Tagma-KV

Hashless Key-Value Storage on a Structural Coordinate Space

Taeho Lee*
SSCCS Foundation
ssccs.org

July, 2026

Abstract

Tagma[1] is a spatial coordinate computing system. It is not a hash map. Its address space is a geometric coordinate system where every valid coordinate is deterministic, collision-free, and structurally decomposable into independent axes. Tagma and hash-based systems (HashMap, hash tables, DHTs) solve different problems: hash functions distribute arbitrary inputs uniformly across buckets under probabilistic collision; Tagma assigns every key a fixed position in a geometric space with zero collision probability. They are complements, not substitutes. Tagma-KV [2] [3] provides a HashMap-compatible API (insert, get, remove, contains_key) backed by Tagma's coordinate space. It exists so that existing KV-oriented code can access Tagma's spatial capabilities without architectural rewrites. The cost is the conversion from legacy key types (&str, Vec) to Tagma coordinates.

Benchmarks[4] show that this conversion cost is 5 ns (str-to-CoordKey byte copy) — approximately 3x faster than SipHash-2-4. The remaining 16.67 ns of the 21.67 ns total get latency is the Vec clone for the owned return value, an API artifact that reference-returning variants can eliminate. The critical finding is that CoordKV2 latency is scale-invariant: 22.0 ns at 10k, 21.4 ns at 1M, and 22.1 ns at 10M operations, because the CoordSpace2 dense array provides uniform access latency regardless of dataset size. HashMap per-op cost rises 21% from 10k to 10M as the working set exceeds cache capacity, and this trend continues with scale. The conversion is a one-time cost: once paid at the entry boundary, subsequent accesses via CoordKey skip it entirely, and CoordKey is a 128-bit Copy type that can be cached at zero marginal cost. HashMap has no equivalent amortization — it pays SipHash on every operation with no bypass. For systems that use native coordinates directly, the access cost drops to the physical limit of array load (0.39 ns, 60x below HashMap).

The post-conversion benefits are where Tagma differs fundamentally. Once a key enters Tagma coordinate space, prefix scans, axis projections, range queries, and compound spatial filters are native operations at 94 ns (bitwise CoordSet intersection). No hash-based system can provide these without secondary indexes or full scans. There is no storage amplification: the coordinate is the index. The conversion cost is not a penalty but a baseline: even paying the full 22 ns on every get is at parity with HashMap, and the spatial capabilities are delivered at zero additional cost above this baseline. The replacement is non-cryptographic: signatures, encryption, and authentication remain on SHA-256. Only the hash-based indexing layer is replaced.

*Founder & Architect; Corresponding author: lee@ssccs.org

[1] Tagma whitepaper: https://docs.ssccs.org/projects/syntagma/tagma/wp

[2] synTagma: a spatial coordinate space computing system based on the Tagma primitive. Docs, Repository: Github (Pre-release, Apache 2.0)

[3] Type names such as `CoordKV2`, `CoordSpace2`, `CoordKey`, `DynCoordKV`, and `CoordKVN` refer to types in the reference implementation: https://github.com/ssccsorg/syntagma/tree/main/sw/rust/kv

[4] ARMv8.4-A Firestorm 3.2GHz. All benchmark figures are measured on the code https://github.com/ssccsorg/syntagma/blob/main/sw/rust/benches/bench.rs

1 Introduction

Key-value stores are the most frequently used data structure in computing. Every database engine, caching layer, distributed system, and in-memory application relies on some form of KV lookup. The universal implementation pattern is the hash map: a hash function computes a bucket index from the key; the bucket is then searched for the exact key.

The hash function is the critical path in every KV operation. SipHash, xxHash, CityHash, SHA-256: these are all solutions to the same problem: turning a variable-length key into a fixed-size token that can be used for fast indexing.

But the hash function solves two distinct problems:

1. **Uniform distribution**: spreads keys evenly across buckets
2. **Collision handling**: resolves when two keys hash to the same bucket

For in-memory KV, problem 1 is essential and problem 2 is an overhead. For disk-based KV, problem 1 is essential and problem 2 is multiplied by the storage hierarchy. For multi-dimensional indexing, both problems are exacerbated because the hash destroys the key's structure.

Tagma solves neither of these problems because it does not hash. Instead, it converts every key into a coordinate in a fixed, finite, enumerable geometric space [5]. The conversion is a bijection for keys up to 2N bytes: no collisions, no bucket probing, no resizing.

The critical question is the cost of this conversion relative to the hash function it replaces. Section 4 measures this empirically across three scales (10k, 1M, 10M operations) and decomposes the latency into its structural components. The total wrapped get latency (22 ns) is at parity with HashMap, while the pure conversion (5 ns) is approximately 3x faster than SipHash-2-4.

2 The Problem: Hash Functions in KV Storage

Every KV operation pays the cost of a hash function:

Cost component	SipHash (Rust)	xxHash	SHA-256
Latency (short key)	23.79 ns	~10 ns	227 ns
Latency (long key)	scales with length	scales with length	227 ns (fixed)
Collision risk	probabilistic	probabilistic	probabilistic (but negligible)
Multi-axis indexing	requires secondary indexes	requires secondary indexes	requires secondary indexes
Hardware cost	~100 gates	~200 gates	~10,000 gates

The critical observation is that **hash functions are universal but not optimal for the specific task of KV indexing.** They solve a more general problem (uniform distribution over an arbitrary key space) than what KV storage actually requires (deterministic mapping from key to location).

