

Tagma

Hashless coordinate primitive in the fixed 16-bit Unicode composition block

Taeho Lee*

SSCCS Initiative

ssccs.org

July, 2026

doi.org/10.5281/zenodo.21302508

Abstract

Tagma¹ is a computing primitive where the address is the coordinate — not a flat pointer, but a point in an N-dimensional geometric space. This is made possible by a fixed 16-bit contiguous Unicode block whose closed-form composition rule holds over *U+AC00..U+D7A3*, so every valid value is at once a 1-D address, a 3-D coordinate (Axis 0, Axis 1, Axis 2), and a compositional character. This triple interpretation gives a collision-free, hash-less, structurally addressable space [4.3] and a single-cycle combinational decoder². The coordinate arithmetic admits unbounded N-Coord composition — the address space grows as $11,172^N$ while lookup cost remains $O(N)$ per Coord. We present a gate-level decoder specification, a processor pipeline attachment reference (e.g., RISC-V XIF)³, a hardware reference implementation [5] of all three primitive operations (decode, compose, and distance) verified over their input spaces against the Rust reference and formally equivalent to the synthesized netlist, a software reference implementation [A], and a full benchmark suite [B]. Measured lookup latency in the software reference: 0.39 ns at single-Coord and 62 ns at SHA-256 scale (20 Coords), against SHA-256’s 227 ns (582x and 3.7x). Structural prefix queries show the decisive advantage: a direct-address table answers a nonexistent prefix in 1.65 ns where a hash-table scan of 10M entries takes 23.05 ms (14.0Mx); bit-set axis filters resolve compound queries at 329 Melem/s against a 2.4 Melem/s scan (137x, bitwise AND against full scan); and 10M sparse gets complete in 44.9 ms against 1.05 s (23.4x). That standard is open, a public good whose composition rules carry no proprietary encumbrance. A production-scale validation in CERN’s ROOT TTree read path confirms the model in petabyte-scale high-energy physics software [7].

1 Introduction

Unicode is an international standard that assigns each script a fixed address in a shared encoding space. Within this space, this block [1] occupies a contiguous 16-bit segment whose closed-form composition formula embeds three independent structural axes into every code point. Every content-addressable system today generates identifiers through a hash function. This indirection layer imposes a predictable cost: gate area, cycle latency, storage overhead, and probabilistic collision resolution. Tagma shows that this fixed 16-bit block can replace the hash function entirely where cryptographic integrity is not required, using a combinational decoder.

The composition formula [2] [3] is:

$$C(i, m, f) = \text{U+AC00} + 588i + 28m + f, \quad 0 \leq i < 19, \quad 0 \leq m < 21, \quad 0 \leq f < 28$$

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

¹Greek *τάγμα* from *σύνταγμα* (syn-tagma, co-ordinate in English), “a well-ordered arrangement of constituent elements”. *Tagma* is the system name; *Coord* is the concrete implementation type.

²Tagma hardware series: docs.sccs.org/projects/syntagma/tagma/hardware/

³Repository: Github (Pre-release), Benchmark github.com/ssccsorg/syntagma/blob/main/sw/rust/benches/bench.rs

2 The Structural Conjunction

Tagma rests on three structural properties and one durability guarantee. The absence of any of the three structural properties makes a Tagma-compatible coordinate space impossible:

Exception-free three-axis composition. Every valid character decomposes through the closed-form formula entries and no irregular exceptions. This is the definitional property: decoding is arithmetic, not a lookup, and the hardware validity check is a formula check.

Contiguity. The 11,172 valid values form a single contiguous run starting at U+AC00, so $\text{offset} = \text{code} - \text{U+AC00}$ is one subtraction and the space is a flat array: *(base + coord) is the entire lookup. Address is coordinate.

BMP membership. The block lies on Unicode's Basic Multilingual Plane Zero, the foundational layer that every implementation must support, so one character is one UTF-16 code unit: one u16, one register, one SRAM word.

Every valid 16-bit value carries **three simultaneous interpretations**: a 1-D Unicode address, a 3-D coordinate, and a compositional character. Internally, the 16 bits are organized as a 3-axis coordinate packed into a 16-bit word:

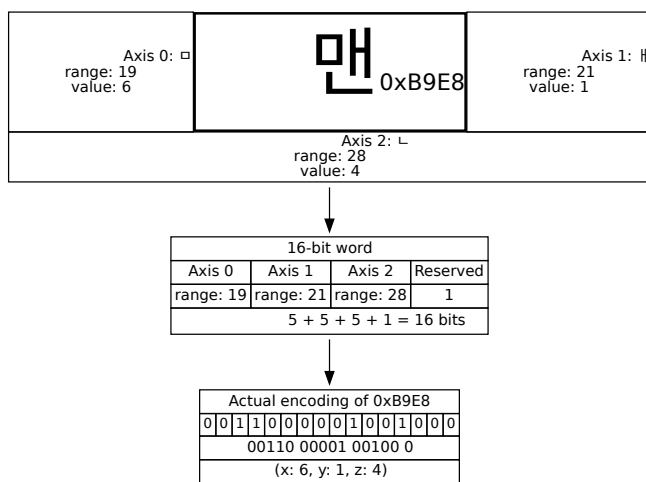


Figure 1: Coordinate 0xB9E8: three-axis coordinate (6, 1, 4) packed into a 16-bit word.

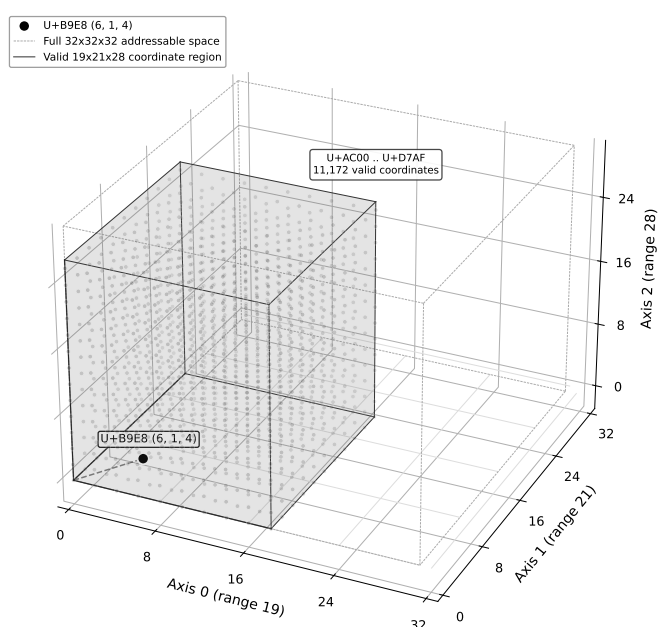


Figure 2: Three-dimensional bitcell organization. 19 x 21 x 28 lattice, 11,172 of 65,536 states valid.

The durability guarantee is the open international standard itself: Unicode's stability policy fixes the block, keeping the constants (U+AC00, U+D7A3, 11,172, 588, 28) permanent, and Hangul is the only writing system in existence that satisfies all three structural properties. The full comparison, including near-miss combinatorial, syllabic, and supplementary-plane cases, is in Appendix: Script Comparison [K].

3 The Encoding Problem

Variable-length encoding has architectural consequences that are independent of any particular implementation. Memory alignment is not guaranteed. Instruction decoding must detect boundary positions. Prefetch and pipeline efficiency are affected. These are structural properties of variable-length design, not engineering limitations. The structural constraint in Tagma arises from the fixed composition formula of this block. As established in the Structural Conjunction section, only 11,172 of 65,536 states satisfy the composition formula. A hardware decoder can distinguish valid from invalid states using combinational logic derived from that formula.

Encoding	Storage unit	Addressable symbols	Wasted bits/unit	Fixed alignment	Structural HW check
ASCII	1 byte (8 bits)	128 of 256	1 bit (12.5%)	yes	no
Latin-1	1 byte	191 of 256	variable	yes	no
UTF-8 (English)	1-4 bytes	unlimited	none for ASCII	no	no
UTF-16 (BMP)	2 bytes	63,488 of 65,536	2,048 reserved	no	no
UTF-32	4 bytes	unlimited	75-88% for common use	yes	no
Tagma	2 bytes	11,172 of 65,536	54,364 structurally invalid (error detection)	yes	yes (54,364 invalid)

Because the coordinate is a fixed 16-bit BMP value, the address role needs no decoder, no parser, and no operating system: $*(base + coord)$ is the entire lookup, from a kernel bootloader to an MMU-less microcontroller. Text input still requires one conversion from the carrier encoding to the code point.

4 The Identity Problem

Every system that stores data by content rather than by location must generate identifiers. Current approaches and their hardware cost:

Method	Identifier size	Generation cost	Collision	Lookup structure	Tagma advantage
Pointer	32-64 bits	zero	none	direct (location-based)	—
Hash (SHA-256)	256 bits	~10K gates, 64-75 cycles	probabilistic	hash table + resolution	17-21x fewer gates, zero collisions
UUID [4]	128 bits	entropy-dependent	probabilistic	hash table	deterministic, no entropy needed
CAM	per-bit comparison	9-16 transistors/bit	none	associative [5]	~2.5x less area for sparse sets
Tagma	16 bits	1 cycle (combinational)	none (formulaic)	direct (coordinate = address)	baseline

A single Coord covers 11,172 identifiers — sufficient for sensor arrays, embedded device registries, and moderate-scale lookups [6]. N-Coord composition (below) extends this to UUID-scale and SHA-256-scale spaces without changing the decoder or the arithmetic.

The practical implication for everyday identifiers is direct. UUID generation requires entropy collection, version/variant bit insertion, and hyphen formatting at approximately 100 ns or more; its 128-bit output expressed in hex or Base64 consumes 36 or 22 characters respectively. Tagma produces a 6-Coord identifier (18 axes, 1.9×10^{24} space) in approximately 150 ns, and a 10-Coord identifier exceeding UUID space in approximately 260 ns, each requiring only multiplications and additions, as a human-readable string with no special characters, no padding, and zero collision probability. Similarly, a SHA-256 hash output as 64 hex characters is matched by 20 Tagma Coords ($11,172^{20} \approx 2^{269}$, exceeding the 2^{256} space) in approximately 480 ns, extrapolated linearly from the 456 ns measured at 19 Coords ($2^{255.5}$) on an x86 CI runner. In all cases Tagma output is shorter, faster, and structurally self-validating. A software reference implementation confirming these measurements is described in Appendix [A].

4.1 N-Coord Composition: From 16 Bits to SHA-256 Scale

A single Tagma Coord provides 11,172 unique identifiers over 16 bits. For larger address spaces, Coords compose linearly:

$$S(N) = 11,172^N \approx 10^{4.05N}$$

Linearization. An N-Coord coordinate $(c_1, \dots, c_N)$ maps to a single linear index via row-major order:

$$\text{index}(c_1, \dots, c_N) = \sum_{k=1}^N c_k \times 11,172^{N-k}$$

This requires exactly $(N - 1)$ multiply-add pairs, a fixed sequence of $2(N - 1)$ arithmetic operations independent of the addressable space. Since N is a compile-time constant, the lookup remains $O(1)$ for any N-Coord composition.

Coords	Axes	Identifier space	Equivalent to
1	3	1.12×10^4	Sensor tags
2	6	1.25×10^8	Database records
3	9	1.39×10^{12}	Distributed nodes
6	18	1.94×10^{24}	Below UUID (3.4×10^{38})
8	24	2.43×10^{32}	Approaches UUID
9	27	2.71×10^{36}	Below UUID (3.4×10^{38})
10	30	3.03×10^{40}	Exceeds UUID (3.4×10^{38})
19	57	8.21×10^{76}	Just below SHA-256 (2^{256})
20	60	9.18×10^{80}	SHA-256 (256-bit) equivalent

Each Coord retains independent hardware-verifiable validity ($19 \times 21 \times 28 = 11,172$ valid triplets per Coord). The N-tuple identity space is the Cartesian product of N independent 3D spaces, yielding a $3N$ -dimensional structural coordinate system. We call this N-Coord sequence a CoordPath, distinct from the atomic Coord.

This is not merely a larger address space, but a coordinate algebra of $3N$ independent axes whose semantics are entirely application-defined: a sensor network may assign axis 0 to device type, axis 1 to geographic zone, and axis 2 to timestamp, while a database system assigns the same axis positions to table, partition, and row. The coordinate arithmetic – composition, decomposition, linearisation, Hamming distance, axis projection – is invariant under semantic reinterpretation because the axes are fungible; the algebra depends only on the invariant structural relations among them, not on what any particular axis represents. This distinguishes Tagma from Euclidean space, where axes are bound to physical dimensions, and from hash space, where all structure is destroyed.

4.2 Recursive State Space Expansion

The composition formula $C(i, m, f)$ accepts three axes whose ranges are fixed: 19, 21, and 28. Nothing in the formula requires these ranges to be atomic. A SynTagma is the structure that results when these axes are themselves composed of CoordPaths. An N -Coord CoordPath can occupy any axis position, replacing the original range with its own $11,172^N$ values.

