PAPER VIII

Packet Operator Algorithms

Exact compilation and GPU realization

Read the finite objects, represented operators, exact compilers, and optimal recompression proofs in their actual source. Each interface states the data it receives and the conclusion it establishes.

Komorebi P.

Lean and evidence guide • 9 October 2026

How to use this volume

This guide teaches the actual finite Lean development. It follows the represented objects, states their hypotheses, and reads representative proofs step by step. It includes 61 exact source selections, the complete 56-result correspondence, four separately tracked classical inputs, and all 458 named audit entries.

The source has 40 modules, 392 theorem declarations (373 for Paper VIII and 19 in the reused foundation), and 66 construction audits. These counts include supporting results and specializations. They do not count independent mathematical discoveries.

Three broader research targets remain open: unrestricted discovery of profitable packet structure, compilation with nonlocal costs and overlapping decompositions, and a packet memory communication theorem for a specified architecture.

The preserved run completed on 6 September 2026. It records successful cumulative compilation and axiom audits. This documentation edition freshly checks source identity, saved logs, declaration names, exact excerpts, and PDF construction. Lean and Lake are unavailable on this host; there is no new kernel execution in this edition.

The traditional paper develops the mathematical arguments. The visual guide works through their constructions. This companion explains their formal counterparts and the exact boundary between proved conclusions and supplied data. The accompanying `README.md` gives the complete monograph regeneration sequence, including static evidence refresh and final page navigation before rebuilding.

Source base: `original_snapshot/formalization/`.

Primary inherited evidence: `original_snapshot/verification/sixth_final/`.

Raw UTF-8 excerpts and their SHA-256 register accompany the printed selections.

Contents

1 Reading a finite formal development 1

2 Navigation through the traditional volume 3

3 Exact packet semantics 5

4 Representation algebra and declared cost models 8

5 Structured algorithms carry their coordinate conventions 11

6 Transfer paths, inverses, and acyclic solves 14

7 Error propagation and exact precision allocation 17

8 Exact packet compilation and cached refinement 21

9 Output contracts, batching, and Krylov certificates 26

10 Small-core optimal recompression from constructed ranges 30

11 Replay, provenance, and the four classical boundaries 38

12 Complete manuscript-result correspondence 41

13 All named theorem and construction audits 58

14 Dependencies and source identities 76

15 Exact excerpt register 81

1 Reading a finite formal development

1.1 The objects precede the theorem names

A packet representation starts with a finite projector family and an operator. A structured representation adds explicit factor matrices or tensor coordinates. A compiler starts with a finite candidate tree and an independently defined set of legal cuts. An error theorem starts with actual errors in a normed space and budgets that bound them. In each case, the theorem follows the object being modeled. Reading only the endpoint name misses the domain, norm, convention, and supplied data on which its conclusion depends.

Use the traditional paper's original item numbers to find the corresponding subsection in Section 12. Then locate its declaration in Section 13 and its module in Section 14. For a first reading, work through the chapters in order. The small-core chapter is longer because the final block proves optimality from several independent ingredients, rather than accepting an optimality certificate as input.

1.2 A short Lean vocabulary

Source form	Meaning in this guide
`Matrix m n K`	A function with row type `m`, column type `n`, and coefficients in `K`. Finite index instances provide finite sums; rectangular shapes are visible in the type.
`[Fintype I]`	A finite enumeration of the index type. Empty index types are allowed unless a nonempty hypothesis is separately stated.
`[DecidableEq I]`	Equality can be decided where the construction needs it. It is not a numerical tolerance test.
`Fin k`	Natural indices smaller than k. The type `Fin 0` is empty.
`[Ring R]`	Addition and multiplication are available; multiplication need not commute.
`IsUnit D`	The matrix or algebra element has a two-sided multiplicative inverse. A displayed inverse expression alone does not establish invertibility.
`X →L[K] X`	A continuous linear map over `K`, with its operator norm.
`Option R`	Either `some v`, a finite candidate value, or `none`, an unreachable / infeasible state.
`let A := ...`	A local definition in the statement. It abbreviates the constructed object; it adds no hypothesis.
`noncomputable def`	A mathematically defined object using classical choice or other noncomputable ingredients. It does not advertise an executable numerical routine.

A theorem statement may inherit `variable` declarations from its section. An `omit ... in` command removes unused instances for that declaration. `autoImplicit false` means an undeclared symbol is rejected rather than silently turned into an implicit parameter. Names such as `SmallCore.spectralNorm` explicitly choose the intended norm; a bare matrix norm also depends on the open norm scope surrounding its declaration.

1.3 How to read a proof

`rw` rewrites by an equality. `simp` applies named simplification facts; `simp only` restricts that collection. `funext` reduces equality of functions to equality at an arbitrary argument, while `ext` exposes entries of a structured object. `refine` supplies a proof expression with remaining holes. `have` names an intermediate proposition and its proof. `exact` closes a goal with a term of the required type.

Induction is visible in the recursive structured algorithms and dynamic programs. A `calc` chain shows the successive equalities or inequalities used in a norm estimate. Arithmetic tactics have limited roles: `ring` normalizes polynomial identities, `omega` handles integer and natural-number constraints, and `linarith/nlinarith` combine supplied real inequalities. They do not supply unrecorded analytic or hardware assumptions.

1.4 Which evidence is current

The inherited sixth-block report records the compiler, source hashes, local dependency hashes, saved log hashes, axiom sets, and two rejected invalid-proof controls. The present guide replays the static identity checks on those retained bytes. It does not reconstruct compiled interfaces or simulate a Lean invocation. The exact distinction is preserved in `generated/documentation_reconciliation.json`.

