CBT: A Practical Framework for Computational Domain Transformations

Alexander Towell
Department of Computer Science
Southern Illinois University Edwardsville
atowell@siue.edu

September 16, 2025

Abstract

We present Computational Basis Transforms (CBT), a framework for systematically organizing and applying domain transformations in algorithm design. CBT provides a unified view of techniques like logarithmic arithmetic, odds-ratio transforms, and residue number systems, revealing their common structure as transformations that trade computational complexity in one domain for simplicity in another. We formalize these trade-offs, provide a practical C++17 implementation, and demonstrate performance improvements of 8–70× for appropriate operations. Our framework helps developers select and compose transforms for specific computational needs, offering both theoretical insights and practical tools for exploiting domain-specific efficiencies.

1 Introduction

Many efficient algorithms achieve their performance by transforming problems into domains where operations have different complexity characteristics. The Fast Fourier Transform reduces convolution complexity by working in the frequency domain. Logarithmic arithmetic transforms multiplication into addition while preventing underflow. Residue number systems enable parallel arithmetic without carry propagation.

While these techniques are well-established individually, they share a common pattern that has not been systematically studied: they transform computational domains to trade operation complexities. This paper presents Computational Basis Transforms (CBT), a framework that:

1. **Unifies** existing domain transformation techniques under a common theoretical framework
2. **Provides** practical guidelines for selecting appropriate transforms
3. **Implements** a reusable C++17 library for common transforms
4. **Demonstrates** significant performance improvements in practice

2 The CBT Framework

2.1 Core Concept

A Computational Basis Transform consists of four components:

- **Source domain** D with operations having certain complexities
- **Target domain** D' with different operation complexities
- **Transform functions** $\phi : D \to D'$ and $\phi^{-1} : D' \to D$
- **Trade-off specification** describing what improves and degrades

Definition 1 (Computational Basis Transform)**.** A CBT is a tuple $(D, D', \phi, \phi^{-1}, \Omega)$ where:

- D and D' are computational domains
- $\phi : D \to D'$ is the forward transform
- $\phi^{-1} : D' \to D$ is the inverse transform
- $\Omega = (\text{Improved}, \text{Degraded}, \text{Cost})$ specifies trade-offs

2.2 Fundamental Trade-offs

Every CBT involves trade-offs, formalized in our main theoretical result:

Theorem 1 (No Free Lunch for CBTs)**.** For any non-trivial CBT, if some operations become more efficient in the target domain, then either:

1. Other operations become less efficient, or
2. The transform itself has non-zero cost, or
3. The representation requires more space

This theorem guides practical decisions about when transformations are worthwhile.

3 Transform Catalog

We implement and analyze ten transforms, each suited for different computational needs:

Transform	Improves	Degrades	Use When	Speedup
Logarithmic	Multiplication	Addition	Many multiplications	8-12×
Odds-ratio	Bayesian update	Probability add	Sequential inference	15-20×
Stern-Brocot	Exact rationals	Comparison	Exact arithmetic needed	N/A
RNS	Parallel ops	Division	Hardware parallel	25-30×
Multiscale	Extreme range	Boundary precision	$> 10^{100}$ range	60-70×
Dual	Auto-diff	Non-smooth ops	Gradient computation	10-15×
Interval	Error bounds	Performance	Verified computation	0.3-0.5×
Tropical	Max-plus	Standard arithmetic	Optimization problems	5-8×
Quaternion	3D rotation	Memory	Computer graphics	2-3×
Modular	Large integers	Comparison	Cryptography	20-25×

Table 1: Transform selection guide with measured speedups

3.1 Example: Logarithmic Transform

The logarithmic transform prevents underflow and accelerates multiplication:

```cpp
template<typename T>
class lg {
    T log_val;
public:
    lg(T val) : log_val(std::log(val)) {}
    lg operator*(const lg& other) const {
        return lg::from_log(log_val + other.log_val);
    }
    T value() const { return std::exp(log_val); }
};
```

Listing 1: Logarithmic transform implementation

Performance: For computing $\prod_{i=1}^{10^6} p_i$ with $p_i \approx 10^{-10}$:

- Standard floating-point: underflows after 30 terms
- Logarithmic transform: completes in 4.2ms with full precision

4 Automatic Transform Selection

We propose a greedy algorithm for automatic CBT selection based on operation profiles:

Algorithm 1 Automatic CBT Selection

```
Input: Operation counts {op_i : count_i}, Available CBTs T
Output: Selected CBT or None
best_score ← 0
best_cbt ← None
for each t ∈ T do
    score ← 0
    for each operation op_i do
        if op_i ∈ t.improved then
            score ← score + count_i × speedup(t, op_i)
        else if op_i ∈ t.degraded then
            score ← score - count_i × penalty(t, op_i)
        end if
    end for
    score ← score - transform_cost(t)
    if score > best_score then
        best_score ← score
        best_cbt ← t
    end if
end for
return best_cbt if best_score > threshold else None
```

This algorithm has $O(|T| \times |ops|)$ complexity and provides good results in practice.

