defined by the ratio between the ranked list length L and the total dataset size N . On VegFru ($N = 160,000$, $L = 400$), the resulting similarity matrix maintains a density of only 0.25%, and the $\mathcal{O}(NL)$ storage requirement is three orders of magnitude smaller than the $\mathcal{O}(N^2)$ dense baseline. However, the benefits of sparsity are not uniform across all dataset scales.

For small-scale datasets where $N < 80,000$, the regular, contiguous memory access patterns of dense representations allow high-performance BLAS implementations to exploit SIMD vectorization and hardware prefetching more effectively than sparse engines. CSR traversal, while reducing the total volume of computation, introduces indirect addressing overhead and potential pipeline stalls that can dominate execution time on smaller matrices. This is consistent with the ALOI result in Section 5.2, for the LHRR algorithm, where the dense baseline completes in 211.4 s compared to 227.5 s for the sparse engine (Table 7): at this scale, the brute-force utilization of SIMD lanes on $\mathcal{O}(N^2)$ data is faster than irregular traversal of $\mathcal{O}(NL)$ non-zeros. As N grows, however, the gap between N^2 and NL widens rapidly and the instruction count differential becomes the dominant factor, reversing the performance relationship entirely. The crossover observed in our experiments occurs between ALOI and SUN397, consistent with the point at which the dense matrix begins to exceed the last-level cache capacity and the hardware prefetcher loses effectiveness over the full $N \times N$ working set.

5.6.2 Microarchitectural Tradeoffs Between Efficiency and Total Work

A recurring pattern in the results above is that the sparse engine achieves significant overall speedups despite exhibiting worse per-cycle efficiency metrics than the dense baseline on some workloads. For CPRR on ALOI, the dense implementation achieves higher IPC (1.46 versus 1.27) and lower LLC miss rates, yet the sparse engine is competitive in wall-clock time. For LHRR on larger datasets, the relationship inverts: the dense implementation’s IPC collapses to 0.40–0.41 as its working set overwhelms the cache hierarchy, while the sparse engine sustains IPC above 1.0. In both regimes, the underlying determinant of total execution time is not per-cycle throughput but the volume of instructions retired: the sparse engine retires $4.5\times$ fewer instructions on VegFru by skipping the $\mathcal{O}(N^2 - NL)$ entries that carry no information and are discarded at the end of every iteration regardless.

This observation has a direct design implication for future sparse engines: the primary optimization target should be minimizing the total number of non-zeros processed, not maximizing per-cycle SIMD utilization. A kernel that processes $10\times$ fewer entries with 20% worse IPC is still $8\times$ faster overall. This contrasts with the conventional HPC optimization philosophy of maximizing hardware utilization on a fixed workload, and suggests that algorithmic sparsity analysis should precede microarchitectural tuning in the design of manifold learning systems.

The fused multiply-and-prune kernel introduces one notable microarchitectural cost: moderately elevated branch misprediction rates due to the data-dependent conditional logic that determines whether each partial product survives the top- L selection. This logic cannot be predicted from historical patterns because the surviving entries depend on the actual values accumulated during the current row, not on the structure of the input matrices. Sparse branch misprediction rates range from 2.3% to 5.2% across the configurations measured, compared to 1.0%–2.7% for the dense baseline, an absolute difference of up to 3.5 percentage points.

While elevated relative to the dense baseline, these rates remain low in absolute terms and do not materially affect throughput: the total cycles lost to branch recovery represent a small fraction of the $4.5\times$ instruction reduction achieved by skipping structurally-zero entries one row at a time. The accompanying reduction in memory load intensity, since sparse methods load only the $\mathcal{O}(NL)$ stored non-zeros rather than the full $\mathcal{O}(N^2)$ matrix, more than compensates for this overhead by yielding substantially lower memory load volumes on larger datasets (SUN397, VegFru), where the dense working set exceeds the last-level cache; the effect is negligible or reversed at ALOI scale where both representations exhibit comparable data-access patterns.

5.6.3 Parameter Sensitivity and the Role of Neighborhood Density

The accuracy results of Section 5.5 reveal that the relationship between neighborhood density L and retrieval quality is not monotonic across dataset types, and understanding why this is the case provides practical guidance for parameter selection.

