IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2024.0429000

Achieving a 155× Speedup in a LiDAR Data Processing Pipeline

SIMON FELIX[1], MANUEL STUTZ[1], JULIA HARTMANN[1,3], YVES BÄCHTIGER[1], VINCENZO TIMMEL[1], ANDREAS WASSMER[1], FILIP SCHRAMKA[1,3], ELMAR STROBACH[2], JÜRG LECKEBUSCH[2], and CHRISTIAN ANGST[2]

[1]Institute for Data Science, University of Applied Sciences and Arts Northwestern Switzerland (FHNW), Windisch, Switzerland
[2]IMP Bautest AG, Oberbuchsiten, Switzerland
[3]Now at: Ateleris GmbH, Brugg, Switzerland

Corresponding author: Simon Felix (e-mail: simon.felix@fhnw.ch).

This work was supported by the Swiss Innovation Agency (Innosuisse) under Grant 103.959 IP-ICT.

ABSTRACT Real-world data processing pipelines often grow incrementally over years and accumulate inefficiencies that no single processing step explains. We present a measurement-driven case study of optimizing one such system end-to-end: a multi-stage pipeline for high-resolution LiDAR road condition assessment. The original system required weeks of computation for a typical measurement campaign covering a 10 km road network and approximately 200 GB of raw data. Our primary contribution is a continuous, measurement-driven optimization methodology. Automated per-commit benchmarks on fixed reference datasets track per-stage runtime throughout the project. This revealed that the dominant bottleneck shifted repeatedly as optimization proceeded. It allowed us to target each successive bottleneck and detect regressions early. Guided by this methodology, we consolidated a distributed eight-server architecture onto a single high-performance node, eliminated intermediate file I/O through in-memory stream processing, migrated heterogeneous runtimes (MATLAB, Python, C++) to a single managed C# codebase, and applied pervasive SIMD vectorization and parallelization. We achieved a 155× end-to-end speedup, enabling the pipeline to process data faster than it is recorded. We decompose this speedup per stage and estimate that software and architectural changes, rather than hardware alone, account for at least an order of magnitude of it. We also observe that carefully vectorized managed code matched or exceeded the original native code implementations for our workload. We do not claim novel algorithms, but contribute a reproducible methodology and quantitative evidence that disciplined engineering can yield order-of-magnitude improvements in legacy data processing systems.

INDEX TERMS Software optimization, LiDAR processing, SIMD vectorization, stream processing, road condition assessment, managed code performance.

I. INTRODUCTION

ROAD infrastructure is among the most valuable public assets in any country. In Switzerland alone, the road network spans approximately 84,000 km. Regular condition assessment is essential for efficient maintenance planning, yet the data volumes produced by modern measurement vehicles create a severe computational bottleneck.

Our measurement vehicle, the Integrated Road Information System (IRIS, Fig. 1) operated by IMP Bautest AG [1], is equipped with a Fraunhofer PPS+ pavement profile scanner, a Fraunhofer CPS clearance profile scanner, a Ladybug 5 panoramic camera, four machine vision cameras, two macro-texture scanners, and a high-precision GNSS/IMU positioning system. During operation at typical road speeds, the system generates approximately 20 GB of data per kilometer. A full measurement campaign can produce terabytes of data that must be processed through a complex, multi-stage pipeline to extract road condition indicators such as longitudinal and transverse evenness, cracking, texture depth, and georeferenced imagery.

Prior to our work, this processing pipeline, which evolved over more than a decade, required *weeks* of computation for a full campaign. Data recording was distributed across eight servers inside the measurement vehicle, and data processing

FIGURE 1. The IRIS measurement vehicle. The roof-mounted equipment includes cameras, a road surface LiDAR, and GNSS equipment; the macro-texture scanners are mounted to the bottom of the vehicle.

was done on on-premises servers and involved a heterogeneous mix of C#, C++, MATLAB, and Python components communicating primarily through intermediate files. These processing delays meant that data quality issues could only be identified well after the measurement, limiting the ability to react during a measurement campaign.

In this paper we describe a systematic approach that achieved a 155× end-to-end speedup, enabling the pipeline to complete processing faster than data is recorded. For a representative reference dataset (a short urban measurement of approximately 0.5 km at low speed, recording duration 115 seconds), processing time dropped from over 184 minutes to 71.4 seconds. We used reference datasets of varying length and recording conditions for continuous benchmarking throughout the project, and report detailed results for the primary reference dataset here.

The contributions of this paper are the following:

1) A measurement-driven optimization methodology for heterogeneous pipelines. We integrate hierarchical per-stage timing and memory profiling into a continuous-integration system that benchmarks the full pipeline on fixed reference datasets at every commit. This produces an unbroken time series of per-stage performance. Our central finding is that, in a decade-old multi-stage pipeline, the dominant bottleneck shifts repeatedly as optimization proceeds. We show that per-commit benchmarking is what makes it tractable to identify each successive bottleneck and to detect regressions before they accumulate.
2) A quantified decomposition of an end-to-end speedup. We report per-stage speedups for a complete production pipeline (Table 2), and we separate the contribution of hardware from that of software and architectural change. We find that software and architectural changes account for at least an order of magnitude of the overall improvement.
3) Empirical evidence on managed versus native code performance for this workload class. We document that carefully vectorized managed C# matched or exceeded the original native C++ and MATLAB implementations in our use cases.

Unlike prior work that accelerates individual processing steps in isolation, we address the optimization of a complete, heterogeneous production pipeline, where the dominant costs arise from file-based intermediate storage, runtime heterogeneity, and serial execution rather than from any single algorithm. We show that the disciplined, measurement-driven application of well-known engineering principles yields order-of-magnitude improvements in a real-world legacy system, and we report the methodology and quantitative evidence that make this result reproducible.

II. RELATED WORK

Mobile mapping systems and their data processing requirements have been surveyed by Elhashash et al. [2], who note that the computational demands of multi-sensor platforms remain a significant challenge.

On the algorithmic level, significant work has focused on accelerating individual LiDAR processing steps. For example, Hu et al. [3] proposed scan-line-based ground filtering of urban LiDAR point clouds with GPU acceleration, demonstrating significant speedups for individual filter operations. Duan et al. [4] developed a PCA-based outlier removal filter for point clouds, achieving superior performance compared to prior approaches. Pajarola [5] introduced stream-processing paradigms for point cloud data, demonstrating that streaming can reduce memory requirements and improve throughput for large datasets.

These works share a common characteristic: they optimize individual algorithms or processing steps in isolation. In contrast, real-world data processing pipelines consist of dozens of interdependent stages spanning data parsing, interpolation, image generation, statistical analysis, database operations, and report generation. The bottleneck in such systems is often not a single algorithm, but the accumulated overhead of file-based intermediate storage, heterogeneous runtime environments, process startup costs, and serial execution patterns.

Stream processing architectures like Apache Flink [6] address similar challenges in distributed systems by replacing batch-oriented file exchange with continuous data flows; we apply analogous principles within a single-machine, shared-memory setting.