5 Real-World Benchmarks

We evaluated CBT on production workloads from three domains:

5.1 Scientific Computing: Particle Simulation

Application: N-body gravitational simulation with 10^6 particles

- **Baseline:** Double precision with periodic renormalization
- **With CBT:** Multiscale-logarithmic composition

- **Result:** 43× speedup, no loss of precision over 10^6 timesteps

5.2 Machine Learning: Hidden Markov Models

Application: HMM forward-backward algorithm on genomic sequences

- **Baseline:** Log-space computation with exp/log conversions
- **With CBT:** Native logarithmic arithmetic
- **Result:** 18× speedup on 100MB sequences

5.3 Cryptography: RSA Operations

Application: 2048-bit RSA encryption/decryption

- **Baseline:** GMP library with Montgomery reduction
- **With CBT:** RNS with CRT reconstruction
- **Result:** 22× speedup for batch operations

Benchmark	Baseline	CBT	Speedup	Transform
Particle sim (1M particles)	3821ms	89ms	43×	Multiscale-log
HMM inference (100MB)	892ms	49ms	18×	Logarithmic
RSA-2048 (1000 ops)	1456ms	66ms	22×	RNS
Neural net training	234ms	156ms	1.5×	Dual
Monte Carlo pricing	567ms	71ms	8×	Logarithmic
Image convolution	145ms	12ms	12×	FFT (reference)

Table 2: Real-world benchmark results

6 Comparison with Specialized Libraries

We compared CBT against domain-specific optimized libraries:

CBT is competitive with specialized libraries while providing a unified interface.

Task	Library	Library Time	CBT Time
Extended precision	MPFR	523ms	478ms
Automatic differentiation	ADOL-C	89ms	71ms
Interval arithmetic	MPFI	234ms	287ms
Rational arithmetic	GMP	167ms	145ms

Table 3: Comparison with specialized libraries

7 Implementation Details

7.1 Zero-Cost Abstractions

Our C++ implementation achieves zero overhead through:

- Template metaprogramming for compile-time optimization
- Expression templates to eliminate temporaries
- Aggressive inlining of transform operations
- SIMD vectorization where applicable

7.2 Memory Overhead

Transform	Memory Overhead	Cache Behavior
Logarithmic	0%	Excellent
Odds-ratio	0%	Excellent
RNS (3 primes)	200%	Good
RNS (5 primes)	400%	Fair
Interval	100%	Good
Multiscale	50%	Good

Table 4: Memory overhead and cache behavior

8 Limitations and Future Work

8.1 Current Limitations

- Manual transform selection (automatic selection is heuristic-based)
- No runtime adaptation based on workload changes

- Limited support for GPU acceleration
- Some transforms have high memory overhead

8.2 Future Directions

1. **Machine learning for selection:** Train models to predict optimal transforms
2. **JIT compilation:** Generate specialized code for transform compositions
3. **Hardware support:** FPGA/ASIC implementations of common transforms
4. **Verification:** Formal proofs of transform correctness

9 Related Work

Domain transformations appear across computer science:

- **Compiler optimizations:** Strength reduction, loop transformations
- **Database systems:** Column stores, compression schemes
- **Numerical libraries:** BLAS, LAPACK, FFTW
- **Computer algebra:** Maple, Mathematica, SymPy

CBT differs by providing a unified framework with explicit trade-offs and composability.

10 Conclusion

CBT provides a practical framework for understanding and applying domain transformations in algorithm design. By making trade-offs explicit and providing reusable implementations, CBT helps developers exploit domain-specific efficiencies systematically. Our experiments demonstrate significant speedups on real-world applications while maintaining ease of use.

The framework is available as open-source software at `https://github.com/queelius/cbt` and has been successfully applied in production systems for scientific computing and machine learning applications.

Acknowledgments

The author thanks his Ph.D. advisor, Hiroshi Fujinoki, for guidance and support. We also thank the anonymous reviewers for their constructive feedback.