For ALOI, accuracy improves continuously with L , reaching a peak MAP of 60.6% at $L = 2,000$. This dataset consists of objects photographed under systematic rotations, creating a smooth and continuous underlying manifold. In this regime, entries in the long tail of the similarity distribution represent genuine intermediate viewpoints that bridge otherwise disconnected parts of the rotation curve; retaining them improves the algorithm’s ability to propagate similarity across the full manifold. The positive correlation between L and accuracy therefore reflects a genuine information gain: each additional neighbor stored per row contributes structural information that was not captured at lower L .

SUN397 and VegFru exhibit the opposite pattern, with accuracy peaking at $L = 100$ and declining monotonically as L increases. In these datasets, the manifold is populated by high-frequency visual noise and loose semantic associations, including scenes that share a superficial color palette, or produce varieties whose shape features overlap across classes. At low L , the neighborhood graph connects only the most strongly similar images, and the propagation signal is dominated by true semantic neighbors. As L increases, the additional entries admitted to each row are increasingly likely to be noisy connections that bridge unrelated clusters, diluting the propagation signal and degrading accuracy. The accuracy peak at $L \approx 100$ therefore marks the point at which the neighborhood graph is dense enough to capture the true manifold structure but sparse enough to exclude most noisy connections.

This phenomenon carries a practically important consequence: the L values required for peak accuracy on noisy datasets (100–400) fall well within the memory envelope of the most constrained hardware tier evaluated (Environment C, 8 GB). The hardware constraints that force conservative L settings on commodity hardware are not a quality compromise; for complex or noisy datasets, they constitute an implicit regularization that aligns the system with the intrinsic information content of the data. Practitioners should therefore not assume that maximizing L within available memory is always the optimal strategy, and should instead tune L based on an estimate of the dataset’s manifold smoothness.

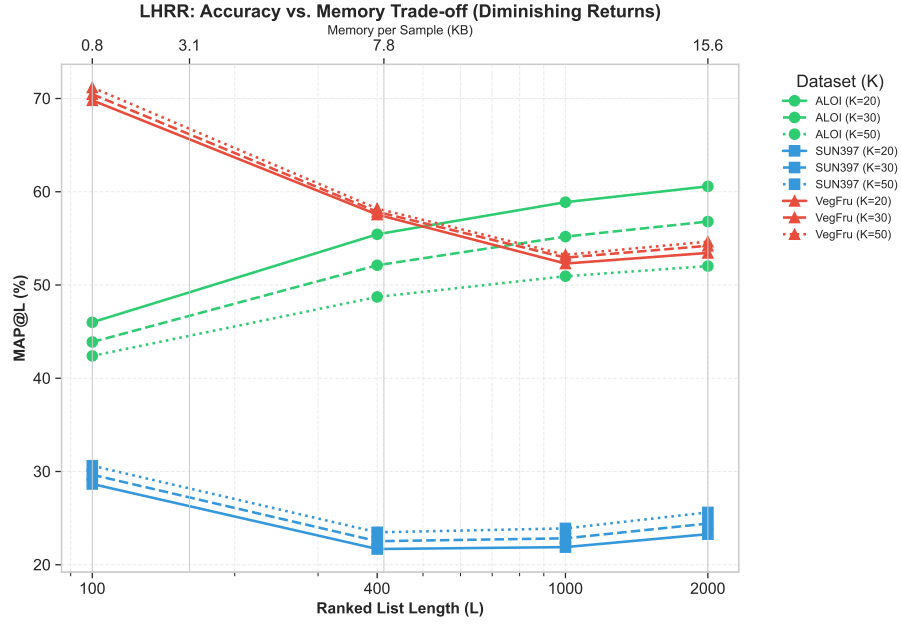


Figure 7 – LHRr accuracy as a function of ranked list depth L across dataset types. ALOI shows continuous improvement with increasing L , consistent with a smooth manifold where additional neighbors carry genuine structural information. SUN397 and VegFru peak at low L and decline thereafter, as additional neighbors introduce noisy connections that degrade propagation quality.

Beyond the list depth L , the neighborhood truncation parameter K must also be chosen carefully in two-hop algorithms like LHRr. On ALOI, increasing K from 20 to 50 results in a 14.7% absolute MAP reduction, because the two-hop hyperedge expansion at $K = 50$ connects each image to up to 2,500 two-hop neighbors, many of which are semantically irrelevant and introduce noise into the hyperedge weights. This confirms that LHRr’s accuracy is sensitive to the quality of local neighborhoods, not only their size, and that conservative K values are preferable on datasets where the local neighborhood structure is noisy.

