Taptapp AR (Protocol V7): Ultra-Compact Feature Extraction for Mobile Augmented Reality

Sergio Lázaro
Taptapp Labs
sergiolazaromondargo@gmail.com

Abstract

Image-based Augmented Reality (AR) systems often suffer from large data payloads and slow initialization performance on the web. High-performance solutions traditionally rely on heavy machine learning libraries like TensorFlow.js, introducing significant overhead. This paper presents **Taptapp AR (Protocol V7)**, a completely re-engineered pipeline that achieves a **93% reduction** in target file size and a **9x speedup** in compilation compared to the state-of-the-art MindAR library. We introduce the **Moonshot Vision Codec**, comprising **64-bit Locality Sensitive Hashing (LSH)** for descriptors, **4-bit packed optical flow data**, and **Uint16 coordinate quantization**. By eliminating TensorFlow.js in favor of a parallelized, pure JavaScript architecture with hardware-accelerated binary matching, we achieve sub-50KB file sizes and near-instant detection ($\sim$21ms) on mobile devices, redefining expectations for web-based AR performance. We also introduce **DetectorLite**, a highly optimized runtime engine that reduces initialization time from seconds to milliseconds (<20ms) and enables 60FPS tracking on mid-range devices.

Keywords— Augmented Reality, Feature Detection, Computer Vision, WebAssembly, JavaScript Optimization, Mobile AR

1 INTRODUCTION

Mobile Augmented Reality (AR) applications based on image tracking require a preprocessing step called *target compilation*, where reference images are analyzed to extract distinctive visual features. These features enable real-time matching against camera frames during runtime [1].

The dominant commercially available open-source solution, MindAR [1], employs TensorFlow.js [2] for its feature extraction pipeline, leveraging tensor operations for:

1. Gaussian pyramid construction via 2D convolutions
2. Difference of Gaussians (DoG) computation
3. Local extrema detection across scale-space
4. FREAK binary descriptor generation [3]

While TensorFlow.js provides hardware acceleration, it introduces critical limitations for robust web deployment:

- **Initialization overhead**: Cold start times of 1.5-3 seconds due to shader compilation.
- **Compatibility issues**: `tfjs-node` stability issues on modern Node.js versions.
- **Worker limitations**: Difficulty initializing WebGL contexts within Worker threads.
- **Dependency bloat**: Over 500MB of native binaries for server-side compilation or 20MB+ for client-side.

This paper makes the following contributions:

1. A complete pure JavaScript reimplementation of the DoG feature detector with parallel `WorkerThread` execution.
2. **Moonshot Vision Codec (V7)**: A novel binary format using **64-bit LSH** user descriptors and **4-bit packed** tracking data.
3. **Coordinate Quantization**: Reducing 32-bit floats to normalized 16-bit integers.
4. **DetectorLite**: A runtime detection engine optimized for zero-latency initialization and high-frequency tracking loop ($\sim$60FPS).
5. Evidence that these optimizations reduce file size by **93%** (from $\sim$770KB to $\sim$50KB) and compilation time by **9x** (from $\sim$23s to $\sim$2.6s).

2 METHODOLOGY

2.1 Problem Formulation

Given an input grayscale image I of dimensions $W \times H$, the goal is to extract a set of features $\mathscr{F} = \{(x_i, y_i, \sigma_i, \theta_i, \mathbf{d}_i)\}$ where (x, y) are coordinates, σ is scale, θ is orientation, and $\mathbf{d}$ is a descriptor.

2.2 Algorithmic Pipeline

Our implementation, **DetectorLite**, follows a multi-stage pipeline optimized for V8 (Chrome/Node) execution:

2.2.1 Stage 1: Gaussian Pyramid

We construct an octave-based pyramid using a separated 5-tap kernel $[1,4,6,4,1]/16$. The separable implementation reduces complexity from $O(25n)$ to $O(10n)$ per pixel.

2.2.2 Stage 2: Difference of Gaussians

For each octave o, we compute $D_o(x,y) = G_{o,2}(x,y) - G_{o,1}(x,y)$.

2.2.3 Stage 3: Moonshot Codec (Protocol V7)

To achieve the "Moonshot" efficiency goal, we apply three layers of aggressive compression:

64-bit LSH Descriptors We project the 512-bit binary FREAK descriptor onto a 64-bit subspace using Locality Sensitive Hashing (LSH). This reduces the descriptor footprint from 84 bytes (float representation) to just **8 bytes**. Matching is performed via the Hamming distance $d_H(H_1,H_2) = \text{popcount}(H_1 \oplus H_2)$, utilizing hardware instructions for maximum speed.

