Accelerated Bundle Adjustment for 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

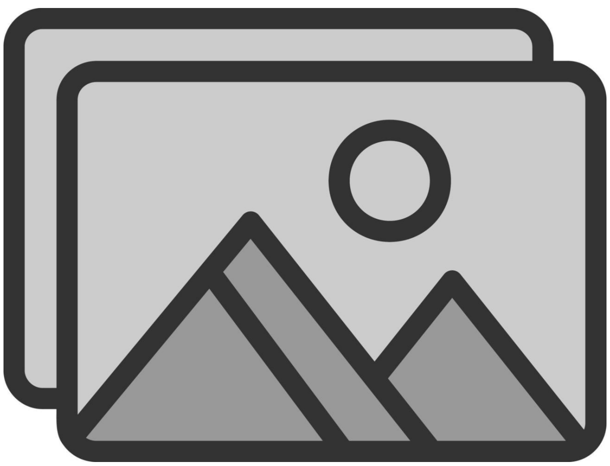

Fig. 1: System pipeline overview (placeholder).

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

Visual SLAM and structure-from-motion (SfM) are fundamental technologies in robotics, mixed reality, and autonomous navigation that reconstruct camera motion and 3D structure from image sequences [1], [2]. These systems consist of three main components: a front-end that handles visual odometry and feature tracking, a back-end that performs optimization through bundle adjustment (BA), and a loop closing module that detects revisited places and corrects accumulated drift [3].

Recent advances have largely solved front-end performance challenges. Modern feature extractors like ORB running on SIMD [4], FPGA-accelerated variants achieving over 1,200 FPS at just 2.1 W [5], and efficient CNN-based detectors like MobileSP running in 4 ms on embedded GPUs [6] now deliver real-time performance even on edge devices.

This progress has shifted the computational bottleneck to the BA back-end, where camera poses and 3D landmarks are jointly refined through nonlinear optimization. On embedded platforms ranging from quad-core Cortex-A53 to automotive-grade octa-core SoCs, BA now consumes 55 % to 70 % of total SLAM runtime, even with modest sliding windows of 10-15 keyframes [7]. The key challenge lies not in raw computational intensity, but rather in inefficient memory access patterns. Specifically, the pointer-heavy sparse data structures and naive sparse matrix implementations used in existing frameworks lead to frequent cache misses and poor data locality. This results in hundreds of milliseconds of stalled CPU pipeline time [8].

Acceleration attempts: Prior work has explored three main approaches:

(i) GPUs While solutions like PBA [9] and MegBA [10] achieve impressive 500 MFLOPS/W efficiency on desktop GPUs [11], their power requirements exceed mobile robot constraints.

(ii) Dedicated hardware FPGA/ASIC accelerators including π-BA [8], BAX [12], BOHA [13] and Archytas [14] enable millisecond-level Schur complement computation. However, they demand substantial on-chip memory (π-BA uses 141.5 BRAMs) or impose strict limits on problem size (BAX: 16 poses, 256 landmarks maximum).

(iii) Multicore CPUs Modern solvers like G2O [15] and CERES [16] employ CSR/CSC storage with parallel processing libraries. While effective on desktop CPUs, their pointer-heavy design causes frequent cache misses on embedded systems. Simpler alternatives like SBA/Levmar [17] run $\approx 2\times$ slower than CERES and face similar memory access inefficiencies.

Key Insight: These observations motivate a fundamental rethink: Can we transform sparse BA into cache-friendly dense operations while maintaining the flexibility of CPU-based execution? Our approach achieves this by converting irregular memory accesses into predictable, BLAS-optimized matrix operations.

We present a cache-optimized BA solver designed from scratch for multicore embedded and desktop CPUs that transforms sparse BA into batched dense GEMMs through four key innovations: (1) *dense parameter layout and landmark-first edge bucketing* that eliminates pointer chasing, (2) *flat connectivity arrays* enabling $\mathcal{O}(1)$ block access without hash tables, (3) a *tiled two-phase Schur complement* (T^2SC) that overlaps memory staging with batched BLAS, and (4) a *unified BLAS interface* that maps all heavy kernels to Level-3 routines for vendor library compatibility.

Our main contributions are:

- A dense local-ID scheme and landmark-first edge bucketing that pack parameters contiguously and eliminate pointer chasing during Jacobian assembly (§IV-A);
- Flat connectivity arrays with closed-form indexing enabling $\mathcal{O}(1)$ block lookup across all matrix structures without STL containers (§IV-B);
- **T^2SC**: a producer-consumer tiled Schur pipeline that stages $\mathbf{Y}$ and $\mathbf{W}$ blocks into contiguous tiles for batched GEMM, reducing Schur build time by up to 12.9× (§IV-C);
- Complete mapping of BA operations ($\mathbf{J}^\top\mathbf{J}$, $\mathbf{Y}\mathbf{W}^\top$, landmark solves) to Level-3 BLAS kernels, enabling drop-in use of OpenBLAS, MKL, or Arm Performance Libraries (§IV-D);
- Comprehensive evaluation on BAL datasets showing 1.2–2.7× speedup on Cortex-A53 and 2.3–8.8× on Core i9-9900K while maintaining accuracy within 0.52% of optimal convergence (§VI).

II. Background and Related Work

This section revisits the classical bottlenecks of bundle adjustment (BA) and surveys three principal acceleration directions—CPU, GPU, and custom hardware—highlighting why a *dense-BLAS, cache-aware CPU design* remains a compelling sweet-spot for embedded robotics.

A. Bundle Adjustment Bottlenecks and the Schur Complement

BA refines camera poses and 3-D landmarks by minimizing re-projection error via damped Gauss–Newton or Levenberg–Marquardt (LM) iterations [18], [2]. The normal equations induce a bipartite block Hessian:

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & \mathbf{W} \\ \mathbf{W}^T & \mathbf{B} \end{bmatrix} \tag{1}$$

where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{W}$ encode pose–pose, landmark–landmark, and cross interactions. Early software such as SBA [17] constructed the full Hessian and used dense/banded factorisation, quickly becoming memory-bound for thousands of landmarks.