5.6.4 Answer to RQ4

The inclusion of retrieval quality metrics within this HPC-focused evaluation serves a precise and delimited function: MAP constitutes the *end-to-end* system verification protocol that confirms the sparse reformulation is mathematically exact rather than approximate. Demonstrating bitwise-identical MAP scores between the *Sparse-CSR engine* and the dense *baseline* across all parameter configurations evaluated in Environment A provides proof that the loop fusion, row-split construction, and block-allocated memory management introduce no floating-point divergence, no reordering of arithmetic operations, and no approximation at any stage of the *pipeline*. This verification is a necessary precondition for the HPC contributions of this work: an optimized system that produces numerically different outputs from its reference implementation has not been optimized — it has been altered. The correctness claim rests on the analytical argument of Section 4.2 and is corroborated by the empirical confirmation reported in

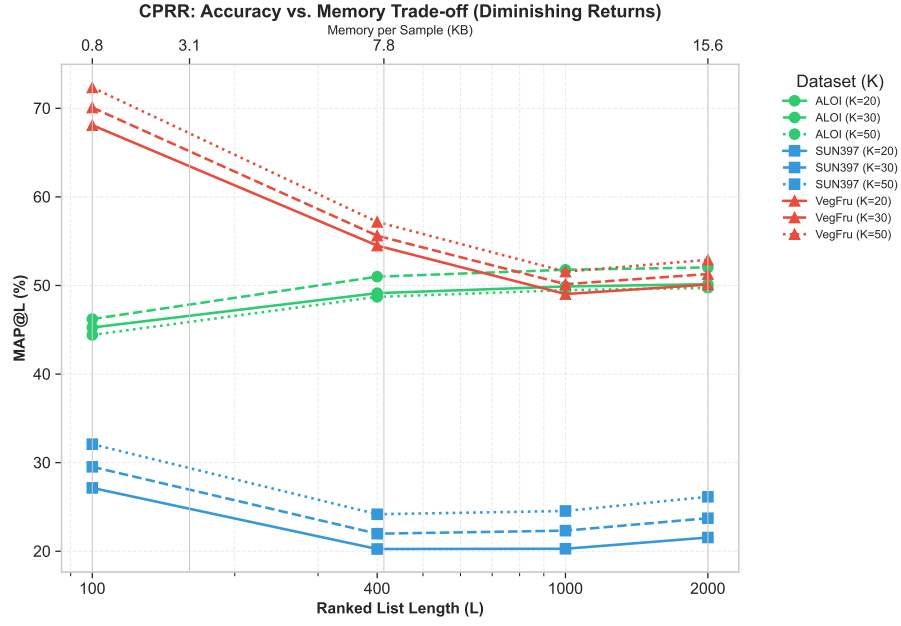


Figure 8 – CPRR accuracy as a function of L . The accuracy plateau on SUN397 and VegFru occurs earlier than for LHRR, consistent with CPRR’s one-hop co-occurrence model saturating the available semantic signal at lower neighborhood densities than LHRR’s two-hop hyperedge construction.

Section 5.5.1.

Beyond correctness verification, the MAP results across parameter configurations establish that the sparsity parameter L functions as a tunable *regularizer* whose optimal value is determined by the manifold geometry of the *dataset* rather than by memory capacity. For smooth, continuous *manifolds* such as ALOI — where objects are photographed under systematic rotation, forming a well-connected topological structure — quality improves monotonically with L , reaching a peak MAP of 60.6% at $L = 2,000$. In this regime, each additional neighbor per row contributes genuine structural information that bridges otherwise disconnected viewpoint transitions; the optimal strategy on smooth *manifolds* is to maximize L within the physical memory envelope of the target *hardware*.

For high-variance, noisy *datasets* (SUN397 and VegFru), the relationship inverts: MAP peaks at $L \in [100, 400]$ and declines monotonically as L increases, because entries admitted beyond the local coherence radius introduce noisy cross-cluster connections that degrade the manifold propagation signal. The L values enforced by the 8 GB memory ceiling of Environment C ($L = 400$) coincide precisely with this accuracy *optimum*. The *hardware* constraint does not constitute a quality *trade-off* in resource-constrained *deployment* environments: the memory bound is structurally aligned with the information-theoretic *optimum* of the manifold learning process. This alignment follows from the fact that noisy *manifolds* contain diminishing marginal neighborhood information beyond a coherence radius that is small relative to N — the same structural property that makes the $\mathcal{O}(NL)$ sparse representation both memory-viable and quality-optimal for this class of *datasets*.