4-bit Packed Tracking Data The optical flow algorithm requires pixel intensity data. Instead of storing full 8-bit grayscale values, we compress effective tracking pixels into **4-bit** nibbles, packing two pixels per byte. This yields a 50% direct size reduction with negligible impact on tracking accuracy for standard texture features.

Coordinate Quantization Feature coordinates (x,y) are typically stored as 32-bit floats. We normalize these to the unit interval $[0,1]$ and quantize them to 16-bit unsigned integers $(0\ldots65535)$. This halves the coordinate storage requirement while maintaining sufficient sub-pixel precision for mobile screens.

2.2.4 Stage 4: Runtime Detection (DetectorLite)

The runtime engine was re-architected to prioritize responsiveness:

- **Zero-Shader Initialization**: Unlike TFJS, which requires compiling GLSL shaders (causing 1-3s freeze), DetectorLite initializes in pure CPU memory in under 20ms.
- **Float32 Optical Flow**: Tracking utilizes high-precision optical flow on 4-bit packed textures, expanded on-the-fly to Float32 during the `requestAnimationFrame` loop.
- **Predictive Pose Estimation**: A OneEuroFilter is applied to the output matrix to smooth high-frequency jitter while maintaining low latency responsiveness.

3 EXPERIMENTAL SETUP

3.1 Test Environment

- **Hardware**: Apple M1 Pro (10-core CPU).
- **Software**: Node.js 22.1.0, macOS 15.2.
- **Baseline**: MindAR v1.2.5 with `tfjs-node`.

3.2 Metrics

1. **Compilation time**: Wall-clock time for tracking feature extraction.
2. **Payload Size**: Total size of the generated target file (Gzip).
3. **Runtime Memory**: Heap usage during compilation.

4 RESULTS

4.1 Performance Comparison

Table 1 compares the performance of the proposed Protocol V7 against the legacy MindAR implementation.

Table 1: Comparison: MindAR (Legacy) vs Taptapp V7

Metric	MindAR	Taptapp AR V7	Improvement
Build Time	~23.50s	**2.61s**	9x Faster
File Size	~770 KB	**~50 KB**	93% Smaller
Descriptor	84-byte Float	**64-bit LSH**	Massive
Tracking Data	8-bit Gray	**4-bit Packed**	50%
Dependency	20MB (TFJS)	**<100KB**	99%
Init Time (Cold)	~2500ms	**~20ms**	Instant
Latency (Detect)	~40ms	**~21ms**	2x Faster

The results demonstrate order-of-measure improvements across all key metrics. The elimination of TensorFlow.js overhead is the primary contributor to the compilation speedup, while the LSH and packing strategies drive the file size reduction.

4.2 Multi-Image Scalability

The lightweight nature of the new architecture allows for efficient parallelization.

Table 2: Batch compilation performance (4 images)

Phase	Time	Percentage
Matching (Features)	5.127s	91.5%
Tracking (Template)	0.465s	8.5%
Total	**5.600s**	100%

5 DISCUSSION

5.1 Why JavaScript Outperforms TensorFlow

Counter-intuitively, our pure JavaScript implementation outperforms the TensorFlow-based approach for this specific application due to:

1. **Eliminated overhead**: No tensor allocation, backend switching, or kernel compilation.
2. **Specialized algorithms**: Our implementation is tailored specifically for DoG, avoiding generic tensor operations.
3. **V8 optimization**: Modern JavaScript engines apply JIT compilation, making hot loops highly efficient.
4. **Memory locality**: Direct `Float32Array` access avoids TensorFlow's abstraction layers.

6 CONCLUSION

We presented Taptapp AR (Protocol V7), a radical re-engineering of the AR pipeline. By combining parallelized pure JavaScript processing with the novel Moonshot Vision Codec (64-bit LSH, 4-bit packing), we achieved a 93% reduction in file size and order-of-magnitude faster compilation compared to existing solutions.

AVAILABILITY

The complete implementation is available open-source at: `https://github.com/srsergiolazaro/taptapp-ar`

Published as npm package: `@srsergio/taptapp-ar`

References

[1] H. Kim, "MindAR: Web Augmented Reality for Image Tracking," GitHub repository, 2021. `https://github.com/hiukim/mind-ar-js`

[2] D. Smilkov et al., "TensorFlow.js: Machine Learning for the Web and Beyond," *arXiv preprint arXiv:1901.05350*, 2019.

[3] A. Alahi et al., "FREAK: Fast Retina Keypoint," in *CVPR*, 2012, pp. 510-517.