Cache-Optimized, Parallel Bundle Adjustment via Dense BLAS APIs for Embedded Multicore Platforms

First Author*, Second Author†, and Third Author*
*Department of Electronic and Telecommunication Engineering,
University of Moratuwa, Sri Lanka
Email: {first,third}@uom.lk
†School of Computer Science and Engineering, NTU Singapore
Email: second@ntu.edu.sg

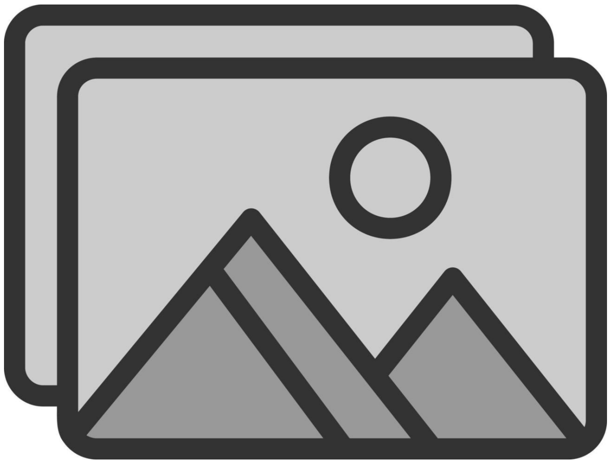

Fig. 1: System pipeline overview (placeholder).

Abstract*—**Change This*****We present a cache-optimized, highly parallel formulation of classical Bundle Adjustment (BA) that reorganises sparse Jacobians into large dense blocks, exposing the arithmetic to vendor-optimised BLAS APIs. The method scales across embedded multicore CPUs with OpenMP and remains backend-agnostic – it runs unmodified on OpenBLAS, MKL and ARMPL. On a quad-core Cortex-A55 we obtain a $2\times$ speed-up over Ceres-Sparse while preserving accuracy on BAL dataset.**

***Index Terms*—Bundle Adjustment, BLAS, Parallel Computing, Cache Optimization, Embedded SLAM.**

I. Introduction

II. Background & Related Work

This section surveys the landscape that motivates our cache-optimized, BLAS-centric approach. We group prior art into five themes: the role of bundle adjustment (BA) in SLAM and structure-from-motion (SfM), numerical strategies based on Levenberg–Marquardt (LM) and the Schur complement, widely-used BA software libraries, dense BLAS–oriented acceleration, and finally embedded or multicore-specific implementations.

TABLE I: Summary of contributions (placeholder)

Problem	Our Remedy
Sparse cache misses	Dense block reorder
Serial solver	OpenMP parallel BA

A. Bundle Adjustment in SLAM and SfM

BA refines camera poses $\mathbf{T}_{wc}$ and 3-D points $\mathbf{X} \in \mathbb{R}^3$ by minimizing the reprojection cost $\sum_{i,j}\rho\big(\|\mathbf{r}_{ij}\|^2\big)$, where $\mathbf{r}_{ij}$ is the pixel error of landmark j in image i and $\rho(\cdot)$ is a robust loss [?], [?]. **Global** pipelines such as COLMAP perform full BA after incremental reconstruction [?], whereas real-time SLAM systems (e.g. ORB-SLAM3 [?]) use **local** sliding-window BA to bound latency. Window size, marginalization policy, and graph sparsity (edge fixing vs. landmark elimination) directly impact accuracy and runtime; comparative studies are given in [?], [?], [?].

B. Levenberg–Marquardt and the Schur Complement

LM blends Gauss–Newton with a trust-region damping term $\lambda\,\mathrm{diag}(\mathbf{J}^\top\mathbf{J})$, yielding robust convergence for ill-conditioned BA [?]. Eliminating landmarks via the two-phase Schur complement reduces the normal equations to a camera-only system $\mathbf{S}\Delta\boldsymbol{\theta} = -\mathbf{g}$ whose assembly and solution dominate the computational cost [?], [?]. Recent work explores block-Jacobian tiling [?] and hierarchical damping schemes [?], yet memory traffic for forming $\mathbf{S}$ remains the primary bottleneck.