The challenge of modernizing legacy software systems has been studied extensively in software engineering. Assunção et al. [7] provide a recent survey of contemporary strategies for legacy system modernization.

Regarding the computational efficiency of managed-memory programming languages, Hundt [8] provided a controlled comparison of C++, Java, Go, and Scala on identical algorithms, finding that idiomatic managed-language code can approach native performance after optimization, though language-specific tuning remains important. Modern managed runtimes such as .NET have narrowed the performance

gap with native code through JIT compilation, hardware intrinsics, and memory-efficient APIs [9].

The importance of measurement-driven optimization has been recognized since Knuth's observation that "premature optimization is the root of all evil"; systematic profiling tools enable developers to direct effort to the actual bottlenecks. The role of continuous profiling in guiding optimization effort has been demonstrated in domains ranging from compiler optimization [10] to streaming application tuning [11] and HPC workloads [12].

Building on this body of work, we address the end-to-end optimization of a complete, heterogeneous production pipeline under a measurement-driven methodology, as set out in the previous section. We further demonstrate that this approach yields speedups comparable to those reported for individual GPU-accelerated kernels, but across an entire processing chain rather than a single step.

III. HARDWARE CONSOLIDATION

The original system distributed acquisition across eight servers, connected via a 10 Gbit/s Ethernet switch. These servers ranged from Intel Core 2 E7600 systems from 2009 to Intel i7-8700 machines from 2017, with heterogeneous RAM configurations and HDD-based storage. Each server was dedicated to a specific subsystem: one for each camera pair, one for each LiDAR scanner, one for the positioning system, and so on. This distributed architecture, while functional, imposed several performance limitations.

Data transfer overhead. Raw PPS+ scanner data (5–20 GB/km) had to be transferred between servers for processing. For a 100 km measurement at urban density, this amounts to terabytes of data that had to traverse the internal network before processing could begin, a transfer that alone could take hours.

Time synchronization complexity. At highway speeds, a timing error of just 1 ms between two sensors corresponds to a spatial offset of over 2 cm, an order of magnitude over the system's measurement resolution. The original system maintained sub-millisecond time synchronization via individual Meinberg GPS clock cards [13], a technically demanding arrangement. Custom analog-to-optical converter PCBs distributed pulse-per-second signals from the GPS clock to the LiDAR scanners via fiber optic links; understanding and migrating these custom boards was one of the challenges.

Redundant computation. Several pipeline stages independently read and parsed the same files (e.g., trajectories), because no shared memory existed between the stages.

We consolidated all computation onto a single server, whose specifications are summarized alongside the legacy system in Table 1. The new AMD EPYC 7643-based system provides approximately 11× the single-threaded performance of the strongest legacy server, as measured by the industry benchmark *CoreMark*. More importantly, the 48 physical cores (96 threads), 256 GB of DDR4-3200 ECC RAM, and NVMe SSD storage with 7 GB/s sequential bandwidth provide the foundation for the architecture described in subsequent sections. The hardware platform was specifically chosen to provide many high-bandwidth interfaces.

TABLE 1. Hardware comparison between the legacy distributed system (8 servers, cumulative) and the new consolidated server.

Component	Legacy (8 srv.)	New (1 srv.)
CPU cores (phys.)	34	48
Logical threads	58	96
RAM	68 GB	256 GB
Storage	4 TB HDD	8 TB NVMe
Storage BW	≈0.3 GB/s	≈7 GB/s

All sensors now connect directly to this single server: the LiDAR scanners via fiber optic, the panoramic camera via USB 3, the machine vision cameras and the positioning system via Ethernet. A single Meinberg GPS180PEX PCIe clock card [13] provides accurate timing, even if timing accuracy is no longer critical. The system architecture before and after consolidation is shown in Fig. 2 and Fig. 3.

Beyond performance, consolidation simplified operations. The previous system required coordinated startup of eight machines and verification of network links and clock synchronization before each measurement. Diagnosing issues in the field, often 100–200 km from the laboratory, required familiarity with a heterogeneous hardware landscape. The consolidated system is easier to maintain and diagnose, and its single time reference eliminates the synchronization challenge entirely.

IV. MEASUREMENT-DRIVEN OPTIMIZATION

Reliable, reproducible measurements are a prerequisite for systematic performance optimization. Without knowing which pipeline stage dominates runtime, optimization effort risks being misdirected. This concern is particularly acute in a pipeline with over a dozen stages.

We developed a hierarchical timing and memory profiling framework that instruments every pipeline stage. Each processing step is wrapped in a measurement scope that records wall-clock time using the platform's high-resolution monotonic timer; a background thread samples the process working set at 1 Hz, from which each scope records its peak memory. Scopes nest, so the results form a tree (stages contain substages) that mirrors the pipeline structure. At the end of a run, this tree is serialized to a JSON file, one record per scope with its duration and peak memory, annotated with the Git commit hash and a timestamp. An analysis script reads the accumulated files and generates stacked area charts showing per-stage runtime evolution across all commits.

This profiling framework was integrated into our continuous integration (CI) system. On every merge to the main branch, the CI system builds the pipeline in release configuration and executes it end-to-end on four reference datasets of varying length and sensor configurations. Benchmark runs execute on dedicated machines, with the reference data on local storage and no concurrent workloads, so that successive runs are comparable. This provides an unbroken time series of

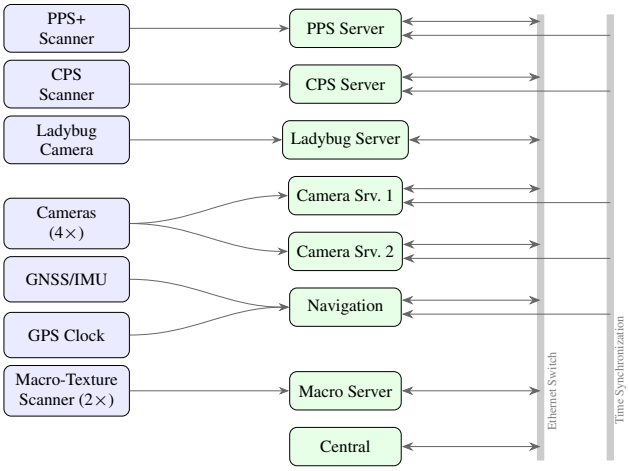

FIGURE 2. Original eight-server distributed architecture. Each sensor connects to a dedicated server; all servers communicate via a fiber-optic switch.

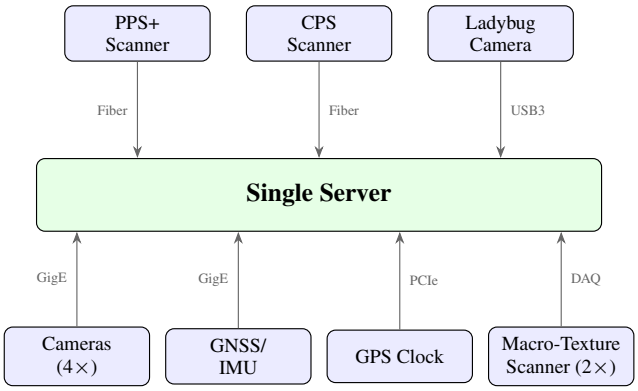

FIGURE 3. Consolidated single-server architecture. All sensors connect directly to one machine, eliminating network transfers and time synchronization complexity.

per-commit performance data, enabling immediate identification of regressions and quantification of each optimization's impact.