The allowed transitive axiom policy is `propext`, `Classical.choice`, and `Quot.sound`, or subsets of them. These are the policy accepted by the recorded run. A passing audit excludes an admitted-proof axiom under that policy; it does not independently rebuild all imported Mathlib proofs. Manuscript correspondence remains mathematical review.

2 Navigation through the traditional volume

The traditional volume contains 124 physical PDF pages. Printed pages are its visible page labels. PDF pages count the cover and opening pages as well. The locations below are resolved from the current source map and verified against the actual PDF labels and destinations. Original result numbers remain the correspondence keys.

2.1 All 56 registered finite results

The page map locates each manuscript statement. The formal hypotheses in Section 12 describe its actual finite counterpart. These page locations do not expand the formal contract.

Item	Number	Statement	Printed	PDF
2.3	2.3	Exact evaluation	7	13
2.5	2.5	Block composition	8	14
2.6	2.6	Adjoint and tensor semantics	9	15
2.9	2.9	Contract conversion	9	15
3.1	3.1	Dense and packet application counts	10	16
3.2	3.2	Equal diagonal packets	10	16
3.4	3.4	BDLR application and storage	11	17
3.6	3.6	Streaming byte count	11	17
3.8	3.8	Roofline lower bound	12	18
3.9	3.9	Capacity threshold	12	18
3.10	3.10	Materialization cost	13	19
4.1	4.1	Equal block banded count	13	19
4.2	4.2	Sparse plus low rank application	14	20
4.3	4.3	Circulant spectral representation	14	20
4.4	4.4	Toeplitz circulant embedding	15	21
4.5	4.5	Kronecker modewise application	15	21
4.6	4.6	Equal tensor factors	15	21
4.7	4.7	Walsh Hadamard recursion	16	22
5.1	5.1	Exact BDLR product	16	22
5.2	5.2	Rank growth in a product chain	17	23
5.3	5.3	Woodbury inversion	18	24
5.4	5.4	Inverse norm bound	18	24
5.6	5.6	Small core recompression	18	24
5.8	5.8	Product chain perturbation	19	25
5.9	5.9	Certified repeated recompression	20	26
6.1	6.1	Path support of powers	21	27
6.2	6.2	Polynomial support	21	27
6.3	6.3	Acyclic transfer nilpotence	21	27
6.5	6.5	Level schedule for a triangular solve	22	28
6.6	6.6	Sequential depth	22	28
6.7	6.7	Inverse support lies in the transitive closure	23	29
6.8	6.8	Schur support inclusion	23	29
7.2	7.2	Invariant packet Krylov decomposition	24	30
7.3	7.3	Approximate packet Krylov bound	24	30
7.4	7.4	Block Jacobi condition bound	25	31
7.6	7.6	Certified packet preconditioning	25	31
8.1	8.1	Packet perturbation certificate	26	32
8.3	8.3	Mixed precision product chain	27	33
8.4	8.4	Continuous precision allocation	27	33

Item	Number	Statement	Printed	PDF
8.5	8.5	Clipped precision allocation	28	34
8.7	8.7	Exact discrete precision program	29	35
9.2	9.2	Exact block discovery certificate	30	36
9.4	9.4	Exact Lagrangian packet compiler	31	37
9.5	9.5	Shared compilation for an operator family	31	37
9.6	9.6	Monotone Lagrangian frontier	32	38
9.7	9.7	Supported Pareto points	32	38
9.8	9.8	Quantized budget dynamic program	33	39
9.9	9.9	Pseudo polynomial complexity	33	39
9.11	9.11	Compiler stability	33	39
9.12	9.12	Local recompilation	34	40
10.2	10.2	Launch reduction by shape	35	41
10.3	10.3	Acyclic schedule depth	35	41
10.4	10.4	Amortization threshold	35	41
10.5	10.5	Graph replay threshold	35	41
10.6	10.6	Materialization threshold	36	42
11.3	11.3	Fair matching is an equivalence relation	36	42

2.2 The four separately supplied classical inputs

These inputs are tracked separately from the 56 finite results. Their exact boundaries are discussed in the replay chapter.

Item	Number	Classical input	Printed	PDF
5.5	5.5	Eckart Young Mirsky theorem	18	24
7.5	7.5	Preconditioned conjugate gradient convergence	25	31
8.2	8.2	Floating point matrix product model	27	33
9.1	9.1	Simultaneous spectral decomposition	30	36

2.3 The open research boundary

The traditional boundary discussion is on printed page 42 (PDF page 48). It retains the three broader targets: discovery of profitable structure for arbitrary operators, controlled compilation for overlapping decompositions and nonlocal costs, and a communication theorem for a specified packet memory architecture. Their finite special cases do not settle those broader targets.

3 Exact packet semantics

3.1 The reused projector family

The unchanged `QuantumPacketCore` foundation uses a ring `R` and finite packet type `I`. The projectors multiply orthogonally and sum to one. Their block $P_b A P_a$ uses the target index first. This algebraic model already covers noncommutative operator multiplication.

Source: `formalization/QuantumPacketCore.lean`.