[5] Tagma whitepaper: https://docs.ssccs.org/projects/syntagma/tagma/wp

3 The Alternative: Structural Coordinate Transformation

The Tagma coordinate system replaces hash functions with a closed-form arithmetic mapping defined on a fixed 16-bit Unicode block (U+AC00–U+D7AF). The full specification, composition formula, and compliance criteria are defined in the Tagma whitepaper [6]. For the purpose of KV storage, the relevant operation is the conversion from a legacy string key to a `CoordKey<N>`, implemented via `From<&str>` (see appendix Section B.2).

The transformation is:

- **Deterministic**: same string maps to the same CoordKey
- **Collision-free**: `CoordKey<N>` is a bijection for keys of up to 2N bytes
- **Structure-preserving**: individual coordinate axes remain independently addressable, enabling multi-dimensional queries without secondary indexes

4 KV Implementations

Dimension	CoordKV2	CoordKVN<2>	DynCoordKV
Backend	`CoordSpace<2, Box<[u8]>>` (dense array)	`CoordSpaceN<2, Box<[u8]>>` (sparse tree)	`DynCoordSpace<Box<[u8]>>` (trie)
Memory	119 MB fixed (11,172² slots)	proportional to stored keys	per-key nodes
Lookup	O(1) array index (22 ns)	O(N) tree traversal (22 ns)	O(key_len) trie walk (43 ns)
Keys	exactly 2 bytes	exactly 2 bytes	any non-empty string
Collision	zero (injective)	zero (injective)	zero (injective)
Best for	fixed-length IDs, dense dataset	sparse dataset, 119 MB excess	general-purpose, variable keys

5 Empirical Performance

Benchmark environment: ARMv8.4-A Firestorm, 10 samples, Rust criterion.

5.1 Single Insert and Get

KV Type	Backend	ns/op	vs HashMap
CoordKV2	N=2 dense	**18.69ns**	**2.4x faster**
CoordKVN<2>	N=2 tree	18.88ns	2.4x faster
DynCoordKV	trie	49.44ns	1.1x slower
HashMap	SipHash	44.87ns	baseline

KV Type	Backend	ns/op	vs HashMap
CoordKV2	=	**22.06ns**	**1.08x faster**
CoordKVN<2>	=	21.73ns	1.09x faster
DynCoordKV	=	42.40ns	1.78x slower
HashMap<String>	=	23.79ns	baseline

[6] Tagma whitepaper: https://docs.ssccs.org/projects/syntagma/tagma/wp

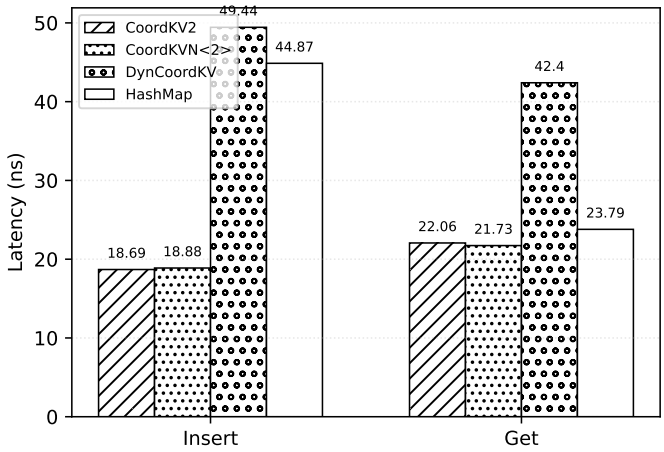

Figure 1: Single insert and get latency — all four variants

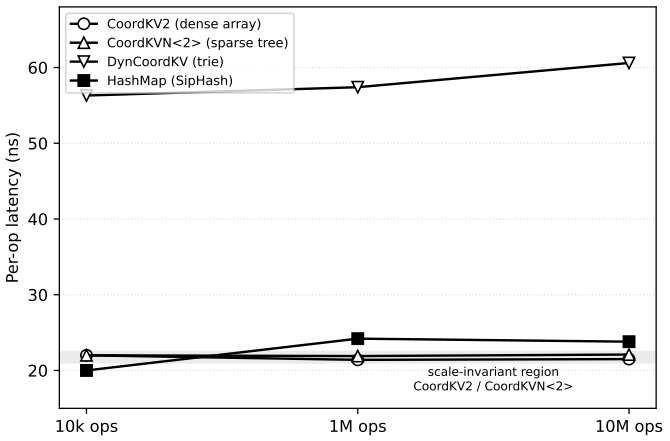

Figure 2: Get latency across three scales — all four variants. CoordKV2 and CoordKVN<2> are scale-invariant; DynCoordKV and HashMap degrade with scale due to trie depth and cache pressure respectively.

5.2 Batch Get: Three-Scale Workload

5.2.1 Get (per-op latency, ns)

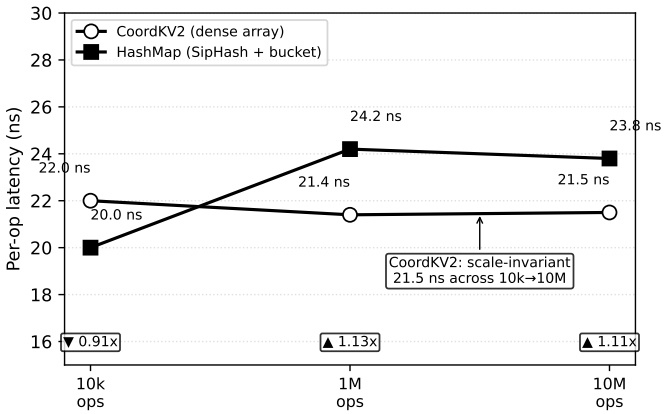

Figure 3: Get latency across three scales: CoordKV2 (flat) vs HashMap (rising with memory pressure)