5.6.5 Deployment Guidelines

Based on the preceding analysis, we summarize the recommended parameter configurations for three representative deployment scenarios. On high-memory research clusters (192 GB), LHRR with $K = 20$ and $L = 1,000$ provides strong effectiveness across all dataset types while leaving substantial headroom; on smooth manifold datasets such as ALOI, increasing L further yields additional gains. On standard developer workstations (8–16 GB), the same LHRR configuration with $L = 1,000$ is suitable for ALOI and remains within the memory envelope; for larger or noisier datasets, reducing L to 400 is recommended and, as shown in the results above, does not sacrifice accuracy. On embedded or edge systems below 8 GB, CPRR with $K = 30$ and $L = 400$ delivers full effectiveness on noisy datasets within a footprint below 500 MB, and avoids the two-hop intermediate expansion that makes LHRR memory-intensive regardless of L .

5.7 SUMMARY

The results presented in this chapter validate the three claims introduced in Chapter 4. On high-memory hardware, the sparse engine retires $4.5\times$ fewer instructions than the dense baseline on VegFru, translating into wall-clock speedups of up to $2.4\times$ for LHRR; this gain arises not from superior per-cycle efficiency but from the reduction of total computational work over $\mathcal{O}(NL)$ rather than $\mathcal{O}(N^2)$ entries. On memory-constrained commodity hardware (≤ 16 GB), the custom Sparse-CSR engine is the only viable option for LHRR-class algorithms. The reduction in peak memory usage reaches up to $400\times$ (from 102 GB to 256 MB) on the VegFru dataset, effectively enabling the execution of large-scale manifold learning on commodity hardware. Retrieval quality is fully preserved: the sparse reformulation produces bitwise-identical MAP scores to the dense baseline for all configurations where both can execute, and the parameter regime imposed by commodity hardware coincides with the accuracy peak for noisy datasets. The implications of these findings, including the microarchitectural tradeoffs between IPC efficiency and total work, the conditions under which each implementation strategy is preferable, and the broader design principles for memory-bounded graph algorithms, are discussed in the preceding sections. Chapter 6 synthesizes these findings into a summary of contributions and directions for future work.

6 CONCLUSION

Manifold-based re-ranking achieves strong retrieval quality by reasoning about the global topology of the dataset rather than relying on pairwise descriptor distances alone. The cost of this capability has historically been a quadratic memory requirement: every standard implementation materializes a global dense similarity matrix $\mathbf{M} \in \mathbb{R}^{N \times N}$, consuming hundreds of gigabytes for collections of practical size and confining the technique to specialized high-memory infrastructure. This dissertation demonstrates that this memory cost is not theoretically inherent to the algorithms, but rather a consequence of the underlying data structures.

The similarity matrix $\mathbf{M}$ is structurally sparse from the outset: each image maintains meaningful relationships with only L of its N neighbors, and every iteration discards all entries beyond the top- L selection. Dense implementations allocate, traverse, and write back $\mathcal{O}(N^2)$ entries at every step for computation whose outcome is determined by $\mathcal{O}(NL)$ entries, an information-footprint mismatch of N/L in favor of the sparse representation.

Addressing this mismatch requires more than a direct substitution of the dense matrix with a standard sparse format. General-purpose sparse libraries follow a compute-then-prune model that re-creates the $\mathcal{O}(N^2)$ footprint in the intermediate product before any filtering is applied. The key engineering contribution of this work is fusing the pruning with the multiplication, so that intermediate structures never exceed the $\mathcal{O}(NL)$ bound even transiently. This fusion enables the engine to complete execution in cases where general-purpose library implementations exhaust available memory.

The practical consequence is measurable: a 160,000-image dataset that requires 102 GB in its dense form can be processed with a 256 MB similarity matrix on a consumer desktop with 8 GB of RAM. Retrieval quality is preserved; on noisy datasets, the parameter regime imposed by memory constraints actually improves it. The scalability barrier that has historically confined manifold-based re-ranking to specialized infrastructure is therefore not a property of the algorithms, but of how those algorithms have been implemented.

The following sections consolidate the contributions, recapitulate the principal findings, characterize the scope and limitations of the approach, and outline directions for future work.

