SIMD-Accelerated JSON Diff, Patch, and Merge

Darach Ennis

January 2026

Abstract

"What is changed is not lost."

— W.B. Yeats

We present a high-performance implementation of JSON structural operations-diff, patch, and merge-with SIMD acceleration for Rust. By leveraging AVX2, SSE2, and NEON instruction sets for byte-level comparison, we achieve **3.7-4x speedup** for identical document detection and **1.7x improvement** for large document diffing compared to the `json-patch` crate. The implementation provides full RFC 6902 (JSON Patch) and RFC 7396 (JSON Merge Patch) compliance while maintaining cross-platform compatibility.

Contents

1 SIMD-Accelerated JSON Diff, Patch, and Merge **2**
- 1.1 1. Introduction . . . 2
- 1.2 2. Architecture . . . 2
  - 1.2.1 2.1 SIMD Comparison Layer . . . 2
  - 1.2.2 2.2 Fast Equality Short-Circuit . . . 2
  - 1.2.3 2.3 Diff Algorithm . . . 3
- 1.3 3. Benchmark Results . . . 3
  - 1.3.1 3.1 Test Environment . . . 3
  - 1.3.2 3.2 JSON Diff Performance . . . 3
  - 1.3.3 3.3 Patch Application . . . 4
  - 1.3.4 3.4 Throughput Summary . . . 4
- 1.4 4. RFC Compliance . . . 4
  - 1.4.1 4.1 RFC 6902 (JSON Patch) . . . 4
  - 1.4.2 4.2 RFC 7396 (JSON Merge Patch) . . . 4
- 1.5 5. API Reference . . . 5
  - 1.5.1 5.1 Diff Functions . . . 5
  - 1.5.2 5.2 Patch Functions . . . 5
  - 1.5.3 5.3 Merge Functions . . . 5
  - 1.5.4 5.4 SIMD Utilities . . . 5
- 1.6 6. Cross-Platform Support . . . 6
- 1.7 7. Use Cases . . . 6
  - 1.7.1 7.1 Document Caching . . . 6
  - 1.7.2 7.2 Real-Time Collaboration . . . 6
  - 1.7.3 7.3 Configuration Management . . . 6

1.8 8. Limitations and Future Work 6
1.8.1 8.1 Current Limitations 6
1.8.2 8.2 Future Optimizations 7
1.9 9. Conclusion 7
1.10 References 7
1.11 Appendix: Raw Benchmark Data 7

1 SIMD-Accelerated JSON Diff, Patch, and Merge

1.1 1. Introduction

JSON structural operations are fundamental to modern distributed systems:

- **Version control** uses diff/patch for document history
- **Real-time collaboration** requires efficient merge operations
- **API synchronization** relies on patch application for incremental updates
- **Caching systems** need fast change detection

Traditional implementations process JSON through recursive tree traversal with string-by-string comparison. This approach leaves performance on the table, particularly for identical or near-identical documents-a common scenario in caching and synchronization workloads.

We present `fionn::diff`, a module applying SIMD acceleration to JSON comparison, achieving **multi-gigabyte-per-second throughput** for change detection.

1.2 2. Architecture

1.2.1 2.1 SIMD Comparison Layer

The foundation is a set of SIMD-accelerated byte comparison functions:

```rust
/// SIMD-accelerated byte slice equality check.
pub fn simd_bytes_equal(a: &[u8], b: &[u8]) -> bool;

/// Find first position where slices differ.
pub fn simd_find_first_difference(a: &[u8], b: &[u8]) -> Option<usize>;
```

Implementation Strategy:

1. **Runtime feature detection** with cached results via `OnceLock`
2. **Tiered dispatch**: AVX2 (32 bytes) -> SSE2 (16 bytes) -> NEON -> scalar
3. **Chunked processing**: Process 32/16 bytes per SIMD iteration
4. **Tail handling**: Scalar comparison for remaining bytes

1.2.2 2.2 Fast Equality Short-Circuit

The diff algorithm begins with fast equality detection:

```rust
fn values_equal_fast(a: &Value, b: &Value) -> bool {
    match (a, b) {
        (Value::String(a), Value::String(b)) => {
            simd_bytes_equal(a.as_bytes(), b.as_bytes())
```

```
        }
        (Value::Array(a), Value::Array(b)) => {
            a.len() == b.len() &&
            a.iter().zip(b).all(|(x, y)| values_equal_fast(x, y))
        }
        (Value::Object(a), Value::Object(b)) => {
            a.len() == b.len() &&
            a.iter().all(|(k, v)| b.get(k).is_some_and(|bv| values_equal_fast(v, bv)))
        }
        _ => a == b
    }
}
```

This enables **early termination** when subtrees are identical, avoiding unnecessary patch generation.