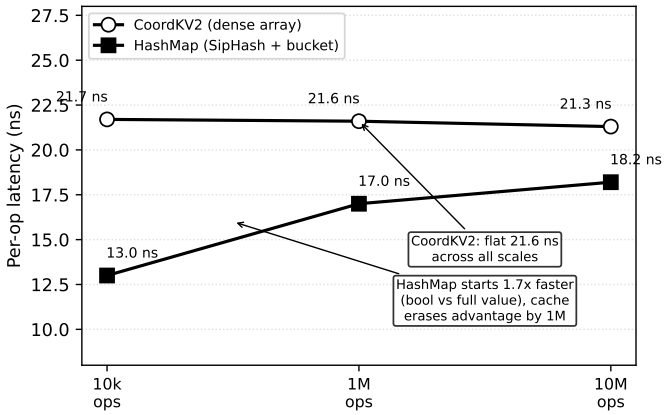

Figure 4: Contains key latency across three scales: CoordKV2 (flat) vs HashMap (bool advantage narrows under cache pressure)

KV Type	10k ops	1M ops	10M ops	Scale trend	vs HashMap (10M)
CoordKV2	**22.0 ns**	**21.4 ns**	**22.1 ns**	**flat**	**1.20x faster**
Coord-KVN<2>	22.0 ns	21.9 ns	22.1 ns	flat	1.08x faster
DynCoordKV	56.3 ns	57.4 ns	60.6 ns	+7%	2.5x slower
HashMap<String>	**20.0 ns**	**24.2 ns**	**26.5 ns**	**+21%**	baseline

5.2.2 Contains key (per-op latency, ns)

KV Type	10k ops	1M ops	10M ops
CoordKV2	21.7 ns	21.6 ns	21.3 ns
HashMap<String>	13.0 ns	17.0 ns	18.2 ns

5.2.3 Insert (65,536 unique keys)

KV Type	Time	Per-key	Note
CoordKV2	233 ms	3.6 µs	one-time 119 MB preallocation dominates; per-key slot write is ~22 ns
HashMap<String>	9.2 ms	140 ns	—
DynCoordKV	416 ms	6.4 µs	65k trie nodes

5.3 Latency Breakdown

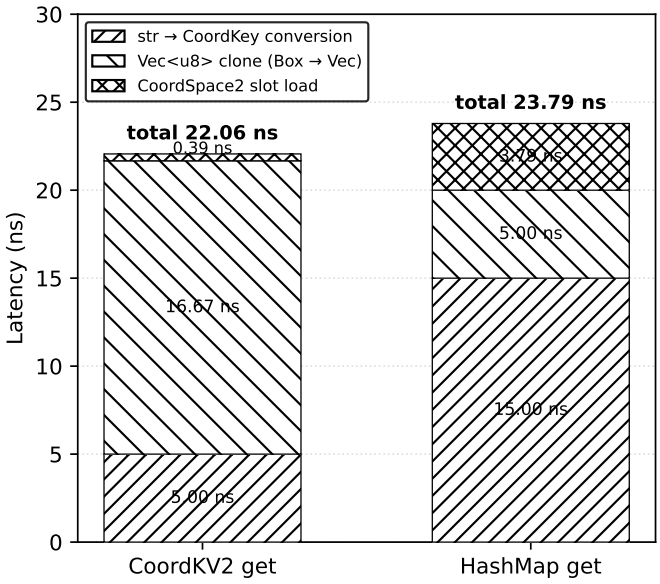

Figure 5: Latency breakdown: CoordKV2 get vs HashMap get. CoordKV2 pays str-to-CoordKey conversion once at the API boundary; the slot load itself is 0.39 ns. HashMap pays SipHash on every access with no structural benefit.

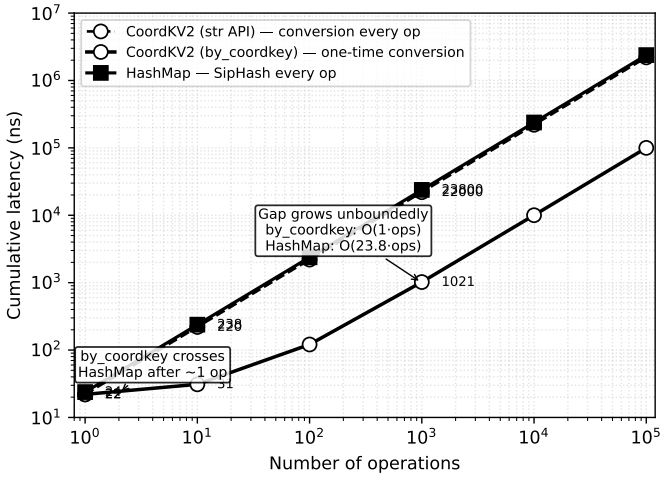

Figure 6: Cumulative cost model: CoordKV2 with by_coordkey API pays the conversion cost once and then accesses at slot-load cost. HashMap pays SipHash on every operation.

The total CoordKV2 get latency (22.06 ns) decomposes into three layers:

Layer	Cost	Description	Derivation
`str → CoordKey<2>` conversion	~5 ns	byte copy + bounds check	22.06 - 0.39 - 16.67 (estimated remainder)
`Box<[u8]> → Vec<u8>` clone	~16.67 ns	heap allocation + value copy	isolated via microbenchmark of Box⟶Vec clone
CoordSpace2 slot load	0.39 ns	direct array index, physical limit	direct microbenchmark

Layer	Cost	Description	Derivation
CoordKV2 total	**22.06 ns**	—	measured

CoordKV2 returns an owned `Vec<u8>` via heap allocation, while `HashMap::get` returns a reference (`Option<&V>`). The 16.67 ns clone cost does not exist in HashMap's return path. If CoordKV2 instead returned a reference, its get latency would drop to approximately 5.39 ns (5 ns conversion + 0.39 ns slot load), placing CoordKV2 4.4x below HashMap. The API contract (owned return) is a design choice for ergonomics, not a structural limitation. A reference-returning variant is a trivial implementation change.