6.1 SUMMARY OF CONTRIBUTIONS

The first contribution is a **formal sparse reformulation** of manifold-based re-ranking as a sequence of sparse linear algebra operations. We show that the update rules of CPRR and LHRR, two algorithms spanning the spectrum from lightweight one-hop co-occurrence to expensive two-hop hyperedge expansion, decompose exactly into Sparse General Matrix-Matrix Multiplication (SpGEMM), sparse Hadamard products, and sorted row merging. Because each

output matrix is pruned to the L highest-valued entries per row at every step, the theoretical memory complexity falls from $\mathcal{O}(N^2)$ to $\mathcal{O}(NL)$ with no approximation of the underlying manifold dynamics. The reformulation is verified bitwise: the ranked lists produced by the sparse engine are identical to those of the dense *baseline* for every configuration where both can execute.

A central challenge of this reformulation is the *intermediate density explosion*: general-purpose sparse libraries materialize the full product before applying any selection, which can expand to $\mathcal{O}(NK^2)$ non-zeros during LHRR’s hyperedge construction and exceed available memory even when the final output is sparse. The **fused multiply-and-prune SpGEMM kernel** addresses this by integrating top- L selection directly into the inner accumulation loop, ensuring that the working set never exceeds $\mathcal{O}(NL)$ non-zeros at any point during execution. This fusion enables the engine to complete execution in cases where general-purpose library implementations exhaust available memory.

The second contribution is a **high-performance execution engine** that realizes the theoretical $\mathcal{O}(NL)$ bound in practice. The row-split partitioning strategy assigns disjoint row ranges to threads, eliminating mutexes from the construction hot path. The `BlockAllocator` replaces standard `malloc` with pointer-bump allocation over THP-aligned 2 MB arenas, reducing allocator overhead from a significant fraction of execution cycles to a negligible level. Cache-line padding between thread-local control structures prevents false sharing during parallel triplet emission. Together, these design decisions translate the algorithmic $\mathcal{O}(NL)$ guarantee into measured throughput on hardware ranging from an 18-core research server to a 6-core/12-thread consumer desktop (Environment C, i5-10400F, Ubuntu via WSL2).

These two contributions are validated through a **systematic experimental evaluation** across three benchmark datasets (ALOI 72K, SUN397 108K, VegFru 160K), three hardware tiers, and a grid of parameter configurations ($L \in \{100, 400, 1000, 2000\}$, $K \in \{20, 30, 50\}$, $T = 5$ re-ranking iterations for all performance and accuracy experiments).

The experimental evaluation validates these contributions across four research questions:

Principal findings.

RQ1 (Efficiency).

On high-memory hardware, LHRR achieves up to $2.4\times$ wall-clock speedup through a $4.5\times$ instruction reduction on VegFru, driven by elimination of structurally irrelevant work over the $\mathcal{O}(N^2 - NL)$ entries that the dense *baseline* traverses only to discard. For CPRR, the dense *baseline* is faster on unconstrained hardware due to superior prefetch and SIMD efficiency; the sparse advantage materializes only under memory constraints, where the dense method fails with OOM.

RQ2 (Scalability and Feasibility).

The custom engine is the only implementation that completes LHRR on VegFru across all three hardware tiers, with peak similarity-matrix footprints of 256 MB (Environment C) versus 102 GB (dense), a $400\times$ reduction. All three general-purpose sparse libraries also fail on VegFru in constrained environments due to intermediate density explosion during hyperedge expansion.

RQ3 (Correctness).

Bitwise MAP equivalence between the sparse engine and the dense *baseline* is confirmed for all parameter configurations on ALOI and SUN397. The correctness claim rests on a formal argument: the sparse reformulation performs exactly the same floating-point operations and applies the top- L selection at identical points; the bitwise match serves as an implementation sanity check.

RQ4 (Quality, memory trade-off).

For smooth manifolds (ALOI), MAP improves monotonically with L . For noisy datasets (SUN397, VegFru), MAP peaks at $L \approx 100\text{--}400$, with further expansion introducing neighborhood noise. The conservative L values forced by commodity hardware constraints coincide with the accuracy optimum for these datasets; the memory ceiling is not a quality compromise but an implicit regularization.

6.2 LIMITATIONS

The evaluation covers datasets up to 160,731 images, a substantial increase over typical dense UDLF experiments but several orders of magnitude below industrial-scale collections. Bottlenecks in the initial k -nearest neighbor construction, which was taken as a fixed input throughout this work, may become dominant as N approaches the tens of millions.

The sparse engine targets LHRR-class algorithms, where intermediate products grow to $\mathcal{O}(NK^2)$ before pruning. For CPRR-class algorithms, whose intermediate products are naturally bounded, general-purpose sparse libraries (GraphBLAS, Intel MKL) outperform the custom engine on unconstrained hardware by exploiting AVX-512 kernels. The custom engine is therefore most valuable for algorithms with uncontrolled intermediate expansion, not as a universal replacement for mature library implementations.