Exact lines 22–25. Excerpt `packet_family`. SHA-256 in Section 15.

```
structure PacketFamily (R I : Type*) [Ring R] [Fintype I] [DecidableEq I] where
  proj : I → R
  mul_proj : ∀ i j, proj i * proj j = if i = j then proj i else 0
  sum_proj : ∑ i, proj i = 1
```

Proof reading. The structure fields are assumptions defining a packet family. Idempotence is derived by setting the two indices equal; reconstruction follows by expanding the two sums of projectors. A finite projector family is supplied to later theorems, so discovery of an appropriate family is a separate mathematical or application task.

3.2 Deleting exact zero blocks

`activeBlocks P A` filters the finite set of ordered packet pairs by the predicate that the actual block is unequal to zero. `evalActive` sums only those surviving blocks.

Source: `formalization/PacketIR.lean`.

Exact lines 41–49. Excerpt `active_evaluation`. SHA-256 in Section 15.

```
theorem exact_active_evaluation (A : R) : evalActive P A = A := by
  classical
  have term (e : I × I) :
      (if P.block A e.1 e.2 ≠ 0 then P.block A e.1 e.2 else 0) =
      P.block A e.1 e.2 := by
    by_cases h : P.block A e.1 e.2 = 0 <;> simp [h]
  simp only [evalActive, activeBlocks, Finset.sum_filter]
  simp_rw [term]
  simpa only [Fintype.sum_prod_type] using P.reconstruct A
```

Proof reading. The local lemma `term` checks two cases: a block equals zero or it does not. In either case, replacing it by an if-expression preserves the value. The proof expands the filtered sum, rewrites every term by this lemma, and applies the foundation's reconstruction theorem. No thresholded support is used.

Source: `formalization/PacketIR.lean`.

Exact lines 145–149. Excerpt `batch`. SHA-256 in Section 15.

```
theorem exact_matrix_batch {C : Type*} (P : PacketFamily (Matrix X X K) I)
    (A : Matrix X X K) (Z : Matrix X C K) :
    A * Z = ∑ e ∈ activeBlocks P A, (P.block A e.1 e.2) * Z := by
  rw [← Matrix.sum_mul]
  exact congrArg (fun a : Matrix X X K => a * Z) (exact_active_evaluation P A).symm
```

Proof reading. A rectangular right-hand-side matrix `Z` may have any column type `C`. Distribution of matrix multiplication over the finite sum turns exact operator reconstruction into equality of the whole output batch. The result concerns actual matrix multiplication, rather than a Boolean sparsity specification.

3.3 Support composition and cancellation

Source: `formalization/PacketIR.lean`.

Exact lines 63–77. Excerpt `support_inclusion`. SHA-256 in Section 15.

```
theorem product_support_inclusion (A B : R) (b a : I)
    (h : P.block (A * B) b a ≠ 0) :
    ∃ c, P.block A b c ≠ 0 ∧ P.block B c a ≠ 0 := by
  classical
  by_contra no
  apply h
  rw [P.compose_blocks]
  apply Finset.sum_eq_zero
  intro c _
  by_cases hc : P.block A b c = 0
  · simp [hc]
  · have hd : P.block B c a = 0 := by
      by_contra hd
      exact no ⟨c, hc, hd⟩
    simp [hd]
```

Proof reading. The premise says the product block is nonzero. Assume no intermediate packet has both nonzero constituent blocks. For each intermediate index, either the first block is zero or the assumption forces the second to be zero. Every summand in the composition formula vanishes, contradicting the premise. This proves an inclusion of support relations. It supplies no converse: nonzero paths can cancel. The separate `BoundaryChecks.paths_can_cancel` declaration records a concrete control.

3.4 An adjoint needs the right projector hypothesis

Source: `formalization/PacketIR.lean`.

Exact lines 95–97. Excerpt `adjoint`. SHA-256 in Section 15.

```
theorem exact_adjoint_block (A : R) (b a : I) :
    star (P.block A b a) = P.block (star A) a b :=
  P.adjoint_block hstar A b a
```

Contract. The surrounding section has a star ring and an explicit `hstar : ∀ a, star (P.proj a) = P.proj a`.

Proof reading. The source calls the foundation's adjoint identity with that self-adjointness proof. Adjoint reversal swaps target and source. Orthogonal idempotence in an arbitrary ring alone does not establish self-adjointness. Tensor semantics are separately proved with actual Kronecker products; their address types are pairs of packet indices.

3.5 The Python prototype is a different object

The historical Python prototype's support accessor returns stored block keys, which may include explicitly stored zeros. The formal `activeBlocks` set instead tests exact nonzero matrix blocks. The two objects agree only when the stored representation has already removed all zero blocks and realizes the same matrices. No automatic equivalence between the prototype and the Lean representation is asserted here.

4 Representation algebra and declared cost models

4.1 Why a count theorem can have a short proof

`PacketCostModels` defines finite address and event types before counting them. The block widths are natural numbers; a diagonal block has a pair of within-block indices. A thin factor has one ambient and one rank index. An application event additionally has a right-hand-side index and a Boolean distinguishing multiplication from accumulation. The Boolean gives the declared two-scalar-operation convention.

Source: `formalization/PacketCostModels.lean`.

Exact lines 68–73. Excerpt `bdlr_count`. SHA-256 in Section 15.

```
theorem bdlr_application_count {I : Type*} [Fintype I]
    (width : I → ℕ) (n r q : ℕ) :
    Fintype.card (BDLROperation I width n r q) =
      2 * q * ((∑ i, width i ^ 2) + 2 * n * r) := by
  simp [BDLROperation, pow_two]
  ring
```

Proof reading. The type `BDLROperation` is a sum of the diagonal-action, right-factor, and left-factor event types. `simp` evaluates cardinalities of finite products and sums; `ring` rearranges the resulting polynomial. The count includes the accumulation into the diagonal result through the declared third kernel. It does not establish how many GPU instructions run.

