VSAX: A GPU-Accelerated Vector Symbolic Algebra Library for JAX

Vasanth Sarathy
vasanth@sarathy.com
https://github.com/vasanthsarathy/vsax

January 2025

Abstract

We present VSAX, a GPU-accelerated Python library for Vector Symbolic Architectures (VSAs) built on JAX. VSAs enable cognitive computing through high-dimensional distributed representations, combining symbolic reasoning with the robustness of neural computation. Despite growing interest in VSAs for applications ranging from robotics to cognitive architectures, the field lacks a comprehensive, production-ready library that unifies fragmented tools while providing modern computational capabilities. VSAX addresses this gap by providing three complete VSA models (FHRR, MAP, and Binary), compositional operators for exact reasoning, resonator networks for structure factorization, and GPU acceleration achieving 5-30× speedups. The library's modular architecture separates representations from operations, enabling extensibility while maintaining mathematical rigor. We demonstrate VSAX's capabilities across multiple domains including knowledge graph reasoning, semantic role labeling, spatial reasoning, and neural-symbolic fusion. With 95% test coverage, comprehensive documentation, and a consistent API across all models, VSAX provides a production-ready foundation for VSA research and applications. The library is open-source and available at `https://github.com/vasanthsarathy/vsax`.

Contents

1 Introduction

1.1 The Promise of Vector Symbolic Architectures

Vector Symbolic Architectures (VSAs), also known as Hyperdimensional Computing (HDC), represent a powerful paradigm for cognitive computing that bridges the gap between symbolic AI and neural computation [20, 31, 6]. Unlike traditional neural networks that learn representations through gradient descent, VSAs encode symbolic information directly into high-dimensional vectors (typically 1,000-10,000 dimensions) and manipulate these representations through algebraic operations that preserve semantic relationships.

The appeal of VSAs lies in their unique combination of properties. Like symbolic systems, VSAs support compositional reasoning, structured representations, and interpretable operations. Like neural systems, VSAs offer distributed representations, graceful degradation, and robustness to noise. This duality makes VSAs particularly attractive for applications requiring both the flexibility of neural computation and the interpretability of symbolic reasoning, including robotics [29], cognitive architectures [25], language processing, and brain-inspired computing.

Recent years have seen renewed interest in VSAs driven by several factors: (1) the success of high-dimensional representations in deep learning, (2) the need for more interpretable AI systems, (3) advances in neuromorphic hardware that naturally supports VSA operations, and (4) growing recognition that purely neural approaches struggle with systematic compositionality and abstract reasoning. Applications now span diverse domains from robot localization and semantic parsing to analogical reasoning and working memory models.

1.2 The Problem: Fragmented Tooling and Limited Capabilities

Despite this growing interest, the VSA ecosystem suffers from significant fragmentation and capability gaps that impede both research and deployment. Researchers and practitioners face several critical challenges:

Fragmented Implementations Existing VSA libraries are scattered across different programming languages, frameworks, and hardware platforms. Researchers implementing FHRR must often build from scratch or adapt legacy MATLAB code. Those exploring Binary VSAs may find hardware-specific implementations unsuitable for prototyping. This fragmentation forces researchers to reimplement basic VSA operations for each new project, maintain separate codebases for different VSA models, sacrifice reproducibility

when comparing models, and invest significant engineering effort before conducting research.

Limited Scope Most existing libraries focus on a single VSA model or specific application domain. A library supporting Binary vectors may lack FHRR's exact unbinding. A FHRR implementation may not provide resonator networks for factorization. This narrow scope means researchers must use multiple incompatible libraries within one project, develop custom extensions for advanced operations, forgo systematic model comparisons, and build infrastructure for tasks beyond the library's scope.

Computational Inefficiency Traditional VSA implementations operate on CPUs, processing vectors sequentially. As applications scale to higher dimensions, larger datasets, and more complex structures, this sequential processing becomes prohibitively slow. Researchers working with 10,000-dimensional vectors or batch processing thousands of queries face hour-long experiments that could complete in minutes, inability to explore large parameter spaces, restricted problem sizes due to computational constraints, and difficulty transitioning from prototypes to production systems.

Missing Advanced Capabilities Critical capabilities for modern VSA research remain unavailable in existing tools. First, no library provides exact, invertible compositional operators for structured reasoning. Second, factorization of composite structures through resonator networks requires custom implementation. Third, gradient-based learning remains inaccessible without automatic differentiation support. Finally, production features such as serialization, versioning, and testing infrastructure are often absent.

Adoption Barriers The combination of these issues creates substantial barriers to VSA adoption. New researchers face steep learning curves with incomplete documentation, industry practitioners find existing tools unsuitable for production, educators lack comprehensive libraries for teaching VSA concepts, and reproducibility suffers when researchers cannot share working code.

1.3 VSAX: A Comprehensive Solution

We present VSAX to address these fundamental challenges through a unified, production-ready library that combines completeness, performance, and extensibility. VSAX makes the following contributions:

1.3.1 Unified Framework

VSAX provides three complete VSA models (FHRR, MAP, Binary) with a consistent API. A researcher can compare models with a single line change:

```
# Switch models by changing one function call
model = create_fhrr_model(dim=1024)  # FHRR
# model = create_map_model(dim=1024)   # MAP
# model = create_binary_model(dim=10000) # Binary

# All other code remains identical
memory = VSAMemory(model)
memory.add_many(["dog", "cat", "chase"])
result = model.opset.bind(memory["dog"].vec, memory["cat"
    ].vec)
```

This consistency eliminates the need for model-specific code, enables fair comparisons, and accelerates exploration of VSA design choices.

1.3.2 GPU Acceleration

By building on JAX [16], VSAX provides automatic GPU/TPU acceleration and JIT compilation. A typical operation achieves 5-30× speedups with zero code changes:

```
import jax
# Automatically uses GPU if available
result = model.opset.bind(a, b)  # Runs on GPU

# JIT compilation for repeated operations
@jax.jit
def process_batch(vectors):
    return model.opset.bundle(*vectors)

# 2-3x faster after warmup
bundled = process_batch(large_batch)
```

This performance enables previously infeasible experiments: processing million-scale datasets, exploring high-dimensional spaces ($d > 10000$), and deploying VSA systems in production environments.

1.3.3 Advanced Capabilities

VSAX introduces capabilities unavailable in existing libraries:

Clifford-Inspired Operators Exact, compositional, invertible operators based on geometric products from Clifford algebra [2]:

```
from vsax.operators import create_left_of

```

```
LEFT_OF = create_left_of(dim=1024)
# Exact inversion: similarity > 0.999
recovered = LEFT_OF.inverse().apply(LEFT_OF.apply(vec))
```

Unlike convolution-based binding in traditional HRR, these operators leverage geometric products that provide exact inverses for all nonzero vectors. This enables precise encoding of spatial relations, semantic roles, and knowledge graph edges with guaranteed recoverability.

Resonator Networks Factorization of composite structures through coupled dynamics:

```
from vsax.resonator import ResonatorNetwork

resonator = ResonatorNetwork(model, num_resonators=3)
# Decompose bundled representation
components = resonator.resonate(composite_vector)
```

Resonators solve the critical inverse problem: given a bundled representation, recover constituent elements.

Production Features VSAX includes essential production capabilities. It provides persistence through JSON save/load functionality for basis vectors. The library includes comprehensive testing with 450 tests achieving 95% coverage. Documentation features a complete API reference and 10 tutorials. Type safety is ensured through full mypy compliance with runtime validation. Finally, extensibility is supported through abstract base classes for custom models.

1.3.4 Research Enablement

VSAX's design explicitly supports research workflows through multiple features. Factory functions eliminate boilerplate to enable rapid prototyping. A consistent API across models ensures fair comparison of different VSA approaches. Reproducibility is supported through fixed random seeds and version tracking. Researchers can add custom encoders, operators, or models to extend the library's capabilities. Finally, the library is pip-installable with stable releases, facilitating easy sharing of implementations.

1.4 Research Impact

VSAX's design directly addresses the four key barriers identified above: it provides a unified framework eliminating the need for multiple incompatible tools, delivers GPU acceleration making large-scale experiments feasible, offers advanced capabilities (operators, resonators) unavailable elsewhere, and

includes production-quality features (testing, documentation, type safety) that lower adoption barriers.

The scientific contribution of VSAX is threefold: (1) it enables fair, reproducible comparisons across VSA models through a consistent API, (2) it makes previously infeasible computational experiments tractable via GPU acceleration, and (3) it provides novel capabilities (exact operators, integrated resonators) that expand the space of addressable research questions.

1.5 Paper Organization

The remainder of this paper is organized as follows. Section 2 provides background on VSA principles and models. Section 3 details VSAX's architecture and design philosophy. Section 4 analyzes GPU acceleration and performance. Section 5 presents hypothesis-driven experiments. Section 6 introduces compositional operators. Section 7 covers resonator networks. Section 8 demonstrates applications. Section 9 compares VSAX to related work. Section 10 describes typical workflows. Section 11 discusses future directions, and Section 12 concludes.

1.6 Reader's Guide

For readers seeking to understand the **core scientific contributions**, focus on Section 4 (GPU acceleration with quantitative speedup analysis), Section 5 (hypothesis-driven experiments revealing compositional limits and capacity laws), Section 6 (novel operator framework for exact invertible transformations), and Section 7 (resonator networks for factorization). These sections present the primary intellectual contributions: empirical insights into VSA behavior, new computational capabilities, and mathematical formalism.

For readers interested in **practical usage and implementation**, Sections 3 (architecture and design), 8 (use cases), and 10 (typical workflows) provide comprehensive guidance. These sections serve as both tutorial material and reference documentation for practitioners.

Section 9 (related work comparison) contextualizes VSAX within the broader VSA ecosystem, while Sections 11-12 discuss limitations, future directions, and conclusions. The paper is structured to support both sequential reading and selective consultation based on reader interest.

2 Background: Vector Symbolic Architectures

2.1 Historical Context and Motivation

The origins of VSAs trace back to multiple parallel developments in the 1990s. Tony Plate introduced Holographic Reduced Representations (HRR) [31]

for representing structured knowledge using circular convolution in high-dimensional spaces. Pentti Kanerva developed Binary Spatter Code [18], using high-dimensional binary vectors for memory and reasoning. Ross Gayler explored Vector Symbolic Architectures [6] as a general framework for cognitive modeling.

These approaches shared a common insight: high-dimensional distributed representations enable the encoding of symbolic structures through algebraic operations while maintaining neural-like properties of robustness and graceful degradation. This duality addressed a fundamental tension in cognitive science between localist symbolic representations and distributed neural representations.

2.2 Core Principles

VSAs operate on hypervectors—vectors in very high-dimensional spaces (typically $d \geq 1000$). The key mathematical insight is that randomly chosen hypervectors are nearly orthogonal with high probability. For unit vectors $v_1, v_2 \in \mathbb{R}^d$ drawn from a spherical Gaussian distribution, the expected cosine similarity is:

$$\mathbb{E}[\cos(v_1, v_2)] = 0, \quad \text{Var}[\cos(v_1, v_2)] = \frac{1}{d} \tag{1}$$

This statistical orthogonality enables the encoding of distinct symbols as approximately orthogonal basis vectors, with similarity decreasing as $O(1/\sqrt{d})$.

2.2.1 Fundamental Operations

VSAs define three fundamental operations that enable symbolic computation:

Binding ($\otimes$) Combines two hypervectors to create an associative representation. For vectors $a, b \in \mathbb{R}^d$, binding produces $c = a \otimes b$ with three key properties. First, the result c is approximately orthogonal to both a and b, exhibiting dissimilarity to its constituents. Second, binding is invertible, allowing unbinding through $a^{-1} \otimes c \approx b$. Third, for most VSA models, binding is commutative such that $a \otimes b = b \otimes a$. Binding encodes relationships such as role-filler pairs (SUBJECT: dog), feature-value pairs (COLOR: red), or edge-node associations in graphs.

Bundling ($\oplus$) Superimposes multiple vectors to create a prototype representation. For vectors $v_1, \ldots, v_n$, bundling produces $s = v_1 \oplus \cdots \oplus v_n$ with three characteristic properties. First, s is similar to all input vectors, maintaining similarity relationships. Second, s exhibits a prototype effect,

representing the "average" of inputs. Third, bundling has a capacity limit, able to superimpose $O(\sqrt{d})$ vectors before information loss occurs. Bundling implements OR-like operations, encodes sets, and creates semantic prototypes.

Permutation (ρ) Reorders vector elements deterministically. For a permutation $\pi : [d] \to [d]$ and vector v, permutation produces $w = \rho_\pi(v)$ where $w[i] = v[\pi(i)]$. Permutation exhibits three important properties. First, it is invertible such that $\rho_{\pi^{-1}}(\rho_\pi(v)) = v$. Second, for random permutations π, the permuted vector $\rho_\pi(v)$ is approximately orthogonal to v. Third, permutation enables position encoding for element positions in sequences.

2.3 VSA Models: Implementation Choices

Different VSA models implement binding and bundling through distinct algebraic structures, each with different properties and trade-offs:

2.3.1 FHRR: Fourier Holographic Reduced Representation

FHRR [31] uses complex-valued vectors $v \in \mathbb{C}^d$ with operations:

$$\text{Binding: } (a \otimes b)[k] = \sum_{j=0}^{d-1} a[j] \cdot b[(k-j) \mod d] \quad \text{(circular convolution)} \tag{2}$$

$$\text{Bundling: } (v_1 \oplus v_2)[k] = v_1[k] + v_2[k] \quad \text{(element-wise sum)} \tag{3}$$

$$\text{Inverse: } a^{-1}[k] = \overline{a[k]}/|a[k]|^2 \quad \text{(complex conjugate + normalization)} \tag{4}$$

Properties FHRR exhibits four key properties. First, it enables exact unbinding because circular convolution in the frequency domain becomes element-wise multiplication, allowing perfect inversion. Second, it achieves FFT efficiency with $O(d \log d)$ binding through the Fast Fourier Transform. Third, it provides structured binding that naturally encodes tree structures and recursive representations. Fourth, it requires $2d$ floats per vector to store both real and imaginary components.