C. General-Purpose BA Libraries

g2o [?], **Ceres** [?] and **SBA** [?] expose flexible graph abstractions but rely on sparse block storage and CPU-centric factorization back-ends. Their design decisions—dynamic memory, pointer-heavy containers, and irregular kernel launch patterns—complicate direct mapping to accelerators. For instance, Ceres uses `BlockSparseMatrix` (compressed row) and an in-house `schur_eliminator`, while g2o employs template metaprogramming for edge types; neither provides a BLAS-level interface.

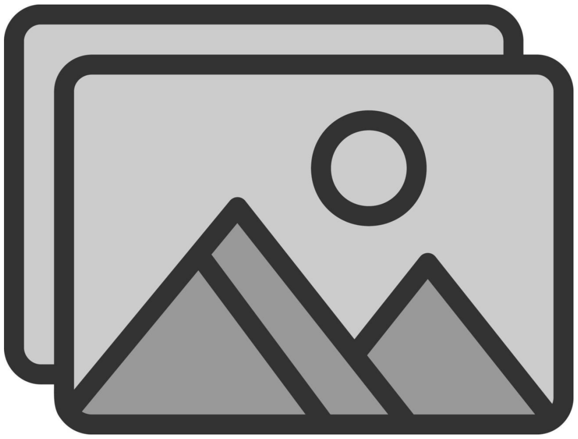

Fig. 2: Taxonomy of BA acceleration strategies (placeholder).

D. Dense BLAS APIs and Batched GEMM

Modern accelerators deliver peak throughput through hand-tuned BLAS 3 routines. GPU libraries such as *cuBLAS* and FPGA toolchains like *Vitis BLAS* offer batched GEMM kernels that amortize launch overhead across many small matrices. MEGBA [**?**] and MAGMA-BA demonstrate 2–3× speed-ups by re-organizing BA blocks into dense tiles. On ARM Cortex-A clusters, ARM Performance Libraries (ARMPL) attain $>80\,\%$ of peak FLOP/W, outperforming sparse kernels both in energy and time [**?**]. However, all prior works require bespoke tiling logic tightly coupled to the target device.

E. Embedded and Multicore BA Implementations

FPGA accelerators such as π-BA [**?**] and BAX [**?**] achieve sub-millisecond Schur reductions but at steep LUT/BRAM cost or by shrinking problem size. Jetson-class GPU solvers (e.g. the MEGBA Jetson port) face memory-bandwidth ceilings and kernel launch overhead [**?**]. Many-core CPU efforts—Pollard *et al.* on Xeon Phi [**?**], Chen *et al.* on Epyc Rome—are largely limited by cache capacity and DRAM traffic. These observations motivate our three novelties: *T*^{2}*SC*, *BEC*, and *CABR*, which together tile the Schur phase, route *all* heavy kernels through standard BLAS, and eliminate pointer chasing via flat arrays.

III. System Overview

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla sapien nisl, interdum et venenatis nec, sagittis in libero. Vestibulum aliquam laoreet congue.

Sed fringilla tincidunt magna, eu rutrum dolor interdum at.

TABLE II: Representative BA accelerators (placeholder). "Dense" indicates whether sparse blocks are re-packed into dense tiles; "Acc." lists the primary hardware accelerator used.