The storage count concerns stored scalar values; sparse index metadata and workspace are outside the corresponding models. The materialization theorem counts an explicit dense low-rank multiplication followed by one addition per output entry. Its quadratic remainder is exactly n^2. This is a count for that declared construction, rather than a lower bound for every conceivable implementation.

Source: `formalization/BandedCount.lean`.

Exact lines 73–77. Excerpt `band_count`. SHA-256 in Section 15.

```
theorem banded_block_count (p w : ℕ) (hw : w < p) :
    Fintype.card (BandedAddress p w) = p*(2*w+1)-w*(w+1) := by
  rw [bandCount_card]
  have he := bandCount_balance p w hw
  omega
```

Proof reading. The banded-count development counts the actual subtype of pairs in `Fin p × Fin p` whose distance is at most `w`. The premise `w < p` controls the two triangular boundary losses. The preceding recurrence and finite-sum lemmas establish the cardinality; this endpoint presents the closed formula. Replacing p by n/b additionally requires positive block width and exact divisibility.

4.2 Capacity inequalities are hypotheses

Source: `formalization/PacketCostModels.lean`.

Exact lines 118–124. Excerpt `roofline`. SHA-256 in Section 15.

```
theorem roofline_time_bound (F B P W T : ℝ) (hP : 0 < P) (hW : 0 < W)
    (hF : F ≤ P * T) (hB : B ≤ W * T) : max (F / P) (B / W) ≤ T := by
  apply max_le
  · apply (div_le_iff₀ hP).2
    nlinarith
  · apply (div_le_iff₀ hW).2
    nlinarith
```

Proof reading. Positive peak arithmetic and bandwidth rates make division preserve order. The assumed inequalities `F ≤ P * T` and `B ≤ W * T` yield the two lower bounds on time, then `max_le` combines them. Lean does not measure a device rate or validate that an experimental execution satisfies these capacity assumptions.

4.3 An exact low-rank product

Contract. `LowRankAlgebra` works with actual finite matrices over a field with an involution. The left and right factors have explicit possibly different width types.

Source: `formalization/LowRankAlgebra.lean`.

Exact lines 57–65. Excerpt `bdlr_product`. SHA-256 in Section 15.

```
theorem bdlr_product (DA DB : Matrix X X K) (UA VA : Matrix X RA K)
    (UB VB : Matrix X RB K) :
    (DA + UA * VA.conjTranspose) * (DB + UB * VB.conjTranspose) =
      DA * DB + bdlrProductLeft DA UA VA UB * (bdlrProductRight DB VA VB).conjTranspose := by
  simp only [bdlrProductLeft, bdlrProductRight,
    Matrix.conjTranspose_fromCols_eq_fromRows_conjTranspose,
    Matrix.fromCols_mul_fromRows, Matrix.conjTranspose_mul, Matrix.conjTranspose_conjTranspose,
    Matrix.add_mul, Matrix.mul_add, Matrix.mul_assoc]
  abel
```

Proof reading. The constructors `bdlrProductLeft` and `bdlrProductRight` concatenate columns, so the new correction width is the sum of the two input widths. The proof expands the concatenated product, reverses conjugate-transposes of products, distributes multiplication, and normalizes additive terms. No rank identity is assumed in this algebraic equality. A separate theorem bounds the correction rank by its column count. Packet diagonality of the base product is also proved separately.

Source: `formalization/LowRankChains.lean`.

Exact lines 56–66. Excerpt `chain_factors`. SHA-256 in Section 15.

```
theorem chainFactors_correct (n : ℕ) :
    chainProduct (fun j => D j + U j * (V j).conjTranspose) n =
      chainProduct D n + (chainFactors r D U V n).1 * (chainFactors r D U V n).2.conjTranspose :=
      ↪ by
```

```
  induction n with
  | zero =>
    ext i j
    simp [chainProduct, chainFactors, ChainColumns]
  | succ n ih =>
    rw [chainProduct, ih]
    exact bdlr_product (D n) (chainProduct D n) (U n) (V n)
      (chainFactors r D U V n).1 (chainFactors r D U V n).2
```

Proof reading. The empty chain is the identity and has an empty correction-column type. For the successor step, rewrite the current chain using the induction hypothesis and invoke the actual two-factor identity. The source thereby constructs factors for every length; the subsequent rank theorem bounds their width by the sum of input widths, including zero widths.

4.4 Woodbury exposes both invertibility assumptions

Source: `formalization/LowRankAlgebra.lean`.

Exact lines 89–94. Excerpt `woodbury`. SHA-256 in Section 15.

```
theorem bdlr_woodbury (D : Matrix X X K) (U V : Matrix X Rk K)
    (hD : IsUnit D) (hS : IsUnit (1 + V.conjTranspose * D⁻¹ * U)) :
    (D + U * V.conjTranspose)⁻¹ =
      D⁻¹ - D⁻¹ * U * (1 + V.conjTranspose * D⁻¹ * U)⁻¹ * V.conjTranspose * D⁻¹ := by
  simpa using Matrix.add_mul_mul_inv_eq_sub D U (1 : Matrix Rk Rk K)
    V.conjTranspose hD isUnit_one (by simpa using hS)
```

Proof reading. `hD` proves the base matrix is a unit. `hS` proves the small update matrix is a unit. The endpoint specializes Mathlib's Woodbury identity, with the middle rank-space matrix equal to one. The update's invertibility, batch solve formula, and L2 operator norm bound are independent declarations in the same module. A singular base or singular update matrix cannot be passed through the displayed theorem simply because an inverse notation exists.