Let $\mathbb{T}_0$ be the set of all valid 19-Coord CoordPaths:

$$S_0 = |\mathbb{T}_0| = 11,172^{19} \approx 8.21 \times 10^{76}$$

Define $\mathbb{T}_1$ as the Cartesian product of three $\mathbb{T}_0$ coordinates:

$$\mathbb{T}_1 = \mathbb{T}_0 \times \mathbb{T}_0 \times \mathbb{T}_0$$

$$S_1 = |\mathbb{T}_1| = S_0^3 = (11,172^{19})^3 = 11,172^{57}$$

A CoordPath of length $3k \cdot N$ Coords can be interpreted as either a flat sequence of $3kN$ atomic coordinates or a k -deep nested structure of SynTagma triplets. Both interpretations produce the same linear index through the same arithmetic.

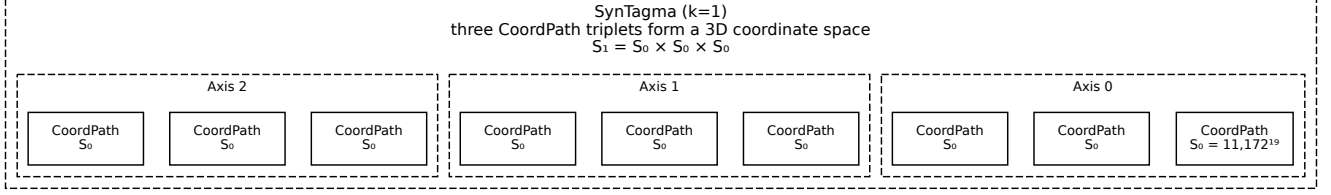


Figure 3: SynTagma contains three CoordPath triplets as its axes.

Access follows from the recursive linearisation formula:

$$\text{index}(a, b, c) = a \cdot S_{k-1}^2 + b \cdot S_{k-1} + c$$

Each of a, b, c is itself linearised recursively until atomic Coords are reached. The number of integer operations per lookup is proportional to the total Coord count, independent of the number of stored entries.

4.2.1 Sparse Allocation

The identifier space S_k is not a storage allocation. A CoordSpace (Appendix [A]) allocates a root array of 11,172 pointers (approximately 89 KB on 64-bit systems) and creates child nodes only along CoordPaths that are actually written to. Memory consumption is proportional to the number of stored entries, not the size of the address space.

4.2.2 Magnitude

For $k = 1$ with a 19-Coord base:

$$S_1 \approx 5.54 \times 10^{230}$$

The estimated number of atoms in the observable universe is approximately 10^{80} . The ratio:

$$\frac{S_1}{10^{80}} \approx 5.54 \times 10^{150}$$

A single recursive step on a 19-Coord base exceeds the atomic count of the observable universe by 150 orders of magnitude. For $k = 2$:

$$S_2 = S_1^3 \approx 1.70 \times 10^{692}$$

No physically meaningful comparison remains. There is no mathematical terminal; the formula admits unbounded k . The practical limit is in the silicon budget of the implementation.

4.2.3 From Observable Horizon to the Whole Universe

The comparison value 10^{80} (atoms in the observable universe) is a familiar reference point, but it is defined by the cosmic light horizon at approximately 46.5 billion light-years. This is a causal limit, not a physical boundary. The actual universe may be many orders of magnitude larger or infinite. The coordinate space exceeds not only this local figure but any physically conceivable finite universe. Even the most generous estimates for the total baryon count of a finite universe under standard Λ CDM curvature bounds fall short of S_1 by more than 145 orders of magnitude. The coordinate space is not bounded by cosmological horizons; it is a mathematical object indexed by arithmetic composition.

4.3 Hash-less computation: Direct structural addressing

In conventional computing, identity is established through indirection:



Figure 4: Traditional hash-based identity model.

Each step adds cost: hash computation, collision handling, dynamic resizing, cache-unfriendly access patterns. Tagma replaces this with direct structural addressing:

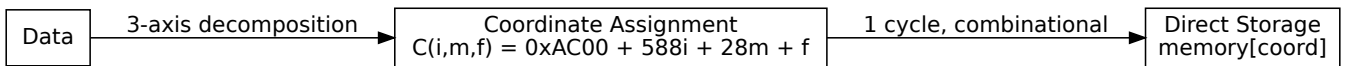


Figure 5: Tagma direct structural addressing.

This one-cycle path from data to address eliminates the scaling relationship between entry count and retrieval cost, a property that standard complexity analysis cannot capture.

4.4 Structural Consequences

The transition from hash-based indirection to coordinate-based addressing produces several interrelated consequences that define Tagma's operational regime. They are not independent properties of the implementation; they are necessary corollaries of the coordinate identity model.

Query reduces to arithmetic. A coordinate's three axes are independent fields in a 16-bit word. A query that filters by axis value is a range projection over that field — a bitmask and comparison, not an index scan. Proximity search is field-wise Hamming distance, computable as a single bitwise operation. The coordinate arithmetic is the query engine; no hash tables, B-trees, or CAM cells are needed.

Index structures are eliminated. In current systems, identity generation and index maintenance are separate concerns: SHA-256 produces an opaque identifier, then a separate hash table or B-tree maps that identifier to a storage location. The coordinate is identity and address simultaneously. A Tagma identifier can be used directly as a memory address, array index, or routing path without an intervening translation layer. The elimination is not an optimization of the index layer; the index layer ceases to exist as a separate structure.

The coordinate space is a physical structure, not a software abstraction. The decoder that maps a 16-bit coordinate to its three axis fields is a combinational circuit operating on a fixed encoding formula. A software library implementing the same formula on a general-purpose CPU is a faithful simulation of that circuit, not a design intent of its own. The space exists at the gate level before any software runs. This reverses the conventional relationship between software and hardware: software does not define the addressing scheme and then implement it in gates; the addressing scheme is already a gate-level structure, and software either uses it (on a Tagma-equipped processor) or simulates it (on a general-purpose CPU).

For example, the software reference implementation reveals this inversion clearly. The `Coord::to_axes()` method decomposes a `Coord` into its three axis fields using arithmetic:

```
pub fn to_axes(self) -> (u8, u8, u8) {
    let v = self.0 as usize;
    let initial = (v / 588) as u8;
    let medial = (v % 588 / 28) as u8;
    let final_ = (v % 28) as u8;
    (initial, medial, final_)
}
```

The code computes division and modulo because it is simulating a gate-level structure on a sequential processor. In hardware, the same fields are present as three contiguous 5-bit groups in the 16-bit word. The hardware does not compute; it reads. The division and modulo in the software are artifacts of simulating a parallel bit extraction on a sequential ALU. A Tagma-equipped processor executes `to_axes()` in zero cycles — the axes are already present on the output wires of the decoder. The software complexity inversely indicates the hardware simplicity: the more arithmetic the reference implementation requires, the more directly the hardware structure implements it. This is the opposite of hash functions, where software and hardware complexity are proportional: SHA-256 is expensive in both domains.

The coordinate space is a search engine, not a store. Tagma replaces the key-value model with a searchable coordinate space. A hash table answers only “value at this exact key”; it cannot answer “entries satisfying a condition” without scanning. Tagma inverts this: the coordinate space is itself the execution plan, with projections, proximity searches, and set operations all reducing to coordinate arithmetic. The departure is fundamental: hash tables answer “get(key)”, coordinate spaces answer “navigate(condition)” – and the latter is what most queries need.

Property	Key-value store (HashMap)	Coordinate space (Tagma)
Input	Key (string, hash)	Coordinate (Coord, CoordPath)
Output	Value	Value + position in coordinate space
Query model	Exact key match only	Axis projection, range, proximity, set operations
String dependency	String keys require encoding, parsing, hashing	String is a display layer; the address is structural
Index requirement	Separate index structures for conditional queries	Identity is the coordinate; no separate index exists
Scaling variable	Entry count (collision chains, rehashing)	Coord Depth d (fixed per schema, data-independent)

The shift from string-based to coordinate-based addressing eliminates an entire class of costs that conventional systems treat as unavoidable:

Vanishing cost	String-based	Tagma (Coord-based)	Effect
Memory allocation	heap allocation	a stack-allocated u16	Zero heap fragmentation; allocator and GC load eliminated
Encoding validation	Every input string must be verified as valid UTF-8	A Coord is a pre-validated 16-bit integer; any valid u16 in range is structurally valid	Validation circuit and cycles eliminated

Vanishing cost	String-based	Tagma (Coord-based)	Effect
Comparison	<code>str1 == str2</code> iterates bytes up to the shorter string length	<code>coord1 == coord2</code> is a single CPU compare instruction	$O(n) \rightarrow O(1)$; one cycle
Hashing (storage)	A string key must hash its entire byte sequence before the map can be indexed	A Coord is itself a complete array index	Hash function call eliminated; zero collision resolution
Serialization / parsing	Keys in JSON, MessagePack, or protocol buffers must be parsed, validated, and copied	A Coord is a fixed 2-byte binary value; the serialized form is the in-memory form	Parsing overhead eliminated; zero-copy direct mapping

These are not optimizations of the string path but its elimination. A system that never uses strings as addresses never pays string costs. The question of whether Tagma could be faster than a hash table on a particular benchmark is therefore misdirected: Tagma does not compete with hash tables on hash-table workloads. It competes by making the addressing substrate invisible — an operation that hash tables cannot attempt because they depend on the very indirection Tagma removes.

The question of string keys dissolves under this framing. An application that stores “user:123:item:456” must parse, hash, and index that string before it can retrieve anything. Tagma assigns a CoordPath directly — Coord for user type, Coord for user ID, Coord for item — and the path is both identifier and address. The string remains a human-readable label generated from the CoordPath for display, not parsed into one for storage. This separation eliminates hashing from the critical path.

The deeper distinction is between a map and a space. A hash map is lossy compression: it preserves only the mapping from each key to its value, discarding all structural relationships between keys. Two keys that differ by a single bit land in completely unrelated buckets; proximity is lost, axis structure is lost, the geometry of the identifier space is destroyed by the hash function. No amount of magnification applied to a map reveals the terrain it abstracts. Tagma is the uncompressed original: every coordinate retains its full structural position in the space, and every operation — lookup, proximity search, axis projection, set membership — operates on the coordinates directly. The space does not compress away structure because the structure is the address.

4.5 The Complexity of Structure: Coord Depth

Standard complexity analysis classifies algorithms by how their cost scales with input size N — the number of data items. Tagma’s retrieval cost does not scale with data volume at all. It scales with **Coord Depth** d , the number of Coords required to uniquely identify an entity under a given scheme. These are fundamentally different kinds of quantities: one measures data inventory, the other measures structural dimensionality.

d is a system constant determined at schema design time. A sensor identifier requires $d = 1$ (single Coord, 11,172 identifiers). UUID-scale identity uses $d = 10$. SHA-256-scale identity uses $d = 20$; the 19-Coord type ($2^{255.5}$) sits just below it. Adding a billion entries does not increase d .

$$\text{Retrieval cost} = O(d), \quad d \ll \log N_{\text{entries}}$$

Metric	Hash table	Tagma
Scaling variable	Entry count N	Coord Depth d (fixed per scheme)
Growth with data	Increases	None
Worst case	$O(N)$ (collision chain)	$O(d)$ (same as average)
Operation	Chop and mix bits	Decompose along structural axes

Metric	Hash table	Tagma
Lookup	Hash + resolve + dereference	Direct array access
Output	Opaque 256-bit value	Self-describing 16-bit value

A hash function receives a key and produces an address by destroying the key's internal structure. The lookup is indirect: hash, resolve collision, dereference. Coord inverts this: the Coord sequence carries its own address — the data is already the address. This inversion makes Coord Depth a distinct scaling regime, expressible neither as $O(1)$ (which implies independence from any parameter) nor as $O(N)$ (which implies dependence on data volume). Coord Depth $O(d)$ parameterizes retrieval by the identifier's inherent dimensionality, a quantity bounded by the application schema, not by the data set.

The engineering frontier. The mathematical bound holds for any fixed d : $2(d - 1)$ arithmetic operations per lookup. On a general-purpose CPU, however, each operation incurs mechanical overhead: instruction dispatch, register pressure, pipeline latency, and cache hierarchy effects. At $d = 1$ these are negligible (one array access). At $d = 19$ the 18 multiply-add pairs accumulate approximately 456 ns on a GitHub Actions CI runner (x86_64), far from the $O(1)$ ideal of a single cycle. This gap follows from running a structurally parallel operation on a sequential von Neumann machine rather than from a flaw in the theory. A dedicated combinational decoder will execute all d Coord decodes concurrently at gate level, collapsing the gap entirely.

The elimination of separate index structures is a direct consequence of coordinate identity:

Layer	Current (SHA-256)	Tagma
Identity	$H = \text{SHA256}(\text{data})$ (256-bit opaque)	$\text{Tagma}(i, m, f) = U + AC00 + 588i + 28m + f$ (16-bit transparent)
Index	separate structures: HashMap, OrderedIndex	none: identity = coordinate
Query	$\text{Vec}[o] \cap \text{Vec}[c] \cap \text{Vec}[t] - O(K \times N)$ intersection	$\text{coord.decompose}() - O(1)$ direct field extraction

This property, verified in the reference implementation, is the software-level manifestation of the same principle the hardware decoder implements: the coordinate IS the identity, and the identity IS the coordinate. No indirection, no intersection, no hash tables.