The **Schur complement** eliminates landmarks analytically [19], forming the reduced camera system:

$$(\mathbf{A} - \mathbf{W}\mathbf{B}^{-1}\mathbf{W}^T)\boldsymbol{\delta}_{\text{poses}} = \mathbf{b}_{\text{poses}} - \mathbf{W}\mathbf{B}^{-1}\mathbf{b}_{\text{landmarks}} \tag{2}$$

Although this reduces factorisation cost from $\mathcal{O}((n_p + n_l)^3)$ to $\mathcal{O}(n_p^3)$, constructing $\mathbf{H}_{\text{act}} = \mathbf{A} - \mathbf{W}\mathbf{B}^{-1}\mathbf{W}^T$ now dominates runtime (often ≥60%) because it requires per-landmark inversions and irregular block accumulations that thrash caches—especially on small L2/L3 embedded CPUs.

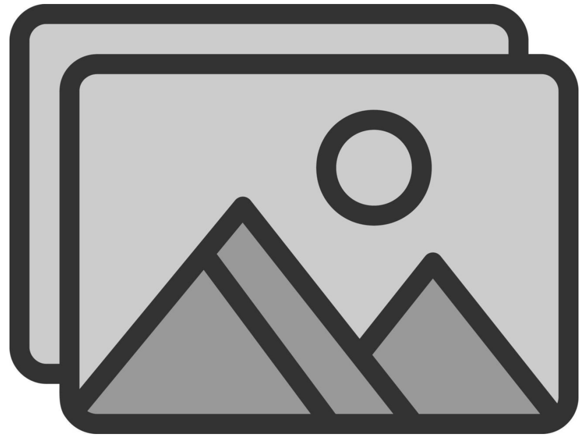

Fig. 2: Taxonomy of BA acceleration approaches: (1) CPU solvers like g2o/Ceres using CSR formats and scatter-gather operations, (2) GPU solutions like PBA/MegBA leveraging thousands of threads but requiring high power, (3) FPGA/ASIC accelerators like π-BA/BAX achieving millisecond latency with resource constraints, and (4) our dense BLAS approach enabling portable acceleration via vendor-optimized libraries.

B. Acceleration Approaches

CPU-based Solvers: Widely-used libraries like G2O [15] and CERES [16] implement the Schur complement using compressed sparse row (CSR) formats with scatter-gather operations that suffer from irregular memory access. While multithreaded Jacobian assembly and block-aligned storage [9] improve locality and demonstrate 2-3× speedups on embedded ARM processors, existing solvers still spend over 50% of runtime in memory-bound operations with frequent cache misses, especially when scaling beyond one NUMA socket.

GPU Approaches: Solutions like PBA [9] and MegBA [10] parallelize Schur complement computation across thousands of GPU threads, achieving impressive 10-30× speedups over single-threaded CPU implementations. However, their high power requirements (≥200W) and PCIe transfer overheads make them unsuitable for mobile robotics applications.

Dedicated Hardware: FPGA/ASIC accelerators including π-BA [8] and BAX [12] achieve millisecond-level Schur complement computation by streaming operations fully on-chip. While this enables significant energy savings, these designs face flexibility constraints - requiring substantial hardware resources (141.5 BRAMs for π-BA, nearly exhausting the 144 BRAMs available on modern K26 SoCs) or imposing strict problem size limits (16 poses, 256 landmarks for BAX).

C. BLAS APIs for Hardware Acceleration

Modern processors and accelerators expose optimized matrix operations through standardized BLAS APIs [20], enabling portable access to vendor-tuned kernels like batched `GEMM` and `SYRK`. Key implementations include Intel oneMKL, Arm Performance Libraries (ArmPL), OpenBLAS, NVIDIA cuBLAS, and Xilinx Vitis BLAS [21], [22], [23], [24], [25].

These libraries deliver 80–95% of peak FLOPS for BA-relevant matrix sizes (32–96 blocks) by leveraging specialized instructions: oneMKL uses AMX tiles on Xeon, ArmPL employs SVE2 on Cortex-A, and cuBLAS dispatches Tensor Cores via `cublasGemmEx`. However, existing BA solvers fail to exploit this potential due to irregular sparse operations that bypass Level-3 BLAS entirely. Our approach systematically maps the complete Schur complement pipeline to dense BLAS calls, achieving portable acceleration across all these platforms.

Despite these advances, existing BA solvers remain memory-bound due to pointer-based sparse structures and irregular access patterns that bypass Level-3 BLAS optimizations entirely. Our approach addresses this fundamental limitation through comprehensive redesign of the Schur complement pipeline using dense parameter ordering (§IV-A), flat connectivity arrays (§IV-B), T²SC tiling (§IV-C), and complete BLAS mapping (§IV-D) to achieve cache-friendly, portable acceleration across CPU architectures.

III. System Overview

This section offers a birds-eye view of our cache-optimized bundle–adjustment (BA) pipeline, tracing the path from raw vision data to dense BLAS calls. Full implementation details follow in §IV; here we focus on the high-level ideas and data flow.

A. Problem Formulation

Bundle Adjustment maintains two vertex sets: camera poses (Type–1, 6 or 9 parameters each) and 3-D landmarks (Type–2, 3 parameters each). Pixel observations link one pose to one landmark and become edges in a bipartite graph. The cost function is the weighted reprojection error, minimized with the damped Levenberg–Marquardt (LM) scheme summarized in Algorithm 1.