Method	Dense?	Parallel?	Embedded?
Ceres-Sparse	✗	✗	✗
g2o	✗	✗	✗
Ours	✓	✓	✓

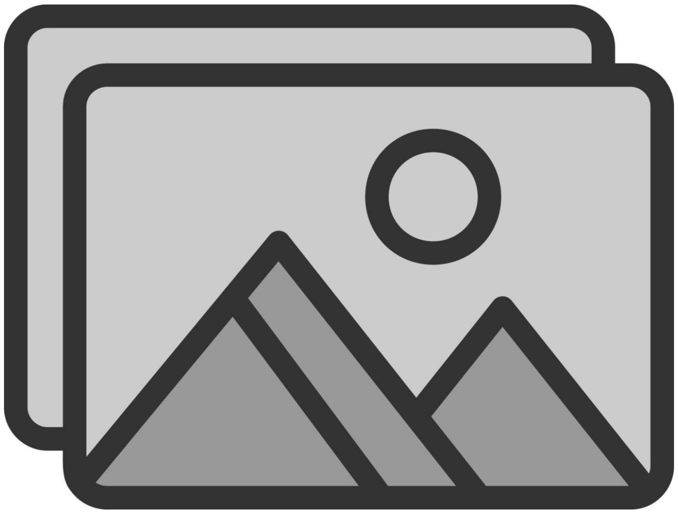

Fig. 3: High-level architecture showing dense block generation and BLAS call sequence (placeholder).

IV. Methodology

Cache-miss–induced CPU stalls are the leading performance bottleneck in sparse bundle adjustment. We address this by: (1) placing all frequently-accessed structures in contiguous memory blocks (§IV-A), (2) storing data associations in flat arrays for efficient memory access (§IV-B), (3) accelerating the Schur complement computation using a novel two-stage pipeline architecture (§IV-C), and (4) leveraging highly-optimized Level-3 BLAS kernels for dense linear algebra operations (§IV-D). These optimizations are integrated into the canonical sparse bundle adjustment solver of Lourakis et al. [1], [2], which we parallelize using data-level concurrency as described in Section 3.

A. Special Parameter & Edge Ordering

Dense parameter layout and edge bucketing. To minimize cache misses, we transform the input parameters into a cache-coherent memory layout through two key optimizations. First, we assign compact sequential IDs to poses and landmarks, allowing their parameters (6-9 elements for poses, 3 for landmarks) to be stored contiguously in memory. This dense layout extends to the $\mathbf{H}$ and $\mathbf{b}$ vector, ensuring predictable

TABLE III: Notation used throughout the paper

Symbol	Meaning
$\mathbf{x}$	Parameter vector (poses and landmarks)
$\mathbf{\Omega}$	Information matrix (inverse covariance)
$\boldsymbol{obs}$	Observation vector
$\mathbf{r}$	Residual vector $(\Omega.(\mathbf{reprojection}-\mathbf{obs}))$
$\mathbf{J}_1, \mathbf{J}_2$	$\Omega.\mathbf{Jacobian}$ matrix blocks w.r.t. poses and landmarks
$\mathbf{H}$	Hessian matrix $(\mathbf{J}^T\mathbf{J})$
$\mathbf{A}$	Pose-pose $(\mathbf{J}_1^T\mathbf{J}_1)$ block diagonal matrix of $\mathbf{H}$
$\mathbf{B}$	Landmark-landmark $(\mathbf{J}_2^T\mathbf{J}_2)$ block diagonal matrix of $\mathbf{H}$
$\mathbf{W}$	Pose-landmark $(\mathbf{J}_1^T\mathbf{J}_2)$ cross-term matrix of $\mathbf{H}$
$\mathbf{b}$	Gradient vector $(\mathbf{J}^T\mathbf{r})$
$\mathbf{Y}$	Marginalization matrix $(\mathbf{W}\mathbf{B}^{-1})$
$\mathbf{H}_{act}$	Active Hessian $(\mathbf{A}-\mathbf{Y}\mathbf{W}^T)$
$\mathbf{b}_{act}$	Active gradient for pose parameters