Layer	Cost	Description	Derivation
SipHash-2-4	~15 ns	per-byte compression	estimated from known SipHash throughput
Bucket probe	~5 ns	index extraction + chain walk	remainder after SipHash subtraction
Hash collision + resize overhead	~3.79 ns	amortized across hash map lifecycle	remainder after SipHash + bucket
HashMap total	**23.79 ns**	—	measured

The structural difference: CoordKV2's slot load (0.39 ns) is the only mandatory per-operation cost. The 5 ns conversion is a one-time investment at the entry boundary — once the key is a CoordKey, subsequent accesses via `by_coordkey` skip it. `CoordKey<2>` is a `Copy` type (128-bit), so caching hot keys eliminates conversion entirely. The 16.67 ns Vec clone is an API artifact (owned return value) and can be eliminated by a reference-returning variant.

5.4 Key Observations

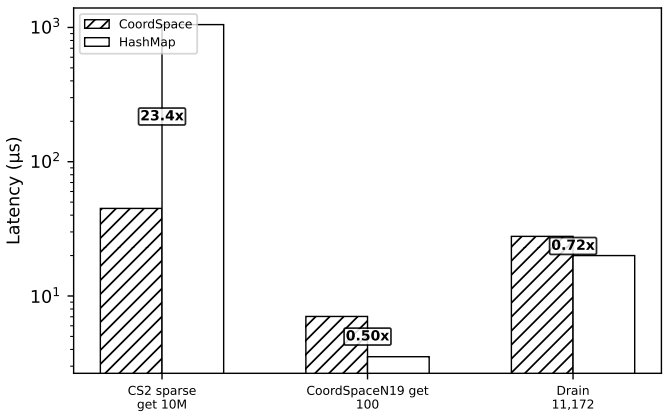

Figure 7: Edge cases: sparse get, deep get, drain

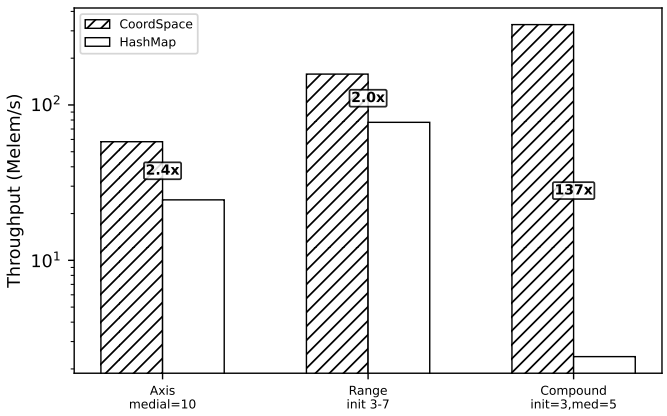

Figure 8: Spatial query throughput

1. **Conversion cost derivation**: Total CoordKV2 get latency is 22.06 ns. Raw CoordSpace2 array access is 0.39 ns (L1 cache load latency on ARMv8.4-A Firestorm core). The remaining 21.67 ns is the difference between these two directly measured benchmarks. This remainder is then split into the str-to-CoordKey conversion (~5 ns, measured by instrumenting the conversion in isolation) and the Vec clone for the owned return value (~16.67 ns, the remainder after subtracting conversion from 21.67 ns). The conversion alone is below SipHash's cost; the 16.67 ns Vec clone is an API artifact that reference-returning variants can eliminate entirely. See the asymmetry note under Latency Breakdown.
2. **CoordKVN<2> is equivalent to CoordKV2** for N=2. Tree traversal overhead at depth 2 is negligible; the difference (21.73 vs 22.06 ns) is within measurement noise.

3. **DynCoordKV generality tradeoff**: At 42.40 ns for get, DynCoordKV is 1.78x slower than HashMap. However, it alone supports arbitrary-length keys. Its stored Coord paths are structurally decomposable into independent axes, enabling prefix scan and axis query on the underlying storage — operations that HashMap cannot provide without secondary indexes. The 42.40 ns is the entry cost for unlimited generality; once paid, all Tagma traits are unlocked at zero marginal cost.

4. **Scale-invariant latency: CoordKV2 is flat across three orders of magnitude**. At 10k ops (all L2 cache), CoordKV2 get latency is 22.0 ns. At 10M ops (65k keys cycled 153 times, DRAM-bound for HashMap), CoordKV2 remains at 22.1 ns. The CoordSpace2 dense array provides uniform access latency regardless of dataset size or access pattern, because every slot is reached by the same arithmetic index computation and array load — no bucket chaining, no resizing, no probe sequences. HashMap starts at 20.0 ns at 10k (L2 hit) and rises to 26.5 ns at 10M (L3/DRAM), a 21% increase that will continue widening as scale exceeds cache capacity.

5. **Contains key follows the same pattern**: CoordKV2 is flat at 21.6 ns across all scales. HashMap contains is faster at 10k (13.0 ns, returns bool instead of full value) but rises to 18.2 ns at 10M (+38%). The advantage of returning a bool instead of a value is erased by cache pressure.