B. Data Ingestion & Pre-processing

Vertex-first loading. The dataset loader first emits a contiguous array of pose parameters and then the landmark array; only afterwards is the observation list streamed. This order enables the dense local-ID mapping and landmark-first edge bucketing introduced in §IV-A.

Reprojection plug-in. Before calling `initialize()`, the solver is bound to a user-supplied reprojection functor that maps $(\text{pose}, \text{landmark}) \to$ pixel. Swapping camera models therefore requires only exchanging this functor.

C. Iterative Optimization Pipeline

Per outer LM iteration we: (1) build Jacobians in parallel over edges, (2) aggregate dense $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{W}$ blocks, (3) construct the reduced pose–pose system with the tiled two-phase Schur complement (T^{2}SC), (4) solve the dense system via vendor BLAS (Cholesky or LDLT) and (5) update parameters and damping. Convergence is declared when either the infinity-norm of the gradient or the update step falls below user thresholds, or after ten LM iterations to match real-time SLAM budgets.

Algorithm 1 High-Level SBA-Style Levenberg–Marquardt for Bundle Adjustment

Require: Measurement vector $\mathbf{x}$, initial parameters $\mathbf{p}_0$, tolerance $\varepsilon_{\mathrm{g}}, \varepsilon_{\mathrm{p}}$, damping seed τ, iteration cap $k_{\max}$
Ensure: Optimised parameters $\mathbf{p}^{+}$

$k \leftarrow 0$, $\nu \leftarrow 2$, $\mathbf{p} \leftarrow \mathbf{p}_0$
Compute $\mathbf{J}$, $\boldsymbol{\varepsilon} = \mathbf{x} - f(\mathbf{p})$, $\mathbf{g} = \mathbf{J}^T\boldsymbol{\varepsilon}$, $\mathbf{A} = \mathbf{J}^T\mathbf{J}$
$\mu \leftarrow \tau \cdot \max_i A_{ii}$; *stop* $\leftarrow (\|\mathbf{g}\|_\infty \le \varepsilon_{\mathrm{g}})$
while not *stop* **and** $k < k_{\max}$ **do**
  $k \leftarrow k + 1$
  repeat
    Solve $(\mathbf{A} + \mu\mathbf{I})\,\delta\mathbf{p} = \mathbf{g}$ ▷ Damped normal eqn.
    if $\|\delta\mathbf{p}\| \le \varepsilon_{\mathrm{p}}\,(\|\mathbf{p}\| + \varepsilon_{\mathrm{p}})$ **then**
      stop $\leftarrow$ **true**
    else
      $\mathbf{p}_{\text{new}} \leftarrow \mathbf{p} + \delta\mathbf{p}$
      $\rho \leftarrow \dfrac{\|\boldsymbol{\varepsilon}\|^2 - \|\mathbf{x} - f(\mathbf{p}_{\text{new}})\|^2}{\delta\mathbf{p}^T(\mu\,\delta\mathbf{p} + \mathbf{g})}$
      if $\rho > 0$ **then**
        $\mathbf{p} \leftarrow \mathbf{p}_{\text{new}}$; Recompute $\mathbf{J}, \boldsymbol{\varepsilon}, \mathbf{g}, \mathbf{A}$
        stop $\leftarrow (\|\mathbf{g}\|_\infty \le \varepsilon_{\mathrm{g}})$
        $\mu \leftarrow \mu\,\max\big(\tfrac{1}{3}, \min\big(\tfrac{2}{3}, 1 - (2\rho - 1)^3\big)\big)$
        $\nu \leftarrow 2$
      else
        $\mu \leftarrow \mu\,\nu$; $\nu \leftarrow 2\,\nu$
      end if
    end if
  until $\rho > 0$ **or** *stop*
end while
return $\mathbf{p}$

D. Memory & Parallel Strategy

All hot data structures live in contiguous memory: a structure-of-arrays parameter vector and five flat connectivity maps. Producer–consumer tiling keeps Schur tiles resident in L2/L3 cache, while every heavy kernel maps cleanly to Level-3 BLAS. This design yields a $\approx 3\times$ wall-clock speed-up over g2o and consistent per-iteration gains on both a 16-core x86-64 and a quad-core Cortex-A53 platform.

E. Complexity Note

Each LM iteration costs $\mathcal{O}(|E|\,b^2 + |C|^3)$; cache tiling and dense BLAS reduce the constant factor by an order of magnitude relative to classic CSR-based solvers.

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

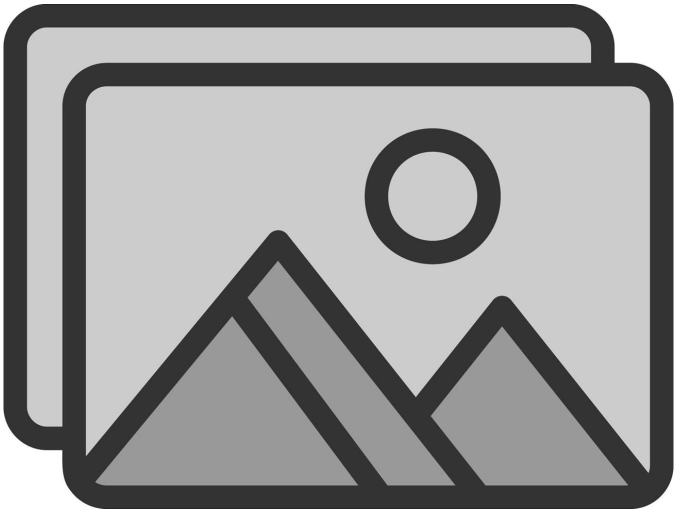

Fig. 3: System architecture showing data flow and major BLAS-backed kernels (placeholder).