5 Structured algorithms carry their coordinate conventions

5.1 Vectorization and recursive tensor contraction

Source: `formalization/StructuredTensor.lean`.

Exact lines 19–22. Excerpt `kronecker`. SHA-256 in Section 15.

```
theorem kronecker_modewise {M N : Type*} [Fintype M] [Fintype N]
    (A : Matrix M M ℂ) (B : Matrix N N ℂ) (X : Matrix N M ℂ) :
    (A ⊗ₖ B) *ᵥ Matrix.vec X = Matrix.vec (B * X * Aᵀ) := by
  exact Matrix.kronecker_mulVec_vec B X A
```

Proof reading. The right factor in the matrix expression is $\mathtt{A}^{\mathrm{T}}$, the ordinary transpose. It is not the conjugate transpose. The argument order reflects Mathlib's column-major `Matrix.vec` convention: $(A \otimes B)\,\mathrm{vec}(X) = \mathrm{vec}(BXA^T)$. The one-line proof invokes that exact library identity.

`TensorIndex d 0` is `Unit`; the successor index is `Fin d × TensorIndex d k`. This makes the tensor dimension d^k visible without an implicit flattening convention. `tensorMatrix` builds iterated actual Kronecker matrices; `tensorApply` contracts trailing slices before the first mode.

Source: `formalization/StructuredTensor.lean`.

Exact lines 73–96. Excerpt `tensor_apply`. SHA-256 in Section 15.

```
theorem tensor_apply_correct (d k : ℕ) (A : Fin k → Matrix (Fin d) (Fin d) ℂ)
    (x : TensorIndex d k → ℂ) :
    tensorApply d k A x = tensorMatrix d k A *ᵥ x := by
  induction k with
  | zero =>
    funext i
    cases i
    change x PUnit.unit = ∑ i : Unit, (1 : ℂ) * x i
    simp only [one_mul]
    symm
    apply Finset.sum_eq_single PUnit.unit
    · intro b _ h
      cases b
      exact (h rfl).elim
    · intro h
      exact (h (Finset.mem_univ _)).elim
  | succ k ih =>
    funext p
    change (∑ j, A 0 p.1 j * tensorApply d k (fun i => A i.succ) (fun u => x (j,u)) p.2) = _
    simp_rw [ih]
    change (∑ j, A 0 p.1 j * (∑ u, tensorMatrix d k (fun i => A i.succ) p.2 u * x (j,u))) =
      ∑ i : Fin d × TensorIndex d k,
        (A 0 p.1 i.1 * tensorMatrix d k (fun j => A j.succ) p.2 i.2) * x i
    simp only [Fintype.sum_prod_type, Finset.mul_sum, mul_assoc]
```

Proof reading. The zero-mode case reduces to the one-coordinate identity. In the successor case, expand one contraction, rewrite trailing contractions using the induction hypothesis, and express the full matrix sum over a product type as two nested sums. Associativity and distribution align the two

formulas. The event type counted later follows this recursive contraction, yielding $2kd^{k+1}$ operations in the declared model and kd^2 factor values. Cases $k = 0$ and $d = 0$ are represented by their actual index types.

5.2 The Walsh butterfly proves an implemented recursion

Source: `formalization/WalshTransform.lean`.

Exact lines 60–84. Excerpt `walsh_apply`. SHA-256 in Section 15.

```
theorem walsh_apply_correct (k : ℕ) (x : TensorIndex 2 k → ℂ) :
    walshApply k x = walshMatrix k *ᵥ x := by
  induction k with
  | zero =>
    funext i
    cases i
    change x PUnit.unit = ∑ i : Unit, (1:ℂ)*x i
    simp only [one_mul]
    symm
    apply Finset.sum_eq_single PUnit.unit
    · intro b _ h
      cases b
      exact (h rfl).elim
    · intro h
      exact (h (Finset.mem_univ _)).elim
  | succ k ih =>
    funext p
    rcases p with ⟨i,u⟩
    change (if i = 0 then walshApply k (fun v => x (0,v)+x (1,v)) u
      else walshApply k (fun v => x (0,v)-x (1,v)) u) =
        ∑ v : Fin 2 × TensorIndex 2 k, (walshTwo i v.1 * walshMatrix k u v.2) * x v
    simp_rw [ih]
    fin_cases i <;>
      simp [walshTwo, Matrix.mulVec, dotProduct, Fintype.sum_prod_type,
        Fin.sum_univ_two, mul_add, Finset.sum_add_distrib, sub_eq_add_neg]
```

Proof reading. The recursively constructed `walshMatrix` matrix is paired with the actual sum/difference routine `walshApply`. The proof follows the same split of tensor coordinates and expands the two butterfly branches. Its equality is an algorithm-correctness statement. The separate operation theorem counts additions and subtractions, giving $N \log_2 N$ for $N = 2^k$, including $N = 1$. Orthogonality uses the ordinary transpose of these real-valued complex matrices.

5.3 Toeplitz padding is concrete

Contract. The matrix and generator definitions assume `[NeZero n]`, so the length is positive. The zero-padding helper itself is meaningful without that instance.

Source: `formalization/ToeplitzEmbedding.lean`.