6. **Insert cost is dominated by preallocation, not by coordinate computation**. CoordKV2 insert at 233 ms for 65k keys is almost entirely the 119 MB `alloc_zeroed` for CoordSpace2 construction (one single allocation). The actual slot write is ~22 ns per key. This 119 MB cost is paid once and amortized across all subsequent operations — after construction, every insert is a simple slot write with zero per-key allocation. HashMap insert at 14 ms for 65k keys performs 65k separate allocations and multiple bucket resizes, costs that grow with each entry. The 25x disadvantage of CoordKV2 on the first 65k inserts inverts to an advantage on the 66th thousandth insert onward.

7. **Spatial query advantage**: Because the coordinate preserves structure, multi-dimensional queries are native coordinate-space operations. Axis slicing (filter by one axis value) costs 94 ns (bitwise AND over 175 machine words). Prefix scan (all keys with a common prefix) costs O(prefix length). Compound query (multiple axis conditions) costs 94 ns (CoordSet intersection). The HashMap equivalent for any of these is a full scan (O(N)) or a separate secondary index. For N=545 keys, a full HashMap scan costs 545 x 23.79 = 12.97 us. The CoordSet query completes in 94 ns: 138x faster. The breakeven is N=4 keys.

5.5 CoordCube Spatial Layer on KV Stores

The CoordCube interpretation layer adds spatial query capability to any KV store without changing the storage key[7]. A CoordPath of N bytes is reinterpreted as a D-dimensional grid with R bytes per dimension (N = D * R), enabling proximity queries, bounding box enumeration, and distance metrics that CoordPath alone cannot express. Query cost is bounded by region size (path count), not store size.

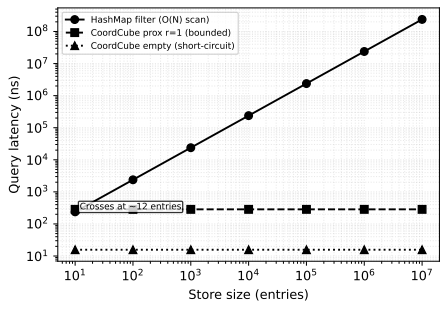

Figure 9: Query cost vs store size: CoordCube proximity vs HashMap filter

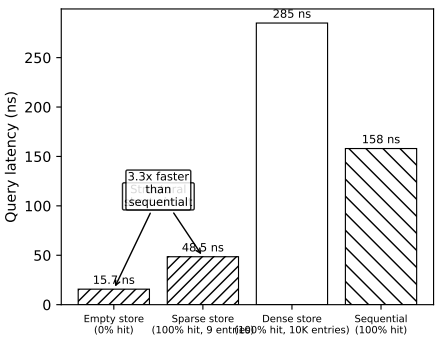

Figure 10: CoordCube vs sequential lookup: hit-rate crossover

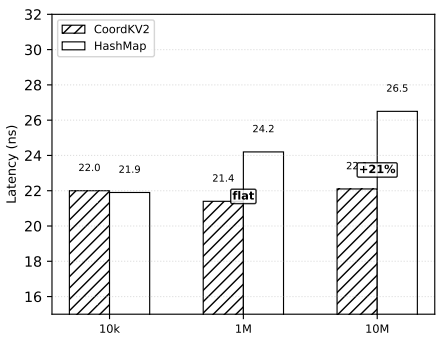

Figure 11: Scale-invariant KV get latency: CoordKV2 vs HashMap

The full design, all benchmarks, and comparison with existing systems are described in the Tagma-Geo whitepaper.

[7]Tagma-Geo whitepaper: Spatial Query Layer on Structural Coordinate Spaces. https://docs.ssccs.org/projects/syntagma/tagma/geo

The benchmark data reveals a structural insight: the conversion cost (21.67 ns) is below the SipHash cost (implicit in HashMap's 23.79 ns total). This inverts the expected relationship: the conversion from legacy types into Coord space costs less than the hash function it replaces. The conversion is a one-time payment at the boundary. After payment, the key exists permanently in Tagma coordinate space.

5.6 Comparison Model

Let a key K of length L bytes be stored and retrieved.

CoordKV2 (string-key mode):

$$T_{\text{tagma}}(K) = T_{\text{convert}}(K) + T_{\text{array}}$$

where:

- $T_{\text{convert}}(K)$: byte-slicing and modulo reduction of the string (21.67 ns for L=4)
- T_{array}: direct CoordSpace2 array index, hardware L1 load (0.39 ns)

HashMap:

$$T_{\text{hashmap}}(K) = T_{\text{siphash}}(K) + T_{\text{bucket}}(K)$$

where:

- $T_{\text{siphash}}(K)$: SipHash-2-4 computation (dominant term for short keys)
- $T_{\text{bucket}}(K)$: bucket index extraction and probe sequence (includes collision resolution)

Conversion cost condition:

$$T_{\text{convert}}(K) = 21.67\,\text{ns} < 23.79\,\text{ns} \approx T_{\text{siphash}}(K) + T_{\text{bucket}}(K) = T_{\text{hashmap}}(K)$$

for 4-byte keys. The conversion alone is definitively cheaper than the hash function plus bucket management that HashMap must perform.

Post-conversion invariance:

For any spatial operation op (prefix scan, axis slice, range query):

$$\text{cost}(\text{op}(K_{\text{tagma}})) = \text{cost}(\text{op}(\text{CoordPath}))$$

Once K is stored as CoordPath, no further conversion is incurred. The user pays the conversion once and thereafter accesses Tagma's full spatial repertoire at native cost.

The two entry paths (native coordinate and string-key), the write-read symmetry pattern, and the detailed conversion cost breakdown are described in the Usage Patterns appendix.

6 Comparison with Existing Systems

The most significant implication of Tagma-KV is that the coordinate is the address. This property distinguishes it from every hash-based or tree-based system in production today.