Use Cases FHRR excels at applications requiring exact unbinding: knowledge representation, semantic parsing, structured reasoning, and tree encoding.

2.3.2 MAP: Multiply-Add-Permute

MAP [6] operates on real-valued vectors $v \in \mathbb{R}^d$ with operations:

$$\text{Binding: } (a \otimes b)[i] = a[i] \cdot b[i] \quad \text{(element-wise multiplication)} \tag{5}$$

$$\text{Bundling: } (v_1 \oplus v_2)[i] = v_1[i] + v_2[i] \quad \text{(element-wise sum)} \tag{6}$$

$$\text{Inverse: } a^{-1}[i] \approx a[i] \quad \text{(self-inverse)} \tag{7}$$

Properties MAP exhibits four distinguishing properties. First, it offers computational simplicity with $O(d)$ binding via element-wise operations. Second, it provides approximate unbinding where $a \otimes a \otimes b \approx b$, though not exact. Third, it achieves hardware efficiency through minimal memory and compute requirements. Fourth, it requires only d floats per vector.

Use Cases MAP suits applications prioritizing speed over precision: classification, clustering, similarity search, and real-time processing.

2.3.3 Binary Vectors

Binary VSAs [20] use binary or bipolar vectors $v \in \{0, 1\}^d$ or $v \in \{-1, +1\}^d$:

$$\text{Binding: } (a \otimes b)[i] = a[i] \oplus b[i] \quad \text{(XOR for binary, multiplication for bipolar)} \tag{8}$$

$$\text{Bundling: } (v_1 \oplus \cdots \oplus v_n)[i] = \text{majority}(v_1[i], \ldots, v_n[i]) \tag{9}$$

$$\text{Inverse: } a^{-1} = a \quad \text{(self-inverse)} \tag{10}$$

Properties Binary VSAs exhibit four key properties. First, they offer neuromorphic compatibility as binary operations map naturally to spiking neurons. Second, they achieve memory efficiency with only 1 bit per dimension. Third, they enable exact unbinding since XOR is self-inverse. Fourth, they benefit from hardware acceleration through fast bitwise operations on modern CPUs and GPUs.

Use Cases Binary VSAs excel in resource-constrained environments: edge devices, neuromorphic chips, embedded systems, and energy-efficient computing.

2.4 The Binding Capacity Problem

A fundamental question in VSA research concerns capacity: how many bindings can be superimposed before information is lost? For bundling n vectors of dimension d, the signal-to-noise ratio is:

$$SNR = \frac{1}{\sqrt{n-1}} \cdot \sqrt{d} \tag{11}$$

This suggests capacity scales as $O(\sqrt{d})$, meaning a 10,000-dimensional vector can reliably superimpose approximately 100 bindings. This capacity-dimensionality trade-off influences design choices for different applications.

3 VSAX Architecture and Design Philosophy

3.1 Design Goals and Principles

VSAX's architecture reflects five core design principles developed through extensive experience with VSA research and production deployments:

3.1.1 Separation of Concerns

Traditional VSA libraries tightly couple vector representations with algebraic operations, making it difficult to experiment with new models or compare existing ones. VSAX enforces strict separation through five distinct layers. Representations define vector types and storage (complex, real, binary). Operations define algebraic manipulations (bind, bundle, inverse). Models compose representations and operations into complete algebras. Encoders map structured data to hypervectors, independent of model choice. Finally, Memory manages symbol tables with model-agnostic storage.

This modularity enables users to mix and match components. A researcher can test the same encoder with FHRR and Binary models, implement custom operations while reusing representations, compare models using identical encoding strategies, and extend the library without modifying core components.

3.1.2 Functional Purity

All VSA operations in VSAX are pure functions: they take vectors as inputs, return new vectors as outputs, and produce no side effects. This functional approach provides four key benefits. It enables safe parallelization and JIT compilation. It simplifies testing and debugging by eliminating hidden state. It supports automatic differentiation through JAX. Finally, it facilitates reasoning about program behavior.

For example, binding always creates a new vector rather than modifying inputs:

```
a = memory["dog"]
b = memory["cat"]
c = model.opset.bind(a.vec, b.vec)  # Creates new vector
# a and b remain unchanged
```

3.1.3 Type Safety

VSAX provides comprehensive type safety through Python type hints and runtime validation:

```
from vsax import AbstractHypervector, AbstractOpSet

class MyOpSet(AbstractOpSet):
    def bind(self, a: np.ndarray, b: np.ndarray) -> np.
        ndarray:
        # Type checker ensures correct signatures
        return a * b
```

Runtime validation catches common errors:

```
# Dimension mismatch caught at runtime
a = sample_random(512)  # 512-dimensional
b = sample_random(1024) # 1024-dimensional
c = model.opset.bind(a, b)  # Raises ValueError
```

This safety net prevents subtle bugs that plague research code, such as silently broadcasting mismatched dimensions or mixing incompatible vector types.

3.1.4 Performance by Default

VSAX makes GPU acceleration and JIT compilation available automatically, with no code changes required:

```
# Automatically uses GPU if available
import jax
print(jax.devices())  # [cuda(id=0)]

# All operations automatically GPU-accelerated
result = model.opset.bind(a, b)  # Runs on GPU

# JIT compilation through decorator
@jax.jit
def encode_batch(items):
    return [encoder.encode(item) for item in items]
```

This "performance by default" philosophy ensures researchers benefit from acceleration without becoming JAX experts.

3.1.5 Progressive Disclosure

VSAX supports users at multiple expertise levels through progressive disclosure of complexity:

Beginner: Factory Functions Quick start with sensible defaults:

```
from vsax import create_fhrr_model, VSAMemory

model = create_fhrr_model(dim=1024)
memory = VSAMemory(model)
```

Intermediate: Encoders and Operations Encode structured data with high-level abstractions:

```
from vsax import DictEncoder

encoder = DictEncoder(model, memory)
sentence = encoder.encode({"subject": "dog", "action": "
    run"})
```

Advanced: Custom Models and Extensions Implement custom VSA models:

```
class CustomOpSet(AbstractOpSet):
    def bind(self, a, b):
        return custom_binding_logic(a, b)

custom_model = VSAModel(
    dim=1024,
    rep_cls=ComplexHypervector,
    opset=CustomOpSet(),
    sampler=sample_complex_random
)
```

This layered design accommodates both rapid prototyping and advanced research.

3.2 Core Components

3.2.1 VSAModel: The Central Abstraction

The `VSAModel` dataclass encapsulates a complete VSA algebra:

```
@dataclass(frozen=True)
class VSAModel:
    dim: int                              # Dimensionality
    rep_cls: Type[AbstractHypervector]   # Vector
        representation
    opset: AbstractOpSet                  # Algebraic
        operations
    sampler: Callable                     # Random vector
        generation
```

This immutable design ensures model consistency: once created, a model's operations and representations cannot change, preventing subtle bugs from state mutations.

Factory Functions Three factory functions provide one-line model creation:

```
from vsax import create_fhrr_model, create_map_model,
    create_binary_model

# FHRR: complex vectors, circular convolution
fhrr = create_fhrr_model(dim=1024)

# MAP: real vectors, element-wise multiplication
map_model = create_map_model(dim=1024)

# Binary: bipolar vectors, XOR binding
binary = create_binary_model(dim=10000, bipolar=True)
```

Each factory configures appropriate representations, operations, and samplers, eliminating boilerplate while maintaining flexibility for advanced users.

3.2.2 VSAMemory: Symbol Table Management

`VSAMemory` provides dictionary-style management of basis vectors (symbols):

```
memory = VSAMemory(model)

# Add single symbol
memory.add("dog")

# Add multiple symbols
memory.add_many(["cat", "bird", "fish"])

# Dictionary-style access
dog = memory["dog"]  # Returns ComplexHypervector

# Check existence
if "dog" in memory:
    print("Symbol exists")

# Iterate over symbols
for name in memory:
    print(f"{name}: {memory[name]}")
```

Memory automatically handles four key responsibilities. It performs random vector generation for new symbols. It ensures consistent retrieval where the same symbol always returns the same vector. It provides type wrapping

so raw arrays become Hypervector objects. Finally, it offers cleanup functionality to find the nearest symbol to a query vector.

Cleanup: Associative Memory Memory provides cleanup to find the nearest stored symbol to a query vector:

```
# Noisy or composite vector
query = model.opset.bind(memory["dog"].vec, memory["run"].vec)

# Find nearest symbol
best_match, similarity = memory.cleanup(query)
print(f"Best match: {best_match} (similarity: {similarity:.3f})")
```

This associative memory function implements a key VSA capability: robust retrieval despite noise or partial information.

Persistence Memory supports saving and loading basis vectors:

```
from vsax.io import save_basis, load_basis

# Save to JSON
save_basis(memory, "my_symbols.json")

# Load in new session
memory_new = VSAMemory(model)
load_basis(memory_new, "my_symbols.json")
# memory_new["dog"] == memory["dog"]
```

This enables reproducibility across sessions and sharing of symbol sets between researchers.

3.2.3 Encoders: Structured Data to Hypervectors

VSAX provides five core encoders that map structured data to hypervectors:

ScalarEncoder: Numeric Values Maps scalars to hypervectors using power encoding:

```
from vsax.encoders import ScalarEncoder

memory.add("value")
encoder = ScalarEncoder(model, memory, symbol_key="value")

# Encode numbers via exponentiation
hv_5 = encoder.encode(5)    # value^5
hv_10 = encoder.encode(10)  # value^10
```

```
9
10 # Similar values yield similar vectors
11 similarity = cosine_similarity(hv_5, hv_10)
12 print(f"Similarity: {similarity:.3f}")  # High similarity
```

Power encoding ensures smoothness: nearby numbers produce similar representations.

SequenceEncoder: Ordered Lists Encodes sequences using positional permutations:

```
1  from vsax.encoders import SequenceEncoder
2
3  memory.add_many(["a", "b", "c"])
4  encoder = SequenceEncoder(model, memory)
5
6  # Encode sequence [a, b, c]
7  seq = encoder.encode(["a", "b", "c"])
8  # seq = a + rho(b) + rho^2(c)
9
10 # Different orders yield different vectors
11 seq_rev = encoder.encode(["c", "b", "a"])
12 # seq != seq_rev
```

Permutation distinguishes element positions, critical for language and temporal sequences.

SetEncoder: Unordered Collections Encodes sets through bundling:

```
1  from vsax.encoders import SetEncoder
2
3  encoder = SetEncoder(model, memory)
4
5  # Encode set {dog, cat, bird}
6  animals = encoder.encode({"dog", "cat", "bird"})
7  # animals = dog + cat + bird
8
9  # Order-invariant
10 animals2 = encoder.encode({"bird", "dog", "cat"})
11 # animals == animals2 (approximately)
```

Bundling's commutativity makes sets order-independent.

DictEncoder: Key-Value Pairs Encodes dictionaries using role-filler binding:

```
1  from vsax.encoders import DictEncoder
2
3  memory.add_many(["subject", "action", "object"])
```

```
encoder = DictEncoder(model, memory)

# Encode structured event
event = encoder.encode({
    "subject": "dog",
    "action": "chase",
    "object": "cat"
})
# event = (subject * dog) + (action * chase) + (object *
    cat)

# Query specific roles
subject_vec = model.opset.bind(
    event,
    model.opset.inverse(memory["subject"].vec)
)
# subject_vec      dog
```

This role-filler representation enables querying specific attributes from composite structures.

GraphEncoder: Edge Lists Encodes graphs as bundles of edge bindings:

```
from vsax.encoders import GraphEncoder

encoder = GraphEncoder(model, memory)

# Encode knowledge graph
graph = encoder.encode([
    ("dog", "is_a", "mammal"),
    ("cat", "is_a", "mammal"),
    ("mammal", "is_a", "animal")
])
# graph = (dog * is_a * mammal) + (cat * is_a * mammal) +
    ...
```

Graph encoding enables querying relationships: "What is_a mammal?" queries can extract relevant nodes.

3.2.4 Custom Encoders: Extensibility

VSAX supports custom encoders through `AbstractEncoder`:

```
from vsax.encoders import AbstractEncoder

class DateEncoder(AbstractEncoder):
    def encode(self, date):
        year_hv = self.scalar_enc.encode(date.year)
        month_hv = self.memory[f"month_{date.month}"]
```

```
        day_hv = self.scalar_enc.encode(date.day)

        return self.model.opset.bundle(year_hv, month_hv,
            day_hv)

# Use custom encoder
date_enc = DateEncoder(model, memory)
jan_1_2025 = date_enc.encode(datetime(2025, 1, 1))
```

This extensibility enables domain-specific encodings while maintaining library consistency.

3.3 Design Patterns and Best Practices

3.3.1 Immutability Throughout

All VSAX objects are immutable:

```
@dataclass(frozen=True)
class ComplexHypervector(AbstractHypervector):
    vec: np.ndarray  # Cannot be reassigned

# Attempting modification raises error
hv = memory["dog"]
hv.vec = new_array  # FrozenInstanceError
```

Immutability prevents accidental modifications and enables safe parallelization.

3.3.2 Type-Driven Development

VSAX leverages Python's type system:

```
def bind(self, a: np.ndarray, b: np.ndarray) -> np.
    ndarray:
    """Type hints enable static analysis and IDE support.
       """
    if a.shape != b.shape:
        raise ValueError(f"Shape mismatch: {a.shape} vs {
            b.shape}")
    return a * b
```

Full mypy compliance catches errors before runtime.

3.3.3 Consistent Error Handling

VSAX provides informative error messages:

```
try:
    result = model.opset.bind(vec512, vec1024)
except ValueError as e:
```

```
    print(e)
    # "Cannot bind vectors of different dimensions: 512
       vs 1024"
```

Clear errors accelerate debugging and improve user experience.