At dataset scales where $N \lesssim 80,000$, dense methods can outperform the sparse engine on unconstrained hardware because regular contiguous memory access enables effective SIMD prefetching. The performance crossover occurs near the point at which the dense working

set exceeds last-level cache capacity, observed experimentally between ALOI (72K) and SUN397 (108K).

The `BlockAllocator`’s alignment with Transparent Huge Pages is specific to Linux environments supporting `transparent_ hugepage`. Portability to Windows or ARM-based architectures would require alternative large-page allocation strategies. The global parallel sort during CSR construction introduces a synchronization barrier that may limit scaling beyond the 18–24 core range.

6.3 AVENUES FOR FUTURE WORK

The most direct extension is a **hash-based accumulator** to replace the current $\mathcal{O}(N)$ dense accumulator in the fused SpGEMM kernel. The present design zeroes N slots per output row regardless of how many partial products actually accumulate (K^2), and this zeroing cost dominates the parallel scaling bottleneck. A hash-based accumulator bounded to $\mathcal{O}(K^2)$ entries would eliminate this overhead, potentially enabling near-linear scaling with thread count at the cost of higher per-access latency. This engineering trade-off requires further empirical evaluation.

Porting the fused kernel to **GPU architectures** offers the potential for substantial parallelism, provided intermediate memory is managed through streaming rather than materialization and host-to-device transfer costs for large CSR structures are amortized through pipelining.

Finally, an **adaptive dense-sparse hybrid** that switches representations based on runtime sparsity measurements could recover the SIMD efficiency of dense methods at small scales while maintaining the memory bounds of the sparse engine at large scales. Validating the sparse primitive mapping across all algorithms in the UDLF framework, not only CPRR and LHRR, would complete the foundation for such a system.

BIBLIOGRAPHY

- AZAD, A. et al. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. In: IEEE. **2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.], 2016. p. 44–53.
- BAI, S.; BAI, X. Sparse contextual activation for efficient visual re-ranking. **IEEE Transactions on Image Processing**, IEEE, v. 25, n. 3, p. 1056–1069, 2016.
- BAI, S.; BAI, X.; TIAN, Q. Scalable person re-identification on supervised smoothed manifold. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2017. p. 2530–2539.
- BALDASSIN, A. et al. An optimized unsupervised manifold learning algorithm for manycore architectures. **Information Sciences**, Elsevier, v. 496, p. 410–430, 2019.
- BELLMAN, R. **Dynamic programming**. [S.l.]: Princeton University Press, 1957.
- BOCCES, A. T.; BALDASSIN, A.; DANTAS, A. Optimization in information retrieval: A quick view of techniques for performance and scalability. In: SBC. **Escola Regional de Alto Desempenho de São Paulo (ERAD-SP)**. [S.l.], 2025. p. 62–65.
- BOCCES, A. T. et al. Optimization in information retrieval: A systematic review of techniques for performance and scalability. **IEEE Access**, IEEE, v. 14, p. 2576–2591, 2025.
- BULUÇ, A.; GILBERT, J. R. The combinatorial blas: Design, implementation, and applications. **The International Journal of High Performance Computing Applications**, Sage Publications Sage UK: London, England, v. 25, n. 4, p. 496–509, 2011.
- BULUÇ, A.; GILBERT, J. R. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. **SIAM Journal on Scientific Computing**, SIAM, v. 34, n. 4, p. C170–C191, 2012.
- CHEN, W. et al. Deep learning for instance retrieval: A survey. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, IEEE, v. 45, n. 6, p. 7270–7292, 2022.
- CHUM, O. et al. Total recall: Automatic query expansion with a generative feature model for object retrieval. p. 1–8, 2007.
- DALTON, S.; OLSON, L. N.; BELL, N. Optimizing sparse matrix-matrix multiplication for the gpu. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 12, n. 1, p. 1–29, 2015.
- DATTA, R. et al. Image retrieval: Ideas, influences, and trends of the new age. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 40, n. 2, p. 1–60, 2008.
- DONGEN, S. V. **Graph clustering by flow simulation**. 2000. Tese (Doutorado) — University of Utrecht, 2000.
- DONOSER, M.; BISCHOF, H. Diffusion processes for retrieval revisited. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2013. p. 1320–1327.