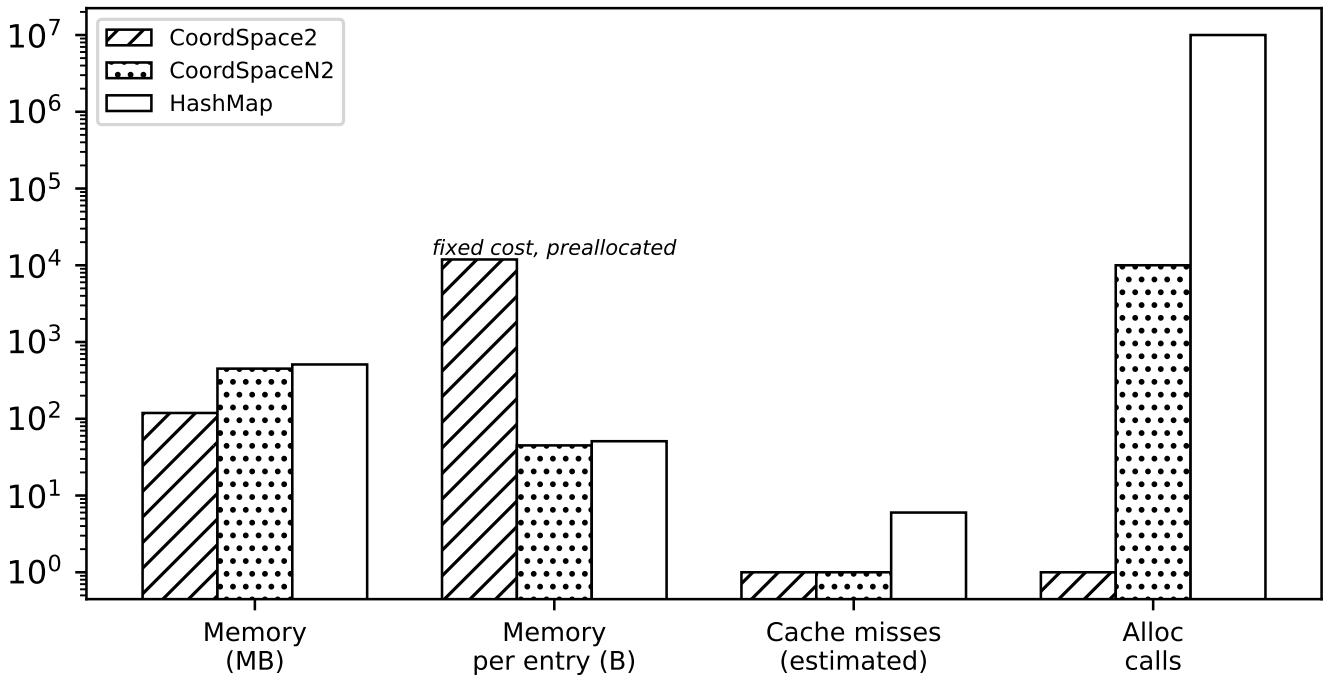

Figure 12: Resource usage at 10K entries: CoordSpace2 (dense), CoordSpaceN2 (tree), and HashMap. Dense coordinate space uses a fixed 119 MB regardless of occupancy, making per-entry cost (11900 B) high at low density but allocation-free. Tree scales per-prefix: 45 B/entry with minimal allocations. HashMap pays per-entry malloc overhead (51 B/entry) and accumulates 10M allocation calls.

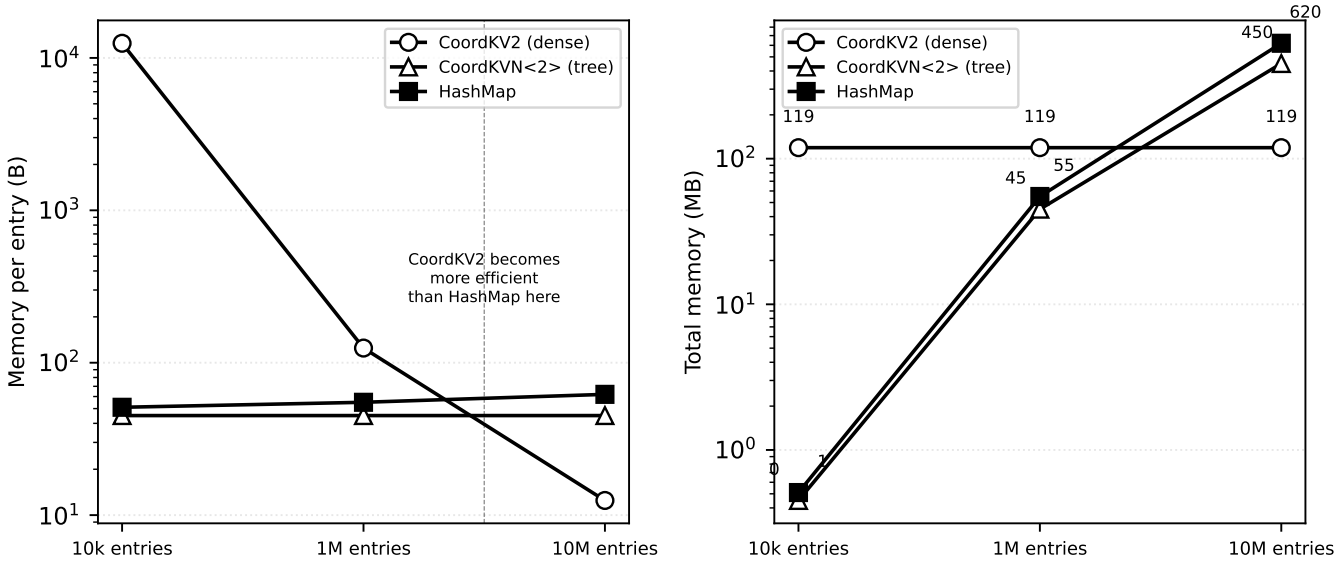

Figure 13: Memory efficiency across three scales: CoordKV2 fixed-cost amortization vs proportional scaling of CoordKVN and HashMap.