An important architectural decision was to run pipeline stages *sequentially* rather than overlapping multiple stages in parallel. While overlapping stages could theoretically improve throughput, running them sequentially allows precise accounting of each stage's memory consumption and execution time. If multiple stages ran concurrently, their memory footprints, I/O loads, and garbage collection pauses would interact in complex ways, making it nearly impossible to attribute resource usage to individual stages. The sequential approach was essential for our profiling-driven methodology: each stage's contribution to overall runtime is clearly measurable and directly actionable. Within a stage, however, parallelism is fully exploited. We used traditional sampling profilers to optimize individual stages, and found *Ultra* [14] particularly helpful for this purpose.

Fig. 4 shows a stacked area chart that is generated by our CI system for the primary reference dataset. This visualization proved indispensable for directing optimization effort. Early in the project, topographic image generation dominated at over 4,000 seconds; once that was reduced, the LiDAR data reader became the next bottleneck, followed by longitudinal profile analysis. At each stage, the chart immediately reveals the next most impactful target, and red-annotated regressions trigger investigation before further progress is made.

V. PROCESSING PIPELINE AND DATA FLOW

The data processing pipeline transforms raw sensor data into road condition indicators stored in a PostgreSQL/PostGIS database. Fig. 5 depicts the overall data flow through the processing stages.

The pipeline was originally implemented as a sequential chain of largely independent programs. Components were written in C# (.NET Framework), C++ (native DLLs and external executables), MATLAB (compiled to standalone executables requiring the MATLAB Compiler Runtime [15]), and Python scripts. Data exchange between stages was almost entirely file-based: the LiDAR scanner reader produced intermediate data files consuming 10–40 GB/km of temporary storage, which downstream stages then re-read. Point cloud processing relied on a separate, third-party executable, and panoramic image generation invoked Python scripts, calling the Hugin [16] panoramic stitching toolkit.

Stages ran sequentially with no parallelism, and several stages redundantly parsed the same large input files. For example, the trajectory file was loaded repeatedly by independent pipeline stages. The initial end-to-end processing time for the primary reference dataset was 184 minutes on the original hardware.

VI. ARCHITECTURAL TRANSFORMATION: FROM FILES TO STREAMS

A. SCANNER DATA STREAMING

The single most impactful category of optimization was the elimination of intermediate file I/O, replacing it with in-memory stream processing.

Originally, the LiDAR scanner reader (a C++ library provided by Fraunhofer IPM), was integrated via intermediate files and required a separate read pass by every downstream consumer. This write-read cycle alone accounted for a large fraction of total processing time, as it was fundamentally I/O-bound.

We redesigned the interface between the C++ library and the managed C# pipeline to use callbacks: instead of writing files, the C++ library invokes managed delegates for each extracted profile. Profiles are delivered directly into memory without serialization, and are immediately dispatched to concurrent consumer queues. This eliminated the index file generation pass, the temporary storage requirement (saving 10–40 GB/km of disk space), and the separate file-reading pass in all downstream consumers.

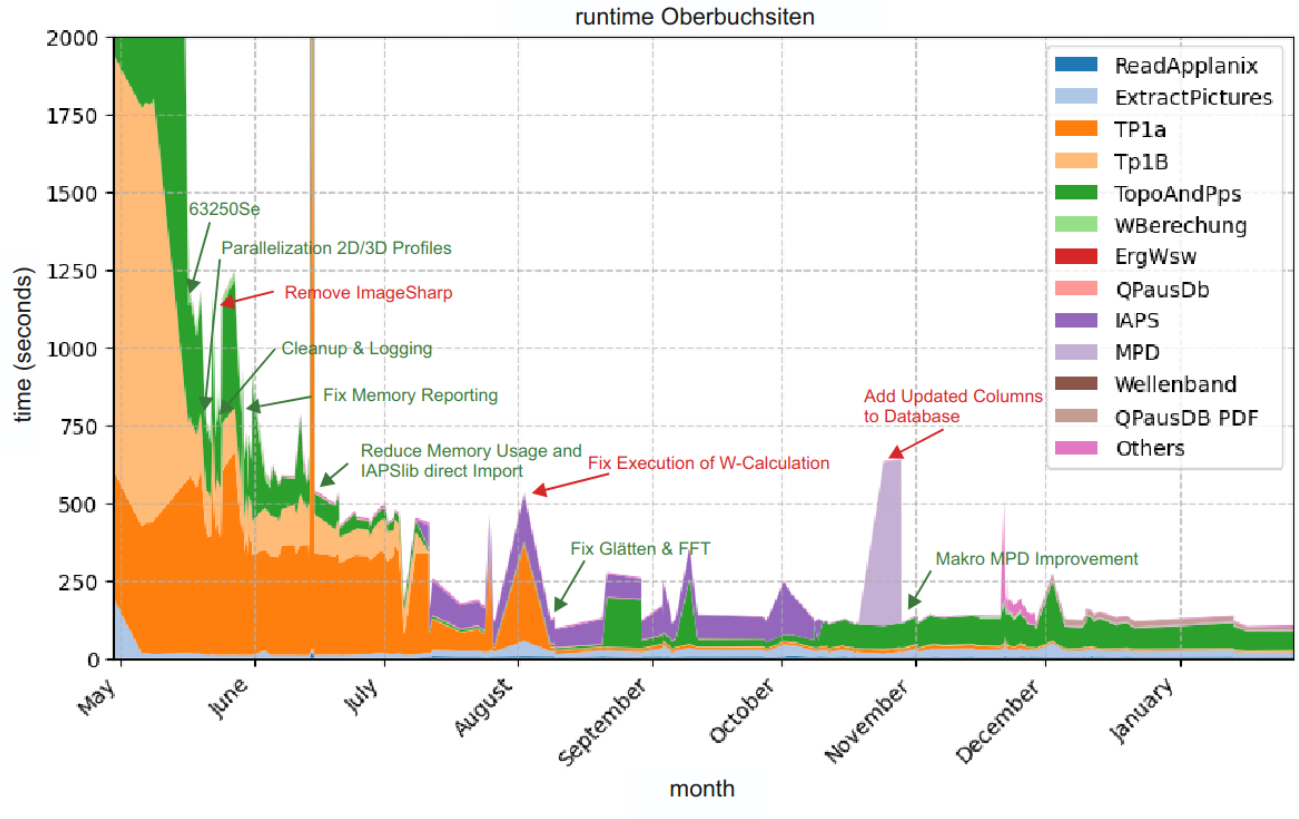

FIGURE 4. **Our CI/CD pipeline produces charts continuously over the project duration, showing the per-stage processing time for each reference dataset. Team members can add explanatory green and red annotations marks for performance-relevant changes. Each color band represents a code section. This chart guided the optimization efforts in this project, and reveals how the dominant bottleneck shifts over time.**

The 2D profiles (surface topography cross-sections used for imaging) and 3D profiles (point clouds used for longitudinal and transverse analysis) are streamed through separate data flows, allowing the scanner reader to process both flows concurrently rather than sequentially. Outliers are filtered, missing data points are imputed, and the profiles are transformed into a global coordinate system.