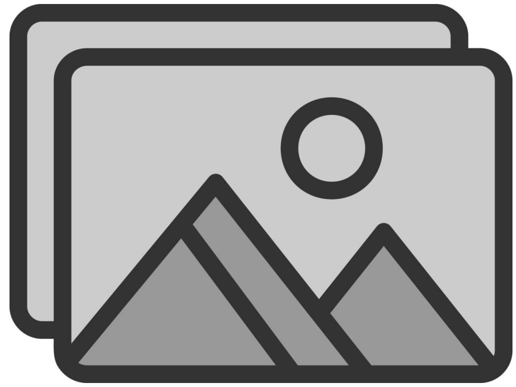

Fig. 4: Edge bucketing layout showing landmark-first organization with sequential pose/landmark IDs.

sparse bundle adjustment solver of Lourakis *et al.*,[26], [17], 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 state into a cache-coherent memory layout through two key steps. First, we assign compact zero-based IDs to poses and landmarks and pack their parameters contiguously (6or9 elements per pose, 3 per landmark). The ordering translates directly to the global Hessian $\mathbf{H}$ and right-hand side $\mathbf{b}$, giving stride-1 access in every kernel. Second, we bucket edges in a *landmark-first* hierarchy—each landmark stores a consecutive list of the poses that observe it. During Jacobian assembly a pose therefore touches its connected landmarks sequentially, preventing pointer chasing. Intermediate Schur blocks are addressed via closed-form index math to keep cache lines hot throughout matrix multiplication.

Forward link $\rightarrow$ The dense IDs serve as direct indices for the flat pattern maps of §IV-B.

TABLE I: 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

B. Sparse Graph Connectivity-Pattern Storage

STL-free flat-array architecture. Building on the IDs above we replace pointer-heavy STL containers with five monotonic arrays that encode the bipartite graph:

```
edge_v1[|E|]         // Edge -> Pose
edge_v2[|E|]         // Edge -> Landmark
v1_v2Edge[|V_p|]   // Pose  -> {V_l, E} list
v2_v1Edge[|V_l|]   // Landm -> {V_p, E} list
v1_edge_pairs[idx] // Pose-pose co-obs list
```

with the closed-form mapping

$$\mathrm{idx}(r, c) = r(r-1)/2 + c, . \tag{3}$$

Every matrix block—Jacobian, Hessian, Schur—can therefore be located in $\mathcal{O}(1)$ time without hash tables.

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 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^2SC 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

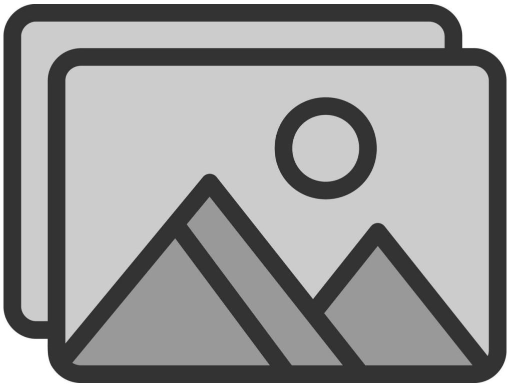

Fig. 5: Left: g2o CSR storage. Right: proposed flat-array encoding with $\mathcal{O}(1)$ block lookup.

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^2SC pipeline (§IV-C).

Forward link $\rightarrow$ The contiguous edge lists become the work-queue for the T^2SC tiles (§IV-C).

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

Pipeline motivation. Schur complement construction—$\mathbf{H}_{act} = \mathbf{A} - \mathbf{Y}\mathbf{W}^\top$—dominates run-time. Fetching scattered $\mathbf{Y}$ and $\mathbf{W}$ blocks thrashes caches. T^2SC sidesteps this by (*i*) copying blocks into pre-allocated contiguous tiles and (*ii*) feeding those tiles to batched GEMM.

Tile book-keeping. At start-up we allocate a tile per pose pair (r, c) sized from the co-observation count available via the flat arrays. Tiles live in a NUMA-aware arena so all threads share a single logical address space.

Phase I — block staging. Producer threads copy each required $\mathbf{Y}$ block once per LM iteration into its tile; $\mathbf{W}$ is constant and staged only at the first iteration. The copy walks the landmark-first edge lists sequentially so the memory system observes efficient prefetching.

Phase II — GEMM consumption. Consumer threads de-queue ready tiles and compute $\mathbf{Y}\cdot\mathbf{W}^\top$ through Level-3 BLAS. Diagonal tiles are emitted first so the pose-pose factorization can overlap with the outer-product of off-diagonals.

Measured impact. On a 16-core Intel i9-9900K the pipeline reduces Schur build time from $50\,\mathrm{ms}$ to $3.9\,\mathrm{ms}$ on the smallest BAL scene ($V_p = 49$, $V_l = 7776$, $E = 31843$) (12.9×). Across the five datasets the end-to-end solver is 3.6,× faster than g2o when limited to ten iterations—the ORB-SLAM real-time budget. On the bandwidth-constrained quad-core Cortex-A53 we still obtain 1.5,× overall speed-up, and T^2SC alone accounts for a consistent 2.5,× cut in Schur time.

Forward link $\rightarrow$ Tiles are handed to vendor BLAS libraries as detailed in §IV-D.

D. Exposing Matrix Operations via Standard BLAS APIs

BLAS-centric design. All heavy kernels reduce to Level-3 routines, allowing us to link against OpenBLAS, Intel MKL, Vitis BLAS or Arm Performance Libraries unmodified.

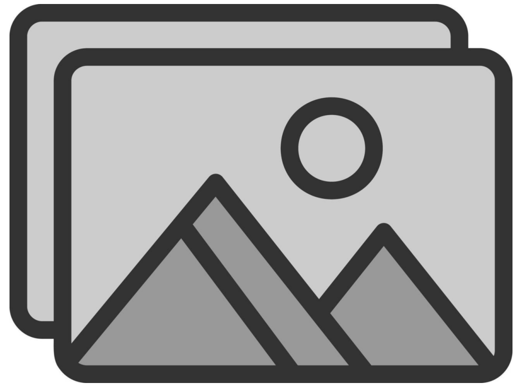

Fig. 6: T^2SC pipeline implementation: In producer stage, we copy the $\mathbf{Y}$ blocks into contiguous tiles, and in consumer stage, we compute the $\mathbf{Y}\cdot\mathbf{W}^T$ products using dense BLAS operations.