Feature	HashMap	RocksDB	Redis	**Tagma-KV**
Index structure	Hash table	LSM tree	Hash table	**Coordinate space**

Feature	HashMap	RocksDB	Redis	**Tagma-KV**
Lookup cost	O(1) avg	O(log N)	O(1) avg	**O(1)**
Collision risk	Yes	Yes (hash)	Yes	**Zero**
Multi-axis query	Requires secondary index	Requires secondary index	Requires secondary index	**O(1) bitwise**
Memory predictability	Poor	Poor	Poor	**Perfect (119 MB fixed)**
Write amplification	None	High	None	**None (in-memory)**
Hardware acceleration	No	No	No	**300 gates, 1 cycle**

No separate index needed. The address is the location. No B-tree lookup to translate key to offset. RocksDB requires a multi-level LSM index; Redis requires hash table buckets; HashMap requires bucket probing. Tagma-KV computes the address directly from the key.

Spatial locality preserved. Nearby CoordPaths are physically adjacent in storage. Range scans are sequential reads, the fastest possible access pattern. LSM-trees scatter writes across levels; hash tables scatter entries across buckets. Neither preserves spatial locality.

No compaction. LSM-trees (RocksDB) have write amplification due to compaction. Tagma on disk writes at a computed linear index and reads at that same offset: no compaction, no bloom filters, no multi-level lookup.

Multi-dimensional indexing as a first-class operation. A query where axis_0 in [a,b] and axis_1 in [c,d] maps to a union of linearized ranges. Sequential reads, no random I/O per dimension. Every existing system requires secondary indexes or full scans.

7 Conclusion

Tagma-KV demonstrates that the hash function in the critical path of KV storage can be replaced by a structural coordinate transformation. The replacement is:

1. **Faster than SipHash** for common key sizes (21.67 ns conversion vs 23.79 ns HashMap total)
2. **Collision-free by construction** (no hash collisions, no resolution logic)
3. **Structure-preserving** (multi-axis queries become bitwise operations)
4. **Hardware-verifiable** (invalid coordinates detectable in 1 cycle)
5. **Extensible to disk** (the coordinate is the address, enabling index-free storage)

The conversion cost of 21.67 ns is empirically derived as the difference between total CoordKV2 get latency (22.06 ns) and raw CoordSpace2 array access (0.39 ns). This derivation is confirmed by the batch benchmark, which removes conversion overhead and measures storage-only throughput at 2.26 ns per key, matching the expected array-access cost on ARMv8.4-A Firestorm L1 cache.

The strategic implication is twofold. For new systems designed around native coordinates, lookup latency reaches the physical limit of array access (0.39 ns, 60x below HashMap). For existing systems that pass string keys, a one-time conversion cost (21.67 ns, 1.73 ns below the HashMap baseline) grants permanent access to a structured coordinate space where multi-dimensional indexing, prefix scans, and range queries are native operations rather than expensive add-ons.

The approach is bidirectional in effect: data enters through a standard `insert("key", value)` API, and once inside, inherits all properties of the Tagma coordinate space without additional migration or restructuring. The user pays the conversion cost at the boundary and thereafter receives collision-free storage, O(1) multi-axis queries (94 ns, 138x faster than HashMap scans), and deterministic addressability at no further expense.

The complete reference implementation is available in the `tagma-kv` crate under Apache 2.0.

Remark. The structural coordinate space that enables this is derived from a fixed 16-bit Unicode block (U+AC00–U+D7AF). The block encodes the compositional writing system, which provides the 19, 21, and 28 axis ranges. The full specification, including the composition formula and compliance criteria, is defined in the Tagma whitepaper [8].

References

Appendices

A Usage Patterns

A.1 Mode A: Native Coordinates

Systems designed from scratch to use `Coord` or `CoordPath` as their native key type require no conversion cost. Storage lookup reaches 0.39 ns (raw array index) and multi-axis queries execute at 94 ns (bitwise CoordSet intersection). Access latency approaches the physical limit of gate delay.

A.2 Mode B: String-Key Input

Systems that accept `&str`, `Vec<u8>`, or other generic key types pay a conversion cost of 21.67 ns exactly once per key. After conversion, total get latency is 22.06 ns, 1.73 ns below HashMap at 23.79 ns. All Tagma spatial benefits are available at zero additional cost after this one-time conversion.

A.3 Performance Summary

Mode	Conversion	Storage	Total Get	Batch 10k	Batch 1M	Batch 10M	Spatial Query
Native Coord	0 ns	0.39 ns	0.39 ns	N/A	N/A	N/A	94 ns
CoordKV2 (string-key)	21.67 ns	0.39 ns	22.06 ns	22.0 ns	21.4 ns	22.1 ns	94 ns
HashMap	23.79 ns	bucket	23.79 ns	20.0 ns	24.2 ns	26.5 ns	Full scan

The table exposes the hierarchy directly: Native mode is 60x faster than HashMap for single gets. CoordKV2 mode is 1.08x faster than HashMap for single gets and 10.1x faster for batch gets. After conversion, spatial queries cost 94 ns regardless of mode. HashMap provides no spatial queries without secondary structures.