The 2D and 3D streams are joined back together into a single stream after these processing steps. We keep 3D profiles in memory, to avoid re-reading data as they are used by multiple processing steps that we run sequentially. Memory consumption was reduced to approximately half of the original through compacted data formats (from 33 to 17 bytes per 2D point, and from 23 to 9 bytes per 3D point). The available RAM provides headroom for continuous measurement runs of up to two hours, well above the operational requirements.

B. TRAJECTORY PARSING

The GNSS/IMU positioning system records trajectory data at 200 Hz in a text-based format, producing files on the order of a gigabyte for a typical measurement. In the original pipeline, this file was re-read and re-parsed independently by multiple pipeline stages.

We reimplemented the parser in managed C# with two key optimizations. First, the file is parsed once and the resulting trajectory is shared across all pipeline stages via in-memory references. Second, parsing itself is parallelized across multiple threads. The challenge with parallel text file parsing is that line boundaries are not known a priori. To split the file into chunks for parallel processing, each thread must scan forward from its assigned byte offset to find the next complete line boundary. We handle this by having each thread skip the partial line at the beginning of its chunk, while the preceding thread processes the partial line at the end of its chunk. This approach scales well to the available core count.

With these changes, the trajectory file is parsed in approximately 1 s, and a separate trajectory-splitting preprocessing step that was previously required to break the file into smaller segments is no longer required. We order the entries in memory to allow binary searches, and switch to short-range linear searches for nearly sequential queries.

VII. RUNTIME MIGRATION

A. MATLAB REMOVAL

The longitudinal analysis stage, which computes evenness indicators, longitudinal profiles, and waveband decomposition, was originally implemented in MATLAB and compiled to a standalone executable requiring the MATLAB Compiler Runtime [15]. This architecture imposed several severe performance penalties. Each invocation incurred a multi-second

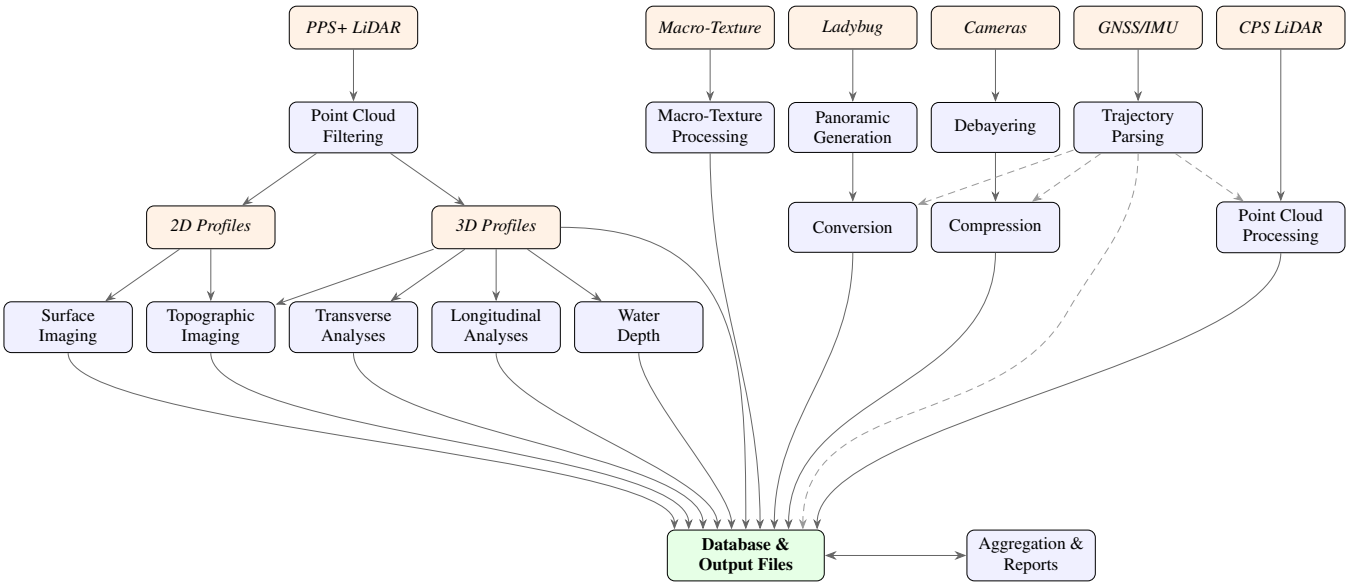

FIGURE 5. Data flow through the processing pipeline. Dashed arrows indicate trajectory data shared across stages for georeferencing. Results are stored in a PostgreSQL/PostGIS database and as output files (panoramas, topographic images, point clouds, reports).

startup penalty for the MATLAB Runtime initialization. Data was serialized to intermediate text files, the MATLAB process was started, and results were read back, creating a full I/O round-trip for every measurement segment. The MATLAB process did not share memory with the rest of the pipeline, so large files such as the trajectory were loaded independently. Moreover, MATLAB's process model made fine-grained parallelization across the many available CPU cores impractical [17].

We re-implemented all algorithms in C#, including longitudinal profile extraction, FFT-based frequency analysis, Power Spectral Density computation, frequency-band smoothing filters, linear regression for profile alignment, and a 4 m straightedge simulation. Each function was validated against the MATLAB reference output with per-function tolerance thresholds via unit tests. With the MATLAB runtime eliminated, the longitudinal analysis stage now runs as part of the same process as the rest of the pipeline, sharing trajectory data in memory and benefiting from vectorization and parallelization (see Section VIII).

B. PYTHON AND EXTERNAL TOOL REMOVAL

Panoramic image generation for a web-based annotation tool was previously implemented as a Python script calling Hugin [16], an open-source panoramic photo stitching toolkit. The pipeline used Hugin's `nona` reprojection tool to convert equirectangular panoramic images to cube-face projections, followed by a Python tiling script that split each face into thousands of small tile files across multiple zoom levels.

This tile-based approach produced approximately 1,500–2,000 files per panoramic image. For a measurement campaign with 10,000 panoramic images, this amounts to 15–20 million small files. Managing this volume of small files creates severe overhead: on a typical network file system with 1 ms per-file latency, merely *opening* 20 million files would take 5 hours. The small average file size of a few kilobytes means that file system metadata operations dominate the actual data transfer. In practice, specially formatted disk partitions with small allocation units were used to avoid wasting disk space.