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.

TABLE II: Bundle-adjustment operation $\rightarrow$ BLAS mapping

BA Operation	**BLAS Kernel**	**Source blocks**
$\mathbf{J}^\top\mathbf{J}$	`GEMM_BATCH`	Contiguous Jacobians
$\mathbf{J}^\top\mathbf{r}$	`GEMM`	Sequential residuals
$\mathbf{Y}\mathbf{W}^\top$	`GEMM_BATCH`	T^2SC tiles
Landmark solve $\mathbf{B}^{-1}$	`POTRF/POTRS`	Dense blocks

Floating point internal operations. Re-building the solver in `float32` cuts memory footprint by $40\,\%$. On the BAL suite the χ^2 at iteration 5 differs by ¡0.01 single precision suffices for real-time SLAM windows while doubling the working-set capacity on the Kria board.

Additional numerical optimizations:

- **Mixed precision Jacobian computation:** We use `double` precision for finite difference calculations to maintain numerical accuracy, then cast results to `float32` for storage and subsequent operations. This balances accuracy with memory efficiency.
- **Parameter scaling:** Following Ceres[16] Solver's approach, we implement Jacobi scaling with $s_i = 1/\sqrt{||\mathbf{J}_{\mathrm{col}_i}||^2 + \epsilon}$ to improve numerical conditioning by normalizing parameter magnitudes across different scales (poses vs. landmarks).

Section summary. The pipeline of (1) dense parameter layout, (2) flat connectivity arrays, (3) T^2SC tiling, and (4) BLAS offload transforms sparse BA into cache-friendly dense GEMMs. The design achieves 1.5–3.6,× lower wall-time than g2o within the 10-iteration budget on ARM and x86 while maintaining accuracy parity with Ceres.

V. Experimental Setup

This section details the hardware, datasets, baselines, and evaluation metrics used to benchmark our cache-optimized bundle-adjustment (BA) solver.

A. Benchmark Platforms

We evaluate on two heterogeneous CPUs to expose the impact of cache capacity versus memory bandwidth:

- **Quad-core Cortex-A53** (Kria KR260 starter kit) (@1.5 GHz), 512 kB shared L2 and 4 GB LPDDR4. The solver is compiled with `gcc 13` using `-O3 -march=cortex-a53` and links against *Arm Performance Libraries 23.04* BLAS.
- **16-core Intel(R) Core(TM) i9-9900K CPU** (@3.60 GHz), 16 MB LLC and 32 GB DDR4-3200. We compile with `gcc 13` and link Eigen's BLAS shim to *OpenBLAS 0.3.8*; flags: `-O3 -march=native -fopenmp`.

B. Datasets and Problem Sizes

We evaluate on the Bundle Adjustment in the Large (BAL) dataset [27], which provides a diverse set of real-world reconstruction problems. Since our target application is real-time SLAM with bounded window sizes, we focus on five medium-scale scenes: *Ladybug-1*, *Dubrovnik-1*, *Ladybug-2*, *Trafalgar-3*, and *Ladybug-3*. These scenes span a representative range of problem sizes while remaining within typical SLAM memory budgets. Table III summarizes the key statistics for each scene: basic problem structure (cameras, landmarks, observations, co-observations) and derived complexity metrics including density ratios and the number of $\mathbf{Y}_i\mathbf{W}_i^\top$ operations required for Schur complement construction (derived from landmark co-visibility patterns). Each camera in the BAL dataset has 9 parameters: 3 for rotation, 3 for translation, 1 for focal length, and 2 for radial distortion coefficients. These camera intrinsics (focal length) and distortion parameters are jointly optimized along with the camera poses during bundle adjustment.

TABLE III: BAL dataset statistics showing problem structure and complexity.

Problem Structure				
Dataset	Cameras	Landmarks	Observations	Co-Obs
Ladybug-1	49	7776	31843	91243
Dubrovnik-1	16	22106	83718	170046
Ladybug-2	73	11032	46122	147691
Trafalgar-3	50	20431	73967	172026
Ladybug-3	138	19878	85217	328220
Complexity Metrics				
Dataset	O/C	O/L	L/C	# of $\mathbf{Y}_i\mathbf{W}_i^\top$
Ladybug-1	649.9	4.10	158.7	123086
Dubrovnik-1	5232.4	3.79	1381.6	253764
Ladybug-2	631.8	4.18	151.1	193813
Trafalgar-3	1479.3	3.62	408.6	245993
Ladybug-3	617.5	4.29	144.0	413437

C. Synthetic Scalability Suite

To validate that our runtime grows *only* with the true computational workload (the number of $\mathbf{Y}_i\mathbf{W}_i^\top$ tiles) we created a controlled, synthetic bundle-adjustment family. Each landmark is observed exactly four times, yielding a fixed sparse pattern that isolates Schur-complement work from visibility changes.

- **Pose sweep:** Landmarks fixed at **10k**; poses varied from 10 to 100 in steps of 10 (detailed in Appendix B).
- **Landmark sweep:** Poses fixed at **40**; landmarks varied from 10k to 100k in steps of 10k (detailed in Appendix B).

D. Baselines and Solver Configuration

We compare against the widely used open-source BA library:

- **g2o 2.3.5** in Levenberg–Marquardt mode using the block-sparse CSR backend with OpenMP, Cholmod, and CSparse support enabled. The solver is built with SuiteSparse dependencies (AMD, CAMD, CCOLAMD, CHOLMOD, COLAMD, SPQR) and Intel TBB 2022.0.