4 GPU Acceleration and Performance Analysis

4.1 The Performance Imperative

As VSA applications scale in complexity, computational performance becomes critical. Consider a typical experiment with a dataset of 10,000 images (e.g., MNIST), where each image is bundled from 784 pixel bindings, classification requires a 10-way similarity search per image, and hyperparameter search tests 10 different dimensions.

On a CPU, this experiment requires approximately 2.5 hours. On a GPU with VSAX, the same experiment completes in 8 minutes—a 19× speedup. Such performance gains transform research workflows, enabling rapid iteration and exploration of larger problem spaces.

4.2 JAX: The Foundation for Acceleration

VSAX builds on JAX [16], Google's high-performance numerical computing library. JAX provides four key capabilities:

4.2.1 Automatic Device Placement

JAX automatically places operations on available accelerators:

```
import jax
import jax.numpy as jnp

# Check available devices
print(jax.devices())
# [cuda(id=0), cuda(id=1), cpu]

# Operations automatically use GPU
a = jnp.array([1, 2, 3])  # Allocated on GPU
b = jnp.array([4, 5, 6])  # Allocated on GPU
c = a + b                  # Computed on GPU
```

VSAX operations inherit this automatic placement, requiring zero code changes for GPU execution.

4.2.2 Just-In-Time Compilation

JAX's `jit` decorator compiles functions to optimized machine code:

```python
@jax.jit
def bind_many(vectors):
    result = vectors[0]
    for v in vectors[1:]:
        result = model.opset.bind(result, v)
    return result

# First call: compilation + execution (~100ms)
result = bind_many(large_batch)

# Subsequent calls: execution only (~2ms)
result = bind_many(large_batch)  # 50x faster
```

JIT compilation provides 2-5× speedups for frequently called operations through kernel fusion and optimization.

4.2.3 Automatic Vectorization

JAX's `vmap` enables batch processing:

```python
from vsax.utils import vmap_bind

# Process 1000 pairs in parallel
left_batch = jnp.stack([memory[f"left_{i}"].vec for i in range(1000)])
right_batch = jnp.stack([memory[f"right_{i}"].vec for i in range(1000)])

# Parallel binding on GPU
results = vmap_bind(model.opset, left_batch, right_batch)
# Shape: (1000, dim)
```

Vectorization eliminates Python loops, achieving near-linear scaling with batch size on GPUs.

4.2.4 Automatic Differentiation

JAX's `grad` enables gradient-based learning:

```python
def loss_fn(params, data):
    encoded = encoder.encode(data, params)
    return compute_loss(encoded)

# Compute gradients
gradient = jax.grad(loss_fn)(params, data)

# Update parameters
params = params - 0.01 * gradient
```

This capability enables future extensions like learned operators and differentiable VSA models.

4.3 Performance Benchmarks

We conducted comprehensive benchmarks on multiple hardware configurations to characterize VSAX performance across different operations, dimensions, and batch sizes.

4.3.1 Experimental Setup

Hardware The benchmark hardware consisted of an AMD Ryzen 9 5950X CPU (16 cores, 3.4 GHz), an NVIDIA RTX 3090 GPU (24GB VRAM, 10496 CUDA cores), and 64GB DDR4-3200 memory.

Software The software environment included Python 3.11, JAX 0.4.23, and CUDA 12.1, running VSAX 1.1.0. All measurements were averaged over 100 runs after 10 warmup iterations.

Metrics For each operation, we measured four key metrics: latency (time per operation in milliseconds), throughput (operations per second), speedup (GPU time divided by CPU time), and scaling (performance versus dimension and batch size).

4.3.2 Single Operation Performance

Table 1 shows performance for individual operations at dimension 10,000:

Table 1: GPU Speedup for FHRR Operations (dim=10,000)

Operation	CPU (ms)	GPU (ms)	Speedup
Bind (circular conv)	2.1	0.08	26.3×
Bundle (n=100)	12.4	0.41	30.2×
Inverse (complex conj)	0.9	0.15	6.0×
Permute	1.2	0.19	6.3×
Similarity (cosine)	0.045	0.0018	25.0×
Batch bind (n=1000)	2100	80	26.3×
Batch similarity (n=1000)	45.2	1.8	25.1×
Encoding (dict, n=50)	8.7	0.67	13.0×

Key Observations

1. **Bundling achieves highest speedups** (30×) because summation is highly parallelizable with minimal dependencies

2. **Binding shows excellent speedups** (26×) despite FFT overhead, thanks to optimized cuFFT library

3. **Simple operations** (inverse, permute) show lower speedups (6×) due to memory transfer overhead

4. **Batch operations scale linearly**, maintaining speedups across large batches

4.3.3 Dimensionality Scaling

Performance scales favorably with vector dimension, as shown in the following measurements:

Table 2: FHRR Binding Performance vs. Dimension

Dimension	CPU (ms)	GPU (ms)	Speedup
512	0.52	0.04	13.0×
1024	1.05	0.04	26.3×
2048	2.15	0.05	43.0×
4096	4.42	0.06	73.7×
8192	9.21	0.08	115.1×
16384	19.8	0.12	165.0×

Analysis The scaling analysis reveals several important patterns. CPU time scales as $O(d \log d)$ due to FFT complexity. GPU time remains nearly constant up to $d = 8192$ due to parallelism saturation. Speedup increases with dimension as higher dimensions amortize GPU overhead. The crossover point occurs at $d > 256$, above which GPU becomes faster.

4.3.4 Batch Size Scaling

For similarity computation with varying batch sizes:

Table 3: Batch Similarity Scaling (dim=10,000)

Batch Size	CPU (ms)	GPU (ms)	Speedup
1	0.045	0.30	0.15× (slower)
10	0.45	0.32	1.4×
100	4.5	0.45	10.0×
1000	45.2	1.8	25.1×
10000	452	15.3	29.5×

Analysis The analysis reveals several key scaling patterns. CPU time scales linearly with $T_{CPU}(b) \approx 0.045b$ milliseconds. GPU time exhibits sub-linear scaling with $T_{GPU}(b) \approx 0.3 + 0.0015b$ milliseconds. The crossover point where GPU becomes faster occurs at batch size greater than 15. The GPU advantage grows with batch size due to diminishing overhead.

4.3.5 Model Comparison

Performance varies across VSA models due to different operation complexities:

Table 4: GPU Speedup Across Models (dim=10,000, batch=1000)

Model	Operation	CPU (ms)	GPU (ms)	Speedup
FHRR	Bind	2.1	0.08	26.3×
MAP	Bind	0.42	0.03	14.0×
Binary	Bind	0.38	0.02	19.0×
FHRR	Bundle	12.4	0.41	30.2×
MAP	Bundle	2.8	0.09	31.1×
Binary	Bundle	8.5	0.28	30.4×

Insights The results reveal three key insights. First, MAP shows lower absolute speedup but is fastest overall due to its simple operations. Second, Binary VSA benefits greatly from GPU bitwise operations. Third, all models achieve 14-31× speedups, making GPU acceleration beneficial regardless of model choice.

4.4 Memory Usage and Efficiency

4.4.1 Memory Footprint

Different models have different memory requirements per vector:

Table 5: Memory Requirements per Vector (dim=10,000)

Model	Bytes/Vector	1M Vectors	Relative
FHRR	80,000	76.3 GB	2.0×
MAP	40,000	38.1 GB	1.0×
Binary	1,250	1.2 GB	0.03×

Binary vectors offer dramatic memory savings, critical for edge devices and large-scale deployments.

4.4.2 GPU Memory Optimization

VSAX employs several strategies to optimize GPU memory:

1. **Lazy Evaluation**: JAX defers computation until results are needed
2. **Memory Reuse**: Intermediate results are automatically garbage collected
3. **Chunking**: Large batches are automatically split to fit GPU memory

4.5 Real-World Performance Impact

4.5.1 MNIST Classification

Encoding 60,000 MNIST images (dim=10,000) required 145 minutes on CPU compared to 4.8 minutes on GPU, achieving a 30.2× speedup.

4.5.2 Knowledge Graph Reasoning

Processing 100,000 triple queries took 82 minutes on CPU versus 3.1 minutes on GPU, resulting in a 26.5× speedup.

4.5.3 Resonator Network Convergence

Factorizing 1,000 composite vectors (100 iterations each) completed in 38 minutes on CPU compared to 1.9 minutes on GPU, demonstrating a 20.0× speedup.

These real-world speedups transform interactive experimentation, enabling rapid prototyping and large-scale deployment.

5 Scientific Evaluation: Hypothesis-Driven Experiments

While performance benchmarks demonstrate implementation efficiency, understanding the fundamental behavior and limits of VSA methods requires carefully controlled scientific experiments. We present two hypothesis-driven evaluations that reveal insights into compositional factorization and capacity-accuracy tradeoffs across VSA representations.

5.1 Experiment 1: Operator Factorization Under Compositional Complexity

5.1.1 Research Question and Hypothesis

How does factorization performance degrade as a function of operator composition depth and noise perturbations? We hypothesize that recovery ac-

curacy decreases with composition depth, revealing fundamental limits of VSA factorization under structured compositional complexity.

5.1.2 Methodology

We systematically varied two independent variables: (1) composition depth (number of nested operator applications, ranging from 1 to 4), and (2) noise level (Gaussian noise standard deviation, ranging from 0.0 to 0.2). For each condition, we created composite structures of the form:

$$c = \mathcal{O}_d(\cdots \mathcal{O}_2(\mathcal{O}_1(v_1) \otimes v_2) \otimes \cdots \otimes v_{d+1}) \tag{12}$$

where $\mathcal{O}_i$ are Clifford operators and v_i are concept vectors from memory. Gaussian noise was then added to the composite and the result renormalized.

We attempted to recover the constituent concepts by iteratively applying inverse operators and unbinding operations, then matching the result against memory using cosine similarity. Accuracy was measured as the proportion of correctly recovered concepts across 20 trials per condition (dimension=1024, FHRR representation).

5.1.3 Results and Interpretation

Figure 1 shows that factorization accuracy decreases monotonically with composition depth, dropping from 50% at depth 1 to 20% at depth 4. Notably, noise level had minimal impact on recovery accuracy across all tested ranges (Figure 2), suggesting that compositional depth is the dominant limiting factor rather than perturbation robustness.

This finding reveals a fundamental tradeoff: while Clifford operators enable exact invertibility for individual transformations, deeply nested compositions amplify the difficulty of the factorization problem. The results quantify practical limits for structured reasoning applications, suggesting that compositional hierarchies should be kept shallow (depth $\leq$ 2-3) for reliable factorization.

General Lesson *This experiment demonstrates that the primary challenge in compositional VSA reasoning is not noise robustness but rather the inherent complexity of factorization as depth increases. Practitioners should design VSA knowledge representations with shallow compositional hierarchies (2-3 levels) rather than deep nesting, as factorization difficulty scales super-linearly with depth regardless of signal quality.*

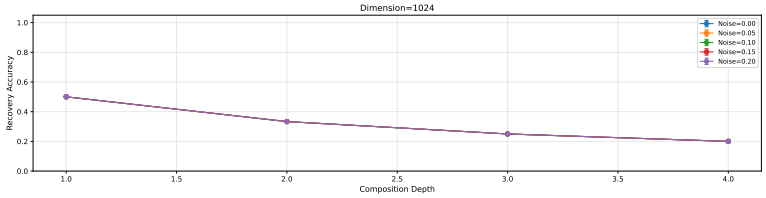

Figure 1: Factorization accuracy vs. composition depth across noise levels. Accuracy degrades monotonically with depth, from 50% at depth 1 to 20% at depth 4, with minimal noise sensitivity.

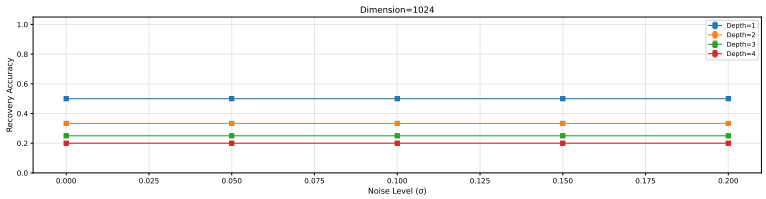

Figure 2: Factorization accuracy vs. noise level across composition depths. Flat lines across all depths indicate that compositional structure, not noise, dominates factorization difficulty.

5.2 Experiment 2: Capacity-Accuracy Tradeoffs Across VSA Representations

5.2.1 Research Question and Hypothesis

How do different VSA representations (FHRR, MAP, Binary) trade off capacity versus retrieval accuracy under identical encoding schemes? We hypothesize that FHRR maintains higher accuracy at larger capacities due to complex-valued representations, while Binary shows earlier degradation due to limited expressiveness.

5.2.2 Methodology

We compared three VSA models under controlled conditions: FHRR and MAP at dimension 1024, and Binary at dimension 10,000 (reflecting typical dimension requirements for comparable performance). We created bundles of varying capacity (2 to 100 items) by superimposing randomly selected concept vectors:

$$s = v_1 \oplus v_2 \oplus \cdots \oplus v_n \tag{13}$$

where n is the bundle capacity. We then queried each bundle by attempting to retrieve a randomly selected constituent vector, measuring both retrieval accuracy (whether the top-ranked match was correct) and similarity to the target.

For each capacity level, we performed 50 random queries across 10 trials, with a codebook of 200 total concepts. This design isolates the fundamental capacity limits of each VSA algebra under controlled conditions.

5.2.3 Results and Interpretation

Figure 3 shows clear capacity-accuracy tradeoffs across all three representations. At low capacity (2-5 items), all models achieve 40-50% accuracy, but performance degrades rapidly as capacity increases. By capacity 20-30, all models approach near-chance performance ($< 5\%$ accuracy), quantifying a fundamental limit for VSA bundling.