Exact lines 54–58. Excerpt `toeplitz_apply`. SHA-256 in Section 15.

```
theorem toeplitz_embedding_action (c r x : Fin n → ℂ) (i : Fin n) :
```

```
    (Matrix.circulant (toeplitzGenerator c r) *ᵥ zeroPad x) (Fin.castAdd n i) =
      (toeplitzMatrix c r *ᵥ x) i := by
  simp only [Matrix.mulVec, dotProduct, Fin.sum_univ_add, zero_pad_left, zero_pad_right,
    mul_zero, Finset.sum_const_zero, add_zero, toeplitz_circulant_block]
```

Proof reading. The preceding theorem identifies the actual top-left Toeplitz block of the length-$n + n$ circulant. This application theorem expands the zero-padded vector with `Fin.addCases`; terms in the right half vanish, leaving precisely the Toeplitz action. The generator reverses the row tail in its second half. Its first entry uses the column datum `c 0`; `r 0` is irrelevant. No symmetry or Hermiticity is assumed.

5.4 Fourier diagonalization is exact, with no FFT timing claim

Source: `formalization/CirculantFourier.lean`.

Exact lines 150–153. Excerpt `fourier`. SHA-256 in Section 15.

```
theorem circulant_spectral_representation (c : ZMod n → ℂ) :
    Matrix.circulant c = (fourierMatrix n)ᴴ *
      Matrix.diagonal ((Real.sqrt (n:ℝ) : ℂ) • (fourierMatrix n *ᵥ c)) * fourierMatrix n := by
  rw [normalized_fourier_eigenvalues, circulant_spectral_dft]
```

Proof reading. For a positive length n, the module defines the actual circulant and the unitary-normalized Fourier matrix on `ZMod n`. The proof rewrites the DFT diagonalization using the normalization identities and actual Fourier unitarity. Supporting declarations establish the complex exponential entry formula, eigenvalues, and both unitarity products. The final application formula uses two transforms. This establishes the represented algebraic action; an FFT algorithm, its execution cost, and hardware timing remain distinct tasks.

6 Transfer paths, inverses, and acyclic solves

6.1 A path expansion covers length zero

Contract. `TransferSupport` uses a finite packet family in a ring. `TransferPath P A n b a` records a path of exactly n active edges, from source a to target b.

Source: `formalization/TransferSupport.lean`.

Exact lines 37–52. Excerpt `power_paths`. SHA-256 in Section 15.

```
theorem power_path_expansion (A : R) (n : ℕ) (b a : I) :
    P.block (A ^ n) b a = activePathSum P A n b a := by
  classical
  induction n generalizing b a with
  | zero =>
      by_cases h : b = a
      · subst b
        simp [activePathSum, PacketFamily.block, P.mul_proj]
      · simp [activePathSum, PacketFamily.block, P.mul_proj, h]
  | succ n ih =>
      rw [pow_succ', P.compose_blocks]
      simp only [activePathSum]
      apply Finset.sum_congr rfl
      intro c _
      rw [ih]
      by_cases h : P.block A b c = 0 <;> simp [h]
```

Proof reading. Induct on the exponent. At length zero, the identity block is a projector on the diagonal and zero off the diagonal. At a successor, use block composition and the induction hypothesis for each intermediate packet. A zero first edge contributes nothing; a nonzero first edge extends a path. The result is an exact sum of path products, so cancellation still allows its support inclusion to be strict.

Source: `formalization/TransferSupport.lean`.

Exact lines 86–90. Excerpt `nilpotent`. SHA-256 in Section 15.

```
theorem nilpotent_of_path_bound (A : R) (h : ℕ)
    (hpaths : ∀ b a, ¬ TransferPath P A (h + 1) b a) : A ^ (h + 1) = 0 := by
  apply all_blocks_zero P
  intro b a
  exact power_block_zero_of_no_path P A (h + 1) b a (hpaths b a)
```

Proof reading. If no pair has a path of the stated length, every block of that power is zero. Reconstruction then forces the whole power to be zero. The level-based version derives a path bound from a supplied strict level function; it does not assume that nonzero paths always survive in the operator.

6.2 Inverse support uses finite dimensionality

Contract. The inverse-support module has a field `K`, a `K`-algebra `R`, and `FiniteDimensional K R`. It defines the subspace of elements with zero components at every packet unreachable from the starting packet. Reachability includes zero-length paths.

Source: `formalization/InverseSupport.lean`.

Exact lines 94–100. Excerpt `inverse_support`. SHA-256 in Section 15.

```
theorem inverse_block_zero_of_no_path (A B : R) (hBA : B*A = 1) (b a : I)
    (h : ¬ Reachable P A b a) : P.block B b a = 0 := by
  have hm := invariant_space_inverse (K := K) A B hBA
    (reachableSpace (K := K) P A a) (reachableSpace_invariant P A a)
    (P.proj a) (seed_mem_reachableSpace P A a)
  have hz := hm b h
  simpa only [PacketFamily.block,mul_assoc] using hz
```

Proof reading. The source first proves that the forward multiplication by `A` preserves the reachable subspace. A left inverse makes that restricted map injective; finite dimensionality makes it surjective. The helper `invariant_space_inverse` therefore shows that multiplication by `B` preserves the same subspace. Apply it to the seed projector and read its unreachable component. The matrix specialization supplies the relevant inverse hypothesis from matrix invertibility. No inverse sparsity pattern is assumed.

6.3 Schur fill follows routes through eliminated packets

Source: `formalization/SchurSupport.lean`.