5 Hardware Reference Implementation

This part states the hardware reference implementation of the primitive on its own terms, so that it can be read without the software benchmark sections above. The same formula the Rust reference evaluates in arithmetic is evaluated here in gates, and every claim in this part is reproduced by the hardware verification gate in the syntagma repository under `hw/`.

5.1 The primitive in gates

The primitive is three combinational functions over the same 16-bit value, each a separate module verified independently:

Operation	Module	Input	Output
decode	<code>tagma_decoder</code>	16-bit code point	three axes, <code>valid</code>
compose	<code>tagma_compose</code>	three 5-bit axes	14-bit coordinate index, <code>valid</code>
distance	<code>tagma_dist</code>	two 16-bit code points	three 5-bit distances

Axis 0, Axis 1, and Axis 2 are the initial, medial, and final axes of the composition formula; the RTL names them *i*, *m*, and *f*. Decode and distance take a code point; compose returns the coordinate index, the offset from U+AC00, which is the internal representation of the Rust Coord, so the code point is U+AC00 + index. The distance unit instantiates two decoders, one per operand, which is why it is the largest of the three. A fourth module, `tagma_segment_store`, realizes the address path as an 11,172-slot store and is described under Memory hierarchy penetration below.

5.2 The decoder

The decoder extracts three 5-bit fields from a 16-bit input in three combinational stages:

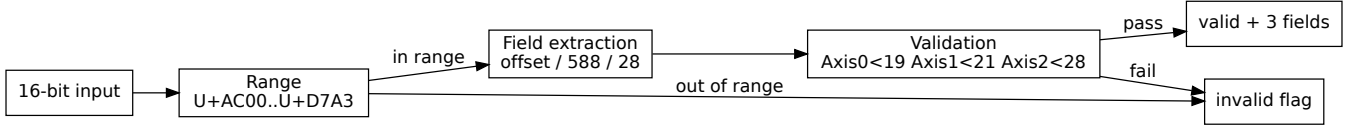


Figure 6: Three-stage decoder flow: range check, field extraction, validation.

Stage 1: Range check. Two comparators verify the input is within [U+AC00, U+D7A3]. An out-of-range value produces an immediate invalid flag.

Stage 2: Field extraction. For in-range values:

Step	Operation	Output	Range
1	input - 0xAC00	offset	0-11,171
2	offset \div 588	Axis 0	0-18
3	offset \bmod 588	remainder	0-587
4	remainder \div 28	Axis 1	0-20
5	remainder \bmod 28	Axis 2	0-27

Stage 3: Validation. Three comparators check: Axis 0 < 19, Axis 1 < 21, Axis 2 < 28. Any failure sets the invalid flag. Tagma coordinates support proximity-based search without dedicated search hardware. Given two coordinates, the field-wise Hamming distance is computed by three banks of parallel XOR gates. This enables associative lookup without hash tables, B-trees, or CAM cells.

The division structure determines the cost. Two implementations of the same arithmetic were synthesized and measured with the open toolchain (Yosys 0.65) in three flows and placed and routed on the iCE40 UP5K:

Metric	shift-subtract (naive)	multiply-shift (current)
Generic cells (ev schema)	206	478
Gate-level estimate (2-input)	232	588
iCE40 LUT4 + SB_CARRY	95 + 66	201 + 41
Logic levels	72	33
Critical path (UP5K PnR)	115.90 ns	59.55 ns
Fmax	8.63 MHz	16.79 MHz
Meets 12 MHz board clock	no	yes

The ~300 gate claim of earlier documents holds for the naive shift-subtract decoder (206 to 232 cells), but timing closure on a 12 MHz board requires the multiply-shift structure at about 2.5x the gate count. The current RTL implements the constant divisions with multiply-shift, exact over the valid domain: division by 28 becomes division by 7 on `offset >> 2` (at most 2792), and division by 21 applies to values at most 398. Both structures are functionally identical, which the verification pipeline proves over all 2^{16} inputs. The decoder is verified exhaustively over all 11,172 valid characters in three simulation channels (formula, Rust reference, gate netlist), plus formal

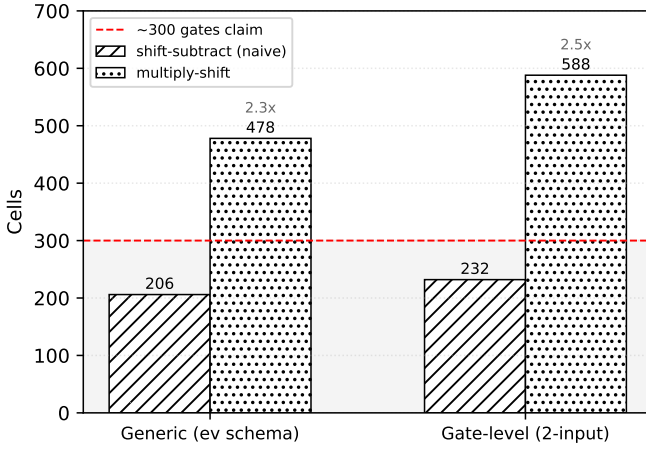


Figure 7: Gate count by synthesis flow: shift-subtract (naive) vs multiply-shift. The shaded region is below the ~300 gate claim; the ratios mark the cost of timing closure.

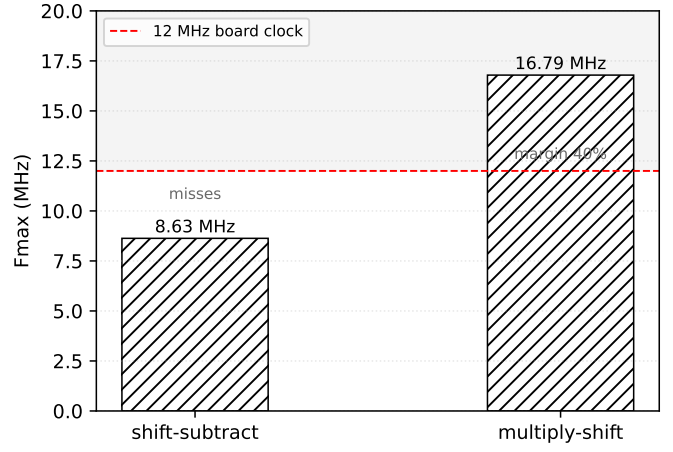


Figure 8: FPGA clock frequency before and after the multiply-shift optimization. The shaded region is above the 12 MHz board clock; the margin is the headroom of the optimized version.

equivalence over all 2^{16} inputs. The full compliance criteria for Tagma-compatible implementations are defined in Section [9].

5.3 Coprocessor attachment

Tagma can be attached to any processor pipeline through a standard coprocessor interface for custom instructions that does not require modifying the core pipeline. An example implementation uses the RISC-V XIF interface [7], [8]. The Rust reference implementation (Coord) serves as the functional golden model: every hardware instruction must produce results bit-exact with the corresponding Rust function.

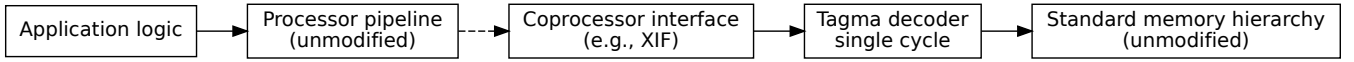


Figure 9: Tagma position in the system stack.

Three custom instructions implement the core Coord API:

tagma_check rd, rs1. Decodes the lower 16 bits of rs1. Returns a 16-bit value packed as: valid flag (bit 15, 1 = valid), reserved (bit 14, reads as zero), Axis 0 (bits 13–9, 5 bits, range 0–18), Axis 1 (bits 8–4, 5 bits, range 0–20), Axis 2 (bits 3–0, 5 bits, range 0–27). Invalid inputs produce a zero valid flag with undefined field values. Corresponds to `Coord::new(rs1).map(|c| c.to_axes())`. One cycle, combinational.

tagma_compose rd, rs1, rs2, imm5. Composes a coordinate from three 5-bit axis fields. rs1 supplies the Axis 0 field (selected by imm5 = 0), rs2 supplies the remaining two fields, or the immediate fields are extracted from rs1[14:0] directly (Axis 0 at bits 14–10, Axis 1 at bits 9–5, Axis 2 at bits 4–0). Returns the 16-bit coordinate index (0–11171) in rd, or zero if any axis is out of range. Corresponds to `Coord::from_axes(initial, medial, final).map(|c| c.index())`. One cycle, combinational.

tagma_dist rd, rs1, rs2. Computes field-wise absolute difference between the two 16-bit coordinates. Returns three 5-bit distances packed as: Axis 0 (bits 14–10), Axis 1 (bits 9–5), Axis 2 (bits 4–0); bit 15 is reserved. Corresponds to `Coord::hamming_distance(rs1, rs2)`. One cycle, combinational.

All three instructions are realized in the hardware tree (hw/rtl/) as the modules `tagma_decoder`, `tagma_compose`, and `tagma_dist`, each verified in the same channels as the decoder; the cell counts are in Synthesis and timing below.

5.4 Verification

The coordinate space is exhaustively enumerable: 65,536 values, of which 11,172 are valid. A verification harness generates all 65,536 inputs, applies the decoder specification, and records the results. The bijection is verified by enumeration: no collisions, no unassigned values within the block.

The hardware reference implementation closes the loop against the Rust reference with four independent channels: formula simulation, golden anchors exported from `Coord::to_axes`, gate-level netlist simulation, and formal equivalence of the RTL against the synthesized netlist over all 2^{16} inputs. All three simulation channels pass all 11,172 valid characters. The exhaustive tests surfaced the boundary correction: the last valid character is U+D7A3, not U+D7AF, and the testbench boundary was wrong initially. The pipeline is reproducible from a clean checkout with `make -C hw check` in the syntagma repository, and runs in the hw CI job.

The three units cover their own input spaces in the same channels:

Unit	Simulation	Formal equivalence
decoder	all 11,172 valid characters	all 2^{16} inputs
compose	all 32^3 axis combinations	all 2^{15} axis combinations
distance	11,172 code points against two references and a varying-operand sweep	all 2^{32} input pairs
segment store	11,172 slots, reserved addresses, read-during-write (behavioral model)	not applicable

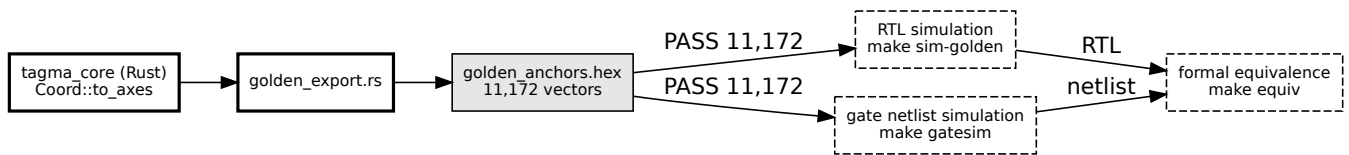


Figure 10: Hardware verification pipeline. The Rust reference is the single source of truth; every simulation and the formal equivalence check re-validate against it.

5.5 Architectural Mapping

A coordinate in this space is a 16-bit value whose identity is its structure. The composition rules of Axis 0, Axis 1, and Axis 2 correspond to the Scheme that defines which coordinates are valid. Validity predicates (range checks) implement the Field by defining admissible coordinate ranges. Enumerating coordinates under constraints implements Observation: a hardware-level selection of valid states.

5.6 Memory hierarchy penetration

The comparison is measured:

Path	Latency	Note
<code>CoordSpace.get</code> (software)	0.38 ns	whole space in L1 cache
<code>Coord::to_axes</code> (software)	1.44 ns	reference decode
HW decoder (UP5K PnR)	59.55 ns	worst-case critical path at 16.79 MHz

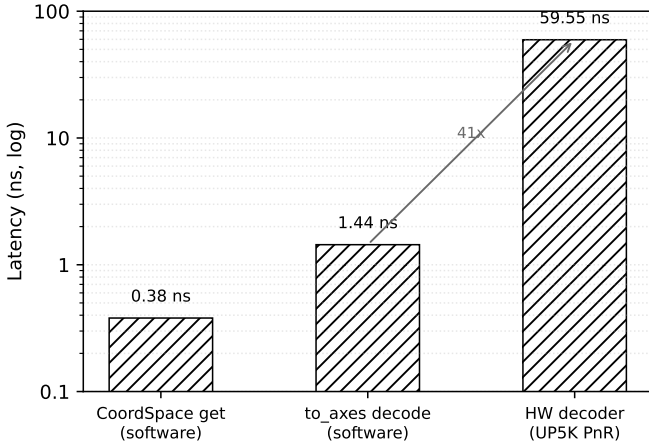


Figure 11: Per-decode latency, log scale: software reference vs hardware worst case on the UP5K. The arrow marks the software-to-hardware gap of the same decomposition.