As done in ORB-SLAM3 [3], all solvers are limited to **ten outer LM iterations** to match the typical real-time budget of sliding-window local Bundle Adjustment in visual SLAM systems.

E. Evaluation Metrics

Reported quantities are:

1) Wall-clock time for the first ten LM iterations.
2) Mean time per outer iteration (runtime / 10).
3) χ^2 reprojection cost after iteration 10, and after full convergence of Ceres solver to evaluate each method's relative performance.
4) Peak memory usage (RAM) measured through the resident set size (RSS) metric using `/proc/$PID/statm` system file.

Each experiment is repeated three times; we report the mean with standard deviation below 2 %.

VI. Results & Discussion

We first compare accuracy and total runtime against g2o, then dissect per-iteration trends, and finally analyze two ablations: (a) disabling the T^{2}SC pipeline and (b) switching to single-precision arithmetic.

A. Runtime and Accuracy Overview

Table V summarizes both accuracy (χ^2 cost) and wall-clock runtime performance. Our solver achieves near-optimal convergence within the 10-iteration budget, with final χ^2 costs deviating by at most 0.52% from fully converged values. This rapid convergence, combined with our cache-optimized implementation, yields significant speed-ups over g2o: 1.2–2.7× faster on the Cortex-A53 and 2.3–8.8× faster on the Core i9. The speed-up is particularly pronounced on larger problems like Ladybug-3, where our solver completes in 1.2s on the i9 compared to g2o's 4.4s.

Beyond raw performance, our solver demonstrates superior numerical stability. G2o struggles with convergence on several datasets, most notably Dubrovnik-1 where its final χ^2 cost (124,093) is 3.4× higher than our method's (36,068). This validates the effectiveness of our parameter scaling and mixed-precision differentiation strategies in maintaining both speed and accuracy.

TABLE IV: Accuracy comparison after ten LM iterations.

	χ^2 **after iteration 10**			
Dataset	Ours	g2o	Final	% vs Final
Ladybug-1	26700.7	30980.5	26688.7	+0.04%
Dubrovnik-1	36067.7	124093.1	36067.7	+0.00%
Ladybug-2	34288.4	34391.6	34198.9	+0.26%
Trafalgar-3	103754.9	141586.9	103218.1	+0.52%
Ladybug-3	118788.8	129259.0	118749.2	+0.03%

TABLE V: Runtime comparison for 10 iterations [ms].

	Time for 10 iterations [ms]			
Dataset	Ours A53	g2o A53	Ours i9	g2o i9
Ladybug-1	3202	8601	189	1662
Dubrovnik-1	9651	17418	680	1562
Ladybug-2	7214	14561	471	3000
Trafalgar-3	9138	16999	836	2159
Ladybug-3	36325	43487	1223	4433

B. Per-Iteration Scaling on the Synthetic Suite

Figure 7 plots the mean time per Levenberg–Marquardt (LM) iteration against the number of $\mathbf{YW}^\top$ tiles. Both sweeps—one varying pose count and the other landmark count—exhibit a strong linear trend ($R^2 > 0.98$). This confirms that our tiled two-phase Schur (T^2SC) pipeline scales directly with the *true* computational workload rather than abstract graph metrics like vertex or edge counts. A unified linear fit across both synthetic series reveals a consistent cost of approximately $0.11\,\mu s$ per $\mathbf{YW}^\top$ tile on the i9 CPU. This result supports our central claim: cache-friendly data organization transforms the workload into a predictable, compute-bound problem dominated by matrix multiplication.

C. Ablation Study

a) (a) Impact of T^2SC.: Table VI shows that disabling T^2SC consistently slows performance on the memory-bandwidth-limited A53 by 2.3–3.2×. On the i9, which has a much larger cache, the benefits are even more pronounced on smaller problems (up to 12.9×) but diminish as the total tile memory footprint grows. Nonetheless, it provided a clear net benefit in all tested scenarios.

b) (b) Single vs. double precision.: Table VII confirms that float32 reduces memory footprint by 38.9–42.0% while maintaining χ^2 accuracy within 0.01% of double-precision results, with no observable loss in convergence quality.

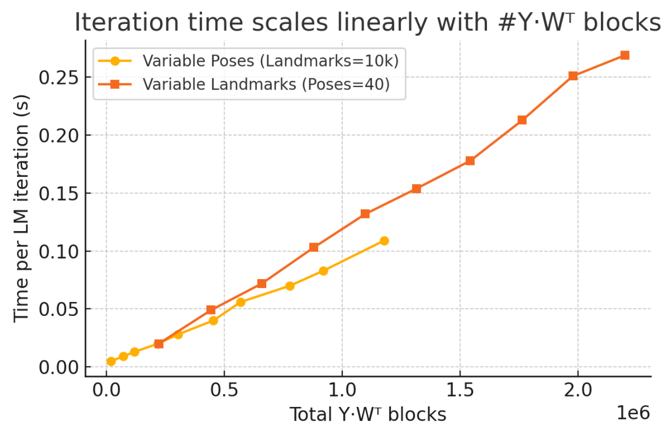

Fig. 7: Time per LM iteration on the Core i9-9900K as a function of the number of $\mathbf{YW}^\top$ tiles in the synthetic scalability suite. *Circles:* landmarks fixed (10 k) while poses increase. *Squares:* poses fixed (40) while landmarks increase. The shared slope ($\approx 0.11\,\mu s$ per tile) indicates that $\mathbf{YW}^\top$ multiplication dominates runtime.

TABLE VI: Ablation (a): T^2SC on/off runtime comparison (ms).