1.2.3 2.3 Diff Algorithm

1. Call `json_diff(src, tgt)`
2. Check `values_equal_fast(src, tgt)` (SIMD-accelerated)
3. If true: skip (no patch needed)
4. If Objects: compare fields recursively, generate add/remove/replace
5. If Arrays: simple diff or LCS-based optimization

Object Comparison: - Iterate source keys, generate `remove` for missing in target - Iterate target keys, generate `add` for new keys - Recurse on keys present in both

Array Comparison: - Simple mode: Compare common prefix, handle additions/removals at ends - LCS mode: O(n*m) Longest Common Subsequence for middle insertions

1.3 3. Benchmark Results

1.3.1 3.1 Test Environment

- **Platform**: x86_64 Linux (Intel CPU with AVX2)
- **Rust**: Edition 2024, release profile with LTO
- **Baseline**: `json-patch` crate v4.1.0
- **Framework**: Criterion.rs with 100 samples per benchmark

1.3.2 3.2 JSON Diff Performance

Scenario	fionn	json-patch	Speedup
identical_small (74B)	25.1 ns / **2.9 GiB/s**	93.4 ns / 796 MiB/s	**3.7x**
identical_medium (13KB)	3.57 µs / **3.6 GiB/s**	14.1 µs / 935 MiB/s	**4.0x**
small_field_change	208 ns / 357 MiB/s	122 ns / 608 MiB/s	0.59x
medium_field_add	13.6 µs / 282 MiB/s	8.0 µs / 477 MiB/s	0.59x
large_document (217KB)	137 µs / **1.54 GiB/s**	236 µs / 919 MiB/s	**1.72x**

Key Findings:

1. **Identical documents**: SIMD comparison provides **3.7-4x speedup** by detecting equality at byte level without recursive traversal
2. **Large documents**: Even with changes, SIMD-accelerated subtree equality checks provide **1.7x improvement**
3. **Small changes**: The baseline is faster for trivial cases due to lower function call overhead-our implementation optimizes for the common case (no change or large documents)

1.3.3 3.3 Patch Application

Scenario	fionn	json-patch	Ratio
small_field_change	141 ns	151 ns	**1.07x**
medium_field_add	17.5 us	14.9 us	0.85x
array_append	1.24 us	1.03 us	0.83x
deep_nested_change	553 ns	481 ns	0.87x

Patch application shows comparable performance. The `json-patch` crate has slightly optimized hot paths for simple operations.

1.3.4 3.4 Throughput Summary

Operation	fionn Peak	Baseline Peak
Identical detection	**3.6 GiB/s**	935 MiB/s
Large document diff	**1.54 GiB/s**	919 MiB/s
Patch application	264 MiB/s	279 MiB/s

1.4 4. RFC Compliance

1.4.1 4.1 RFC 6902 (JSON Patch)

Full support for all operations:

```
[
  { "op": "add", "path": "/foo", "value": "bar" },
  { "op": "remove", "path": "/baz" },
  { "op": "replace", "path": "/qux", "value": 42 },
  { "op": "move", "from": "/old", "path": "/new" },
  { "op": "copy", "from": "/source", "path": "/dest" },
  { "op": "test", "path": "/check", "value": true }
]
```

Paths use JSON Pointer syntax (RFC 6901) with proper `~0`/`~1` escaping.

1.4.2 4.2 RFC 7396 (JSON Merge Patch)

```
pub fn json_merge_patch(target: &Value, patch: &Value) -> Value;
pub fn merge_many(documents: &[Value]) -> Value;
```

Semantics: - Objects are recursively merged - `null` values indicate deletion - Other values replace existing ones

1.5 5. API Reference

1.5.1 5.1 Diff Functions

```rust
/// Generate JSON Patch transforming source into target.
pub fn json_diff(source: &Value, target: &Value) -> JsonPatch;

/// Generate patch with options (move detection, LCS optimization).
pub fn json_diff_with_options(
    source: &Value,
    target: &Value,
    options: &DiffOptions
) -> JsonPatch;

pub struct DiffOptions {
    pub detect_moves: bool,
    pub detect_copies: bool,
    pub optimize_arrays: bool,  // Use LCS
}
```