The coordinate space maps onto physical memory as a flat array: 11,172 slots of 16 bits each, addressed by the 14-bit offset from the decoder. The store is configured as a single-port SRAM macro for SkyWater 130nm (16-bit words, 11,172 words, one read-write port): about 22 KB of raw storage, or 44 to 89 KB with values depending on the value width. Its behavior is verified today as a behavioral model over all 11,172 slots, the 5,212 reserved addresses, and the read-during-write case; the macro generation is pending the OpenRAM PDK install. 11,172 is not a power of two; the fallback is 16,384 words (2^{14}) with 5,212 reserved entries, since the 14-bit address space covers the valid range either way.

The software numbers come from the hardware-comparison baseline (sw/rust/benches/bench_hw.rs); the main benchmark center measures 0.39 ns for `CoordSpace.get [B]`.

For the $N=1$ dense case, hardware is not a speedup. The entire working set fits in L1 cache, and the software already runs at memory speed. The hardware value is on other axes: energy per access (22 KB SRAM at 16 MHz; orders of magnitude below a host cache hierarchy, an estimate pending a power model), latency determinism (fixed one cycle), physical embedding (the primitive becomes a device), and streaming throughput (one code point per clock when pipelined). The acceleration case appears at N greater than 1, where software falls back to sparse trees (0.87 to 53 ns per operation), and in embedded or IO paths where energy and determinism dominate over raw speed.

The Sky130 standard cell flow runs in CI (ORFS, hw job in the syntagma repository) and produced the first measured PDK numbers: the registered demo top at $4631 \mu\text{m}^2$ with 35% utilization, an 11.19 ns critical path with +72.33 ns slack, and 0.897 mW; the pure decoder at 388 Sky130 cells and $2826 \mu\text{m}^2$. The FPGA demo on the Upduino 3.1 closes the onboard 12 MHz clock at 16.79 MHz.

The coprocessor path (above) attaches Tagma to a linear memory system: the coordinate is decoded and the result is used as an address in conventional memory. Two extensions remain future directions:

5.6.1 Memory controller mapping

The Tagma coordinate space can be mapped to a physical address region configured to bypass virtual-to-physical translation through memory protection or non-standard memory region attributes. The controller then receives each coordinate directly as a physical address and maps `Axis0-Axis1-Axis2` to row-column within the existing memory array. General-purpose platforms support custom address mapping logic at the memory controller or system level.

5.6.2 Three-dimensional decode

The bitcell array can be organized as a $19 \times 21 \times 28$ grid (11,172 cells) rather than a linear array. Alternatively, a memory controller can map the 11,172 valid coordinates to a dense linear array through a simple packed-index transformation. The three coordinate fields each have their own decoder:

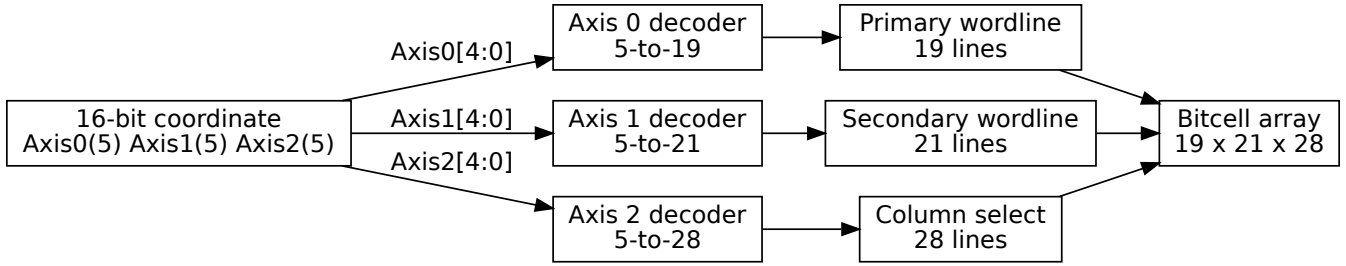


Figure 12: Three parallel decode paths from the 16-bit coordinate fields to the bitcell array wordlines.

Parameter	Linear (SRAM)	Tagma 3D (SRAM)
Decoder type	1 decoder (N-to-2 ^N)	3 decoders (5-to-19, 5-to-21, 5-to-28)
Worst-case decoder delay	O(2 ^N)	O(32) max
Address mapping	row + column	Axis 0 + Axis 1 + Axis 2
Address translation	required	none
Structural validity	external (ECC)	built-in

The bitcell array remains standard 6T SRAM. The change is in the peripheral circuitry: three independent decoders replace the single row-column decoder pair, and the sense amplifier layout must accommodate three-axis access. This increases peripheral complexity but leaves the bitcell array unchanged [9].

In a 28 nm CMOS process, a 6T SRAM high-density bitcell occupies approximately $0.127 \mu\text{m}^2$. The 11,172-cell core array therefore covers approximately $1,419 \mu\text{m}^2$ (0.0014 mm^2). With peripheral circuitry – three decoders, sense amplifiers, and wordline drivers – the total area approximately doubles to $0.003\text{--}0.004 \text{ mm}^2$, roughly 30x smaller than a single 32-bit multiplier in the same process.

5.7 Synthesis and timing

The three units are synthesized with Yosys; the decoder is also placed and routed on the iCE40 UP5K and mapped to a Sky130 standard cell library. Every figure names its library or node, and figures from different flows are not comparable.

Decoder metric	Value	Flow
generic cells	478	Yosys proc; synth, ev-compatible schema
2-input gate estimate	588	abc -g AND, NAND, OR, NOR, XOR, XNOR
iCE40 LUT4 + SB_CARRY	201 + 41	synth_ice40
iCE40 placement	255 ICESTORM_LC of 5280 (4%)	nextpnr-ice40, --freq 12
critical path	59.55 ns	icetime, host
Fmax	16.79 MHz	Homebrew toolchain icetime, host
Sky130 area	388 cells, $2826 \mu\text{m}^2$	Homebrew toolchain OpenROAD, yosys stat -liberty

The compose and distance units are measured in the 2-input gate library only: 258 and 1329 cells. The distance figure carries two decoders, because a pair needs both operands decoded; the comparators and subtractors around them are the remaining about 153 cells.

The registered demo top, which adds clocking and the LED output, is measured on Sky130 at the 12 MHz board-equivalent clock (83.33 ns):

Demo metric	Value
design area	4631 μm^2 , 35% utilization
critical path delay	11.19 ns
worst setup slack	+72.33 ns
total power	0.897 mW

Place and route numbers are tool-version dependent: the image (nextpnr 0.6, Ubuntu) measured 62.35 ns / 16.04 MHz on the demo, while the host Homebrew toolchain measured 59.55 ns / 16.79 MHz.

5.8 Hardware compliance

A hardware implementation is Tagma-compatible iff, in addition to the conditions in Section [9], it satisfies:

1. **Decode bit-exactness.** Decode is a combinational function bit-exact with `Coord::to_axes` over all 2^{16} inputs, reporting validity for the 11,172 valid values.
2. **Compose bit-exactness.** Compose is bit-exact with `Coord::from_axes`: it returns the coordinate index for an in-range axis triple and reports invalidity otherwise.
3. **Distance bit-exactness.** Distance is bit-exact with `Coord::hamming_distance` for valid operand pairs.
4. **Netlist equivalence.** The RTL and the synthesized netlist are formally equivalent over the full input space of each operation.

The hardware contract mirrors the software contract: the two agree on every input, and the verification pipeline enforces it rather than assuming it.

5.9 Status and reproduction

Realized and verified: the decode, compose, and distance units in RTL, each exhaustively simulated against the Rust reference, simulated at gate level, and formally equivalent to its synthesized netlist; the segment store as a behavioral model over all 11,172 slots, the reserved addresses, and the read-during-write case.

Pending: the OpenRAM macro for the segment store (needs a PDK install), the physical FPGA board bring-up (the bitstream is generated), and the power model over the VCD activity trace.

Reproduction: `make -C hw check` runs the whole gate (simulation, synthesis, and equivalence) from the sources recorded with this document’s version, and the `hw CI` job runs the same gate. The HDL lives under `hw/` in the syntagma repository [10].

6 Comparison with Existing Paradigms

Content-Addressable Memory provides content-based lookup in one cycle but at 9-16 transistors per bit versus 6 for SRAM [5]. At the Tagma maximum of 11,172 entries, Tagma with direct mapping uses standard SRAM cells at approximately 2.5x less transistor count than CAM, at the cost of 54,364 unused states. Tagma is more area-efficient when the stored set is sparse; CAM is more efficient when dense.

The power difference is larger than the transistor count difference. Every CAM search cycle precharges all matchlines (one per stored word) and conditionally discharges them through the comparison cells, consuming approximately 0.3–1.0 pJ per bit per search. By contrast, an SRAM read of the addressed word dissipates approximately 0.05–0.2 pJ per bit [5]. For an 11,172-entry, 16-bit array, a full CAM search activates all 11,172 matchlines and 16 searchlines simultaneously, whereas a Tagma direct SRAM read activates a single wordline and 16 bitlines, yielding an energy difference of approximately four orders of magnitude per operation.

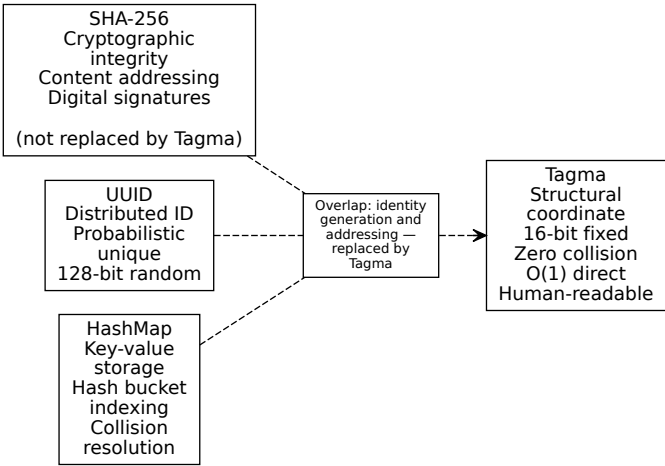


Figure 13: Tagma occupies the structural coordinate space, replacing the hash-based addressing role in all three while leaving cryptographic functions to SHA-256.

A comprehensive comparison across dimensions not covered by individual paradigms above:

Attribute	Traditional (Hash-based)	Tagma
Identifier generation	Hash (thousands of cycles)	Arithmetic (1-2 cycles)
Collision resistance	Probabilistic (finite hash space)	Guaranteed (coordinate is unique)
Memory layout	Dynamic hash table (variable size)	Fixed array (predictable size)
Concurrency	Requires synchronization	Lock-free (slot-level atomic)
Self-validation	Separate checksum/hash	Built-in (16-bit range check)
Output form	Opaque hex or Base64	Displayable as Unicode text
Structural operations	None (random access only)	Proximity search, slicing, distance
Hardware mapping	Complex (hash function gates)	Simple (decoder + address lines)

In addition to the differentiators listed above, structural coordinates enable two capabilities unavailable to hash-based addressing:

- **Proximity search:** Coordinates (x+1, y+1, z+1) are neighboring positions, enabling locality queries without index intersection.
- **Axis slicing:** All coordinates with a fixed medial value are extracted by projecting a single axis.

What Tagma replaces, partially replaces, and does not replace:

Domain	Hash Role	Tagma Replaces?	Tagma Alternative
ID generation	hash(data) to produce unique identifier	Yes	Coordinate assignment (1-2 cycles, zero collision)
Hash map key	hash(key) to compute bucket index	Yes	CoordSpace (direct array index)
Content addressing	data hash as storage address	Partial	Coordinate-to-data mapping; requires coordinator
Cache key	file metadata hash as cache key	Partial	Composite Tagma coordinate from content attributes

Existing addressing paradigms each optimize a different axis: pointers minimize generation cost (zero) but provide no content-addressability; hash functions maximize identifier space at the cost of indirection; CAM cells offer full associativity at a transistor premium. Tagma occupies a fourth category, structural addressing, where identifier, address, and structure converge into a single 16-bit value decodable at the gate level. Every coordinate maps to exactly one slot, in one cycle, with zero collision resolution. Hash-based systems accept probabilistic guarantees and variable latency; Tagma guarantees both latency and uniqueness by construction. Combined with N-Coord composition (yielding up to 1.94×10^{77} identifiers at 19 Coords), the bound dissolves for practical purposes while per-Coord determinism is preserved.

Domain	Hash Role	Tagma Replaces?	Tagma Alternative
Integrity verification	data hash to detect tampering	No	Retain SHA-256 or BLAKE3
Digital signatures	hash-then-sign for non-repudiation	No	Retain Ed25519 or ECDSA
Key derivation	HKDF for key expansion	No	Retain HKDF
Password hashing	bcrypt/Argon2 for work-factor	No	Retain bcrypt or Argon2

The rows marked No retain cryptographic functions. The tagma-sec layer (Appendix [G]) composes them with the coordinate primitives for coordination traffic, providing integrity, authorization, audit, and non-repudiation.

7 Application Domains

The deterministic single-cycle decode makes Tagma suitable for real-time and safety-critical systems. The three-decoder topology of the future path reduces worst-case decode latency compared to a single linear decoder.