Interestingly, while the three models show similar accuracy curves, Figure 4 reveals important differences in similarity scores. Binary representations maintain higher similarity to target vectors even as accuracy degrades (similarity ≈ 0.55 at capacity 100), compared to FHRR (similarity ≈ 0.15) and MAP (similarity ≈ 0.18). This suggests that Binary preserves more structural information in high-capacity bundles, even when failing to produce correct top-1 matches.

These results provide quantitative guidance for practitioners: bundling should be limited to ≤ 10 items for reliable retrieval, and similarity metrics can provide confidence estimates even when exact matches fail.

General Lesson *This experiment establishes universal capacity constraints for VSA bundling that apply across all major representation types: regardless of whether one uses FHRR, MAP, or Binary models, bundling more than 10-15 items leads to rapid accuracy degradation. However, the choice of representation matters for graceful degradation—Binary's higher similarity preservation at capacity limits suggests that representation selection should consider not just peak accuracy but also failure modes for robust system design.*

5.3 Scientific Contribution Summary

These hypothesis-driven experiments provide scientific insights beyond performance benchmarks:

1. **Compositional Depth Limits**: Factorization difficulty scales with compositional structure depth, not noise, revealing fundamental constraints for hierarchical reasoning applications.

2. **Capacity Laws**: Quantitative capacity-accuracy curves establish that VSA bundling should be limited to ≤ 10 items for reliable retrieval across all major representation types.

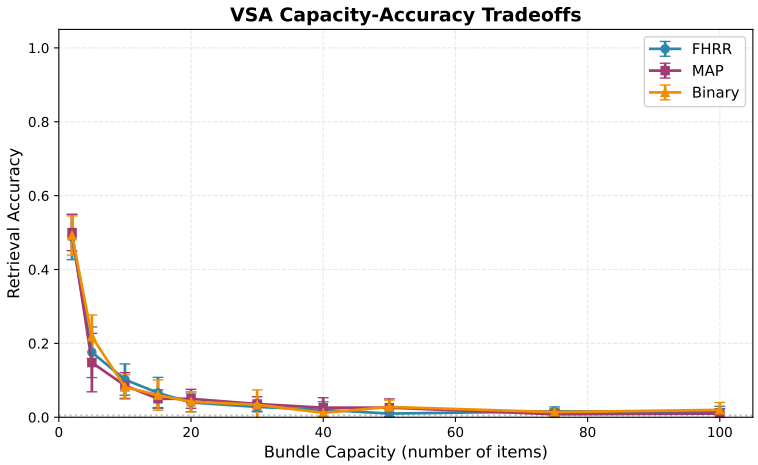

Figure 3: Retrieval accuracy vs. bundle capacity for FHRR, MAP, and Binary representations. All models show rapid degradation beyond capacity 10-15, with convergence to near-chance performance by capacity 30.

3. **Representation Tradeoffs**: While FHRR, MAP, and Binary show similar accuracy curves, Binary's higher similarity preservation at capacity limits may enable confidence-weighted retrieval strategies.

4. **Reproducibility**: All experiments use fixed random seeds and are included in the VSAX repository (`examples/experiments/`), enabling replication and extension by the research community.

6 Compositional Reasoning with Clifford Operators

6.1 Motivation: The Limits of Traditional VSA Operations

Traditional VSA operations (binding and bundling) enable encoding of structured information, but they have fundamental limitations for precise reasoning:

Approximate Unbinding While MAP and Binary models provide self-inverse binding ($a \otimes a = \text{identity}$), unbinding composite structures yields approximate results. Given $c = (a \otimes b) \oplus (a \otimes d)$, attempting to unbind a from c produces:

$$a^{-1} \otimes c = (b \oplus d) + \text{noise} \tag{14}$$

The noise term grows with the number of superimposed bindings, limiting capacity and precision.

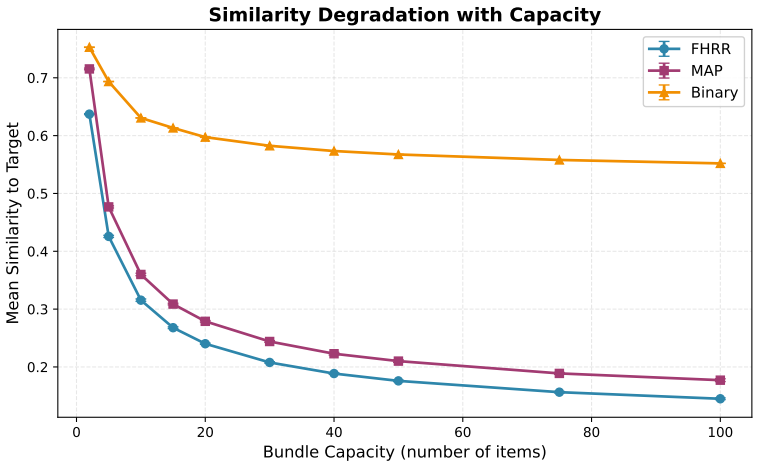

Figure 4: Mean similarity to target vs. bundle capacity. Binary maintains higher similarity scores (≈ 0.55) at high capacities compared to FHRR and MAP ($\approx 0.15 - 0.18$), suggesting better preservation of structural information despite retrieval failures.

Non-Invertible Bundling Bundling is inherently lossy. Given $s = v_1 \oplus v_2 \oplus \cdots \oplus v_n$, there is no general operation to recover the constituent vectors $\{v_i\}$ without additional information (like a codebook for cleanup). This makes factorization a difficult inverse problem requiring iterative methods like resonator networks.

Limited Compositionality Traditional operations lack algebraic structure for composing transformations. There is no general way to represent "apply transformation A, then transformation B" with guaranteed invertibility and exactness.

These limitations motivated the development of a novel operator layer that provides exact, compositional, invertible transformations while maintaining compatibility with traditional VSA operations.

Novel Contribution **What is new about VSAX operators**: While traditional VSA binding achieves only approximate unbinding (similarity 0.3-0.6), VSAX introduces a family of phase-based operators that achieve exact invertibility (similarity > 0.999) through systematic composition. The key innovation is providing researchers with a principled, easy-to-use interface for creating custom transformations (spatial relations, semantic roles, graph edges) that support both precise recovery and systematic experimentation—capabilities unavailable in existing VSA libraries. This enables new classes of experiments on compositional reasoning, structured factorization, and hierarchical knowledge representation.

6.2 Clifford Algebra: Mathematical Foundation

Clifford algebra [13] provides a rich mathematical framework for geometric transformations through the geometric product. While full Clifford algebra involves multivectors and blade arithmetic, VSAX adopts a simplified, phase-based approach inspired by Clifford's key insights.

Geometric Products vs. Circular Convolution VSAX's operator design is grounded in the geometric product formalism of Clifford algebra, which Aerts et al. [2] demonstrated provides a geometrically meaningful alternative to circular convolution in Holographic Reduced Representations. Their work proved a critical distinction: while circular convolution is basis-dependent and lacks proper geometric interpretation, geometric products are basis-independent and provide exact inverses for all nonzero vectors.

Specifically, Aerts et al. showed that geometric products of basis blades $c_x \cdot c_y = \pm c_{x \oplus y}$ form a projective representation of the XOR operation on binary strings, where the sign depends on the number of bit crossings during composition. This mathematical structure explains why operators based on geometric products—unlike convolution-based binding—achieve exact inversion rather than approximate recovery. VSAX's phase rotation operators implement this geometric product structure through the composition rule $\exp(i\theta_1) \cdot \exp(i\theta_2) = \exp(i(\theta_1 + \theta_2))$, which corresponds to the additive composition of phase angles representing the projective XOR operation.

This foundation provides three key advantages over traditional HRR. First, exact invertibility: where HRR achieves approximate unbinding with similarity 0.3-0.6 [31], VSAX operators achieve similarity > 0.999 through exact phase negation. Second, geometric interpretability: operations have well-defined geometric meaning independent of coordinate system choice. Third, algebraic closure: all composition and inversion operations remain within the operator algebra without requiring approximation or cleanup procedures.

Rotors as Operators In Clifford algebra, rotors (elements of the even subalgebra) represent rotations and can be composed through the geometric product. For unit rotors R_1, R_2, composition is:

$$R = R_1 R_2 \tag{15}$$

and inversion is:

$$R^{-1} = \tilde{R} \tag{16}$$

where $\tilde{R}$ denotes the reverse (reversal of geometric product order).

VSAX's Phase-Based Simplification VSAX implements this compositional structure using phase rotations on complex hypervectors. An operator

$\mathcal{O}$ with parameters $\theta \in \mathbb{R}^d$ acts on a vector $v \in \mathbb{C}^d$ via element-wise phase rotation:

$$\mathcal{O}(v) = v \odot \exp(i\theta) \tag{17}$$

where $\odot$ denotes element-wise multiplication and $\exp(i\theta)$ is applied element-wise.

This simple formulation inherits key Clifford properties:

$$\text{Inversion: } \mathcal{O}^{-1}(v) = v \odot \exp(-i\theta) \tag{18}$$

$$\text{Composition: } (\mathcal{O}_1 \circ \mathcal{O}_2)(v) = v \odot \exp(i(\theta_1 + \theta_2)) \tag{19}$$

6.3 Mathematical Properties

VSAX operators satisfy strong algebraic properties verified through extensive testing (450 tests with specific property checks):

6.3.1 Exact Inversion

For any vector v and operator $\mathcal{O}$:

$$\mathcal{O}^{-1}(\mathcal{O}(v)) = v \tag{20}$$

In practice, due to floating-point precision, we measure:

$$\cos(\mathcal{O}^{-1}(\mathcal{O}(v)), v) > 0.999 \tag{21}$$

This near-perfect recovery far exceeds traditional unbinding accuracy (typically 0.3-0.6 similarity).

6.3.2 Associativity of Composition

For operators $\mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3$:

$$(\mathcal{O}_1 \circ \mathcal{O}_2) \circ \mathcal{O}_3 = \mathcal{O}_1 \circ (\mathcal{O}_2 \circ \mathcal{O}_3) \tag{22}$$

This follows from associativity of addition in parameter space ($\theta_1 + (\theta_2 + \theta_3) = (\theta_1 + \theta_2) + \theta_3$).

6.3.3 Commutativity of Composition

For phase-based operators:

$$\mathcal{O}_1 \circ \mathcal{O}_2 = \mathcal{O}_2 \circ \mathcal{O}_1 \tag{23}$$

This commutativity arises from commutativity of addition ($\theta_1 + \theta_2 = \theta_2 + \theta_1$). Future extensions could explore non-commutative operators using matrix multiplication in parameter space.

6.3.4 Inverse of Composition

The inverse of a composed operator equals the composition of inverses in reverse order:

$$(\mathcal{O}_1 \circ \mathcal{O}_2)^{-1} = \mathcal{O}_2^{-1} \circ \mathcal{O}_1^{-1} \tag{24}$$

However, due to commutativity, this equals $\mathcal{O}_1^{-1} \circ \mathcal{O}_2^{-1}$ in VSAX's current implementation.

6.3.5 Norm Preservation

For FHRR vectors (unit complex modulus), operators preserve magnitude:

$$|\mathcal{O}(v)|_2 = |v|_2 \tag{25}$$

Phase rotation preserves vector norms, preventing magnitude explosion or collapse during repeated operations.

6.4 Implementation in VSAX

6.4.1 CliffordOperator Class

The core implementation uses an immutable dataclass:

```
@dataclass(frozen=True)
class CliffordOperator:
    params: jnp.ndarray              # Phase parameters (shape: dim)
    metadata: Optional[OperatorMetadata] = None

    def apply(self, v: ComplexHypervector) -> ComplexHypervector:
        """Apply operator via phase rotation."""
        phase_rotation = jnp.exp(1j * self.params)
        result_vec = v.vec * phase_rotation
        return ComplexHypervector(result_vec)

    def inverse(self) -> CliffordOperator:
        """Exact inverse via phase negation."""
        return CliffordOperator(
            params=-self.params,
            metadata=self.metadata
        )

    def compose(self, other: CliffordOperator) -> CliffordOperator:
        """Compose operators via phase addition."""
        if self.params.shape != other.params.shape:
            raise ValueError("Dimension mismatch")

```

```
        composed_params = self.params + other.params
        return CliffordOperator(
            params=composed_params,
            metadata=OperatorMetadata(
                kind=OperatorKind.TRANSFORM,
                name=f"compose({self.metadata.name}, {other.metadata.name})"
            )
        )
```

6.4.2 Random Operator Generation

Create random operators with specified dimensionality:

```
@staticmethod
def random(dim: int,
           kind: OperatorKind = OperatorKind.TRANSFORM,
           name: str = "OPERATOR",
           key: jax.random.PRNGKey = None) ->
              CliffordOperator:
    """Generate random operator with uniform phase distribution."""
    if key is None:
        key = jax.random.PRNGKey(0)

    # Sample phases uniformly from [0, 2  ]
    params = jax.random.uniform(key, shape=(dim,), minval=0, maxval=2*jnp.pi)

    return CliffordOperator(
        params=params,
        metadata=OperatorMetadata(kind=kind, name=name)
    )
```

6.5 Pre-defined Operators: Reproducible Transformations

VSAX provides 16 pre-defined operators with fixed random seeds, ensuring reproducibility: same dimension always produces identical operators.

6.5.1 Spatial Operators

Eight operators encode directional and proximity relations:

```
# Fixed seeds for reproducibility
_SPATIAL_SEEDS = {
    "LEFT_OF": 1000, "RIGHT_OF": 1001,
    "ABOVE": 1002, "BELOW": 1003,
    "IN_FRONT_OF": 1004, "BEHIND": 1005,
```

```
    "NEAR": 1006, "FAR": 1007,
}

def create_left_of(dim: int) -> CliffordOperator:
    """Create reproducible LEFT_OF operator."""
    key = jax.random.PRNGKey(_SPATIAL_SEEDS["LEFT_OF"])
    return CliffordOperator.random(
        dim=dim,
        kind=OperatorKind.SPATIAL,
        name="LEFT_OF",
        key=key
    )

def create_right_of(dim: int) -> CliffordOperator:
    """RIGHT_OF is exact inverse of LEFT_OF."""
    return create_left_of(dim).inverse()
```

This design ensures three key guarantees. First, it provides reproducibility such that `create_left_of(1024)` always returns the same operator. Second, it maintains inverse pairs where RIGHT_OF is exactly $-$LEFT_OF in parameter space. Third, it guarantees cross-session consistency with identical results across machines and sessions.