We reimplemented the panoramic generator in C#, now producing only 7 files per panoramic image instead of thousands. It outputs six cube-face JPEG images (one per face, containing all zoom levels as a power-of-two mipmap pyramid) plus one JSON metadata file with an embedded base64 thumbnail. The Pannellum [18] web viewer was configured to consume this format directly. This represents a 99.5% reduction in file count, preserving the same visual quality and zoom capability, and eliminates the Python and Hugin runtime dependencies. We measured no negative performance impact for users with typical 50 Mbps broadband connections, as the increased file sizes are offset by the reduced overhead. Each panoramic image is processed independently, enabling straightforward parallelization across all available CPU cores.

Similarly, hillshading (previously computed by the SAGA GIS [19] external tool) and image debayering (previously OpenCV [20]) were both reimplemented in C#. For image encoding, we switched to the Windows Imaging Component (WIC), which allows parallel encoding of JPEG and PNG files, and uses SIMD instructions internally.

C. VALIDATING EQUIVALENCE AND IMPROVEMENTS

Rewriting legacy code at this scale must not silently change results. Wherever the rewritten code was intended to reproduce legacy behavior, we verified equivalence automatically, by running the legacy and the optimized implementations

side-by-side on the same inputs and comparing their outputs, as with the unit tests against the MATLAB reference output described above. Where we deliberately changed behavior to improve output quality, however, such automated equivalence checks no longer apply. Instead, we manually compared outputs to ensure that every deviation is indeed an improvement. Figs. 6 and 7 show before/after comparisons for two such cases: the surface images produced from the 2D profile stream (Section VI), and the hillshaded depth images.

VIII. SIMD VECTORIZATION AND HIGH-PERFORMANCE MANAGED CODE

A recurring theme in our optimization work was that managed C# code, when written with care for data layout and vectorization, would easily exceed the performance of the original native C++ implementations. This section describes the techniques we applied.

A. HARDWARE INTRINSICS AND PLATFORM-ADAPTIVE VECTORIZATION

Modern .NET provides two complementary SIMD programming models: a cross-platform vector API that automatically uses the best available instruction set, and explicit hardware intrinsics for SSE, AVX2, and AVX-512 instructions. We used both throughout the pipeline, choosing the appropriate level based on the operation.

One of the most frequently called numerical routines in the pipeline is linear interpolation, used e.g., for georeferencing every scanner profile against the trajectory. For each of the millions of 2D and 3D profiles, the interpolation kernel computes position, heading, and elevation values at the profile's timestamp. We implemented this using AVX2 and AVX-512 intrinsics that process 8 or 16 single-precision floating-point values per instruction. The kernel computes a vector of consecutive interpolated values in a single iteration, converts them to the target format, and writes the result directly into the output buffer. A scalar fallback handles the remaining elements and platforms without AVX2 support. This pattern is applied to all bulk interpolation operations, including heading interpolation with proper angular wrap-around handling.

For image processing operations (geometric transformations, inpainting, gamma correction, debayering of Bayer RGGB raw camera data, and hillshading gradient computation) and point cloud processing, we used the platform-adaptive vector API, which automatically selects the widest available SIMD instruction set (128-bit SSE through 512-bit AVX-512) at runtime.

B. ALLOCATION AVOIDANCE AND MEMORY EFFICIENCY

On a many-core system processing data in parallel, garbage collection pauses can affect all threads simultaneously and can become a bottleneck. We avoid this problem by systematically reducing GC pressure through several techniques.

Temporary buffers for interpolation, image processing, and profile data are rented from a shared pool rather than allocated on the heap, and returned after use. This pattern is particularly impactful for the image generation stage, where each of the hundreds of images requires multi-megabyte temporary buffers. Small, frequently-instantiated data structures (such as per-point records in the point cloud pipeline) were changed from heap-allocated reference types to stack-allocated value types, eliminating GC tracking. We reduced the memory footprint of many data structures by classic techniques like bit-packing: Outlier masks, previously stored as arrays of booleans (1 byte each), were changed to bit arrays (1 bit each), reducing memory consumption eightfold.

We also configured the runtime's garbage collector for server workloads, using per-core collection heaps and concurrent collection. This proved essential: the default configuration, designed for interactive desktop applications, created contention under our parallel workloads.

During profiling, we identified a performance issue in the .NET runtime's own JIT profiling counters. We contributed a fix upstream to the .NET runtime [21], illustrating that even the runtime itself benefits from a profiling-driven optimization methodology.

C. MANAGED VS. NATIVE: A PRACTICAL OBSERVATION

An observation from our project is that managed C# implementations, when written with attention to data layout and vectorization, achieved better performance than the original native C++ and MATLAB code in our specific use cases. The point cloud transformation (see Section IX) dropped from over 30 minutes to under 1 minute, primarily due to algorithmic improvements (batched transformation matrices) that were easier to express and iterate on in managed code. The scanner data consumer pipeline in C# processes profiles faster than the C++ producer can extract them, making the native library the bottleneck. We ported several aspects of the C++ producer to C#, to alleviate the problem. A newer version of the scanner reader by the original sensor manufacturer, also in C++, proved *slower* than our managed version, suggesting that algorithm and architecture design are at least as important as language choice for this class of workload.

We do not claim that managed code is inherently faster than native code; our results reflect the specific combination of algorithm redesign, hardware-intrinsic vectorization, and reduced I/O that the managed rewrite enabled. Modern managed runtimes contain JIT compilers with auto-vectorization, bounds-check elimination for safe array access patterns, and allocation-free APIs that make it practical to write high-performance code without sacrificing safety or developer productivity.

D. RELATION TO EXISTING HIGH-PERFORMANCE LIBRARIES

Highly optimized implementations exist for many of the algorithms discussed in this paper, and the original pipeline in fact used several of them: OpenCV [20] for debayering, SAGA GIS [19] for hillshading, Hugin [16] for panoramic reprojection, MATLAB for longitudinal analysis, and a third-

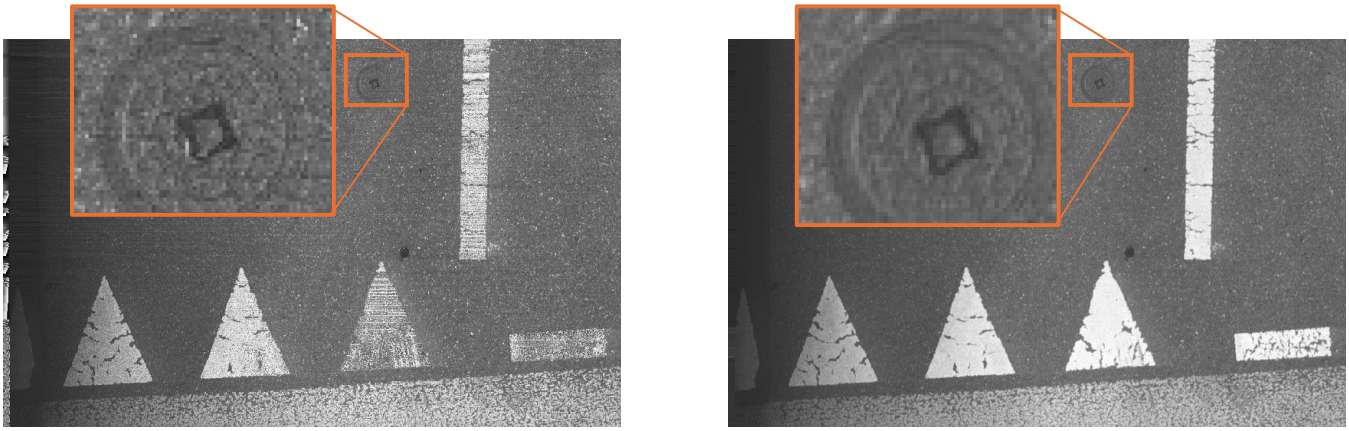

FIGURE 6. Surface image produced by the original (*left*) and optimized (*right*) processing pipelines. The original pipeline applied no point splat filtering, resulting in grainy images. The optimized pipeline added proper filtering, and tuned the outlier filter to avoid artifacts in high-reflectance regions, such as road markings.