Radiation-tolerant computing. The structural validity check embedded in every decode provides inherent error detection. A single-bit upset that maps a valid coordinate outside the valid range produces an immediate invalid flag. Errors that map one valid coordinate to another are not detected by the structural check alone and require ECC supplementation, as noted in Boundaries. The decoder’s small measured gate count makes triplication feasible at lower cost than protecting a full hash unit.

An exact enumeration of all 11,172 valid coordinates confirms the detection rate⁴. Each valid coordinate has 16 possible single-bit-flip destinations. Averaged across all valid states, 12.14 of those 16 destinations remain within [U+AC00, U+D7A3] (standard deviation 0.62; range 8–13). The resulting average SEU detection rate is 24.1% from the structural check alone, before any ECC supplementation. This detection rate comes at zero additional hardware cost, as a free byproduct of the structural encoding. When combined with ECC, the structural check handles the subset of errors that map valid coordinates outside the valid range, while ECC handles the remaining cases.

Real-time object identification. In sensor fusion for autonomous systems, Tagma coordinates serve as deterministic object identifiers that do not require hash computation or lookup tables. Each new object is assigned a coordinate at encoding time; subsequent frames reference the same coordinate without recomputation.

Proximity search. The three-axis structure enables field-wise Hamming distance computation through parallel XOR gates, supporting nearest-neighbor lookup without CAM cells or hash-based index intersection.

KV cache addressing. LLM inference engines maintain key-value caches indexed by token sequence prefixes. Current implementations use hash tables or radix trees with $O(L \times H)$ cost per lookup (sequence length L , hash cost H). Tagma replaces this with CoordPath-based direct access: each prefix maps to a unique CoordPath of length L , and lookup cost is $O(L)$ array accesses with zero hash computation. Production KV cache sizes (typically 10^4 – 10^7 entries) are covered by 2–4 Coords (1.25×10^8 to 1.55×10^{16} identifiers), with deterministic $O(1)$ access and no collision resolution.

Graph adjacency and multi-dimensional query. Graph engines check adjacency via hash lookups or index intersections ($O(\deg(v))$) and query multi-dimensional attributes via composite indexes or join operations. Tagma represents each node as a Coord and each edge type as a CoordSet; adjacency reduces to a single bitwise AND over 175 machine words. Multi-dimensional queries (e.g., “nodes with Axis 0=a, Axis 1=b”) project directly to axis ranges without index intersection, with cost independent of graph size.

Secure coordination traffic. Routing updates, resolver evidence, and audit trails name target paths as CoordPaths. The tagma-sec layer turns the structural path into a security object: authorization over scopes, integrity seals

⁴Computed by exhaustive enumeration: for each of the 11,172 valid Coord values, all 16 single-bit-flip neighbors are tested against the valid range [U+AC00, U+D7A3].

over records and epochs, chained audit evidence, and non-repudiation receipts. Keyed primitives provide the cryptographic guarantees over public coordinate arithmetic. Design and benchmark details are in Appendix [G].

8 Boundaries

The 11,172-identifier bound per Coord is a consequence of the $19 \times 21 \times 28$ composition formula, not a configurable parameter. It defines the single-Coord direct-address range. Applications requiring larger identifier spaces compose multiple Coords via CoordPath (Section N-Coord Composition). The decoder’s structural validity check does not eliminate the need for full error-correcting codes: single-bit errors that map one valid coordinate to another are not detected.

Tagma does not replace cryptographic primitives. SHA-256 remains for signatures, Merkle proofs, and preimage resistance. Encryption, authentication, and key derivation are outside its scope. Tagma replaces the use of hashes as structural identifiers and addresses. Coordination traffic obtains these guarantees from the tagma-sec layer (Appendix [G]), which composes keyed primitives with the coordinate primitives to provide integrity, authorization, audit, and non-repudiation.

For applications requiring content determinism, where the same data must always produce the same identifier, SHA-256 provides content fingerprinting while Tagma provides human-readable encoding of that fingerprint:

Property	SHA-256 hex	SHA-256 with Tagma
Output	64 hex characters	19 Coords
Determinism	Yes	Yes (SHA preserved)
Human readable	No	Yes
Self-validating	No	Yes (each Coord checked)
Collision resistance	2^{256}	2^{256} (SHA preserved)

The combination serves use cases such as file identification, content-addressed storage, and commit hashing where determinism is required but hex output is not. An implementation example combining SHA-256 with Tagma encoding is provided in Appendix [E].

9 Compliance

An implementation is Tagma-compatible iff it satisfies all of the following conditions:

1. **Composition correctness.** The composition formula $C(i, m, f) = U + AC00 + 588i + 28m + f$ must produce the correct Unicode code point for every valid combination of axes ($19 \times 21 \times 28 = 11,172$ triplets).
2. **Structural validity.** Every 16-bit value in the range $[U + AC00, U + AC00 + 11,172)$ must decode to a valid (i, m, f) triplet. Every value outside this range must be rejected, including the 12 filler positions $U + D7A4..U + D7AF$ within the Unicode block but outside the composition formula. Total: 11,172 valid values and 54,364 invalid values in the 16-bit space.
3. **Decomposition correctness.** Decomposition must be the functional inverse of composition: $decompose(compose(i, m, f)) = (i, m, f)$ for all 11,172 valid triplets.
4. **Linearization uniqueness.** The linearization function must be injective over the N-Coord product space. Distinct N-Coord tuples must produce distinct linear indices.
5. **Bit-exactness.** All implementations must produce identical results for the same input across all languages, platforms, and hardware configurations: composition, decomposition, and linearization.
6. **Coord atomicity.** Coord is a single-Coord atomic value. An implementation must not impose application-level semantics on Coord’s three axis fields or assume any particular storage strategy for CoordPaths. Coord’s only invariant is structural validity.

10 Vision

Tagma defines a new type of silicon primitive: a combinational decoder that derives identity from structure in a single cycle. At a cost measured in hundreds of gates, content-addressable operation becomes viable where hash-based approaches are too expensive in power, area, or latency. The decoder is verified exhaustively over all 11,172 valid characters in three simulation channels, plus formal equivalence to the synthesized netlist over all 2^{16} inputs; the compose and distance units are verified the same way. The structural validity check embedded in every decode provides inherent error detection, relevant for radiation-tolerant computing in space environments. The coordinate space that enables this is this Unicode block, an open international standard and a public good. Tagma is released as open-source hardware in hw/ in the syntagma repository, with the FPGA demo closing the 12 MHz board clock at 16.79 MHz, inviting the next conversation: what else becomes possible when identity costs less than a single multiply. The broader hardware design implications of this shift are discussed in Appendix [D].

The SynTagma specification [11] complements this document. It defines how the recursive state space expansion described in the previous section is realised across physical topologies: routing, transport framing, device-boundary resolution, and distributed coordination. Where this document defines the invariant, SynTagma defines the protocol.

References

- [1] Unicode Consortium, “Hangul syllables (u+AC00–u+D7AF).” Unicode Standard, Chapter 3: Conformance, 2020. Available: <https://www.unicode.org/charts/PDF/UAC00.pdf>
- [2] Unicode Consortium, “The arithmetically specified decompositions of precomposed hangul syllables.” Unicode Standard, Chapter 3: Conformance, Section 3.12, 2024. Available: <https://www.unicode.org/L2/L2003/03203-arithmetic-decomp.pdf>
- [3] K. Karlsson, “The arithmetically specified decompositions of precomposed hangul syllables.” Unicode Technical Report L2/03-203, Jun. 12, 2003. Available: <https://www.unicode.org/L2/L2003/03203-arithmetic-decomp.pdf>
- [4] P. Leach, M. Mealling, and R. Salz, “RFC 4122: A universally unique IDentifier (UUID) URN namespace.” Internet Engineering Task Force, Jul. 2005. Available: <https://datatracker.ietf.org/doc/html/rfc4122>
- [5] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey,” *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006, doi: 10.1109/JSSC.2005.864128.
- [6] A. Banerjee and S. W. Hussain, “An 8T single bit-line content addressable memory cell for high-performance searching applications,” in *2024 international conference on microelectronics (ICM)*, 2024, pp. 1–6. doi: 10.1109/ICM63406.2024.10815912.
- [7] OpenHW Group, *Core-v eXtension interface (CV-x-IF) specification*. 2023. Available: <https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/>
- [8] OpenHW Group, *CV-x-IF interface and coprocessor: CVA6 user documentation*. 2023. Available: https://cva6.readthedocs.io/en/latest/01_cva6_user/CVX_Interface_Coprocessor.html
- [9] Intelligent Computing Research Group, *OpenRAM: An open-source static random access memory compiler*. 2022. Available: <https://github.com/VLSIDA/OpenRAM>
- [10] SSCCS Initiative, “Tagma: Content-addressable structural primitive.” GitHub repository, 2026. Available: <https://github.com/ssccsorg/syntagma>
- [11] SSCCS Initiative, “synTagma: Spatial coordinate space computing system based on tagma.” SSCCS Document Suite, 2026. Available: <https://docs.ssccs.org/projects/syntagma/tagma/>
- [12] SSCCS Initiative, “synTagma benchmark suite.” GitHub repository, 2026. Available: <https://github.com/ssccsorg/syntagma/blob/main/sw/rust/benches/bench.rs>

- Whitepaper: PDF / HTML DOI: 10.5281/zenodo.18759106 via CERN/Zenodo, indexed by OpenAIRE. Licensed under *CC BY-NC 4.0*.
- Official repository: GitHub. Authenticated via GPG: BCCB196BADF50C99. Licensed under *Apache 2.0*; hardware sources (HDL, constraints, synthesis and PDK flow) under *CERN-OHL-P v2*.
- Governed by the Foundational Charter and Statute of the SSCCS Initiative (in formation).
- Provenance: Human-in-Command, AI-assisted. Aligns with ISO/IEC JTC 1/SC 42 and C2PA-certified. Full intellectual responsibility with author(s).

Appendices

A Reference Implementation

The coordinate space is implemented as a multi-crate Rust reference implementation⁵. The `Coord` type is defined in the core library and is bit-exact with the hardware decoder. The workspace is published on GitHub [10].

A.1 Core Types

Type	Description	Key property
Coord	16-bit newtype valid in $[0, 11171]$. Three-axis decomposition, composition, Hamming distance, Unicode display.	Bit-exact with hardware decoder
CoordPath<N>	Compile-time N -element <code>Coord</code> array. Index path through multi-level address table. Each element is a direct array index at the corresponding tree depth.	No hashing, no equality comparison
CoordSet	Fixed-size bit array over 11,172-coordinate space. $[\text{u64}; 175]$ (1.4 KB, zero heap, Copy). Bitwise union, intersection, difference.	Single-bit ops; 175-word AND for compound axis filter

These three types are always available (no allocator required). With the `alloc` feature, the `Space` family below is added.

The **CoordPath** types above treat coordinate space as a tree: index paths through multi-level arrays. **CoordCube** (from `tagma-geo`) reinterprets the same `CoordPath` keys as **D-dimensional coordinates**, enabling proximity, bounding box, and distance queries that `CoordPath` alone cannot express [C]. The two access patterns share the same underlying storage; `CoordCube` is a zero-cost view (construction 0.96 ns, axis extraction at raw path speed 319 ps).

A.1.1 Space Family

The `CoordSpace` series provides hash-free, collision-free coordinate-indexed spaces backed by direct array addressing.

Type	Depth	Address space	Allocation	Latency	Exploration pattern
<code>CoordSpace</code>	1	11,172	None (inline array)	0.39 ns	General single-Coord, no_std, MCU
<code>CoordSpace2</code>	2	1.25×10^8	Dense heap (119 MB)	0.39 ns	Two-axis space
<code>CoordSpaceM3</code>	3	1.39×10^{12}	mmap (1.27 TB)	0.39 ns	Large-scale dense space
<code>CoordSpaceN2</code>	2	1.25×10^8	Heap (lazy)	0.90 ns	Two-axis space
<code>CoordSpaceN6</code>	6	1.94×10^{24}	Heap (lazy)	5.44 ns	Below UUID
<code>CoordSpaceN19</code>	19	$2^{255.5}$ (8.2×10^{76})	Heap (lazy)	58.6 ns	Just below SHA-256
<code>DynCoordSpace</code>	Runtime	Unlimited	Heap (lazy)	N/A	Variable-depth paths

⁵Tagma Reference Implementations: docs.sscs.org/projects/syntagma/impl

A.2 Serialization: base11172

A `no_std + alloc` crate providing Tagma's native serialization format. Every coordinate index 0..11171 maps to exactly one Unicode character (U+AC00 + index). A pair of Coords encodes a 16-bit value. The encoding is self-validating: characters outside U+AC00..U+D7A3 are immediately detectable as invalid. No special characters, padding, escaping.

Code-level analysis of all types – including `Coord` bit layout, `CoordSpace` inline array with niche optimization, `CoordSet` bit iteration with `trailing_zeros`, and `CoordSpaceN` sparse tree with lazy node allocation – is in the software reference implementation.