	A53		i9	
Dataset	ON	OFF	ON	OFF
Ladybug-1	45.6	147.3	3.9	50.4
Dubrovnik-1	184.9	315.3	22.9	96.1
Ladybug-2	82.1	238.0	6.5	78.4
Trafalgar-3	164.2	300.1	32.5	106.0
Ladybug-3	187.8	502.0	15.3	169.0

TABLE VII: Ablation (b): float32 vs double precision comparison.

Dataset	χ^2_{f32}	χ^2_{f64}	RAM % save
Ladybug-1	26775.908	26783.331	38.9%
Dubrovnik-1	36067.785	36067.835	39.2%
Ladybug-2	34318.758	34318.709	40.3%
Trafalgar-3	123822.656	123820.275	39.0%
Ladybug-3	119939.984	119940.173	42.0%

D. Peak Memory Consumption

Table VIII contrasts resident set size across different configurations. The T^2SC pipeline requires 2.1–2.5× more memory than the baseline version, with peak usage reaching $290.4\,\mathrm{MB}$ on the largest dataset. Interestingly, our baseline (T^2SC off) uses less memory than g2o on most datasets, while the full T^2SC version trades memory for significant speed improvements. Future work will explore tile compression and out-of-core variants.

E. Discussion and Limitations

The results reveal distinct performance characteristics across platforms. On the i9, the large $16\,\mathrm{MB}$ LLC enables cache-resident tile processing, delivering 2.3–8.8× speed-ups over g2o. The effectiveness of T^2SC varies with problem

TABLE VIII: Peak RAM usage (values in MB).

Implementation	Dataset				
	Lady-1	Dub-1	Lady-2	Traf-3	Lady-3
Ours (+T^2SC)	101.7	219.2	148.5	206.0	290.4
Ours (-T^2SC)	48.0	104.2	64.6	94.9	115.8
g2o	52.9	104.4	74.3	98.4	132.8

size; while smaller problems see dramatic acceleration (up to 12.9×), the advantage diminishes for larger problems as tile memory begins to exceed cache capacity. On the A53 platform, where memory bandwidth is the primary bottleneck, T^2SC provides a consistent and significant 2.3–3.2× speed-up. This suggests our contiguous storage and batched BLAS approach is highly effective in memory-constrained scenarios. A key limitation is memory overhead: T^2SC requires 2.1–2.5× more RAM, reaching $290.4\,\mathrm{MB}$ on the largest dataset—a significant constraint for embedded deployment.

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 …

References

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[2] R. Hartley, *Multiple view geometry in computer vision*. Cambridge university press, 2003, vol. 665.

[3] C. Campos, R. Elvira, J. J. G. Rodríguez, J. M. Montiel, and J. D. Tardós, "Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam," *IEEE transactions on robotics*, vol. 37, no. 6, pp. 1874–1890, 2021.

[4] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International conference on computer vision*. Ieee, 2011, pp. 2564–2571.

[5] W. Fang, Y. Zhang, B. Yu, and S. Liu, "Fpga-based orb feature extraction for real-time visual slam," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 275–278.

[6] Y. Liu, J. Li, K. Huang, X. Li, X. Qi, L. Chang, Y. Long, and J. Zhou, "Mobilesp: An fpga-based real-time keypoint extraction hardware accelerator for mobile vslam," *IEEE transactions on circuits and systems I: regular papers*, vol. 69, no. 12, pp. 4919–4929, 2022.

[7] S. Semenova, S. Y. Ko, Y. D. Liu, L. Ziarek, and K. Dantu, "A quantitative analysis of system bottlenecks in visual slam," in *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, 2022, pp. 74–80.

[8] S. Qin, Q. Liu, B. Yu, and S. Liu, "π-ba: Bundle adjustment acceleration on embedded fpgas with co-observation optimization," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 100–108.

[9] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz, "Multicore bundle adjustment," in *CVPR 2011*. IEEE, 2011, pp. 3057–3064.

[10] J. Ren, W. Liang, R. Yan, L. Mai, X. Liu, and X. Liu, "Megba: A high-performance and distributed library for large-scale bundle adjustment," in *European Conference on Computer Vision (ECCV)*, vol. 2, 2022.

[11] R. Hänsch, I. Drude, and O. Hellwich, "Modern methods of bundle adjustment on the gpu," *ISPRS annals of the photogrammetry, remote sensing and spatial information sciences*, vol. 3, pp. 43–50, 2016.

[12] R. Sun, P. Liu, J. Xue, S. Yang, J. Qian, and R. Ying, "Bax: A bundle adjustment accelerator with decoupled access/execute architecture for visual odometry," *IEEE Access*, vol. 8, pp. 75 530–75 542, 2020.

[13] Y. Liu, K. Huang, N. Zhao, J. Li, S. Hao, Z. Huang, X. Qi, X. Wang, L. Zhou, L. Chang *et al.*, "Boha: A high performance vslam backend optimization hardware accelerator using recursive fine-grain h-matrix decomposition and early-computing with approximate linear solver," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 10, pp. 3827–3831, 2023.

[14] W. Liu, B. Yu, Y. Gan, Q. Liu, J. Tang, S. Liu, and Y. Zhu, "Archytas: A framework for synthesizing and dynamically optimizing accelerators for robotic localization," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 479–493.

[15] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g 2 o: A general framework for graph optimization," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 3607–3613.

[16] S. Agarwal, K. Mierle *et al.*, "Ceres solver: Tutorial & reference," *Google Inc*, vol. 2, no. 72, p. 8, 2012.

[17] 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.

[18] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—a modern synthesis," in *International workshop on vision algorithms*. Springer, 1999, pp. 298–372.

[19] K. Konolige, "Sparse sparse bundle adjustment," in *British Machine Vision Conference*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:13784570

[20] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[21] *Intel oneAPI Math Kernel Library Developer Reference*, Intel Corporation, 2023, https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[22] *Arm Performance Libraries User Guide*, Arm Ltd., 2023, https://developer.arm.com/downloads/-/arm-performance-libraries.