1.5.2 5.2 Patch Functions

```rust
/// Apply patch, returning new value.
pub fn apply_patch(target: &Value, patch: &JsonPatch) -> Result<Value, PatchError>;

/// Apply patch in-place.
pub fn apply_patch_mut(target: &mut Value, patch: &JsonPatch) -> Result<(), PatchError>;
```

1.5.3 5.3 Merge Functions

```rust
/// RFC 7396 merge patch.
pub fn json_merge_patch(target: &Value, patch: &Value) -> Value;

/// Merge multiple documents.
pub fn merge_many(documents: &[Value]) -> Value;

/// Deep merge (preserves nulls, non-RFC).
pub fn deep_merge(base: &Value, overlay: &Value) -> Value;
```

1.5.4 5.4 SIMD Utilities

```rust
/// SIMD byte equality check.
pub fn simd_bytes_equal(a: &[u8], b: &[u8]) -> bool;

/// Find first differing byte position.
pub fn simd_find_first_difference(a: &[u8], b: &[u8]) -> Option<usize>;
```

1.6 6. Cross-Platform Support

Architecture	SIMD Path	Fallback
x86_64	AVX2 -> SSE2	Scalar
aarch64	NEON	Scalar
Other	-	Scalar

Feature detection is performed at runtime and cached:

```rust
static HAS_AVX2: OnceLock<bool> = OnceLock::new();

fn has_avx2() -> bool {
    *HAS_AVX2.get_or_init(|| is_x86_feature_detected!("avx2"))
}
```

1.7 7. Use Cases

1.7.1 7.1 Document Caching

The 3.7-4x speedup for identical detection is ideal for cache invalidation:

```rust
fn should_update_cache(cached: &Value, incoming: &Value) -> bool {
    !json_diff(cached, incoming).is_empty()
}
```

1.7.2 7.2 Real-Time Collaboration

Minimal diffs reduce network bandwidth:

```rust
let patch = json_diff_with_options(
    local, remote,
    &DiffOptions::default().with_array_optimization()
);
```

1.7.3 7.3 Configuration Management

Merge configuration layers:

```rust
let config = merge_many(&[base, env_specific, local_overrides]);
```

1.8 8. Limitations and Future Work

1.8.1 8.1 Current Limitations

- Memory allocation for each diff/patch operation
- No streaming mode-entire documents must fit in memory
- LCS algorithm is O(n*m) for large arrays

1.8.2 8.2 Future Optimizations

- **Arena allocation**: Reduce allocation overhead
- **Tape-based diff**: Operate on simd-json tape directly
- **Parallel subtree comparison**: Use rayon for large trees

1.9 9. Conclusion

The `fionn::diff` module demonstrates that SIMD acceleration provides substantial benefits for JSON structural operations:

- **Change detection**: 3.7-4x faster for identical documents
- **Large documents**: 1.7x improvement at 1.5+ GiB/s throughput
- **Full RFC compliance**: RFC 6902 and RFC 7396 support
- **Cross-platform**: x86_64, aarch64 with automatic fallback

For workloads dominated by change detection and large document comparison-common in caching, synchronization, and collaboration systems-fionn offers significant performance advantages.

1.10 References

1. RFC 6902 - JavaScript Object Notation (JSON) Patch
2. RFC 7396 - JSON Merge Patch
3. RFC 6901 - JavaScript Object Notation (JSON) Pointer
4. simd-json: Parsing gigabytes of JSON per second
5. json-patch crate (v4.1.0)

1.11 Appendix: Raw Benchmark Data

```
json_diff/fionn/identical_small       time: [25.1 ns]   thrpt: [2.9 GiB/s]
json_diff/json_patch_crate/identical_small time: [93.4 ns]  thrpt: [796 MiB/s]

json_diff/fionn/identical_medium      time: [3.57 µs]   thrpt: [3.6 GiB/s]
json_diff/json_patch_crate/identical_medium time: [14.1 µs]  thrpt: [935 MiB/s]

json_diff/fionn/large_document        time: [137 µs]    thrpt: [1.54 GiB/s]
json_diff/json_patch_crate/large_document time: [236 µs]    thrpt: [919 MiB/s]

apply_patch/fionn/small_field_change  time: [141 ns]    thrpt: [264 MiB/s]
apply_patch/json_patch_crate/small_field_change time: [151 ns] thrpt: [247 MiB/s]
```