B Benchmarks

A SHA-256 engine requires approximately 10,000 gates and 64–75 cycles per operation, then needs collision resolution and dynamic resizing. UUID generation requires entropy collection and delivers probabilistic uniqueness. The Tagma decoder replaces this with a combinational decoder and a 16-bit register: one `Coord` covers 11,172 identifiers; six Coords (18 axes) exceed typical distributed system needs; nineteen Coords reach $2^{255.5}$; twenty exceed the SHA-256 2^{256} space.

- Lookup latency: native `CoordSpace` (dense array) is flat at **0.39 ns** across all depths – every `Coord` resolves to a single array load. The tree fallback (`CoordSpaceN`) scales linearly with depth: 2.69 ns at $N=3$, 58.6 ns at $N=19$ ($2^{255.5}$, **↑3.9x** vs SHA-256's 227 ns), and 62 ns at $N=20$ (2^{269} , **↑3.7x**). Native `CoordSpace` reaches **582x** vs SHA-256. Recursive depth is bounded by schema, not data volume: 10^4 and 10^{77} entries both cost N dereferences in the fallback path, while the native dense path costs a constant 0.39 ns.
- Nonexistent prefix lookup: `CoordSpace` **1.65 ns** (structural, navigates to the branch and returns `None`) vs `HashMap` **23.05 ms** (**↑14.0Mx**, full scan – `HashMap` has no structural prefix index). Sparse get at 10M entries: `CoordSpaceN2` completes all 10M operations in 44.9 ms vs `HashMap` 1.05 s (**↑23.4x**).

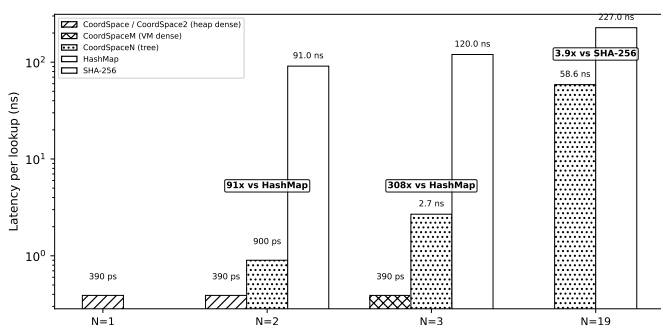


Figure 14: Identity generation latency

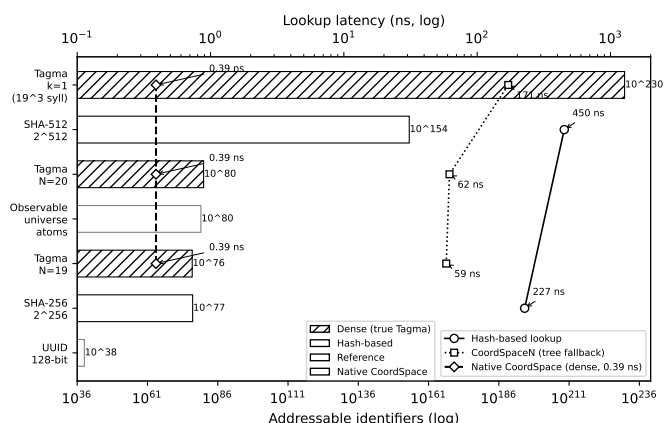


Figure 15: Addressable space (bars, log) and lookup latency (line, right axis).

- Identity generation: SHA-256 lookup costs 227 ns; the tree fallback (`CoordSpaceN`) exceeds 2^{256} at 20 Coords for 62 ns (**↑3.7x**), while the measured 19-Coord depth costs 58.6 ns (**↑3.9x**). The native dense path (`CoordSpace`, `CoordSpace2`, `CoordSpaceM3`) holds at a flat 0.39 ns.
- Address space: tree fallback lookup cost scales as $O(N)$; native dense path is $O(1)$ flat. Tagma recursion $k=1$ reaches 5.5×10^{230} identifiers (about 10^{76} times the SHA-512 space) at 171 ns.

Tagma assigns every point in a geometric space a structural address that is simultaneously a coordinate, an identifier, and a computation target. `HashMap` stores values by hashing keys by comparison. Querying this space is spatial computation: axis projection, set membership, proximity, and coordinate slicing are arithmetic operations. The figures above measure the consequence: `HashMap` degrades with data volume; the coordinate space does not.

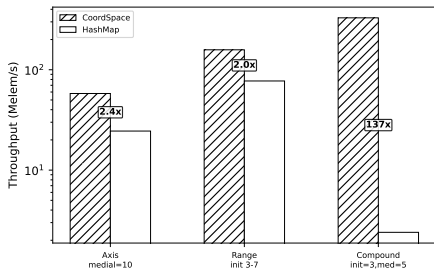


Figure 16: Spatial query throughput

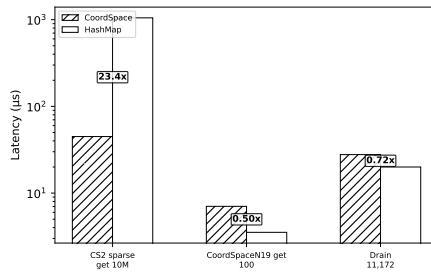


Figure 17: Edge cases: sparse get, deep get, drain

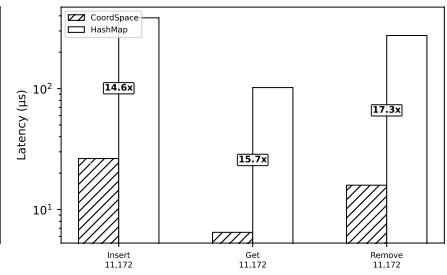


Figure 18: Bulk operations

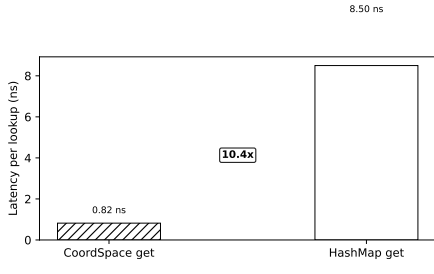


Figure 19: Single-get mark microbench

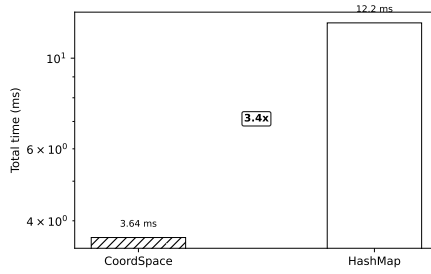


Figure 20: Mixed workload (500k ops)

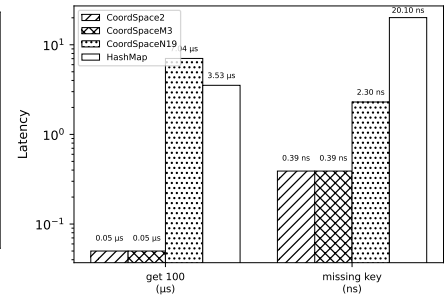


Figure 21: Deep tree: get and non-existent key across the CoordSpace family

Rust's `std::collections::HashMap` compiles to C-grade machine code within 5-10% of theoretical CPU throughput. Whether Tagma matches or exceeds this baseline is incidental: HashMap degrades linearly with collision rate and entry count while Tagma does not. A coordinate-slice query costs the same at 10^4 entries as at 10^{77} entries: one array dereference per Coord.

Metric	SHA-256	CoordSpace (N=1)	CoordSpace2 (N=2)	CoordSpaceM3 (N=3)	CoordSpaceN6 (tree)	CoordSpaceN19 (tree)
Latency	227 ns	0.39 ns	0.39 ns	0.40 ns	5.44 ns	58.6 ns
Backing	hash	inline array	heap alloc_zeroed	mmap MAP_NORESERVE	sparse tree	sparse tree
Allocation	per-entry heap	22 KB	119 MB	1.27 TB (virtual)	per-node	per-node
Identity size	32 bytes	2 bytes	4 bytes	6 bytes	12 bytes	38 bytes
Addressable space	2^{256}	1.12×10^4	1.25×10^8	1.39×10^{12}	1.94×10^{24}	8.21×10^{76}
Collision	probabilistic (2^{-128})	zero	zero	zero	zero	zero
Native	—	Yes (dense)	Yes (dense)	Yes (dense)	No (fallback)	No (fallback)

All figures are software measurements on ARMv8.4-A Firestorm (2020), compiled with `rustc` stable in release mode. Full benchmark source is included in the repository [12].

C CoordCube: Spatial Interpretation Layer

The CoordCube layer reinterprets existing CoordPath storage keys as D-dimensional coordinates without modifying the underlying key, enabling proximity queries, bounding box enumeration, and distance metrics that fall

outside the core CoordPath scope. The full design, benchmarks, and comparison with existing systems are described in the TagmaGeo whitepaper⁶; the reference implementation lives in `sw/rust/geo`⁷.

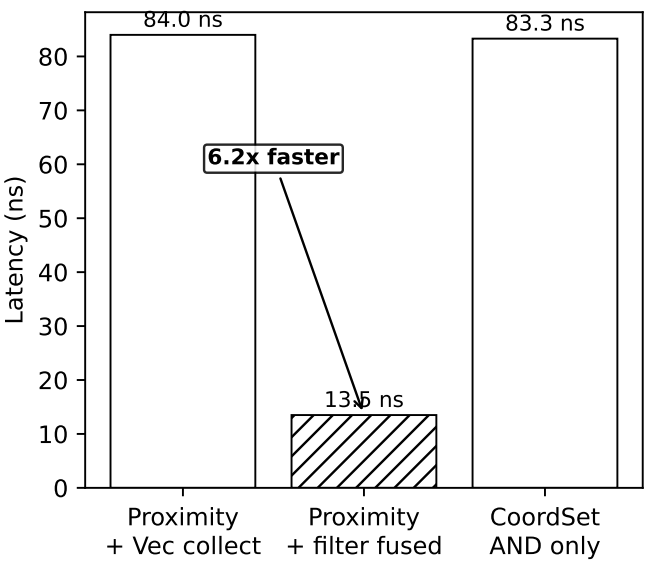


Figure 22: Compound query efficiency: iterator fusion vs collect-then-filter

Query	CoordPath	CoordCube
Point lookup	O(k) direct	O(k) direct
Neighborhood of P	external index required	proximity(r)
Bounding box	manual loop	bounding_box()
Distance metric	manual compute	1.75 ns hamming
Scale cost (10M entries)	O(k log N)	bounded O(paths)
Empty region check	O(k log N) None	15.7 ns immediate
Compound axis filter	O(N) scan	85.7 ns AND

D Hardware Design Implications

Tagma changes the problem that hardware must solve alongside the speed at which it solves it. Conventional hardware spends area and energy on finding data through hash computation, cache tag matching, and address translation. Tagma replaces finding with knowing: the coordinate is known at encoding time, so the hardware need only decode.

Layer	Conventional approach	Tagma-based approach
ISA	Instructions compute or look up addresses	Instructions carry Tagma coordinates as direct operands
Pipeline	Branch prediction, cache miss handling	Predictable access patterns from coordinate regularity
Cache Accelerator	Tag comparison, associative lookup Dedicated hash unit for DHT or content addressing	Direct-indexed cache lines, no tag match Coordinate arithmetic only; hash unit eliminated

⁶TagmaGeo whitepaper (draft): docs.sscs.org/projects/syntagma/tagma/geo
⁷TagmaGeo reference implementation: github.com/ssccsorg/syntagma/tree/main/sw/rust/geo

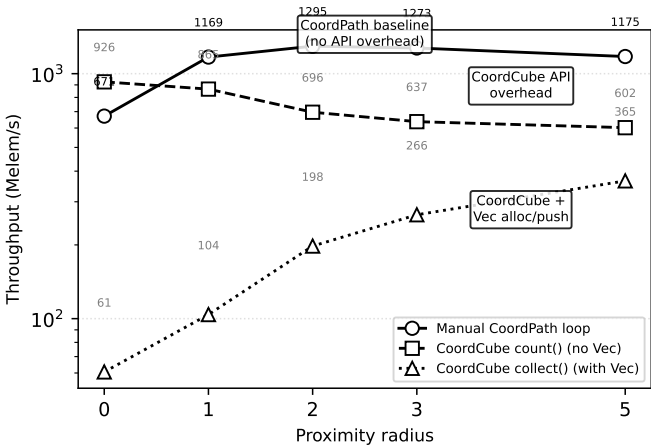


Figure 23: Proximity path generation throughput: manual CoordPath vs CoordCube

Layer	Conventional approach	Tagma-based approach
Energy	Dynamic voltage scaling to cover worst-case hash latency	Fixed, minimal decode path; predictable power

Conventional hardware searches. Tagma hardware interprets. This shifts the hardware design problem from faster computation to simpler decoding. The measured trade is explicit: timing closure costs about 2.5x gates [5.2], and the N=1 dense case is examined in Section [5.6]. All three operations of the primitive are realized and verified in RTL; the reference implementation and its remaining work are stated in Section [5].

E SHA-256 with Tagma Encoding

SHA-256 output is uniformly distributed, so the modulo-11172 mapping to each Coord value is statistically unbiased. The overall collision probability of the 20-Coord output remains 2^{-256} , preserved from the underlying hash. The 20-Coord space (2^{269}) exceeds the 2^{256} domain, so no collision is forced by the encoding.