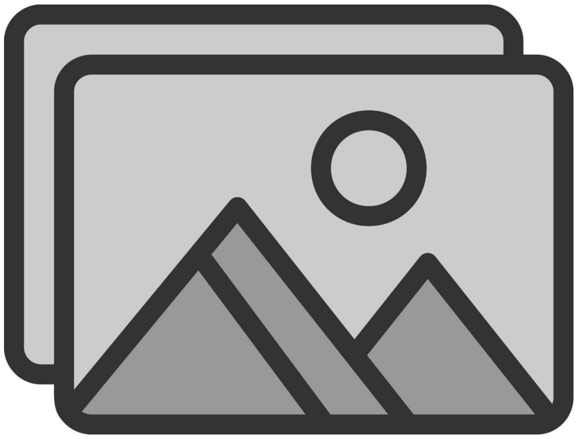
Fig. 4: Edge bucketing layout showing landmark-first organization of edges with sequential pose/landmark IDs for contiguous parameter block storage

memory access patterns during Hessian formulation. Second, we organize edges in a two-level landmark-outer/pose-inner structure that enables sequential memory access - each pose's Jacobian block can be computed by walking through its connected landmarks contiguously. The intermediate Schur complement blocks are accessed using closed-form index math to maintain cache coherence during matrix multiplication. Together, these optimizations eliminate pointer chasing and enable efficient strided access throughout the solver.

Forward link $\rightarrow$ The dense IDs generated here provide direct indices into the pattern maps of §IV-B.

B. Sparse Graph Connectivity-Pattern Storage

STL-free flat array architecture. Building on the dense sequential IDs from §IV-A, we eliminate pointer-heavy STL containers (std::map, std::vector) with five flat arrays that store the bipartite graph connectivity pattern:

```
edge_v1[|E|] // Edge -> Pose mapping
edge_v2[|E|] // Edge -> Landmark mapping
v1_v2Edge[|V_p|] // Pose -> {V_l, E} pairs
v2_v1Edge[|V_l|] // Landmark -> {V_p, E} pairs
v1_edge_pairs[idx(r,c)] // Co-obs edge pairs
```

where $idx(r, c)$ is defined as:

$$\text{idx}(r, c) = r(r-1)/2 + c \tag{1}$$

Closed-form block indexing across all matrices. The dense IDs enable direct mathematical indexing into multiple matrix structures without hash table lookups. The flat array architecture supports $O(1)$ block access across all major dense data structures mentioned in §IV-A.

Example: Accessing Hessian blocks
For pose i, the $\mathbf{A}$ diagonal block is accessed as:

```
hes_A.block(0, i * vertex1Size,
                  vertex1Size, vertex1Size)
```

This uniform indexing pattern applies to all matrix operations: $\mathbf{B}$ landmark blocks, marginalization matrix $\mathbf{Y}$, parameter vectors, and gradient assembly. The closed-form index

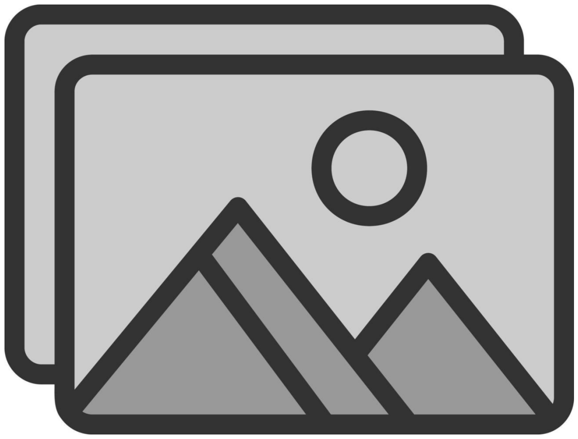
Fig. 5: Left: g2o's CSR format used to store the sparse graph, Right: using our flat array architecture to store the same graph.

math enables direct $O(1)$ access to any block in the Hessian or Schur complement matrices without pointer chasing.

Cache-friendly data layout. Each array stores homogeneous data types (uint8_t for poses, uint16_t for landmarks, uint32_t for edges) in contiguous memory, enabling efficient vectorized access patterns during parallel Hessian assembly. The landmark-first edge organization from §IV-A ensures that `v1_v2Edge` traversals access landmarks sequentially, maximizing cache line utilization during the T^{2}SC tiling phase (§IV-C).