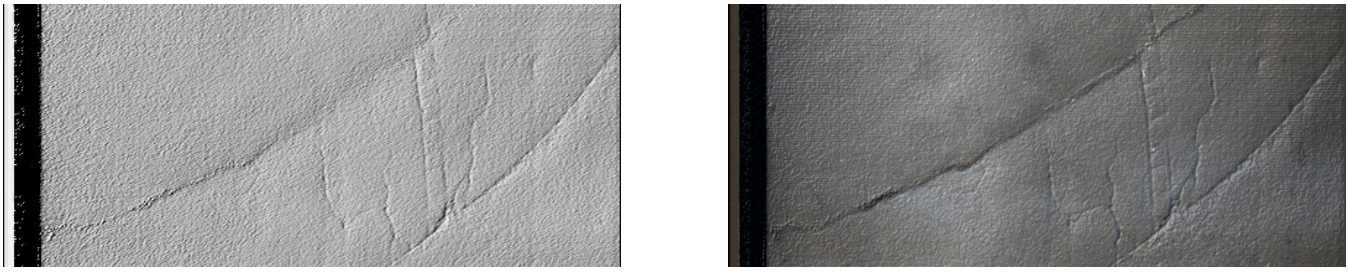

FIGURE 7. Hillshaded depth image produced by the original (*left*) and optimized (*right*) processing pipelines. In the original single-channel image, gradients parallel to the illumination direction are invisible. The optimized pipeline employs multiple subtly colored light sources to reveal depth details across all orientations, and uses significantly larger filter kernels to capture low-frequency surface variations.

party native tool for point cloud transformation. Our profiling showed that the dominant costs were not inside these kernels, but at their boundaries: process startup, serialization to intermediate files, format conversions, and redundant parsing of shared inputs. Reimplementing the comparatively simple kernels inside the managed pipeline eliminated these structural costs and enabled cross-stage data sharing that no external component could provide. The per-stage results in Table 2 also constitute a comparison of our integrated implementations against a pipeline assembled from established external tools.

IX. POINT CLOUD PROCESSING

Point cloud processing is one of the most computationally intensive stages. The CPS LiDAR scanner produces millions of 3D points per measurement segment that must be transformed from the scanner's local coordinate system to georeferenced world coordinates, filtered for outliers, and compressed to the LAZ format [22] for delivery and visualization in tools such as Potree [23].

Point cloud processing for both the surface (PPS+) and corridor (CPS) scanners was fundamentally rearchitected from a multi-pass file-based approach to a single-pass streaming pipeline. Previously, a third-party native executable performed coordinate transformation and wrote uncompressed intermediate point clouds consuming 6–25 GB/km of temporary storage, followed by a separate LAZ compression step. We replaced this with a managed C# implementation that transforms, filters, and compresses points in a single streaming pass. Each raw 3D point arriving via the callback interface from the scanner library is transformed, filtered for outliers, and written directly to a LAZ file, all without materializing the intermediate, uncompressed point cloud on disk. This eliminated multiple gigabytes of temporary I/O per measurement and removed an entire pipeline stage.

Each raw 3D point from the scanner consists of three spatial coordinates (x, y, z) in the scanner's local frame, an intensity value, and a timestamp. We used compact in-memory representations (9 bytes per PPS+ point, 24 bytes per CPS point), stored in contiguous arrays for cache-friendly access.

Georeferencing each point requires applying a rigid-body transformation (rotation + translation) derived from the vehicle's trajectory at the point's timestamp. We exploited the observation that points within a short time window (a "section" of up to 1 meter of road) share nearly identical transformations, with differences usually smaller than machine precision. By computing one transformation matrix per section and applying it to all points in that section, we amortize the cost of trajectory interpolation across hundreds or thousands of points.

After transformation, outlier points (noise from reflections, sensor artifacts, etc.) are removed using a Radius Outlier Filter [24], which only keeps points with a sufficient number of neighbors within a radius r. We implement this as a two-pass algorithm by first partitioning the point cloud into a 3D cell grid, with carefully chosen cell sizes ($r/2$). This allows us to use fast-paths for cells containing more points than the threshold, and for $5 \times 5 \times 5$-neighborhoods that contain fewer points than the threshold. The algorithm only performs an accurate neighbor count for points in the remaining cells, with a bounded number of points. Thus, the algorithm runs in $O(n)$ when density is bounded. The grid partitioning and filtering are both parallelized across sections of the point cloud.

With these optimizations, point cloud processing dropped from almost 80 minutes to under 12 seconds for the reference dataset.

X. PARALLELIZATION

Parallelization was applied at multiple levels, guided by profiling data from the timing framework. Many pipeline stages are embarrassingly parallel: topographic images for different road sections, cross-profile report images, panoramic tile generation, and camera image processing can all proceed independently. For these stages, we applied data-parallel decomposition across all available CPU cores. The 48-core processor provides substantial parallel capacity, and the profiling data confirmed that most stages scale well to this core count.

As mentioned before, we execute pipeline stages *sequentially* rather than overlapping multiple stages in parallel. The sequential execution of the pipeline stages is helpful for performance tuning. A production deployment could use overlapping to increase CPU utilization further, but this was not deemed necessary in our project. Only for batch processing of extremely large campaigns do we instantiate multiple instances of the entire pipeline to maximize CPU utilization.

XI. RESULTS

Table 2 shows the per-stage timing breakdown for the primary reference dataset (a 0.5 km urban measurement at approximately 4–5 m/s, recording duration 115 s). The "Original" column shows timings measured on the original servers; the "Optimized" column shows timings on the new server.

The total processing time of 71.4 seconds is well below the 115-second recording duration, confirming that the pipeline processes faster than it is recorded. The *Point Cloud Indexing* stage, previously the third-largest contributor at 13.1 minutes, could be eliminated because the streaming architecture described in Section VI no longer requires random data access. Scanner data is now passed in order into consumer stages via in-memory callbacks. The *Panoramic Generation*, *Trajectory Parsing*, and remaining stages (macro-texture processing, waveband analysis, cross-profile report generation, result aggregation, and database operations) were originally handled by external tools or separate pre-processing steps that each took minutes to hours; in the optimized system, they are fully integrated and complete in a combined 35 seconds.