[23] X. Zhang *et al.*, *OpenBLAS: An Optimized BLAS Library*, 2023, https://www.openblas.net.

[24] *cuBLAS Library User Guide*, NVIDIA Corporation, 2025, https://docs.nvidia.com/cuda/cublas/.

[25] *Vitis Libraries: BLAS User Guide*, AMD Xilinx, 2024, https://github.com/Xilinx/Vitis_Libraries/tree/master/blas.

[26] 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.

[27] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski, "Bundle adjustment in the large," in *Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part II 11*. Springer, 2010, pp. 29–42.

Appendix

A. Detailed Algorithm Derivations

B. Additional Experimental Results

This section provides detailed scaling characteristics for the synthetic benchmark suite referenced in Section V-C. The following tables show the precise problem sizes and computational complexity metrics for both the pose and landmark scaling sweeps.

TABLE IX: Pose sweep: problem size versus cost drivers.

Poses	Landmarks	Observations †	# $YW^\top$ tiles
10	10 000	40 000	20 044
20	10 000	40 000	70 213
30	10 000	40 000	119 498
40	10 000	40 000	218 828
50	10 000	40 000	303 770
60	10 000	40 000	453 498
70	10 000	40 000	569 260
80	10 000	40 000	777 922
90	10 000	40 000	920 175
100	10 000	40 000	1 177 270

† Four observations per landmark.

TABLE X: Landmark sweep: problem size versus cost drivers.

Poses	Landmarks	Observations	# $YW^\top$ tiles
40	10 000	40 000	221 862
40	20 000	80 000	443 766
40	30 000	120 000	659 369
40	40 000	160 000	878 919
40	50 000	200 000	1 098 073
40	60 000	240 000	1 316 027
40	70 000	280 000	1 543 366
40	80 000	320 000	1 763 846
40	90 000	360 000	1 977 473
40	100 000	400 000	2 198 656

C. Implementation Detailssdsd

Algorithm 2 Detailed Block-Structured BA–LM Solver with Scaling and Schur Complement

Require: Scaled tolerance ε_{g}, damping μ, current pose/landmark parameters $\mathbf{p}_1, \mathbf{p}_2$, observations $\mathbf{x}$, information $\mathbf{\Sigma}^{-1}$
Ensure: Increment $\delta\mathbf{p}$

Compute dense column norms $n_i \leftarrow \|\mathbf{J}_{:i}\|_2^2$; $s_i \leftarrow 1/\sqrt{n_i + \varepsilon}$
Scale Jacobian blocks $\widetilde{\mathbf{J}}_{:i} \leftarrow s_i\,\mathbf{J}_{:i}$ and residual $\widetilde{\boldsymbol{\varepsilon}} \leftarrow \mathbf{\Sigma}^{-1/2}\,(\mathbf{x} - f(\mathbf{p}))$
for all edge $e = (i, j)$ **do**
    $\mathbf{W}_{ij} \mathrel{+}= \widetilde{\mathbf{J}}_1(e)^T \widetilde{\mathbf{J}}_2(e)$
    $\mathbf{A}_i \mathrel{+}= \widetilde{\mathbf{J}}_1(e)^T \widetilde{\mathbf{J}}_1(e)$; $\mathbf{B}_j \mathrel{+}= \widetilde{\mathbf{J}}_2(e)^T \widetilde{\mathbf{J}}_2(e)$
    $\mathbf{b}_{1,i} \mathrel{+}= \widetilde{\mathbf{J}}_1(e)^T \widetilde{\boldsymbol{\varepsilon}}(e)$; $\mathbf{b}_{2,j} \mathrel{+}= \widetilde{\mathbf{J}}_2(e)^T \widetilde{\boldsymbol{\varepsilon}}(e)$
end for
for all i **do** $\mathbf{A}_i \leftarrow \mathbf{A}_i + \mu\mathbf{I}$
end for;
for all j **do** $\mathbf{B}_j \leftarrow \mathbf{B}_j + \mu\mathbf{I}$
end for
for all j **do** $\mathbf{B}_j^{-1} \leftarrow$ LDLT($\mathbf{B}_j$) (SVD fallback)
end for
for all (i, j) **do** $\mathbf{Y}_{ij} \leftarrow \mathbf{W}_{ij}\mathbf{B}_j^{-1}$
end for
for $i = 1$ **to** m **do**
    $\mathbf{H}_{ii} \leftarrow \mathbf{A}_i$
    $\mathbf{b}_{\mathrm{act},i} \leftarrow \mathbf{b}_{1,i}$
end for
for all pose pair (i, k) sharing landmarks **do**
    $\mathbf{H}_{ik} \leftarrow \mathbf{0}$
end for
for all edge $e = (i, j)$ **do**
    $\mathbf{H}_{ii} \mathrel{-}= \mathbf{Y}_{ij}\mathbf{W}_{ij}^T$
    $\mathbf{b}_{\mathrm{act},i} \mathrel{-}= \mathbf{Y}_{ij}\mathbf{b}_{2,j}$
    for all pose $k\,(k < i)$ observing j **do**
        $\mathbf{H}_{ik} \mathrel{-}= \mathbf{Y}_{ij}\mathbf{W}_{kj}^T$
    end for
end for
Solve $\mathbf{H}\,\delta\mathbf{p}_1 = \mathbf{b}_{\mathrm{act}}$ (LDLT, fallback LU)
for $j = 1$ **to** n **do**
    $\delta\mathbf{p}_{2,j} \leftarrow \mathbf{B}_j^{-1}\Big(\mathbf{b}_{2,j} - \sum_i \mathbf{W}_{ij}^T\,\delta\mathbf{p}_{1,i}\Big)$
end for
$\delta\mathbf{p} \leftarrow \delta\mathbf{p} \oslash \mathbf{s}$
return $\delta\mathbf{p}$