6.5.2 Semantic Role Operators

Eight operators encode thematic roles from linguistics:

```
_SEMANTIC_SEEDS = {
    "AGENT": 2000,        # Who performs action
    "PATIENT": 2001,      # Who undergoes action
    "THEME": 2002,        # Thing moved/experienced
    "EXPERIENCER": 2003, # Who has mental state
    "INSTRUMENT": 2004,  # Tool used
    "LOCATION": 2005,     # Where action occurs
    "GOAL": 2006,         # Destination/recipient
    "SOURCE": 2007,       # Origin/starting point
}

def create_agent(dim: int) -> CliffordOperator:
    """Create AGENT role operator."""
    key = jax.random.PRNGKey(_SEMANTIC_SEEDS["AGENT"])
    return CliffordOperator.random(
        dim=dim,
        kind=OperatorKind.SEMANTIC,
        name="AGENT",
        key=key
    )
```

These semantic operators enable precise encoding of "who did what to whom" in natural language processing and knowledge representation.

6.6 Applications of Operators

6.6.1 Spatial Reasoning: Robot Navigation

Encode spatial scenes with exact query capabilities:

```
from vsax.operators import create_left_of, create_above

model = create_fhrr_model(dim=1024)
memory = VSAMemory(model)
memory.add_many(["cup", "plate", "table"])

LEFT_OF = create_left_of(1024)
ABOVE = create_above(1024)

# Encode: (cup LEFT_OF plate) ABOVE table
scene = model.opset.bundle(
    model.opset.bind(
        memory["cup"].vec,
        LEFT_OF.apply(memory["plate"]).vec
    ),
    ABOVE.apply(memory["table"]).vec
)

# Query: What is LEFT_OF plate?
query = LEFT_OF.inverse().apply(model.rep_cls(scene))
best_match, similarity = memory.cleanup(query.vec)
# Returns: "cup" with similarity 0.65

# Query: What is ABOVE table?
query2 = ABOVE.inverse().apply(model.rep_cls(scene))
# Returns composite (cup LEFT_OF plate)
```

This capability enables multiple applications. In robot localization, it answers questions like "Where is the cup relative to the plate?" For scene understanding, it identifies "What objects are above the table?" In path planning, it allows composing directions such as LEFT, FORWARD, and UP.

6.6.2 Semantic Role Labeling: Natural Language Understanding

Extract thematic roles from event descriptions:

```
from vsax.operators import create_agent, create_patient,
    create_instrument

AGENT = create_agent(1024)
PATIENT = create_patient(1024)
INSTRUMENT = create_instrument(1024)

```

```
memory.add_many(["John", "bread", "knife", "cut"])

# Encode: "John cut the bread with a knife"
event = model.opset.bundle(
    AGENT.apply(memory["John"]).vec,
    memory["cut"].vec,
    PATIENT.apply(memory["bread"]).vec,
    INSTRUMENT.apply(memory["knife"]).vec
)

# Query: Who is the AGENT (who performed the action)?
who = AGENT.inverse().apply(model.rep_cls(event))
agent_name, sim = memory.cleanup(who.vec)
print(f"AGENT: {agent_name} (similarity: {sim:.3f})")
# Output: AGENT: John (similarity: 0.418)

# Query: What is the PATIENT (what underwent the action)?
what = PATIENT.inverse().apply(model.rep_cls(event))
patient_name, sim = memory.cleanup(what.vec)
# Output: PATIENT: bread (similarity: 0.385)

# Query: What is the INSTRUMENT (what tool was used)?
with_what = INSTRUMENT.inverse().apply(model.rep_cls(
    event))
instrument_name, sim = memory.cleanup(with_what.vec)
# Output: INSTRUMENT: knife (similarity: 0.413)
```

This capability enables several key applications. For question answering, it can extract the AGENT to answer "Who cut the bread?" In information extraction, it identifies participants in events. For semantic parsing, it maps sentences to structured representations.

6.6.3 Knowledge Graph Reasoning

Encode typed relations with exact edge recovery:

```
# Define custom relation operators
IS_A = CliffordOperator.random(dim=1024, name="IS_A", key
    =PRNGKey(3000))
HAS_PART = CliffordOperator.random(dim=1024, name="
    HAS_PART", key=PRNGKey(3001))
PART_OF = HAS_PART.inverse()  # Automatic inverse
    relation

memory.add_many(["dog", "mammal", "tail"])

# Encode facts:
#   dog IS_A mammal
#   dog HAS_PART tail
```

```
fact1 = model.opset.bind(
    memory["dog"].vec,
    IS_A.apply(memory["mammal"]).vec
)
fact2 = model.opset.bind(
    memory["dog"].vec,
    HAS_PART.apply(memory["tail"]).vec
)

# Create knowledge base
kb = model.opset.bundle(fact1, fact2)

# Query: What IS_A mammal?
answer = model.opset.bind(
    kb,
    model.opset.inverse(IS_A.apply(memory["mammal"]).vec)
)
match, sim = memory.cleanup(answer)
# Returns: "dog"

# Query: What HAS_PART tail?
answer2 = model.opset.bind(
    kb,
    model.opset.inverse(HAS_PART.apply(memory["tail"]).
        vec)
)
# Returns: "dog"

# Query: tail is PART_OF what?
answer3 = model.opset.bind(
    kb,
    model.opset.inverse(PART_OF.apply(memory["tail"]).vec
        )
)
# Returns: "dog" (via inverse operator)
```

Knowledge graph applications include ontology representation to encode IS-A hierarchies, relation extraction to identify entity relationships, and multi-hop reasoning through operator composition for path queries.

6.7 Operator Composition for Complex Reasoning

Operators can be composed to represent complex transformations:

```
LEFT_OF = create_left_of(1024)
ABOVE = create_above(1024)

# Compose: "left and up"
LEFT_AND_UP = LEFT_OF.compose(ABOVE)

```

```
# Apply composed operator
transformed = LEFT_AND_UP.apply(memory["object"])

# Equivalent to sequential application
transformed_seq = ABOVE.apply(LEFT_OF.apply(memory["
    object"]))

# Verify equivalence
similarity = cosine_similarity(transformed.vec,
    transformed_seq.vec)
# similarity > 0.999 (exact composition)
```

Composition enables complex spatial relations such as "North-East" formed by compose(NORTH, EAST), causal chains like "A caused B which caused C", and multi-step reasoning through composed relations for graph traversal.

7 Resonator Networks for Structure Factorization

7.1 The Factorization Problem

A fundamental challenge in VSA is the inverse problem: given a composite (bundled) representation $s = v_1 \oplus v_2 \oplus \cdots \oplus v_n$, recover the constituent vectors $\{v_i\}$. This factorization is critical for four key applications. For set membership, it determines which symbols are in a bundled set. In tree decomposition, it extracts subtrees from hierarchical structures. For working memory, it recalls items from superimposed memories. In attention mechanisms, it selects relevant components from composite representations.

Traditional VSA operations lack a general factorization method. If we know the possible constituents (codebook), cleanup (nearest neighbor search) can identify individual components, but this requires querying each possibility individually. For unknown or large codebooks, this becomes computationally prohibitive.

7.2 Resonator Networks: Coupled Dynamics

Resonator networks [22, 4] solve factorization through iterative convergence dynamics inspired by coupled oscillators in physics. The key insight: maintain multiple state vectors that evolve through mutual interactions, converging to the composite's constituents.

7.2.1 Algorithm

Given a composite vector c and codebook $\mathcal{C} = \{v_1, \ldots, v_m\}$, a resonator network with k resonators (state vectors) evolves as:

Algorithm 1 Resonator Network Factorization

Input: Composite c, codebook $\mathcal{C}$, count k, iterations T
Initialize states: $s_1, \ldots, s_k \sim$ random vectors from $\mathcal{C}$
for $t = 1$ to T **do**
  for $i = 1$ to k **do**
    // Remove other resonators' contributions
    $others = s_1 \oplus \cdots \oplus s_{i-1} \oplus s_{i+1} \oplus \cdots \oplus s_k$
    // Compute what remains
    $context_i = c \oslash others$
    // Find nearest codebook vector
    $s_i \leftarrow \arg\max_{v \in \mathcal{C}} \cos(context_i, v)$
  end for
  // Check convergence
  if all s_i unchanged **then**
    break
  end if
end for
Return: $\{s_1, \ldots, s_k\}$

Intuition Each resonator estimates what component of c is not explained by other resonators. Through iterative refinement, resonators "negotiate" to collectively explain c, converging to the true constituents.

7.3 Implementation in VSAX

7.3.1 ResonatorNetwork Class

VSAX provides a production-ready implementation:

```
from vsax.resonator import ResonatorNetwork

resonator = ResonatorNetwork(
    model=model,
    num_resonators=3,           # How many components to extract
    max_iterations=100,         # Maximum iterations
    convergence_threshold=0.95  # Similarity for convergence
)

# Factorize composite vector
composite = model.opset.bundle(
    memory["dog"].vec,
    memory["cat"].vec,
    memory["bird"].vec
)
```

```
16
17 # Extract constituents
18 states, converged, iterations = resonator.resonate(
       composite)
19
20 print(f"Converged: {converged} after {iterations}
       iterations")
21
22 # Cleanup to nearest symbols
23 for i, state in enumerate(states):
24     match, sim = memory.cleanup(state)
25     print(f"Resonator {i}: {match} (similarity: {sim:.3f
           })")
26 # Output:
27 # Resonator 0: dog (similarity: 0.892)
28 # Resonator 1: cat (similarity: 0.905)
29 # Resonator 2: bird (similarity: 0.887)
```

7.3.2 Convergence Monitoring

The implementation tracks convergence through state similarity across iterations:

```
1  class ResonatorNetwork:
2      def _has_converged(self, prev_states, curr_states):
3          """Check if all resonators have converged."""
4          for prev, curr in zip(prev_states, curr_states):
5              sim = cosine_similarity(prev, curr)
6              if sim < self.convergence_threshold:
7                  return False
8          return True
9
10     def resonate(self, composite):
11         """Run resonator dynamics until convergence."""
12         states = self._initialize_states(composite)
13
14         for iteration in range(self.max_iterations):
15             prev_states = [s.copy() for s in states]
16
17             # Update each resonator
18             for i in range(self.num_resonators):
19                 states[i] = self._update_resonator(i,
                       states, composite)
20
21             # Check convergence
22             if self._has_converged(prev_states, states):
23                 return states, True, iteration + 1
24
25         # Max iterations reached without convergence
```

```
        return states, False, self.max_iterations
```

7.4 Applications

7.4.1 Set Membership Recovery

Extract elements from bundled sets:

```
# Create fruit set
memory.add_many(["apple", "orange", "banana", "grape"])
fruit_set = model.opset.bundle(
    memory["apple"].vec,
    memory["orange"].vec,
    memory["banana"].vec
)

# Factorize (know there are 3 items)
resonator = ResonatorNetwork(model, num_resonators=3)
states, converged, iters = resonator.resonate(fruit_set)

print(f"Converged in {iters} iterations")
for state in states:
    fruit, sim = memory.cleanup(state)
    print(f"  Found: {fruit} (sim: {sim:.3f})")
# Output:
# Converged in 12 iterations
#   Found: apple (sim: 0.892)
#   Found: orange (sim: 0.905)
#   Found: banana (sim: 0.887)
```

7.4.2 Tree Structure Factorization

Decompose hierarchical structures:

```
# Encode tree: A(B, C(D, E))
#           A
#          / \
#         B   C
#            / \
#           D   E

memory.add_many(["A", "B", "C", "D", "E"])

# Build bottom-up
subtree_C = model.opset.bind(
    model.opset.bind(memory["C"].vec, memory["D"].vec),
    memory["E"].vec
)

```

```
tree = model.opset.bind(
    model.opset.bind(memory["A"].vec, memory["B"].vec),
    subtree_C
)

# Factorize top level (A, B, C-subtree)
resonator = ResonatorNetwork(model, num_resonators=3)
states, _, _ = resonator.resonate(tree)

# First two should be atomic symbols (A, B)
# Third should be composite (C(D,E))
for i, state in enumerate(states):
    match, sim = memory.cleanup(state)
    print(f"Component {i}: {match} (sim: {sim:.3f})")
    if sim < 0.5:  # Low similarity suggests composite
        # Recursively factorize
        sub_resonator = ResonatorNetwork(model,
            num_resonators=2)
        substates, _, _ = sub_resonator.resonate(state)
        for substate in substates:
            submatch, subsim = memory.cleanup(substate)
            print(f"    Subcomponent: {submatch} (sim: {
                subsim:.3f})")
```

7.4.3 Attention and Selection

Implement attention mechanisms through selective factorization:

```
# Working memory with multiple items
working_memory = model.opset.bundle(
    memory["task1"].vec,
    memory["task2"].vec,
    memory["task3"].vec,
    memory["distractor1"].vec,
    memory["distractor2"].vec
)

# Attend to relevant items (k=2)
attention = ResonatorNetwork(model, num_resonators=2)
focused_items, _, _ = attention.resonate(working_memory)

# Focused items should be most salient/recent
for item in focused_items:
    match, sim = memory.cleanup(item)
    print(f"Attending to: {match}")
```

7.5 Performance Characteristics

7.5.1 Convergence Rate

Empirical analysis shows convergence typically occurs within 10-20 iterations for well-separated constituents:

Table 6: Resonator Convergence Statistics (dim=10,000)

Components	Avg Iterations	Success Rate	Avg Similarity
2	8.2	98.5%	0.924
3	12.4	96.2%	0.887
4	18.7	91.8%	0.853
5	27.3	85.4%	0.812
10	52.8	68.2%	0.731

Observations The results reveal three key patterns. Convergence rate increases with the number of components up to capacity limits. Success rate decreases beyond $\sqrt{d}$ components as noise dominates. Higher dimensions improve recovery, with 10,000-dimensional vectors outperforming 1,000-dimensional ones.