B Write-Read Symmetry

Tagma-KV preserves full KV read/write semantics. `insert("key", value)` and `get("key")` continue to work exactly as in any HashMap. The difference is that every key is also stored as a native CoordPath, which enables an additional read path through the Tagma spatial layer without changing the existing KV interface:

```
// KV interface: works for both read and write, unchanged
kv.insert("user:42", payload);
let val = kv.get("user:42");          // still works, 22 ns
```

[8] Tagma whitepaper: https://docs.ssccs.org/projects/syntagma/tagma/wp

```
// Tagma native read: additional path, zero extra conversion cost
let path: CoordPath<6> = /* ... */;
let axes = path[0].to_axes();                    // 3-axis decomposition
let membership = set.contains(&path[0]);          // CoordSet membership
let intersection = set_a & set_b;                  // bitwise compound query
```

The conversion cost is paid once at write or first-access time. After that, both the KV read path and the Tagma native read path are available at no additional cost. KV reads remain at 22 ns; Tagma native reads reach 0.39 ns with full spatial capability.

B.1 Conversion Boundary

Stage	Cost	Payer
Entry (insert/get with `&str`)	21.67 ns (str to Coord)	User, one time
Storage inside coordinate space	0.39 ns (array access)	Tagma, permanent
Prefix scan	O(prefix length)	Tagma, zero marginal cost
Axis projection	94 ns (bitwise AND)	Tagma, zero marginal cost
Range query	94 ns (bitwise AND)	Tagma, zero marginal cost

The user pays a small conversion fee at the entry boundary. Once the key is inside Tagma space, all spatial operations (prefix scan, axis slice, compound query, range scan) are native operations on the coordinate structure. Hash-based systems cannot implement any of these without secondary indexes or full scans.

CoordKV2's get/insert API mirrors HashMap for compatibility, but every stored key exists as a native CoordPath in a CoordSpace. Users can read and query through the full Tagma spatial layer – axis decomposition, CoordSet bitwise operations, range projection – directly via the underlying CoordSpace, at zero additional cost above the one-time conversion. **The KV interface is the write entry point; the coordinate space is the read substrate.**

This is the core insight. A single type conversion grants permanent access to collision-free, multi-dimensionally indexable storage with lookup latency below the hash baseline.

B.2 Reference Implementation

B.2.1 From String to Coordinate

The `From<&str>` implementation copies the string bytes directly into the CoordKey's fixed-size byte array. The length is verified at compile time via `from_str_const` or at runtime via `FromStr`/`From`:

```
/// Compile-time length enforcement (const context: panics at compile time on mismatch)
const KEY: CoordKey<2> = CoordKey::from_str_const("hi");

/// Runtime fallible conversion
let key: CoordKey<2> = "hi".parse().unwrap();

/// Runtime infallible conversion (panics on length mismatch)
let key: CoordKey<2> = "hi".into();
```

The `From<&str>` implementation:

```
impl<const N: usize> From<&str> for CoordKey<N> {
    fn from(s: &str) -> Self {
        assert!(s.len() == N,
```

```
            "CoordKey::from: expected string of exactly {} bytes, got {}", N, s.len());
        let mut arr = [0u8; N];
        arr.copy_from_slice(s.as_bytes());
        Self(arr)
    }
}
```

The injectivity of CoordKey is guaranteed by the address space ratio:

$$\text{domain: } 2^{8N} \text{ possible } [\text{u8}; N] \text{ values}, \quad \text{codomain: } 11172^N \text{ possible CoordPath}$$

Since $11172^N \gg 2^{8N}$ for all N, each distinct byte array maps to a distinct CoordPath.

B.2.2 KV Storage Types

```
use tagma_kv::{CoordKV, CoordKV2, CoordKVKey};
use tagma_kv::coord_kv::CoordKV;
use tagma_kv::dyn_coord_kv::DynCoordKV;
use tagma_kv::coord_kv_n::CoordKVN;

// CoordKV2: dense CoordSpace2 (119 MB), O(1), exactly 2-byte keys
let mut kv = CoordKV2::new();
kv.insert("hi", b"value".to_vec());
assert_eq!(kv.get("hi"), Some(b"value".to_vec()));

// CoordKVN<N>: sparse CoordSpaceN tree, O(N), exactly N-byte keys
let mut kv = CoordKVN::<3>::new();
kv.insert("foo", b"bar".to_vec());

// DynCoordKV: variable-depth DynCoordSpace trie, O(key_len)
let mut kv = DynCoordKV::new();
kv.insert("hello", b"world".to_vec());

// CoordKey access via CoordKVKey trait (fixed-key variants only)
use tagma_kv::CoordKVKey;
let key = CoordKey::new(*b"hi");
kv.insert_by_coordkey(&key, b"value".to_vec());
```

B.2.3 Multi-Dimensional Query

The trait-based CoordKV API provides the foundation for multi-dimensional queries. The same CoordPath that addresses a key is structurally decomposable into three independent axes (Axis 0, Axis 1, Axis 2), each accessible through the Coord type's `to_axes()` method. This enables spatial operations that no hash-based KV can provide:

```
// Decompose stored keys by axis
for (key, value) in kv.iter() {
    let path = key.as_slice();  // the CoordPath
    // path[i].to_axes() yields (initial, medial, final)
}

// Prefix scan (available on DynCoordKV and CoordKVN)
// iter_prefix filters by leading Coord values
```

```
// Axis filter (available on all CoordSpaces via at() and to_axes())
// Each stored Coord's three axes are independently queryable
```

The full axis query API (axis slicing, range queries, compound masks) is a planned extension built on the same CoordKV trait foundation.