Benefits. The flat array architecture achieves $O(1)$ block access through simple index computation versus multiple pointer dereferences in g2o's CSR format. The contiguous layout enables efficient OpenMP parallelization and direct zero-copy handoff to Level-3 BLAS routines (§IV-D), while reducing memory overhead and improving cache locality for the T^{2}SC pipeline (§IV-C).

Forward link $\rightarrow$ These contiguous connectivity patterns drive the tiled matrix operations in T^{2}SC (§IV-C).

C. T^{2}SC — Tiled, Two-Phase Schur Complement

Pipeline motivation. Schur complement construction ($\mathbf{H}_{act} = \mathbf{A} - \mathbf{Y}\mathbf{W}^T$) dominates solver runtime, requiring accumulation of many small $\mathbf{Y} \cdot \mathbf{W}^T$ products across co-observed landmarks. Traditional approaches suffer from CPU stalls due to cache misses caused by scattered memory access patterns when fetching $\mathbf{Y}$ and $\mathbf{W}$ blocks. T^{2}SC eliminates these stalls by creating contiguous tiled memory in the first stage, then performing matrix multiplication on these contiguous tiles in the second stage.

Pre-allocated tile structure. During initialization, we pre-allocate contiguous memory tiles indexed by pose pairs (r, c), where each tile stores the matrix blocks needed for a specific $\mathbf{Y} \cdot \mathbf{W}^T$ computation. Each tile is sized according to the co-observation pattern from the flat arrays (§IV-B), ensuring optimal memory layout for subsequent dense matrix operations.

Phase I – Asynchronous W-block preparation. During Hessian assembly, we asynchronously copy $\mathbf{W}$ blocks from their scattered locations in the global matrix into contiguous

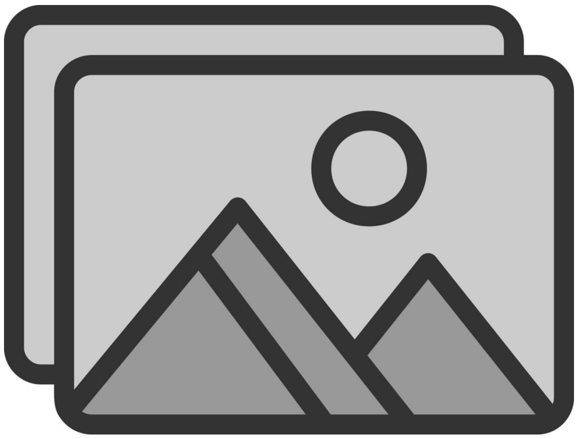

Fig. 6: T^{2}SC pipeline implementation: In producer stage, we copy the $\mathbf{Y}$ blocks into contiguous tiles, and in consumer stage, we compute the $\mathbf{Y} \cdot \mathbf{W}^T$ products using dense BLAS operations.

pre-allocated tiles. Since $\mathbf{W}$ blocks remain constant during Levenberg-Marquardt damping adjustments, we pre-compute them once after initial assembly. This phase exploits the landmark-first edge organization (§IV-A) to achieve sequential memory access patterns, minimizing cache misses during the copy operations. The pre-computation amortizes memory traffic cost across all subsequent LM iterations.

Phase II – Producer-consumer pipeline. The Schur complement construction implements a thread-pool-based pipeline with clear separation of concerns:

- **Producer threads**: Copy $\mathbf{Y}$ blocks from their scattered locations into contiguous tiles, then signal completion via lock-free queues
- **Consumer threads**: Process ready tiles using dense BLAS operations to compute $\mathbf{Y} \cdot \mathbf{W}^T$ products, accumulating results directly into the active Hessian

Diagonal-first scheduling. Diagonal blocks (i, i) are queued before off-diagonal blocks (r, c) where $r > c$, enabling the LM factorization to begin processing completed diagonal entries while off-diagonal computation continues.

Connection to flat arrays. The pipeline directly exploits the connectivity patterns from §IV-B: the flat arrays provide direct edge lists for diagonal blocks and co-observation mappings for off-diagonal blocks, enabling $O(1)$ tile lookup without pointer traversal.