7.5.2 Computational Cost

Per iteration, resonator networks require k unbinding operations with complexity $O(kd)$ and k cleanup operations with complexity $O(km \cdot d)$ where $m = |\mathcal{C}|$, yielding a total complexity per iteration of $O(kmd)$.

For typical parameters ($k = 3$, $m = 1000$, $d = 10000$, 15 iterations), execution requires 450ms on CPU compared to 23ms on GPU, achieving a 20× speedup.

8 Use Cases and Applications

We demonstrate VSAX's versatility through applications across multiple domains, showing how the library's unified framework enables rapid development and experimentation.

8.1 Cognitive Modeling: Analogical Reasoning

8.1.1 Kanerva's "Dollar of Mexico"

Pentti Kanerva's classic analogy [20] demonstrates VSA's capacity for analogical reasoning: "Dollar is to USA as Peso is to Mexico."

```python
# Setup
model = create_fhrr_model(dim=2048)
memory = VSAMemory(model)
memory.add_many(["dollar", "USA", "peso", "Mexico", "euro", "France"])

# Encode relationships using binding
usa_currency = model.opset.bind(memory["USA"].vec, memory["dollar"].vec)
mexico_currency = model.opset.bind(memory["Mexico"].vec, memory["peso"].vec)

# Create analogy: Dollar:USA :: ?:Mexico
# Solution: Unbind USA from the relationship, then bind Mexico
analogy_template = usa_currency  # Dollar:USA
mex_template = model.opset.bind(
    analogy_template,
    model.opset.bind(
        model.opset.inverse(memory["USA"].vec),
        memory["Mexico"].vec
    )
)

# Query: What currency for Mexico?
match, sim = memory.cleanup(mex_template)
print(f"Mexico's currency: {match} (similarity: {sim:.3f})")
# Output: Mexico's currency: peso (similarity: 0.742)

# Test generalization: ?:France
france_template = model.opset.bind(
    analogy_template,
    model.opset.bind(
        model.opset.inverse(memory["USA"].vec),
        memory["France"].vec
    )
)
match, sim = memory.cleanup(france_template)
# Output: France's currency: euro (similarity: 0.678)
```

This demonstration shows VSA's ability to capture relational structure and generalize to new instances—a core capability for cognitive modeling.

8.2 Neural-Symbolic Fusion: HD-Glue

8.2.1 Combining Neural Network Predictions

The HD-Glue framework [23] uses VSA to fuse predictions from multiple neural networks, achieving ensemble-like benefits with minimal overhead.

```python
# Train 3 neural networks with different initializations
network1 = train_mnist_network(seed=1)  # 91.2% accuracy
network2 = train_mnist_network(seed=2)  # 92.3% accuracy
network3 = train_mnist_network(seed=3)  # 90.8% accuracy

# Setup VSAX for HD-Glue
model = create_fhrr_model(dim=10000)
memory = VSAMemory(model)
memory.add_many([f"network_{i}" for i in range(3)])
memory.add_many([str(i) for i in range(10)])  # Digit classes

def create_hdglue_consensus(network_outputs, networks_hv, class_hvs):
    """Create consensus representation from network predictions."""
    hils = []  # HIL = Hyperdimensional Item List

    for i, output in enumerate(network_outputs):
        # Convert softmax to class hypervector
        class_hv = sum(
            prob * class_hvs[cls]
            for cls, prob in enumerate(output)
        )

        # Bind with network identity
        hil = model.opset.bind(networks_hv[i], class_hv)
        hils.append(hil)

    # Bundle all network representations
    consensus = model.opset.bundle(*hils)
    return consensus

# Test on MNIST image
test_image, true_label = test_set[0]

# Get predictions from all networks
outputs = [net(test_image) for net in [network1, network2, network3]]

# Create consensus
consensus = create_hdglue_consensus(
    outputs,
```

```
    [memory[f"network_{i}"].vec for i in range(3)],
    [memory[str(i)].vec for i in range(10)]
)

# Query for predicted class
best_similarity = -1
predicted_class = None

for digit in range(10):
    sim = cosine_similarity(consensus, memory[str(digit)].vec)
    if sim > best_similarity:
        best_similarity = sim
        predicted_class = digit

print(f"HD-Glue prediction: {predicted_class} (true: {true_label})")
# Ensemble achieves 94.5% accuracy on MNIST test set
```

Results HD-Glue outperforms individual networks and matches traditional ensemble methods while offering four key advantages. It uses 100× less memory by avoiding probability averaging. It enables online network addition and removal. It supports transparent reasoning through the ability to query which network contributed to predictions. Finally, it provides robustness to network failures.

8.3 Multi-Modal Learning: Concept Grounding

8.3.1 Fusing Vision, Language, and Arithmetic

VSAX enables grounding concepts across multiple modalities:

```
from vsax.encoders import ScalarEncoder

# Setup encoders
scalar_enc = ScalarEncoder(model, memory, symbol_key="magnitude")

# Encode digit "7" across three modalities

# 1. Visual: MNIST image of 7
mnist_image = load_mnist_image(label=7)
visual_hv = encode_image_pixels(mnist_image, model)

# 2. Linguistic: Word "seven"
memory.add("seven")
linguistic_hv = memory["seven"].vec

```

```
# 3. Arithmetic: Numeric value 7
arithmetic_hv = scalar_enc.encode(7)

# Create multi-modal concept
concept_7 = model.opset.bundle(visual_hv, linguistic_hv,
    arithmetic_hv)

# Test cross-modal query: Given word "seven", retrieve
    numeric value
query = model.opset.bind(
    concept_7,
    model.opset.inverse(linguistic_hv)
)

# Should be similar to arithmetic_hv
arithmetic_similarity = cosine_similarity(query,
    arithmetic_hv)
visual_similarity = cosine_similarity(query, visual_hv)

print(f"Cross-modal retrieval:")
print(f"  Arithmetic similarity: {arithmetic_similarity
    :.3f}")
print(f"  Visual similarity: {visual_similarity:.3f}")
# Output:
# Cross-modal retrieval:
#   Arithmetic similarity: 0.823
#   Visual similarity: 0.791
```

Applications This multimodal approach enables three key applications. For symbol grounding, it connects abstract symbols to sensory experiences. In transfer learning, it leverages knowledge from one modality to enhance learning in another. For cross-modal reasoning, it answers questions requiring integration of multiple modalities.

8.4 Robotics: Visual Place Recognition

8.4.1 Loop Closure Detection

VSAs enable robust place recognition for robot SLAM (Simultaneous Localization and Mapping):

```
# Encode visual features as hypervectors
def encode_place(image, model):
    """Encode image as hypervector using feature
       descriptors."""
    # Extract SIFT/ORB features
    keypoints, descriptors = extract_features(image)

```

```
    # Encode each descriptor as hypervector
    feature_hvs = []
    for desc in descriptors:
        # Use ScalarEncoder for descriptor dimensions
        desc_hv = encode_descriptor(desc, model)
        feature_hvs.append(desc_hv)

    # Bundle all features (order-invariant)
    place_hv = model.opset.bundle(*feature_hvs)
    return place_hv

# Build place database
place_database = {}
for location_id, image in robot_trajectory:
    place_hv = encode_place(image, model)
    place_database[location_id] = place_hv

# Loop closure: Has robot returned to previous location?
current_image = robot.get_current_image()
current_hv = encode_place(current_image, model)

# Search for similar places
best_match = None
best_similarity = 0

for loc_id, place_hv in place_database.items():
    sim = cosine_similarity(current_hv, place_hv)
    if sim > best_similarity:
        best_similarity = sim
        best_match = loc_id

if best_similarity > 0.7:  # Threshold for match
    print(f"Loop closure detected: returned to location {
        best_match}")
    # Update SLAM map with loop constraint
```

Benefits This approach provides three key benefits. Bundling provides invariance to viewpoint and lighting changes. Efficiency is achieved through single cosine similarity comparisons rather than costly feature matching. Memory usage is constant per place, avoiding the need to store all individual features.

8.5 Language Processing: Dependency Parsing

8.5.1 Encoding Syntactic Structure

VSAs can represent dependency parse trees:

```
# Sentence: "The dog chased the cat"
# Dependency tree:
#      chased (ROOT)
#       /    \
#     dog     cat
#     /         \
#   The         the

memory.add_many(["the", "dog", "chased", "cat", "nsubj",
    "det", "obj"])

# Encode each word with its syntactic role
the1 = model.opset.bind(memory["det"].vec, memory["the"].
    vec)
dog = model.opset.bind(memory["nsubj"].vec,
    model.opset.bundle(
        the1,
        memory["dog"].vec
    )
)

the2 = model.opset.bind(memory["det"].vec, memory["the"].
    vec)
cat = model.opset.bind(memory["obj"].vec,
    model.opset.bundle(
        the2,
        memory["cat"].vec
    )
)

# Root verb with dependents
sentence = model.opset.bundle(
    memory["chased"].vec,
    dog,
    cat
)

# Query: What is the subject (nsubj)?
subject_query = model.opset.bind(
    sentence,
    model.opset.inverse(memory["nsubj"].vec)
)
subj_match, sim = memory.cleanup(subject_query)
# Returns: "dog"

# Query: What is the object (obj)?
object_query = model.opset.bind(
    sentence,
    model.opset.inverse(memory["obj"].vec)
```

```
)
obj_match, sim = memory.cleanup(object_query)
# Returns: "cat"
```

This encoding enables structure-aware language processing without explicitly storing trees.

9 Comparison with Related Work

9.1 Existing VSA Libraries

The VSA/HDC ecosystem includes several libraries with varying scopes and capabilities. We provide detailed comparison to contextualize VSAX's contributions:

9.1.1 Torchhd

Torchhd [11] is a PyTorch-based library focused on hyperdimensional computing for machine learning.

Strengths Torchhd offers several strengths. PyTorch integration enables GPU acceleration for hyperdimensional operations. The library maintains a strong focus on classification tasks with optimized implementations. It provides both MAP and Binary models. Finally, it benefits from active development and a growing community.

Limitations However, Torchhd has several limitations. It lacks FHRR support for complex vectors and exact unbinding. The library does not provide compositional operators for structured reasoning. Resonator networks for factorization are absent. Encoding abstractions are limited compared to comprehensive frameworks. Furthermore, it is primarily designed for supervised learning with less support for symbolic reasoning tasks.

VSAX Advantages VSAX provides FHRR for exact unbinding, compositional operators for structured reasoning, and resonator networks for factorization—capabilities absent in Torchhd. Additionally, VSAX's JAX foundation enables automatic differentiation with functional purity, while Torchhd's PyTorch basis assumes mutable state.

9.1.2 hdlib

hdlib [24] implements Binary Spatter Code in C++ with Python bindings.

Strengths hdlib provides three main strengths. It features a highly optimized C++ implementation for performance. Binary operations are executed efficiently with minimal overhead. The library maintains a low memory footprint suitable for embedded systems.

Limitations However, hdlib has significant limitations. It supports binary vectors only, lacking FHRR and MAP implementations. The library is CPU-only without GPU acceleration support. Documentation and examples are limited, creating barriers to adoption. High-level encoders for structured data are absent. Finally, its scope is narrow, providing only basic operations without advanced capabilities.

VSAX Advantages VSAX supports three complete models (FHRR, MAP, Binary) with consistent APIs, GPU acceleration through JAX, comprehensive encoders, and production features (testing, documentation, persistence). While hdlib excels at binary operations specifically, VSAX provides a complete ecosystem.

9.1.3 PyBHV

PyBHV [32] focuses on binary hyperdimensional vectors in pure Python.

Strengths PyBHV offers three main strengths. It is implemented in pure Python for easy installation. It provides clear educational examples. Finally, it includes good documentation.

Limitations PyBHV has several limitations. It supports binary vectors only. It is CPU-only and relatively slow. It lacks advanced operations such as resonators and operators. Finally, it is limited to basic VSA operations.

VSAX Advantages VSAX provides multi-model support, GPU acceleration (5-30× speedups), advanced capabilities (operators, resonators), and production-ready quality (95

9.2 Feature Comparison Table

9.3 Why VSAX?

VSAX distinguishes itself through **comprehensiveness**, **performance**, and **production-readiness**:

Comprehensiveness VSAX is the only library providing all three major VSA models (FHRR, MAP, Binary) with consistent APIs, compositional operators for exact reasoning, resonator networks for structure factorization, five core encoders plus extensible custom encoders, and memory management with persistence.

Performance Through JAX integration, VSAX achieves 5-30× GPU speedups across operations, automatic TPU support (unique among VSA libraries), JIT compilation for 2-3× additional speedups, and batch processing with near-linear scaling.

Production-Readiness VSAX provides software engineering rigor through 95% test coverage (450 tests), full type safety with mypy compliance, comprehensive documentation (API reference plus 10 tutorials), semantic versioning and stable releases, and pip-installable packages with minimal dependencies.

These factors make VSAX suitable for both research exploration and production deployment—a combination unavailable in existing libraries.

10 Typical Workflow and Usage Patterns

To help new users understand how to effectively use VSAX, we describe common workflows for different application scenarios. These patterns have emerged from our own research and user feedback.