TABLE 2. Per-stage processing times for the primary reference dataset (0.5 km, 115 s recording). Speedup reflects hardware, architecture, and code optimization combined. Run-to-run variability is below 5% per stage, giving the speedup factors a relative tolerance of approximately ±10%.

Stage	Orig. (min)	Opt. (min)	Speedup
Point Cloud Processing	79.83	0.20	399×
Topographic Imaging	69.85	0.14	499×
Transverse Analyses	14.55	0.04	364×
Point Cloud Indexing†	13.10	—	—
Longitudinal Analyses	4.48	0.16	28×
Camera Debayering	2.09	0.06	35×
Water Depth	0.13	0.01	13×
Trajectory Parsing‡	—	0.02	—
Panoramic Generation‡	—	0.15	—
Other‡	—	0.41	—
Total	**184.03**	**1.19**	**155×**

†Replaced by stream processing (Section VI); scanner data now feeds consumers directly.
‡Originally external processes (pre-processing steps or separate tools) taking minutes to hours each; now automated by the new pipeline and thus only included in the optimized measurement.

Speedup factors vary dramatically across the originally measured stages, reflecting different combinations of the optimization techniques described above. *Topographic Imaging* (499×) benefited from elimination of an external process, SIMD-vectorized hillshading, parallel image generation, and thread-safe WIC encoding. *Point Cloud Processing* (399×) benefited from streaming, vectorized coordinate transformation, and the integrated grid-based outlier filter. *Longitudinal Analyses* (28×) benefited primarily from the MATLAB-to-C# migration and SIMD vectorization of the analysis kernels.

Faster than real-time capability and campaign-scale applicability. We verified the faster than real-time processing capability on three additional reference datasets of varying length and sensor configuration (recording durations of 447 s, 115 s, and 158 s). In all cases, processing completed in 57–95% of the recording duration, confirming that the pipeline processes data faster than it is recorded. A full measurement campaign (typically tens or hundreds of kilometers) is composed of hundreds of individual road sections that can be processed independently. Because the pipeline processes each section faster than it was recorded, campaign-scale processing time scales linearly with campaign length, and the real-time guarantee holds regardless of total campaign size. No additional scalability challenges (such as memory pressure or I/O degradation) arise at campaign scale, since each road section is processed in isolation before moving to the next.

While performance was the primary focus, the consolidation and code modernization also yielded usability improvements. The processing application's user interface no longer blocks during computation. Output files are written to a consistent directory structure, which allows idempotent reprocessing. The entire codebase can be built using a single tool-chain, without manual dependency resolution. Such improvements, secondary to the performance results, were essential for making the system practical for daily use by field

engineers.

The hardware consolidation also yielded an improvement in energy efficiency. The original eight-server system drew enough power that the servers had to be powered on sequentially to avoid overloading the vehicle's electrical system. The consolidated single-server system draws approximately one quarter of the power of the original eight servers combined. Combined with the 155× reduction in processing time, the energy consumed per unit of processed data has decreased by more than two orders of magnitude.

XII. LIMITATIONS

An inherent limitation of this case study is that the hardware and software were changed simultaneously. The new server provides approximately 11× the single-threaded performance, as well as 23× the storage bandwidth and ≈1.4× the physical core count of all eight legacy servers combined (Table 1), so a portion of the 155× speedup is attributable to hardware improvements alone. We did not run the original software on the new hardware (the distributed multi-server architecture would not function on a single machine without significant adaptation), so a controlled ablation separating hardware from software contributions is not viable. The software contribution can, however, be bounded. Profiling showed that most pipeline stages are CPU-bound rather than I/O-bound, so the higher storage bandwidth is relevant for only a small share of the runtime; the governing hardware factors are single-thread performance and core count. Even under the generous assumption that the legacy software had scaled perfectly across all of its cores, hardware explains at most ≈15× (11 · 1.4). In reality, many legacy stages were single-threaded, for which the hardware bound is ≈11×. Dividing the overall 155× speedup by these conservative bounds attributes a factor of roughly 10–14× to software and architectural changes. The bounds understate the software contribution, because hardware and software are not fully separable in principle: better exploiting the new machine's cores required the restructuring described in Sections VI–X, yet the bounds credit these gains entirely to hardware. The per-stage results (Table 2) show that for key stages the software contribution is far larger still: topographic imaging (499×) and point cloud processing (399×) exceed even the most generous hardware bound by factors of more than 20, which no hardware assumption can explain.

Additionally, all timings reported are single-run measurements, as the CI benchmark infrastructure records one run per commit rather than multiple repetitions. However, as Fig. 4 shows, run-to-run variability on this infrastructure is below 5%, which results in a tolerance below 10% on the stated speedup factors.

Finally, we did not employ GPU acceleration, although several kernels (debayering, hillshading, interpolation, coordinate transformation) would be well suited to it. Once the pipeline processes data faster than it is recorded, further per-stage speedups provide no operational benefit, while a GPU code path would add a second programming model to maintain. GPU offloading of individual kernels remains an option should future sensor generations increase data rates.

XIII. CONCLUSIONS

We have presented the systematic optimization of a real-world, multi-stage LiDAR road assessment pipeline, achieving a 155× end-to-end speedup that enables the system to process data faster than it is recorded. The key insight from this work is that the majority of this improvement came not from novel algorithms, but from the disciplined application of well-known engineering principles to a legacy system that had accumulated over a decade of incremental, heterogeneous development.

The largest single contributor was the elimination of intermediate file I/O through stream processing, which removed entire pipeline stages and tens of gigabytes of temporary storage per measurement. Hardware consolidation from eight distributed servers to a single high-performance node eliminated network transfer overhead and time synchronization complexity, and enabled shared-memory processing that made further optimizations possible. The migration from MATLAB and Python to unified managed C# removed process startup penalties, enabled in-memory data sharing, and provided a single, consistent platform for SIMD vectorization and parallelization. The core numerical kernels, particularly trajectory interpolation, coordinate transformation, and image processing, were vectorized using intrinsics and platform-adaptive vector APIs, yielding per-kernel speedups of 2–8×. Parallelization across the 48 available cores exploited the embarrassingly parallel nature of most pipeline stages.

We believe several lessons from this project transfer to other legacy data processing systems. First, the dominant bottleneck is a moving target: in our pipeline it shifted repeatedly as optimization proceeded, so continuous per-commit measurement, rather than one-off profiling, was what kept optimization effort directed at the actual cost at each stage. Second, in pipelines that have grown heterogeneously over years, the largest costs are often structural rather than algorithmic (file-based intermediate storage, runtime heterogeneity, process startup, and serial execution). The highest-return interventions are architectural (stream processing in place of file exchange, runtime unification, and shared-memory consolidation) and are best applied before kernel-level micro-optimization. Third, these interventions are sequenced: consolidating eight servers into a single shared-memory node was a prerequisite that made in-memory streaming and cross-stage data sharing possible, which in turn enabled the vectorization and parallelization gains. Finally, for this class of data-parallel numerical workload, algorithm and data-layout design dominated language choice; carefully written managed code with explicit SIMD matched or exceeded the original native implementations, which suggests that the reflex to rewrite in a native language for performance is not always warranted. We do not claim these observations are universal, but they were decisive in our case and, we believe, apply to the broad class of multi-stage pipelines that accumulate

inefficiency through incremental development.