F TagmaMap⁸: Key-Value Store on Coordinate Primitives

TagmaMap builds a practical key-value storage engine on top of the coordinate primitives described in this document. Where the core Tagma library provides collision-free $O(1)$ addressing within a single address space, TagmaMap extends the model to handle legacy infrastructure requirements that fall outside the core library scope: multi-node coordinated sharding, persistence to backing stores, protocol adaptation (Redis RESP, S3 REST), and operational tooling.

The structural addressing model handles what legacy systems delegate to hash functions and index structures: key placement, collision resolution, and range partitioning. TagmaMap supplies what the coordinate model intentionally abstracts away: durable storage, wire protocols, and cluster management.

The full design and benchmark results are described in the TagmaMap whitepaper; the reference implementation lives in `sw/rust/map`⁹. Selected CoordCube benchmarks that measure KV-relevant metrics are reproduced below.

F.1 Throughput: Store Density and Proximity

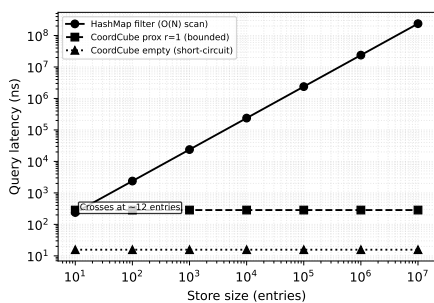


Figure 24: Query cost vs store size:

CoordCube proximity vs
HashMap filter

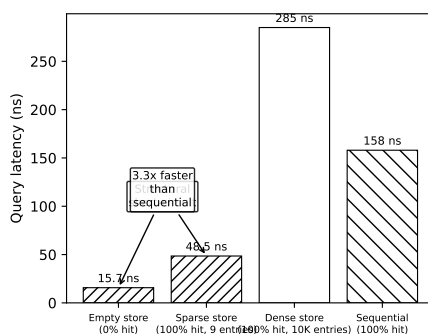


Figure 25: CoordCube vs sequential
lookup: hit-rate crossover

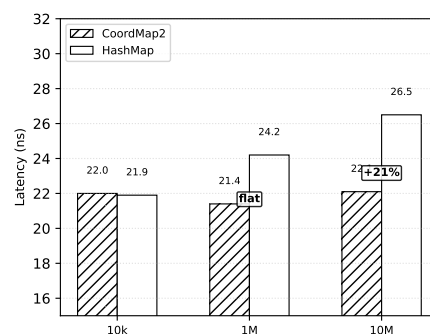


Figure 26: Scale-invariant KV get
latency: CoordMap2 vs
HashMap

⁸TagmaMap whitepaper (DOI: 10.5281/zenodo.21550431): docs.sscs.org/projects/syntaxagma/tagma/map

⁹TagmaMap reference implementation: github.com/ssccsorg/syntaxagma/tree/main/sw/rust/map

- CoordCube proximity on dense stores adds 127 ns overhead over sequential lookup, but this overhead is dominated by Vec allocation/push (87%), not coordinate arithmetic (13%).
- On sparse stores, CoordCube is up to 3.3x faster than sequential lookup (48.5 ns vs 158 ns) because it avoids tree lookups for nonexistent paths.
- On empty stores, CoordCube returns immediately at 15.7 ns (pure path generation cost) — sequential lookup still pays 158 ns for 9 tree misses.
- The crossover point where CoordCube becomes faster than sequential is at roughly 55% hit rate. Below this, generating paths and checking is cheaper than looking up known paths.
- CoordCube query cost is bounded by region size (path count), not store size. HashMap spatial queries cost $O(N)$ full scan. At 10M entries, CoordCube proximity is 285 ns vs HashMap filter at 238 ms — a million-fold advantage.
- Hierarchical queries (CoordCube proximity + manual post-filter) are faster than direct KV proximity on multi-character dimensions (547 ns vs 639 ns).

F.2 Throughput: General KV Operations

- Edge: CS2 sparse get sustains 23.4x at 10M entries; CS19 get shows 19-dereference cost (0.50x); drain is 0.72x on the full space.
- Bulk operations: CoordSpace outperforms HashMap by 14.6–17.3x across all operations on the full 11,172-entry space.
- Single-get microbenchmark isolates the per-operation cost: 0.82 ns vs 8.50 ns.
- Stress test: under 500,000 interleaved insert, get, remove, and update operations, CoordSpace completes in 3.64 ms vs HashMap 12.2 ms.
- Deep tree: CoordSpaceN19 at 100 entries shows the 19-dereference tree traversal cost (7.04 us vs 3.53 us for HashMap). Nonexistent key lookup remains depth-independent: CoordSpace2 0.39 ns, CoordSpaceN19 2.30 ns, HashMap 20.1 ns.

G Tagma Security¹⁰: Security Layer on Coordinate Primitives

Tagma Security builds the security primitive layer for coordination traffic on top of the coordinate primitives described in this document. Routing updates name target paths, resolvers exchange evidence, and the audit trail records what happened; Tagma Security answers the accompanying questions: which principal may act on which path, whether a record is intact, whether the origin of an update can be denied, and what evidence remains. The layer provides integrity, authorization, audit, and non-repudiation over Tagma coordinate objects and coordination traffic.

The security model rests on keyed primitives over public coordinate arithmetic. The composition formula, the linearization rule, and the validity bounds are public specifications, so coordinate structure contributes collision-free addressing while the security guarantees come from keyed hashing (blake3 in the reference implementation). Four modules expose small interfaces: authority (CoordPath Exact/Prefix scope authorization, epoch-scoped revocation), integrity (epoch-bound seals), audit (chained evidence log with inclusion proofs), and channel (non-repudiation receipts). The implementation covers specification milestones 1 to 4 with a legacy pattern as a reverse-verification mirror: the same workflow suite must pass identically on both stacks.

- Route update workflow: 696.1 ns, the exact sum of its module costs (authorize 36.8, seal 171.4, verify 129.2, append 110.3, append 110.3, exchange 132.7). The proxy and trait-object dispatch add no measurable overhead.
- Epoch replay detection: the four-way seal (record, path, principal, epoch) costs 171.4 ns against 138.8 ns for the record-and-path legacy seal, a 33 ns price for binding the epoch.
- The tagma-sec pattern adds ~87 ns per update over the legacy baseline (696.1 vs 609.5 ns), the price of replay detection.

¹⁰Tagma Security whitepaper: docs.ssccs.org/projects/syntaxtagma/tagma/security

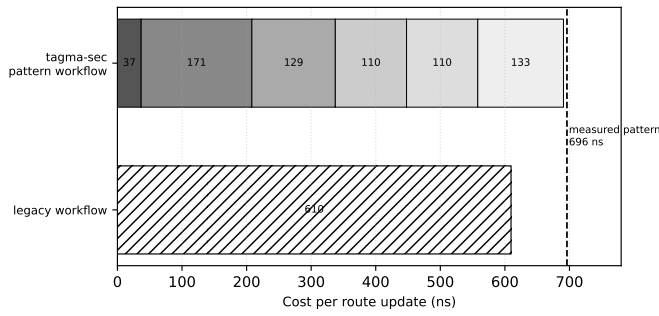


Figure 27: Route update cost: the tagma-sec pattern workflow equals the sum of its module costs

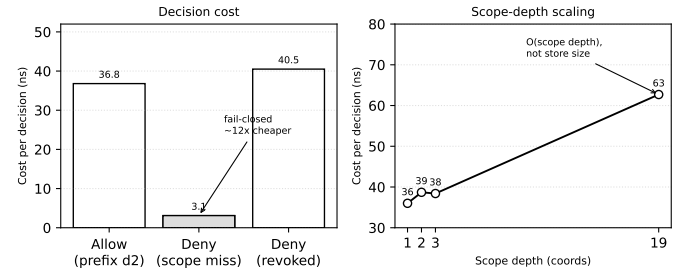


Figure 28: Authorization decision cost and scope-depth scaling

- Authorization is $O(\text{scope depth})$: 36.8 ns for a 2-coord prefix scope, 62.7 ns at 19-coord depth, independent of store size. Deny short-circuits at the first mismatching coord (3.1 ns, ~12x cheaper than Allow); a revoked scope adds one map lookup (40.5 ns).
- Audit chaining: append costs 110.3 ns (keyed payload hash); chain verification over 10,000 entries costs 9.9 μs , about 1 ns per entry prev-link walk with no hashing. Prove and export are about 20 ns per entry and memory-bound, so offline investigation scales with evidence size rather than log size.

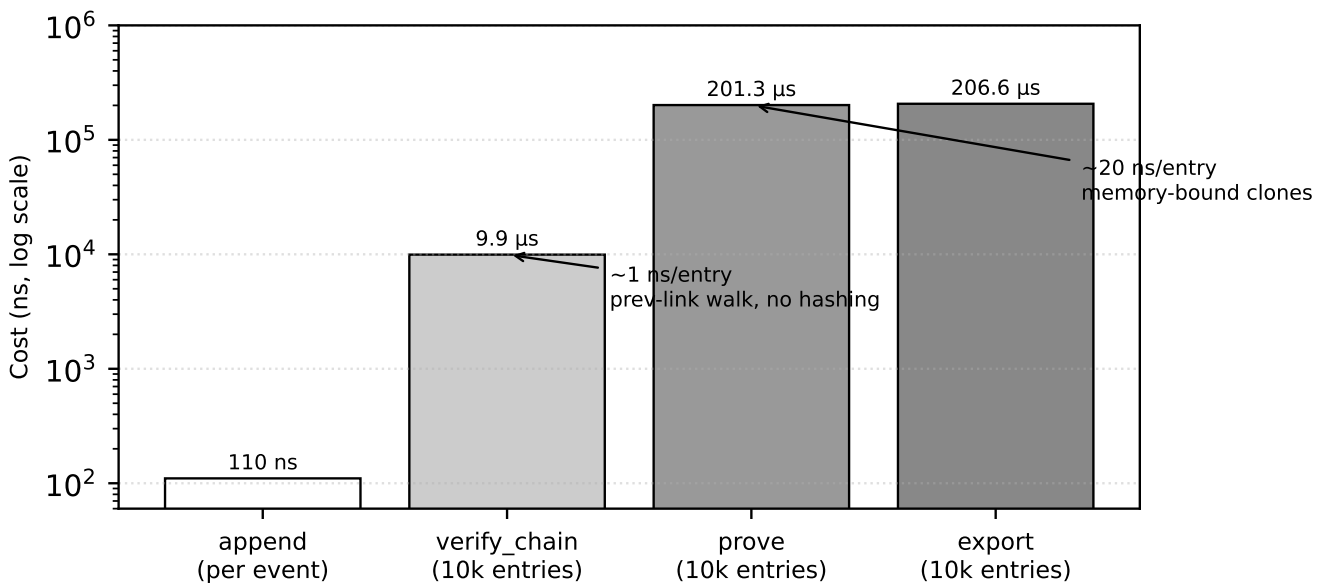


Figure 29: Audit operations: append cost and 10k-entry verification scale

The full design, benchmark charts, and implementation status are described in the Tagma Security whitepaper and in the Tagma security layer specification¹¹ in the synTagma repository.

H TagmaMatrix¹²: Coordinate-Addressed Matrices

TagmaMatrix carries the coordinate model to numeric arrays. An element of a rank-2 matrix already has an address in this system: a row and a column are two Coords, and two Coords are a CoordPath<2>. Where the core library addresses a path and TagmaMap addresses a stored value, TagmaMatrix addresses the element a path names when the elements form a structure the program computes with.

¹¹Tagma Security Layer Specification: github.com/ssccsorg/syntagma/blob/main/docs/spec/tagma-sec.md

¹²TagmaMatrix whitepaper (draft): docs.ssccs.org/projects/syntagma/tagma/matrix

The crate makes one property checkable. The physical order of the backing bytes is a type parameter and never part of the address, so the same coordinate names the same element whether the bytes are stored row-major or column-major: the same logical matrix in the two orders answers the same coordinates with the same values, produces a byte-identical product, and writes identical bytes. That is relocation invariance, held by test, and it is what lets a matrix move between devices without changing its identity. A wire form follows from the same statement, because a matrix can cross as its coordinates and their values while the receiver lays the bytes out as it likes.

The crate adds the integer product over the elements, i8 by i8 into i32, whose accumulator is bounded by the coordinate space itself at 181,612,032, so no widening is needed at any addressable width. It is the family member that takes no allocator: tagma-core is taken with its defaults off, and a link into a program with no operating system and no global allocator is part of the crate’s verification.

The full design and the verification surface are described in the TagmaMatrix whitepaper; the reference implementation lives in `sw/rust/matrix`¹³.

I Structural Enumeration in Practice: RISC-V Verification

ExaVerif¹⁴ exhaustively verifies RISC-V custom instruction encodings. Its standard pipeline generates the full Cartesian product of field domains and filters each combination through constraint checks. At the CVA6 CV-X-IF space (5 fields, 33,554,432 raw combinations), this takes 29.7 seconds and yields 229,376 valid encodings.