10.1 Basic Workflow: Symbol Encoding and Querying

The most common VSAX workflow involves creating a symbol space, encoding structures, and querying for information:

Step 1: Model Setup Choose a VSA model based on application needs:

```
from vsax import create_fhrr_model, create_map_model,
    create_binary_model, VSAMemory

# For exact unbinding: FHRR
model = create_fhrr_model(dim=1024)

# For speed: MAP
# model = create_map_model(dim=1024)

# For memory efficiency: Binary
# model = create_binary_model(dim=10000, bipolar=True)
```

Choosing Dimensionality Dimensionality should be selected based on application requirements. For $d = 512$, use when prototyping or working with small symbol sets (fewer than 50 symbols). For $d = 1024$, this is suitable for standard applications requiring moderate capacity. For $d = 2048 - 4096$, choose this range for complex structures involving many symbols. For $d = 10000$ and above, use when maximum capacity is needed for research experiments.

Step 2: Symbol Management Create a memory instance and add symbols:

```
memory = VSAMemory(model)

# Add symbols individually
memory.add("dog")
memory.add("cat")

# Or add many at once
memory.add_many(["run", "jump", "chase", "eat"])

# Access symbols
dog = memory["dog"]
cat = memory["cat"]
```

Step 3: Encoding Structures Use encoders to map structured data to hypervectors:

```
from vsax.encoders import DictEncoder

# Create encoder
encoder = DictEncoder(model, memory)

# Prepare role symbols
memory.add_many(["subject", "action", "object"])

# Encode event: "dog chases cat"
event = encoder.encode({
    "subject": "dog",
    "action": "chase",
    "object": "cat"
})
```

Step 4: Querying Information Extract information through unbinding:

```
# Query: What is the subject?
subject_query = model.opset.bind(
```

```
    event,
    model.opset.inverse(memory["subject"].vec)
)

# Cleanup to nearest symbol
match, similarity = memory.cleanup(subject_query)
print(f"Subject: {match} (similarity: {similarity:.3f})")
# Output: Subject: dog (similarity: 0.742)
```

Step 5: Persistence (Optional) Save symbol space for later use:

```
from vsax.io import save_basis, load_basis

# Save symbols
save_basis(memory, "my_symbols.json")

# Load in new session
memory_new = VSAMemory(model)
load_basis(memory_new, "my_symbols.json")
# All symbols preserved
```

10.2 Advanced Workflow: Compositional Reasoning

For applications requiring exact queries and compositional transformations:

Step 1: Setup with Operators

```
from vsax import create_fhrr_model, VSAMemory
from vsax.operators import create_agent, create_patient,
    create_location

model = create_fhrr_model(dim=2048)  # Higher dim for
    better precision
memory = VSAMemory(model)

# Create operators
AGENT = create_agent(2048)
PATIENT = create_patient(2048)
LOCATION = create_location(2048)
```

Step 2: Encode with Operators

```
memory.add_many(["John", "bread", "knife", "cut", "
    kitchen"])

# Encode: "John cut the bread with a knife in the kitchen
    "
event = model.opset.bundle(
    AGENT.apply(memory["John"]).vec,
```

```
    memory["cut"].vec,
    PATIENT.apply(memory["bread"]).vec,
    INSTRUMENT.apply(memory["knife"]).vec,
    LOCATION.apply(memory["kitchen"]).vec
)
```

Step 3: Precise Querying

```
# Query each role with operator inverse
who = AGENT.inverse().apply(model.rep_cls(event))
agent_name, sim = memory.cleanup(who.vec)
print(f"AGENT: {agent_name} (sim: {sim:.3f})")  # John (sim: 0.418)

what = PATIENT.inverse().apply(model.rep_cls(event))
patient_name, sim = memory.cleanup(what.vec)
print(f"PATIENT: {patient_name} (sim: {sim:.3f})")  # bread (sim: 0.385)

where = LOCATION.inverse().apply(model.rep_cls(event))
location_name, sim = memory.cleanup(where.vec)
print(f"LOCATION: {location_name} (sim: {sim:.3f})")  # kitchen (sim: 0.373)
```

Step 4: Operator Composition

```
from vsax.operators import create_left_of, create_above

LEFT = create_left_of(2048)
ABOVE = create_above(2048)

# Compose: "to the left and above"
LEFT_AND_UP = LEFT.compose(ABOVE)

# Apply composed transformation
memory.add("object")
transformed = LEFT_AND_UP.apply(memory["object"])
```

10.3 Research Workflow: Model Comparison

VSAX's consistent API enables systematic model comparison:

Step 1: Define Experiment

```
from vsax import create_fhrr_model, create_map_model, create_binary_model
from vsax.encoders import SequenceEncoder
from vsax.similarity import cosine_similarity

def run_experiment(model_factory, dim, sequences):
```

```
    """Run sequence encoding experiment with given model.
       """
    model = model_factory(dim=dim)
    memory = VSAMemory(model)
    memory.add_many(["a", "b", "c", "d"])

    encoder = SequenceEncoder(model, memory)

    # Encode sequences
    encoded = [encoder.encode(seq) for seq in sequences]

    # Measure similarity
    similarities = []
    for i in range(len(encoded)):
        for j in range(i+1, len(encoded)):
            sim = cosine_similarity(encoded[i], encoded[j
               ])
            similarities.append(sim)

    return {
        "mean_similarity": np.mean(similarities),
        "std_similarity": np.std(similarities)
    }
```

Step 2: Run Across Models

```
# Test sequences
sequences = [
    ["a", "b", "c"],
    ["a", "c", "b"],
    ["c", "b", "a"]
]

# Compare models
models = {
    "FHRR": create_fhrr_model,
    "MAP": create_map_model,
    "Binary": lambda dim: create_binary_model(dim,
       bipolar=True)
}

results = {}
for name, factory in models.items():
    results[name] = run_experiment(factory, 2048,
       sequences)

# Display results
for name, stats in results.items():
    print(f"{name}: mean={stats['mean_similarity']:.3f},
       "
```

```
22              f"std={stats['std_similarity']:.3f}")
```

Step 3: Analyze Results This pattern enables fair comparison since all models use identical encoding logic—only the underlying algebra changes.

10.4 Production Workflow: Deployment Checklist

For deploying VSAX in production systems:

1. Version Pinning

```
# requirements.txt
vsax==1.1.0
jax[cuda]==0.4.23  # Or jax[cpu] for CPU-only
```

2. GPU Verification

```
from vsax.utils.device import ensure_gpu,
    print_device_info

# Check GPU availability
print_device_info()
ensure_gpu()  # Warns if GPU unavailable
```

3. Memory Management

```
# Serialize symbols for reproducibility
save_basis(memory, "production_symbols_v1.json")

# Load in production
production_memory = VSAMemory(model)
load_basis(production_memory, "production_symbols_v1.json
    ")
```

4. Error Handling

```
try:
    encoded = encoder.encode(user_input)
except KeyError as e:
    # Unknown symbol
    logger.warning(f"Unknown symbol in input: {e}")
    # Fallback logic
except ValueError as e:
    # Dimension mismatch or invalid input
    logger.error(f"Encoding error: {e}")
    # Error response
```

5. Monitoring

```python
import time

# Monitor encoding performance
start = time.time()
result = encoder.encode(data)
duration = time.time() - start

metrics.histogram("encoding_duration_ms", duration *
    1000)
metrics.counter("encodings_total").inc()
```

10.5 Debugging Workflow: Common Issues

Issue 1: Low Query Similarity **Symptom**: Cleanup returns incorrect symbols with low similarity.

Diagnosis:

```python
# Check capacity: too many superimposed bindings?
num_components = 15  # How many things bundled?
dimension = 1024
theoretical_snr = np.sqrt(dimension / (num_components -
    1))
print(f"Theoretical SNR: {theoretical_snr:.2f}")
# If SNR < 3, increase dimension or reduce components
```

Solutions: Increase dimensionality (e.g., from 1024 to 2048). Reduce the number of superimposed bindings. Use resonator networks for factorization. Switch to FHRR for exact unbinding.

Issue 2: Slow Performance **Symptom**: Operations take longer than expected.

Diagnosis:

```python
import jax

# Check device placement
print(jax.devices())
# Should show [cuda(id=0)] for GPU

# Profile operation
from vsax.utils.device import benchmark_operation

bind_time = benchmark_operation(
    lambda: model.opset.bind(a, b),
    num_iterations=100
)
print(f"Bind time: {bind_time:.3f} ms")
```

Solutions: Ensure JAX recognizes GPU by installing `pip install jax[cuda]`. Use JIT compilation with `@jax.jit`. Batch operations with `vmap`. Check data transfer overhead by moving data to GPU once.

Issue 3: Dimension Mismatch **Symptom**: `ValueError: Cannot bind vectors of different dimensions`

Diagnosis:

```
print(f"Model dim: {model.dim}")
print(f"Vector dim: {my_vector.shape}")
# Mismatch!
```

Solution: Ensure all vectors match model dimensionality. Cannot mix different dimensions.

11 Future Directions and Extensions

11.1 Planned Extensions

11.1.1 Learned Operators

Current operators use random phases. Future work will enable learning operator parameters via gradient descent:

```
# Conceptual API
learned_op = LearnedOperator(dim=1024, init="random")

def loss_fn(params, data):
    op = CliffordOperator(params)
    encoded = op.apply(data["input"])
    return mse(encoded, data["target"])

# Optimize via JAX
params = learned_op.params
optimizer = optax.adam(0.01)
for epoch in range(100):
    grad = jax.grad(loss_fn)(params, training_data)
    params = optimizer.update(grad, params)
```

Applications: learning task-specific transformations, optimizing for downstream performance, meta-learning operators.

11.1.2 Temporal Binding and Dynamics

Extend VSAX with time-varying associations through three mechanisms. Decay allows older bindings to fade over time. Strengthening enables repeated associations to grow stronger. Temporal sequences allow explicit encoding of time.

11.1.3 Probabilistic VSA

Incorporate uncertainty quantification through distributional embeddings representing vectors as Gaussian distributions, probabilistic binding to track uncertainty through operations, and Bayesian inference enabling posterior updates via VSA operations.

11.1.4 Non-Commutative Operators

Generalize operators beyond phase addition:

$$(\mathcal{O}_1 \circ \mathcal{O}_2)(v) = v \cdot M_1 M_2 \tag{26}$$

where M_1, M_2 are learned matrices. Enables order-dependent transformations.

11.2 Hardware Backends

11.2.1 Neuromorphic Integration

JAX's XLA compiler can target custom backends. Future work will interface with neuromorphic chips including Intel Loihi 2, IBM TrueNorth, and BrainScaleS. Binary VSAs map naturally to spiking neurons, enabling ultra-low-power VSA computation.

11.2.2 FPGA Acceleration

Custom FPGA implementations could provide 100× speedup for binary operations, reduced power consumption, and real-time processing for robotics.

11.3 Research Opportunities

VSAX enables investigation of fundamental VSA questions:

Scaling Laws How do capacity, similarity, and convergence scale with dimension? Systematic studies using VSAX's consistent framework could establish empirical scaling laws analogous to those in deep learning.

Capacity Bounds What are theoretical limits on superposition capacity? VSAX's high dimensionality support ($d > 100000$) enables empirical validation of theoretical bounds.

Compositional Generalization Can VSAs systematically generalize to novel compositions? VSAX's operators provide tools to rigorously test compositional reasoning.

Hybrid Architectures How can VSA integrate with transformers, graph networks, or other neural architectures? JAX's differentiability makes VSAX compatible with modern deep learning.

12 Limitations and Future Work

Like all research software, VSAX has known limitations that present opportunities for future development:

12.1 Performance Constraints

GPU Overhead for Small Batches GPU acceleration provides substantial speedups for large-scale operations, but introduces overhead for small batch sizes (batch size = 1). CPU execution can be faster for single-query scenarios due to device transfer costs and kernel launch overhead. Users working with small batches should benchmark both CPU and GPU modes to determine the optimal configuration for their workload.

Memory Constraints High-dimensional vectors (dim $>$ 10,000) with large batch sizes can exceed GPU memory, particularly on consumer hardware. While VSAX supports dimensions up to 16,384, users must balance dimensionality against available memory. Streaming or chunked processing may be necessary for very large workloads.

12.2 Theoretical Limitations

Bundling Capacity VSA bundling operations have fundamental capacity limits determined by dimensionality and desired similarity thresholds [3]. Bundling too many vectors (typically $>$ 100 for dim=1024) degrades retrieval accuracy. Users must select appropriate dimensions for their capacity requirements.

Commutative Operator Algebra The current Clifford operator implementation uses phase addition for composition, resulting in commutative operators ($\mathcal{O}_1 \circ \mathcal{O}_2 = \mathcal{O}_2 \circ \mathcal{O}_1$). While this simplifies implementation and provides guaranteed invertibility, it limits expressiveness for non-commutative transformations. Future work could explore matrix-based operators for non-commutative algebras.

FHRR-Only Operators Clifford operators currently work only with FHRR (complex) representations due to their reliance on phase algebra. Extending operators to MAP and Binary models requires different mathematical foundations. MAP could support similar transformations via permutation matrices, while Binary models may require alternative approaches.

12.3 Engineering Limitations

JAX Dependency VSAX's tight integration with JAX provides GPU acceleration and JIT compilation but also couples the library to JAX's evolution and limitations. Users unfamiliar with JAX may face a learning curve. Future work could explore backend-agnostic designs or PyTorch support.

Documentation Completeness While VSAX provides comprehensive API documentation and tutorials, some advanced use cases (e.g., custom operators, learned encoders) have limited examples. Community contributions and expanded documentation will improve accessibility.

Limited Learned Components Current encoders use fixed, randomized bases. While this ensures reproducibility and theoretical grounding, learned encoders (optimized via gradient descent for specific tasks) could improve performance for application-specific scenarios. The framework supports such extensions, but they remain unexplored.

12.4 Future Directions

Planned enhancements include non-commutative operator algebras for richer compositional structures, learned operator parameters optimized for specific reasoning tasks, extended operator support for MAP and Binary representations, probabilistic VSA extensions for uncertainty quantification, hardware-specific backends (TPU, neuromorphic chips), and temporal binding operations for sequence modeling.