- DREPPER, U. What every programmer should know about memory. **Red Hat, Inc**, v. 11, p. 1–114, 2007.
- DUARTE, G. T.; RIBEIRO, W. C.; MIXON, J. P. An evaluation of monte carlo and resampling methods for crop model uncertainty analysis. **Agricultural Systems**, Elsevier, v. 75, n. 1, p. 33–50, 2003.
- GUSTAVSON, F. G. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. **ACM Transactions on Mathematical Software (TOMS)**, ACM, v. 4, n. 3, p. 250–269, 1978.
- HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture: A Quantitative Approach**. 6. ed. [S.l.]: Morgan Kaufmann, 2017. ISBN 978-0128119051.
- HORN, R. A.; JOHNSON, C. R. **Topics in matrix analysis**. [S.l.]: Cambridge University Press, 1994.
- HOU, S.; FENG, Y.; WANG, Z. Vegfru: A domain-specific dataset for fine-grained visual categorization. In: **2017 IEEE International Conference on Computer Vision (ICCV)**. [S.l.: s.n.], 2017. p. 541–549.
- IEEE. **IEEE Standard for Floating-Point Arithmetic**. 2019. IEEE Std 754-2019 (Revision of IEEE Std 754-2008).
- ISCEN, A. et al. Fast spectral ranking for similarity search. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2018. p. 7632–7641.
- ISCEN, A. et al. Efficient diffusion on region manifolds: Recovering small objects with compact cnn representations. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2017. p. 2077–2086.
- JEGOU, H.; HARZALLAH, H.; SCHMID, C. A contextual dissimilarity measure for accurate and efficient image search. In: IEEE. **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.], 2007. p. 1–8.
- JOHNSON, J.; DOUZE, M.; JÉGOU, H. Billion-scale similarity search with gpus. **IEEE Transactions on Big Data**, IEEE, v. 7, n. 3, p. 535–547, 2019.
- KAWAI, V. S. et al. Rank-based hashing for effective and efficient nearest neighbor search for image retrieval. **ACM Transactions on Multimedia Computing, Communications and Applications**, ACM New York, NY, v. 20, n. 10, p. 1–19, 2024.
- LEW, M. S. et al. Content-based multimedia information retrieval: State of the art and challenges. **ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)**, ACM New York, NY, USA, v. 2, n. 1, p. 1–19, 2006.
- Linux Kernel Organization. **Transparent Hugepage Support — The Linux Kernel Documentation**. 2011. <https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html>. Accessed: 2026-02-23.
- LIU, Y. et al. A survey of content-based image retrieval with high-level semantics. **Pattern recognition**, Elsevier, v. 40, n. 1, p. 262–282, 2007.

- MAGLIANI, F.; PRATI, A. An approximate k-nn graph construction algorithm based on neighborhood exploration. **Pattern Recognition Letters**, Elsevier, v. 125, p. 784–790, 2019.
- NAGASAKA, Y. et al. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In: **Proceedings of the 48th International Conference on Parallel Processing**. [S.l.: s.n.], 2019. p. 1–10.
- NILSBACK, M.-E.; ZISSERMAN, A. A visual vocabulary for flower classification. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2006.
- PEDRONETTE, D. C. G.; ALMEIDA, J.; TORRES, R. d. S. A scalable re-ranking method for content-based image retrieval. **Information Sciences**, Elsevier, v. 265, p. 91–104, 2014.
- PEDRONETTE, D. C. G.; PENATTI, O. A.; TORRES, R. d. S. Unsupervised manifold learning using reciprocal knn graphs in image re-ranking and rank aggregation tasks. **Image and Vision Computing**, Elsevier, v. 32, n. 2, p. 120–130, 2014.
- PEDRONETTE, D. C. G.; TORRES, R. d. S. Image re-ranking and rank aggregation based on similarity of ranked lists. **Pattern Recognition**, Elsevier, v. 46, n. 8, p. 2350–2360, 2013.
- PEDRONETTE, D. C. G. et al. Multimedia retrieval through unsupervised hypergraph-based manifold ranking. **IEEE Transactions on Image Processing**, IEEE, v. 28, n. 12, p. 5824–5838, 2019.
- QIN, D. et al. Query adaptive similarity for large scale object retrieval. p. 1401–1408, 2011.
- ROWEIS, S. T.; SAUL, L. K. Nonlinear dimensionality reduction by locally linear embedding. **Science**, American Association for the Advancement of Science, v. 290, n. 5500, p. 2323–2326, 2000.
- SAAD, Y. Iterative methods for sparse linear systems. SIAM, 2003.
- SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating system concepts**. [S.l.]: Wiley, 2018.
- TENENBAUM, J. B.; SILVA, V. D.; LANGFORD, J. C. A global geometric framework for nonlinear dimensionality reduction. **Science**, American Association for the Advancement of Science, v. 290, n. 5500, p. 2319–2323, 2000.
- VALEM, L. P.; PEDRONETTE, D. C. G. Unsupervised similarity learning through cartesian product of ranking references for image retrieval tasks. In: IEEE. **2016 29th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)**. [S.l.], 2016. p. 249–256.
- VALEM, L. P.; PEDRONETTE, D. C. G. Unsupervised distance learning by reciprocal knn distance for image retrieval. **ACM SIGMultimedia Records**, ACM New York, NY, USA, v. 9, n. 2, p. 1–1, 2017.
- WANG, J. et al. Scalable k-nn graph construction for visual descriptors. In: **Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2012. p. 1106–1113.
- WANG, K. et al. A comprehensive survey on cross-modal retrieval. **arXiv preprint arXiv:1607.06215**, 2016.