The Tagma-based structural pipeline replaces post-hoc filtering with structure-preserving generation. Cross-field constraints (`funct3`→`funct7` mapping, `oneof`, `enable_mask`) are encoded into a `DynCoordSpace<CoordSet>` during enumeration setup. The iterator visits only combinations that satisfy those constraints a priori. Constraint evaluation on visited combinations is identical to the standard pipeline.

The two charts below capture the result. The first shows the speedup across all fixtures from the standard pipeline to structural enumeration: 79x at Ibex scale, 766x at CVA6 R4 scale. The second zooms into the CVA6 encoding space, showing the full 33M space in 29.7 seconds versus 31.3 milliseconds — a 950x reduction in verification time.

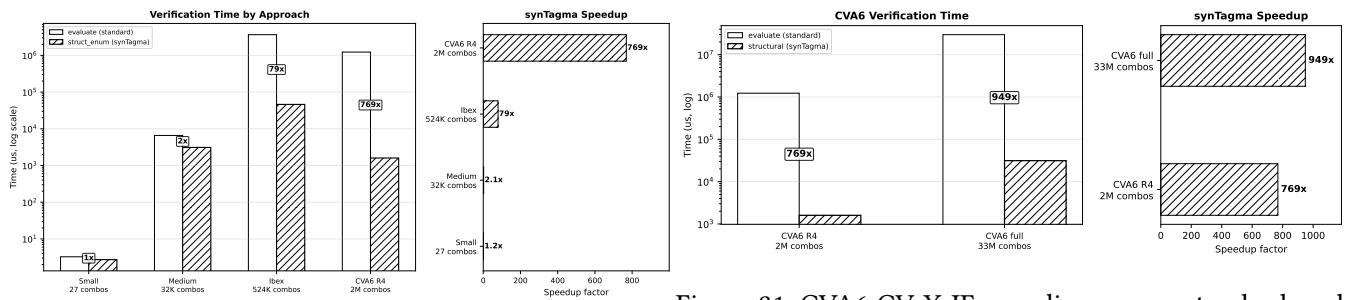


Figure 30: Verification time: standard evaluate vs structural enumeration across four fixture sizes. At CVA6 R4 2M scale, structural enumeration achieves 766x speedup.

Figure 31: CVA6 CV-X-IF encoding space: standard evaluate (29.7s for 33M) vs structural_verify with all constraint checks (31.3ms), a 950x speedup.

Fixture	Raw space	Valid space	Density	Standard evaluate	Structural verify	Speedup
Ibex	524,288	92,160	17.6%	3.66 s	46.1 ms	79x
R-type						
CVA6 R4	2,097,152	12,288	0.6%	1.23 s	1.60 ms	766x
CVA6 full	33,554,432	229,376	0.7%	29.7 s	31.3 ms	950x

¹³TagmaMatrix reference implementation: github.com/ssccsorg/syntagma/tree/main/sw/rust/matrix

¹⁴Project ExaVerif, Exhaustive Verification for RISC-V Custom Instructions docs.sscs.org/projects/ev

The speedup is not algorithmic optimization. It is a change in enumeration strategy: the standard pipeline generates all 33,554,432 combinations then filters, while the structural pipeline generates only the 229,376 valid combinations by construction. Every combination that the structural pipeline visits, it evaluates with the same constraint checks as the standard pipeline. The 99.3% of the space that is invalid is never allocated, never iterated, and never tested — this is not faster filtering; it is the absence of filtering.

J Petabyte-Scale Science Software: CERN ROOT TTree

ROOT is the data analysis framework of the CERN community, and its TTree columnar format stores the events of nearly every physics analysis. A documented production bottleneck limits the read path: a Fermilab CCESOP analysis over CMS NanoAOD issues 372,000 singular reads averaging 4.6 KB, sustains an effective 33 KB/s, and runs for roughly 14 hours. The measured cause is per-request I/O overhead (system call, cache lookup, context switch), so the cost is per request, not per byte.

A fork of CERN’s ROOT replaces branch, basket, and cache traversal with closed-form coordinate arithmetic: an event addressed as (run, luminosity block, event number) resolves to a direct byte offset with no hash and no index scan, while the TTree API and existing analysis code stay unchanged. The store is a fixed-width projection of each event, 320 of the tree’s 1,380 leaves with the 276 array fields truncated to their leading element. The table compares the read paths on the full CMS Run2016G DoubleMuon NanoAOD first file (2,315,223 events, 2,155,974,646 bytes).

Metric	ROOT TTree baseline	Tagma coordinate	Tagma mapped
Full-file read	187.0-188.2 s	2.71-2.81 s (66.9-69.1x)	1.55-1.61 s (116.6-120.5x)
Read throughput	11.5 MB/s	2,190 MB/s	3,819 MB/s
Reads per event	0.20	1.00	1.00
Syscalls per event	0.20	1.00	0.00
Analysis workload (20,861 events)	187.9 s	2.72 s (69.2x)	n/a

The ratios compare against the all-branch baseline, and the rows move different payloads: 2.15 GB of compressed file against 5.93 GB of raw records, so the throughput figures are not like for like. The component controls narrow the claim. Decompression removal accounts for 2.2x to 2.3x; against an uncompressed baseline reading the same columns the store is 13.1x, or 22.8x mapped; and against ROOT’s best configuration for that payload, 40.5x, or 70.5x mapped. The store’s cost does not fall with the number of columns an analysis selects, so on the two-column selection of the analysis workload ROOT’s column-selective reader is 4.9x faster, and the store is the cheaper unit above a crossover near 11 columns for the projection, or about 6 with the store mapped. A one-time conversion of the full dataset costs 226.3 s, so the mapped path breaks even within the second read pass.

The analysis workload reproduces the baseline exactly (20,861 selected events, histogram mean 132.696), and the served bytes match the conversion checksum (233,262,869,086). The full report is the ROOT TTree I/O report¹⁵.

The fork has since carried the store past the projection. The store now carries the whole event, 974 scalar fields and 19 collections at two reads per event, with the read path at 10.2 s against the baseline’s 176.7 s; the whole-event row with the branches active runs 117.6 s, 1.5 times, because the entry layer rather than the read path decides that comparison. The store compresses under the same addressing, 7,082,290,985 to 2,897,372,833 bytes, 2.44 times, with the read path at 12.4 s and 572 MB/s, while the compressed store file remains 1.34 times the baseline file’s 2,155,974,646 bytes.

¹⁵Solving CERN’s ROOT TTree I/O bottleneck: docs.sccs.org/works/cern/root-ttree/; measured evidence reproducible from the ROOT fork, its benchmark, and the measured artifact that records every row.

tagma-doi.org/10.5281/zenodo.21302508-caadaa-261002

The regime the work was aimed at is event-selected access. Reading selected events in list order rather than a scan, the store reads the whole event, the index record and the slice with the 1,380 field branches delivered, in 0.101 s scattered against 0.109 s sequential over 2,000 events, two reads per event, against the baseline’s 202.2 s and 682.6 requests per event: 2,000 times payload for payload and delivery for delivery, or 345 times through the block-compressed store. The coordinate also addresses a dataset rather than one file: one shard per run over 38 runs, a run resolved by arithmetic with no scan against the 11.1 s run-branch scan a TChain needs, at 18.1 ms per file to attach against the chain’s forced 9.3 ms, and the entry layer at 2.8 times the chain’s per-event read over a whole run. The boundaries stay with the results: for the collection store ROOT wins a narrow column selection below about 42 scalar columns for the read path, near 810 with delivery; thread scaling narrows the lead from 14.3 to 11.2 times over eight cores; the entry layer’s per-cell cost rises with the scale of the walk, 54.5 microseconds over a shard’s first 2,000 cells against 78.9 over 368,008; and multi-terabyte samples, the remote medium, and 128 cores are not measured.

The two charts below capture the result. The first shows the speedup across all measured conditions; the second shows the mechanism on the full dataset: the baseline scatters one event across many small requests, while the coordinate path serves every event with one aligned request and the mapped path adds zero application-level read system calls.

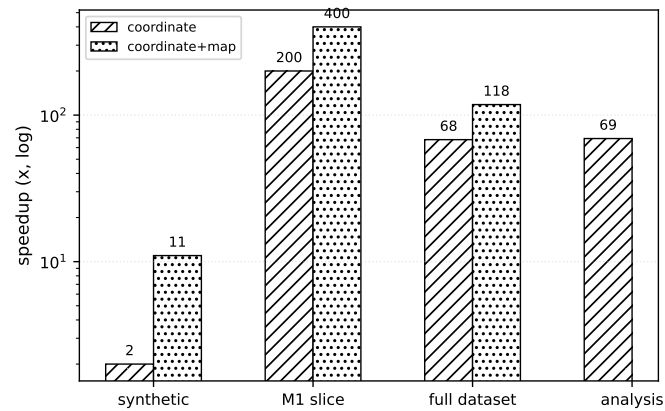


Figure 32: Speedup over the ROOT baseline across measured conditions, log scale. The mapped row is the coordinate+map path and the analysis row is the analysis workload. The synthetic and M1 slice bars are indicative microbenchmark-scale ratios; the full-dataset and analysis bars are the measured claims.

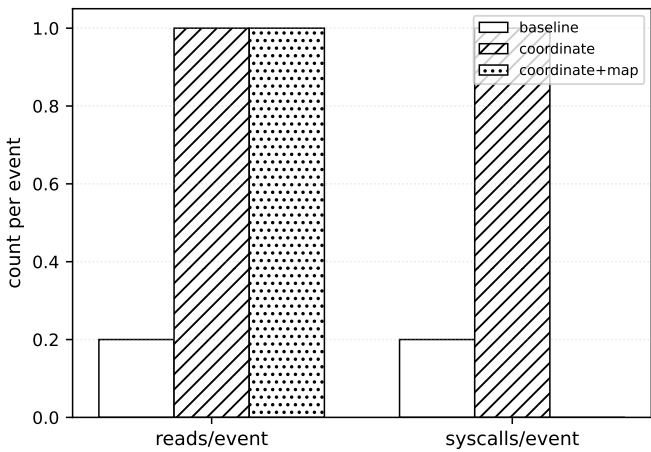


Figure 33: Requests and read system calls per event on the full dataset. The mapped path serves every event with one aligned request and zero application-level read system calls.

K Script Comparison

The full comparison across all examined writing systems. Main-stream computing alphabets (ASCII/Latin, Cyrillic, Greek, Hebrew) are atomic assignments with no combinatorial structure. Devanagari, Thai, and Tibetan combine consonants and vowels with irregular joining rules; not every combination maps to a predefined code point, so no closed-form three-axis decomposition exists. Braille is genuinely combinatorial, but as a 2^6 dot bit pattern rather than a three-axis algebra. Hangul Jamo composes characters as conjoining code-point sequences, so the composition exists but spans multiple code points instead of one closed-form block. Ethiopic and Cherokee are syllabaries: each character is an individually assigned code point, a one-dimensional enumeration. Hangul is the only script that combines a fully contiguous BMP valid range with a complete three-axis decomposition.

Script	Contiguous Address	Combinatorial Structure	3-Axis Decomposition	Compatible?
Hangul	Y U+AC00..U+D7A3	Y 19x21x28, no exceptions	Y Onset-Nucleus-Coda	Y

Script	Contiguous Address	Combinatorial Structure	3-Axis Decomposition	Compatible?
ASCII	Y U+0000-U+007F	N	N	N
Cyrillic	Y U+0400-U+04FF	N	N	N
Greek	Y U+0370-U+03FF	N	N	N
CJK Unified	Y U+4E00-U+9FFF	N (infinite, irregular)	N	N
Ideographs				
Japanese(Kana)	Y U+3040-U+30FF	N	N	N
Arabic	Y U+0600-U+06FF	N	N	N
Hebrew	Y U+0590-U+05FF	N	N	N
Devanagari	N	N (2D consonant+vowel)	N	N
Thai	N	N (2D consonant+vowel)	N	N
Tibetan	N	N (2D consonant+vowel)	N	N
Braille	Y U+2800-U+28FF	Y (2^6 dot bit pattern)	N (bit pattern, no axis algebra)	N
Hangul Jamo	Y U+1100-U+11FF	N (conjoining sequences)	N (multi-code-point composition)	N
Ethiopic	Y U+1200-U+137F	N (1D syllabary)	N	N
Cherokee	Y U+13A0-U+13FF	N (1D syllabary)	N	N

L CoordSpace20

This is a single unique atom's address in our observable universe:

맨가억빈힉씩룩직랏풍첿겨뇨도디뤼뫼빅식짜

Twenty Korean characters (U+AC00..U+D7A3) form a coordinate path. Each character encodes a value drawn from 11,172 possibilities through its initial, medial, and final decompositions, yielding $11,172^{20} \approx 9.2 \times 10^{80}$ possible addresses. This is enough to assign a unique address to each atom in a volume 9.2 times larger than the observable universe. Yet the entire address is a 20-character Korean string. It is not a hash of a larger datum, but the address itself rendered in human-readable Hangul, decodable by anyone who reads Korean without a hex dump.