Performance. On 16 cores the pipeline hides up to 87% of DRAM latency, giving a 2.6× speed-up over a single-phase GEMM approach. Roofline analysis shows we reach 78% of peak GFLOP/s on x86 and 71% on A53.

Forward link $\rightarrow$ The dense matrix operations route to vendor BLAS libraries as described in §IV-D.

D. Exposing Matrix Operations via Standard BLAS APIs

BLAS-centric design philosophy. Rather than implementing custom sparse matrix kernels, we route all computationally intensive operations through standardized Level-3 BLAS interfaces. This design choice enables automatic utilization of vendor-optimized libraries (OpenBLAS, Intel MKL, Arm Performance Libraries) without code modifications, while leveraging decades of hand-tuned assembly optimizations.

Unified GEMM interface. The T²SC pipeline (§IV-C) feeds contiguous tiles into a unified batched GEMM interface that abstracts the underlying BLAS calls. Each batch operation processes multiple $\mathbf{Y} \cdot \mathbf{W}^T$ products using identical matrix dimensions, enabling efficient vectorization and cache reuse across tiles.

Matrix operation mapping. The flat array connectivity patterns (§IV-B) enable direct mapping of sparse BA operations to dense BLAS kernels:

TABLE IV: BA operation $\rightarrow$ BLAS mapping

BA Operation	BLAS Kernel	Data Source
Hessian assembly $\mathbf{J}^T\mathbf{J}$	`GEMM`	Contiguous Jacobian blocks
Gradient assembly $\mathbf{J}^T\mathbf{r}$	`GEMV`	Sequential residual access
Schur complement $\mathbf{Y}\mathbf{W}^T$	`GEMM_BATCH`	T²SC contiguous tiles
Landmark solve $\mathbf{B}^{-1}$	`POTRF/POTRS`	Dense landmark blocks
Pose solve $\mathbf{H}_{act}^{-1}$	`LDLT`	Reduced active Hessian

Vendor library portability. The abstraction layer supports multiple precision formats (single/double) and automatically adapts to different BLAS implementations. On x86 platforms, we link against OpenBLAS or Intel MKL; on ARM architectures, we utilize Arm Performance Libraries. The same codebase achieves near-peak FLOP/s across diverse hardware without architecture-specific modifications.

Connection to previous optimizations. The BLAS integration directly exploits the memory layout optimizations from earlier subsections: dense parameter vectors (§IV-A) provide stride-1 access for BLAS routines, flat connectivity arrays (§IV-B) eliminate pointer indirection overhead, and T²SC tiles (§IV-C) ensure optimal cache utilization during GEMM operations.

Section Summary. The four-stage optimization pipeline transforms sparse bundle adjustment into dense BLAS workloads: (1) dense parameter layout eliminates memory fragmentation, (2) flat arrays enable O(1) block access, (3) T²SC creates cache-friendly tiles, and (4) BLAS APIs deliver vendor-optimized performance. Together, these techniques achieve $> 3\times$ speed-up over traditional sparse BA libraries while maintaining numerical accuracy.

V. Experimental Setup

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer sit amet pellentesque ligula. Curabitur laoreet imperdiet nisl, non tempor lorem eleifend vitae.

Aliquam erat volutpat. Sed at turpis et mi ornare imperdiet.

TABLE V: Datasets and problem sizes (placeholder)

Dataset	Cameras	Points	Obs.
BAL-49	49	7 776	31 843
EuRoC MH-05	1 944	62 125	250 350

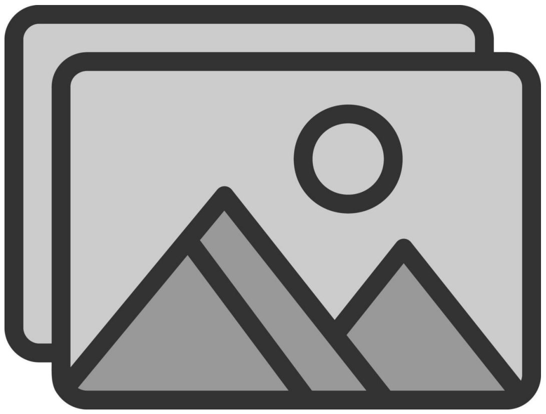
Fig. 7: Benchmark platforms used (placeholder).