Exact lines 57–73. Excerpt `schur_support`. SHA-256 in Section 15.

```
theorem schur_support_inclusion (A : Matrix X X K) (B : Matrix X Y K)
    (H : Matrix Y Y K) (C : Matrix Y X K) (hH : IsUnit H.det) (b a : I)
    (h : P.block (A-B*H⁻¹*C) b a ≠ 0) :
    P.block A b a ≠ 0 ∨
      ∃ j k, P.proj b*B*Q.proj j ≠ 0 ∧ Reachable Q H j k ∧
        Q.proj k*C*P.proj a ≠ 0 := by
  by_cases ha : P.block A b a ≠ 0
  · exact Or.inl ha
  · right
    apply schur_correction_support P Q B H C hH b a
    intro hc
    apply h
    have hz : P.block A b a = 0 := not_ne_iff.mp ha
    change P.proj b*(A-B*H⁻¹*C)*P.proj a = 0
    rw [Matrix.mul_sub,Matrix.sub_mul]
    change P.block A b a - P.block (B*H⁻¹*C) b a = 0
    rw [hz,hc,sub_self]
```

Proof reading. The premise is a nonzero block of the actual Schur complement $A-BH^{-1}C$. If the original A-block is nonzero, the first alternative applies. Otherwise, the correction block must be nonzero. Expand it through the eliminated packet family and use inverse support: a surviving term has an incoming block, a route through H, and an outgoing block. The determinant-unit premise establishes invertibility of H. The conclusion allows fill; it does not say every route creates a nonzero fill block.

6.4 The graph level is constructed

`DAG.Graph` contains a finite edge relation and the hypothesis that no vertex lies on a nonempty directed cycle. The source proves that the ancestor set strictly grows along an edge. Its finite cardinality gives a well-founded recursion. The resulting level is one plus the maximum predecessor level; sources start at level one. The empty graph's maximum and depth are zero.

Source: `formalization/GraphSchedules.lean`.

Exact lines 150–154. Excerpt `schedule_depth`. SHA-256 in Section 15.

```
theorem optimal_depth (s : V → ℕ) (hs : G.ValidSchedule s) :
    G.depth ≤ Finset.univ.sup s := by
  classical
  exact Finset.sup_le (fun v _ => (G.level_minimal s hs v).trans
    (Finset.le_sup (Finset.mem_univ v)))
```

Proof reading. The earlier `level_minimal` theorem proves by well-founded induction that the constructed level is no later than any positive valid schedule at each vertex. Taking the finite maximum yields the displayed lower bound. Separate declarations prove the level schedule valid and its maximum attained by a path, so the construction is optimal in the unit-duration, unrestricted-parallelism model. Device resources, variable task durations, and launch overhead are outside that model.

6.5 Actual forward substitution in dependent block spaces

For each vertex v, `X v` is its block space; unequal block dimensions are allowed. `D v : X v ≃ₗ[K] X v` is an invertible linear map, and `T b a` is a linear transfer from the source space to the target space. The action sums only graph edges.

Source: `formalization/TriangularSolve.lean`.

Exact lines 38–43. Excerpt `solve_correct`. SHA-256 in Section 15.

```
theorem solve_correct (rhs : ∀ v, X v) :
    G.triangularAction X D T (G.solve X D T rhs) = rhs := by
  funext v
  unfold triangularAction
  rw [G.solve_eq X D T rhs v, LinearEquiv.apply_symm_apply]
  exact sub_add_cancel _ _
```

Proof reading. The constructed `solve` uses well-founded forward substitution, applying `(D v).symm` to the right-hand side minus previously solved transfers. Function extensionality chooses one vertex. Rewrite the solve equation, cancel the inverse linear equivalence, then add back the transfer sum. Uniqueness is a separate well-founded induction over the same graph. The staged solver is separately proved correct through each level. The theorem begins with no assumed solution or solve equation.

7 Error propagation and exact precision allocation

7.1 Local block errors are actual operators

Contract. `PacketPerturbation` works over a nontrivially normed field and a normed vector space. `A e` and `Δ e` are continuous linear maps. The source-support hypothesis is `Δ e = (Δ e).comp (P (src e))`.

Source: `formalization/PacketPerturbation.lean`.

Exact lines 35–39. Excerpt `packet_error`. SHA-256 in Section 15.

```
theorem packet_perturbation_bound (P : I → X →L[K] X) (src : E → I)
    (A Δ : E → X →L[K] X) (hΔ : ∀ e, Δ e = (Δ e).comp (P (src e))) (x : X) :
    ‖((∑ e, (A e + Δ e)) - (∑ e, A e)) x‖ ≤ ∑ e, ‖Δ e‖ * ‖P (src e) x‖ := by
  rw [packet_error_action_identity P src A Δ hΔ]
  exact (norm_sum_le _ _).trans (Finset.sum_le_sum (fun e _ => (Δ e).le_opNorm _))
```

Proof reading. Rewrite the total action error as a sum of source-supported error actions. The triangle inequality bounds the norm of that sum. Each map's operator-norm estimate supplies $\|\Delta_e\|\|P_{\mathrm{src}(e)}x\|$. The exact support of the A_e terms is not required for this inequality. A later theorem assumes local relative budgets; another adds contractive source projectors for a uniform certificate. Neither conclusion certifies IEEE arithmetic or a specific CUDA kernel.

7.2 Products preserve multiplication order

Contract. `ProductChainError` uses a normed ring. The product-chain endpoint additionally assumes `NormOneClass R`, so the empty product has norm one. Products are built by left multiplication at each successor step.

Source: `formalization/ProductChainError.lean`.