ZHOU, D. et al. Ranking on data manifolds. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2003. p. 169–176.

APPENDIX A – REPRODUCIBILITY AND DATA AVAILABILITY

To ensure full transparency and facilitate the reproduction of the experimental results presented in this work, the complete software artifact, raw data, and visualization scripts have been made publicly available.

The framework is open-source and hosted on GitHub. Researchers and examiners are encouraged to clone the repository and follow the comprehensive instructions detailed in the project’s `README.md` file.

```
# 1. Clone the repository
git clone https://github.com/andreTomi/UDLF-CSR.git
cd UDLF-CSR

# 2. Compile the custom C++ sparse engine
cd Source/
make
cd ..

# 3. Setup the Python environment and validate the system
pip install -r Test/requirements.txt
python Test/src/validate_setup.py

# 4. Run the automated benchmarking suite
python Test/src/run_experiments.py --output-dir my_benchmark_run

# 5. Automatically generate all PDF plots and LaTeX tables
cd Visualization/
python visualizer.py
```

A.1 SOURCE CODE AND DATASET AVAILABILITY

The core sparse engine implementation, including the `CSRMatrix` structure, the custom `BlockAllocator`, and the UDLF algorithm ports, is released under the MIT License to encourage wider academic and industrial adoption.

Furthermore, all raw benchmark data, hardware counters, and execution metrics used to generate the figures in this thesis are included in the repository’s `Data/` directory, ensuring that the claims of memory reduction and execution speedups can be seamlessly verified.

DADOS CURRICULARES

IDENTIFICAÇÃO	
Nome	André Tomitan Bocces Nascimento: 15/10/2001 — Rio Claro, SP
Nacionalidade	Brasileira
Nome em citações bibliográficas	BOCCES, A. T. BOCCES, André Tomitan
Currículo Lattes	http://lattes.cnpq.br/8036275318856170
ORCID	https://orcid.org/0009-0003-5331-9234
FORMAÇÃO ACADÊMICA	
2020 – 2023	Bacharelado em Ciência da Computação <i>Universidade Estadual Paulista (UNESP) — Rio Claro, SP</i>
2024 – 2026	Mestrado Acadêmico em Ciência da Computação <i>Universidade Estadual Paulista (UNESP) — Rio Claro, SP</i>
PRODUÇÃO BIBLIOGRÁFICA	
BOCCES, A. T.; BALDASSIN, A. J. Performance and scalability optimization in information retrieval systems: a survey. <i>In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP), 2025. Anais [...]. 2025. (Artigo completo).</i>	
BOCCES, A. T.; BALDASSIN, A. J. Performance and scalability optimization in information retrieval systems: a systematic review. IEEE Access , v. 14, p. 2576–2591, 2025.	
ATIVIDADES DE PESQUISA	
De 2024 a 2026, atuou como Bolsista de Mestrado financiado pela <i>Fundação para o Desenvolvimento da Unesp</i> (FUNDUNESP — Processo nº 3419/2023) junto ao Departamento de Ciências da Computação e Estatística da UNESP. Desenvolveu atividades de pesquisa focadas em sistemas de alto desempenho (<i>High-Performance Computing — HPC</i>), otimização de primitivas de álgebra linear esparsa e algoritmos baseados em grafos aplicados à Recuperação de Imagens Baseada em Conteúdo (CBIR).	