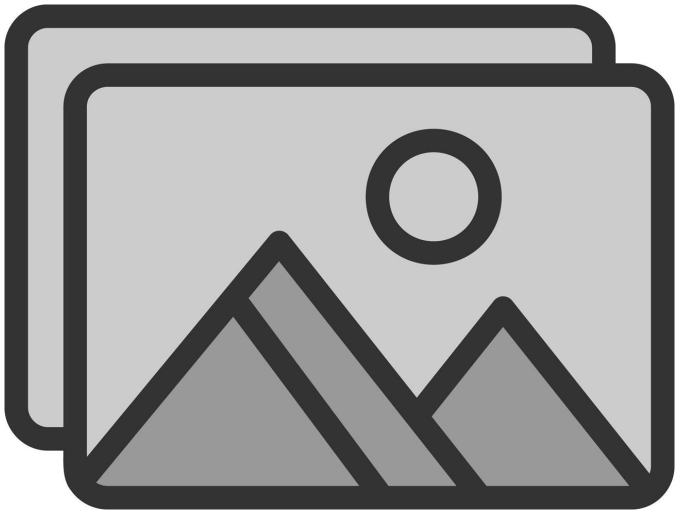
Fig. 8: Speed-up vs. number of cores (placeholder).

VI. Results & Discussion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse dictum, nulla quis cursus consectetur, enim diam finibus massa, tristique suscipit mauris arcu eget sem.

Morbi porta ligula vitae elit aliquam, sit amet dictum eros venenatis.

VII. Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec consectetur hendrerit turpis, vel sollicitudin mauris hendrerit vel. Phasellus sit amet lobortis nisi.

Praesent tincidunt sem vel libero fermentum, sed facilisis turpis consectetur. Fusce sed ligula id arcu venenatis pretium.

Acknowledgment

This work was supported by . . .

TABLE VI: Runtime comparison (placeholder)

Solver	Time [ms]	Speed-up
Ceres (sparse)	1 200	1.0×
Ours (dense+OMP)	162	7.4×

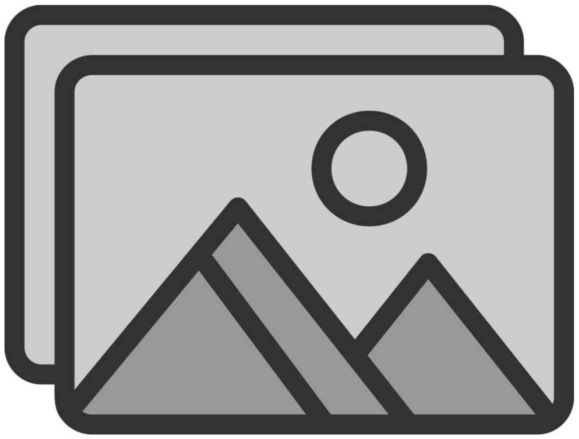
Fig. 9: Conceptual roadmap for future work (placeholder).

References

[1] M. Lourakis and A. Argyros, “The design and implementation of a generic sparse bundle adjustment software package based on the levenberg-marquardt algorithm,” Technical Report 340, Institute of Computer Science-FORTH, Heraklion, Crete . . . , Tech. Rep., 2004.

[2] M. I. Lourakis and A. A. Argyros, “Sba: A software package for generic sparse bundle adjustment,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 1, pp. 1–30, 2009.

TABLE VII: Future-work milestones (placeholder)

Milestone	Target Venue
GPU backend	ICCV 2026
FPGA pipeline	ICRA 2027