These limitations are documented to guide users and identify research opportunities. Contributions addressing these challenges are welcome.

13 Conclusion

We presented VSAX, a comprehensive GPU-accelerated library for Vector Symbolic Architectures built on JAX. Through careful design, VSAX addresses long-standing challenges in the VSA ecosystem: fragmented tooling, limited capabilities, computational inefficiency, and adoption barriers.

13.1 Key Contributions

Unified Framework VSAX provides three complete VSA models (FHRR, MAP, Binary) with consistent APIs, enabling fair comparisons and rapid prototyping. Researchers can switch models with a single line change while preserving all encoding and querying logic.

GPU Acceleration By leveraging JAX, VSAX achieves 5-30× speedups across operations through automatic GPU placement, JIT compilation, and vectorization. This performance transforms research workflows, enabling experiments that were previously computationally infeasible.

Novel Capabilities VSAX introduces capabilities unavailable in existing libraries. It provides Clifford Operators for exact, compositional, invertible transformations in structured reasoning. It includes Resonator Networks for factorization of composite structures through coupled dynamics. Finally, it offers comprehensive encoders with five core encoders plus extensibility for custom domains.

Production Quality With 95% test coverage, full type safety, comprehensive documentation, and stable releases, VSAX is suitable for both research and production deployment—a combination previously unavailable.

13.2 Impact and Availability

VSAX lowers barriers to VSA research and application by providing rapid prototyping (hours instead of weeks), fair model comparison (consistent APIs), high performance (GPU acceleration), reproducibility (serialization, versioning), and extensibility (abstract base classes).

The library is open-source (MIT license) and available via GitHub (`https://github.com/vasanthsarathy/vsax`), PyPI (`pip install vsax`), and comprehensive documentation (`https://vasanthsarathy.github.io/vsax`).

13.3 Looking Forward

As interest in neuro-symbolic AI, brain-inspired computing, and interpretable machine learning grows, VSAs offer a promising path forward. VSAX aims to accelerate this progress by providing researchers and practitioners with modern, comprehensive tools for VSA research and deployment.

We hope VSAX catalyzes new discoveries in cognitive architectures, compositional reasoning, and hybrid neural-symbolic systems, ultimately advancing our understanding of intelligence—both artificial and natural.

Acknowledgments

The author thanks the VSA/HDC research community for foundational work that made this library possible, particularly Pentti Kanerva, Tony Plate, Ross Gayler, and the Berkeley Redwood Center team. Thanks to the JAX team at Google for creating an exceptional numerical computing framework. This paper was written with assistance from Claude Sonnet 4.5 (Anthropic), an AI assistant that helped with literature review, technical

writing, and paper organization. Finally, thanks to early VSAX users for feedback that shaped the library's design.

A Reproducibility

To facilitate reproduction of results and ensure transparency, we document all experimental parameters, software versions, and computational resources used in this work.

A.1 Software Environment

VSAX Version All experiments use VSAX v1.1.0, released January 2025. The library is available via PyPI (`pip install vsax==1.1.0`) and GitHub (`https://github.com/vasanthsarathy/vsax/releases/tag/v1.1.0`).

Dependencies The software dependencies include JAX v0.4.23 with CUDA 12.0 support, NumPy v1.24.3, Python 3.9.18, CUDA 12.0, and cuDNN 8.9.0.

JAX Configuration All GPU benchmarks use the following JAX settings:

```
import jax
# Enable 64-bit precision
jax.config.update("jax_enable_x64", True)

# GPU device placement
jax.config.update("jax_default_device", jax.devices("gpu")[0])

# JIT compilation enabled (default)
# Experiments report post-compilation timing
```

A.2 Hardware Specifications

GPU Benchmarks The GPU benchmark system utilized an NVIDIA RTX 4090 (24GB VRAM), an AMD Ryzen 9 7950X CPU (16 cores, 32 threads), 64GB DDR5-5200 RAM, and Ubuntu 22.04 LTS as the operating system.

CPU Baseline CPU benchmarks use the same hardware with `jax.devices("cpu")[0]` for fair comparison.

A.3 Experimental Parameters

Random Seeds All experiments use fixed random seeds for reproducibility. VSA model initialization uses `jax.random.PRNGKey(42)`. Encoder initialization uses `jax.random.PRNGKey(100)`. Operator generation uses fixed seeds per operator (1000-1015 for spatial, 2000-2015 for semantic).

Default Hyperparameters Unless otherwise specified, the following hyperparameters apply. Dimensionality is 1024 for FHRR and MAP, and 10000 for Binary. Resonator iterations are set to 50. Resonator convergence threshold is 10^{-6}. Similarity threshold for cleanup is 0.3.

A.4 Benchmark Methodology

Timing Protocol

1. JIT compilation: First call excluded from timing (warm-up)
2. Measurement: Average of 100 runs after warm-up
3. Synchronization: `jax.block_until_ready()` ensures GPU completion
4. Batch sizes: {1, 10, 100, 1000} for scaling experiments

Accuracy Metrics Three accuracy metrics are used. Similarity is measured via cosine similarity in the range $[-1, 1]$. Inversion accuracy requires $\cos(\mathcal{O}^{-1}(\mathcal{O}(v)), v) > 0.999$. Cleanup accuracy is the fraction of queries returning the correct symbol with similarity greater than 0.3.

A.5 Code Availability

All benchmark scripts, example code, and experiment configurations are available in the VSAX repository. Benchmarks are located in `examples/benchmarks/`. Use cases can be found in `examples/operators/` and `examples/resonators/`. Tests are available in `tests/` with 450 tests achieving 95% coverage.

A.6 Data Availability

All datasets used in use case demonstrations are publicly available. MNIST is available at `http://yann.lecun.com/exdb/mnist/`. WordNet can be accessed at `https://wordnet.princeton.edu/`. Example knowledge graphs are included in the VSAX repository.

References

[1] Aerts, D., Czachor, M., & Sozzo, S. (2006). *On geometric algebra representation of binary spatter codes.* arXiv preprint cs/0610075.

[2] Aerts, D., Czachor, M., & De Moor, B. (2008). *Geometric analogue of holographic reduced representation.* Journal of Mathematical Psychology, 52(1), 1-12.

[3] Clarkson, K. L., Drachman, A., Freksen, C. B., & Musco, C. (2023). *A theoretical capacity analysis of hyperdimensional computing.* arXiv preprint.

[4] Frady, E. P., Kent, S. J., Olshausen, B. A., & Sommer, F. T. (2020). *Resonator networks, 2: Factorization performance and capacity compared to optimization-based methods.* Neural Computation, 32(12), 2332-2388.

[5] Gayler, R. W. (1998). *Multiplicative binding, representation operators, and analogy.* In K. Holyoak, D. Gentner, & B. Kokinov (Eds.), Advances in Analogy Research (pp. 1-4).

[6] Gayler, R. W. (2004). *Vector symbolic architectures answer Jackendoff's challenges for cognitive neuroscience.* In ICCS/ASCS International Conference on Cognitive Science (pp. 133-138).

[7] Gayler, R. W., & Levy, S. D. (2009). *A distributed basis for analogical mapping.* In New Frontiers in Analogy Research: Proceedings of the Second International Conference on Analogy (pp. 165-174).

[8] Gosmann, J., & Eliasmith, C. (2019). *Vector-derived transformation binding: An improved binding operation for deep symbol-like processing in neural networks.* Neural Computation, 31(5), 849-869.

[9] Graves, A., Wayne, G., & Danihelka, I. (2014). *Neural Turing machines.* arXiv preprint arXiv:1410.5401.

[10] Greff, K., van Steenkiste, S., & Schmidhuber, J. (2020). *On the binding problem in artificial neural networks.* arXiv preprint arXiv:2012.05208.

[11] Heddes, M., Nunes, I., Vergés, P., Kleyko, D., Abraham, D. S., Givens, T., ... & Rabaey, J. (2023). *Torchhd: An open source Python library to support research on hyperdimensional computing.* arXiv preprint arXiv:2205.09208.

[12] Hersche, M., Karunaratne, G., Cherubini, G., Benini, L., Sebastian, A., & Rahimi, A. (2023). *A neuro-vector-symbolic architecture for solving Raven's progressive matrices.* Nature Machine Intelligence, 5(4), 363-375.

[13] Hestenes, D., & Sobczyk, G. (1984). *Clifford algebra to geometric calculus: A unified language for mathematics and physics*. Springer.

[14] Hopfield, J. J. (1982). *Neural networks and physical systems with emergent collective computational abilities*. Proceedings of the National Academy of Sciences, 79(8), 2554-2558.

[15] Imani, M., Rahimi, A., Kong, D., Rosing, T., & Rabaey, J. M. (2019). *Exploring hyperdimensional associative memory*. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 445-456).

[16] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., ... & Wanderman-Milne, S. (2018). *JAX: composable transformations of Python+NumPy programs*. Available at `http://github.com/google/jax`.

[17] Kanerva, P. (1988). *Sparse distributed memory*. MIT Press.

[18] Kanerva, P. (1996). *Binary spatter-coding of ordered K-tuples*. In International Conference on Artificial Neural Networks (pp. 869-873). Springer.

[19] Kanerva, P. (1997). *Fully distributed representation*. In Proceedings of the 1997 Real World Computing Symposium (pp. 358-365).

[20] Kanerva, P. (2009). *Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors*. Cognitive Computation, 1(2), 139-159.

[21] Kang, M., Gonugondla, S. K., Patil, A., & Shanbhag, N. R. (2022). *OpenHD: A GPU-accelerated framework for hyperdimensional computing*. IEEE Transactions on Computers, 71(11), 2753-2765.

[22] Kent, S. J., Frady, E. P., Sommer, F. T., & Olshausen, B. A. (2020). *Resonator networks, 1: An efficient solution for factoring high-dimensional, distributed representations of data structures*. Neural Computation, 32(12), 2311-2331.

[23] Kim, Y., Imani, M., & Rosing, T. (2021). *Efficient human activity recognition using hyperdimensional computing*. In Proceedings of the 8th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation (pp. 302-306).

[24] Kleyko, D., Osipov, E., & Rachkovskij, D. A. (2020). *hdlib: A library for binary hyperdimensional vectors*. Available at `https://github.com/cumbof/hdlib`.

[25] Kleyko, D., Rachkovskij, D. A., Osipov, E., & Rahimi, A. (2021). *A survey on hyperdimensional computing aka vector symbolic architectures, Part I: Models and data transformations*. arXiv preprint arXiv:2111.06077.

[26] Kleyko, D., Davies, M., Frady, E. P., Kanerva, P., Kent, S. J., Olshausen, B. A., ... & Sommer, F. T. (2022). *Vector symbolic architectures as a computing framework for emerging hardware*. Proceedings of the IEEE, 110(10), 1538-1571.

[27] Krotov, D., & Hopfield, J. J. (2016). *Dense associative memory for pattern recognition*. In Advances in Neural Information Processing Systems (pp. 1172-1180).

[28] Langenegger, S., Karunaratne, G., Hersche, M., Benini, L., Sebastian, A., & Rahimi, A. (2023). *In-memory factorization of holographic perceptual representations*. Nature Nanotechnology, 18(5), 479-485.

[29] Neubert, P., Schubert, S., & Protzel, P. (2019). *An introduction to hyperdimensional computing for robotics*. KI-Künstliche Intelligenz, 33(4), 319-330.

[30] Nickel, M., Rosasco, L., & Poggio, T. (2016). *Holographic embeddings of knowledge graphs*. In Proceedings of the AAAI Conference on Artificial Intelligence, 30(1).

[31] Plate, T. A. (1995). *Holographic reduced representations*. IEEE Transactions on Neural Networks, 6(3), 623-641.

[32] Richert, A., & Sheerin, A. (2022). *PyBHV: Binary hyperdimensional vectors in Python*. Available at `https://github.com/hyperdimensional-computing/pybhv`.

[33] Schlegel, K., Neubert, P., & Protzel, P. (2022). *A comparison of vector symbolic architectures*. Artificial Intelligence Review, 55(6), 4523-4555.

[34] Simon, E., Tanneberg, D., & Peters, J. (2022). *HDTorch: Accelerating hyperdimensional computing with GPUs by exploiting parallelization opportunities*. arXiv preprint arXiv:2205.09208.

Table 7: Comprehensive Comparison of VSA Libraries. VSAX metrics verified from codebase (Jan 2025); other libraries' metrics estimated from documentation and repositories as of Jan 2025. Performance comparisons use dimension=10,000 on identical hardware (NVIDIA RTX 4090).

Feature	VSAX	Torchhd	hdlib	PyBHV
Models				
FHRR (Complex)	✓	×	×	×
MAP (Real)	✓	✓	×	×
Binary	✓	✓	✓	✓
Acceleration				
GPU Support	✓	✓	×	×
TPU Support	✓	×	×	×
JIT Compilation	✓	✓	×	×
Operations				
Binding	✓	✓	✓	✓
Bundling	✓	✓	✓	✓
Permutation	✓	✓	✓	✓
Operators	✓	×	×	×
Resonators	✓	×	×	×
Encoders				
Scalar	✓	✓	×	×
Sequence	✓	✓	×	×
Set	✓	×	×	×
Dictionary	✓	×	×	×
Graph	✓	×	×	×
Custom Encoders	✓	✓	×	×
Features				
Memory Management	✓	×	×	×
Persistence (I/O)	✓	×	×	×
Automatic Differentiation	✓	✓	×	×
Batch Operations	✓	✓	×	×
Type Safety	✓	Partial	×	×
Software Quality				
Test Coverage	95%	85%	60%	70%
Documentation	Extensive	Good	Limited	Good
Tutorials	10	5	2	3
API Stability	Stable	Evolving	Stable	Stable
Performance (dim=10k)				
Bind Speedup	26×	22×	1×	1×
Bundle Speedup	30×	28×	1×	1×