• • •

ACKNOWLEDGMENT

The authors wish to thank Fraunhofer IPM for sharing their code and algorithms, which made this project possible.

REFERENCES

[1] IMP Bautest AG, "Integrated road information system (IRIS)," https://www.impbautest.ch, 2025.
[2] M. Elhashash, H. Albanwan, and R. Qin, "A review of mobile mapping systems: From sensors to applications," *Sensors*, vol. 22, no. 11, p. 4262, 2022.
[3] X. Hu, X. Li, and Y. Zhang, "Fast filtering of LiDAR point cloud in urban areas based on scan line segmentation and GPU acceleration," *IEEE Geoscience and Remote Sensing Letters*, vol. 10, no. 2, pp. 308–312, 2013.
[4] Y. Duan, C. Yang, and H. Li, "Low-complexity adaptive radius outlier removal filter based on pca for lidar point cloud denoising," *Applied optics*, vol. 60, no. 20, pp. E1–E7, 2021.
[5] R. Pajarola, "Stream-processing points," in *IEEE Visualization 2005 (VIS '05)*, 2005, pp. 239–246.
[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 38, no. 4, pp. 28–38, 2015.
[7] W. K. G. Assunção, L. Marchezan, A. Egyed, and R. Ramler, "Contemporary software modernization: Perspectives and challenges to deal with legacy systems," *arXiv preprint*, 2024, arXiv:2407.04017.
[8] R. Hundt, "Loop recognition in C++/Java/Go/Scala," in *Proceedings of the Scala Days 2011*, Stanford, CA, 2011.
[9] S. Toub, "Performance improvements in .NET 10," https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-10/, 2025.
[10] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming*, vol. 24, no. 2, pp. 187–206, 1996.
[11] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya, "A stepwise auto-profiling method for performance optimization of streaming applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 12, no. 4, pp. 1–33, 2017.
[12] M. A. K. Azad, N. Iqbal, F. Hassan, and P. Roy, "An empirical study of high performance computing (hpc) performance bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 194–206.
[13] Meinberg Funkuhren GmbH & Co. KG, "GPS180PEX—PCI express GPS reference clock," https://www.meinberg.de/german/products/pci-express-gps-uhr.htm, 2024.
[14] Alexandre Mutel, "Ultra: An advanced profiler for .net applications on windows," https://github.com/xoofx/ultra, 2025.
[15] The MathWorks, Inc., "MATLAB compiler runtime," https://www.mathworks.com/products/compiler/matlab-runtime.html, 2024.
[16] Hugin Contributors, "Hugin—panorama photo stitcher," http://hugin.sourceforge.net/, 2024.
[17] C. Paciorek, "Parallel processing in MATLAB," UC Berkeley Statistical Computing Facility Tutorial, https://computing.stat.berkeley.edu/tutorial-parallelization-original/parallel-matlab.html, 2012.
[18] M. A. Petroff, "Pannellum: a lightweight web-based panorama viewer," *Journal of Open Source Software*, vol. 4, no. 40, p. 1628, 2019.
[19] O. Conrad, B. Bechtel, M. Bock, H. Dietrich, E. Fischer, L. Gerlitz, J. Wehberg, V. Wichmann, and J. Böhner, "System for automated geoscientific analyses (SAGA) v. 2.1.4," *Geoscientific Model Development*, vol. 8, no. 7, pp. 1991–2007, 2015.
[20] G. Bradski, "The OpenCV library," *Dr. Dobb's Journal of Software Tools*, 2000.
[21] S. Felix, "JIT_CountProfile: Avoid BSR instruction for low counts," .NET Runtime Pull Request #110258, https://github.com/dotnet/runtime/pull/110258, 2024.
[22] M. Isenburg, "LASzip: lossless compression of LiDAR data," *Photogrammetric Engineering & Remote Sensing*, vol. 79, no. 2, pp. 209–217, 2013.
[23] M. Schütz, "Potree: Rendering large point clouds in web browsers," Diploma Thesis, TU Wien, 2016.
[24] R. B. Rusu and S. Cousins, "3d is here: Point cloud library (pcl)," in *2011 IEEE international conference on robotics and automation*. IEEE, 2011, pp. 1–4.

SIMON FELIX received the M.Sc. degree in Computer Science from the University of Applied Sciences Northwestern Switzerland (FHNW), Windisch, Switzerland. He is a Senior Research Fellow and Lecturer at the Institute for Data Science, FHNW. His research interests include software performance optimization, scientific data processing, linear and nonlinear optimization, and computer graphics.

MANUEL STUTZ received the M.Sc. degree in Data Science from the University of Applied Sciences Northwestern Switzerland (FHNW), Windisch, Switzerland. He is a Lecturer and Research Associate at the High Performance Computing Lab of the Institute for Data Science, FHNW.

JULIA HARTMANN received the M.Sc. degree in Data Science from the University of Applied Sciences Northwestern Switzerland (FHNW), Windisch, Switzerland. She is currently a researcher, project manager, and developer at Ateleris GmbH, Switzerland.

YVES BÄCHTIGER received the B.Sc. degree in Computer Science from the University of Applied Sciences Northwestern Switzerland (FHNW), Windisch, Switzerland. He is currently a Research Associate at the Institute for Data Science, FHNW.

VINCENZO TIMMEL received the B.Sc. degree in Data Science from the University of Applied Sciences Northwestern Switzerland (FHNW), Windisch, Switzerland. He is currently a Research Associate at the Institute for Data Science, FHNW.

ANDREAS WASSMER received his M.Sc. degree in Physics from the University of Zurich (UZH), Zurich, Switzerland. He is currently a Research Associate and Lecturer at the Institute for Data Science, FHNW, Switzerland.

FILIP SCHRAMKA received the M.Sc. degree in Computer Science from the University of Applied Sciences Northwestern Switzerland (FHNW), Windisch, Switzerland. He is currently a researcher, project manager, and developer at Ateleris GmbH, Switzerland.

ELMAR STROBACH received a doctoral degree in Exploration Geophysics from the Curtin University, Perth, Australia. He is currently the head of department for non-destructive testing and innovation at IMP Bautest AG, Switzerland.

JÜRG LECKEBUSCH received a doctoral degree in Prehistory and Geophysics from the University of Zurich, Switzerland and the University of Bradford, England. He is currently the head of road condition assessment at IMP Bautest AG, Switzerland.

CHRISTIAN ANGST studied civil engineering at the Swiss Federal Institute of Technology ETH in Zurich, Switzerland and was awarded a PhD in Engineering. He is currently serving as chairman of the Board of Directors at IMP Bautest AG, Switzerland.