Exact lines 129–145. Excerpt `chain_error`. SHA-256 in Section 15.

```
theorem product_chain_error_bound [NormOneClass R] (A B : ℕ → R) (δ : ℕ → ℝ)
    (hδ : ∀ j, ‖B j - A j‖ ≤ δ j) (n : ℕ) :
    ‖chainProduct B n - chainProduct A n‖ ≤
      ∑ j ∈ Finset.range n,
        (∏ i ∈ Finset.Ico (j + 1) n, ‖B i‖) *
          (δ j * ∏ i ∈ Finset.range j, ‖A i‖) := by
  rw [← propagatedError_eq_sum]
  apply norm_recurrence_bound (fun k => chainProduct B k - chainProduct A k)
    (fun k => ‖B k‖) (fun k => δ k * ∏ i ∈ Finset.range k, ‖A i‖)
    (by simp [chainProduct]) (fun k => norm_nonneg _)
  intro k
  rw [product_discrepancy_step]
  apply (norm_add_le _ _).trans
  apply add_le_add (norm_mul_le _ _)
  apply (norm_mul_le _ _).trans
  exact mul_le_mul (hδ k) (chainProduct_norm_bound A k) (norm_nonneg _)
    ((norm_nonneg _).trans (hδ k))
```

Proof reading. The discrepancy at a successor splits into the perturbed factor times the previous discrepancy, plus the new local error times the exact prefix. The generic norm recurrence propagates these two contributions. Its unrolled finite sum uses perturbed suffix norms and exact prefix norms in the displayed order. Submultiplicativity and the local error bound close each successor estimate. The empty chain has zero discrepancy. No commutativity is used.

Source: `formalization/ProductChainError.lean`.

Exact lines 100–111. Excerpt `recompression_error`. SHA-256 in Section 15.

```
theorem repeated_recompression_bound (A E : ℕ → R) (δ : ℕ → ℝ)
    (hδ : ∀ j, ‖E j‖ ≤ δ j) (n : ℕ) :
    ‖recompressedProduct A E n - chainProduct A n‖ ≤
      ∑ j ∈ Finset.range n, (∏ i ∈ Finset.Ico (j + 1) n, ‖A i‖) * δ j := by
  rw [← propagatedError_eq_sum]
  apply norm_recurrence_bound
      (fun k => recompressedProduct A E k - chainProduct A k)
      (fun k => ‖A k‖) δ (by simp [chainProduct, recompressedProduct])
      (fun k => norm_nonneg _)
  intro k
  rw [recompression_discrepancy_step]
  exact (norm_add_le _ _).trans (add_le_add (norm_mul_le _ _) (hδ k))
```

Proof reading. Here each iterate has an additive residual `E j` after multiplication by `A j`. The successor error is A_j times the old error plus E_j. The same recurrence yields the product of subsequent operator norms times each certified residual budget. This theorem uses actual residuals with verified bounds; it does not itself choose the recompression algorithm or infer its floating-point error.

7.3 The continuous minimizer is global

The finite edge type is explicitly nonempty. Every sensitivity s_e, cost c_e, and budget is positive. `AllocationFeasible` requires positive tolerances and exact weighted budget equality. The objective is $\sum_e c_e \log(1/\varepsilon_e)$, and the candidate is

$$\varepsilon_e^* = \frac{\mathrm{budget}\, c_e}{s_e \sum_f c_f}.$$

Source: `formalization/ContinuousPrecision.lean`.

Exact lines 80–97. Excerpt `continuous_precision`. SHA-256 in Section 15.

```
theorem precision_allocation_optimal [Nonempty E] (s c : E → ℝ) (budget : ℝ)
    (hs : ∀ e, 0 < s e) (hc : ∀ e, 0 < c e) (hb : 0 < budget)
    (eps : E → ℝ) (heps : AllocationFeasible s budget eps) :
    precisionObjective c (optimalTolerance s c budget) ≤ precisionObjective c eps := by
  have ho := optimal_tolerance_positive s c budget hs hc hb
  have hbalance := relative_budget_identity s c eps budget hs hc hb heps.2
  have hlog : (∑ e, c e * Real.log (eps e / optimalTolerance s c budget e)) ≤ 0 := by
    calc
      _ ≤ ∑ e, c e * (eps e / optimalTolerance s c budget e - 1) := by
        apply Finset.sum_le_sum
        intro e _
        exact mul_le_mul_of_nonneg_left
```

```
          (Real.log_le_sub_one_of_pos (div_pos (heps.1 e) (ho e))) (le_of_lt (hc e))
      _ = 0 := by
        simp_rw [mul_sub, mul_one]
        rw [Finset.sum_sub_distrib, hbalance, sub_self]
  have hd := precision_objective_difference c eps (optimalTolerance s c budget) heps.1 ho
  linarith
```

Proof reading. First prove positivity of the candidate and an exact weighted-ratio balance. Compare each competitor ratio $\varepsilon_e/\varepsilon_e^*$ with the inequality $\log x \le x - 1$. Multiply by nonnegative costs and sum. The balance makes the upper sum zero, and the objective-difference identity changes its sign to obtain the global lower bound. The strict logarithmic inequality at a differing edge gives strictness and then uniqueness in subsequent declarations. The proof uses no stationary-point optimality assumption.

7.4 Caps separate the allocation from its multiplier

`clippedTolerance s c m t e` is $\min(m_e, tc_e/s_e)$. The cap vector is positive; the budget is positive and no larger than the total weighted cap. A continuous monotone total-allocation function yields an attaining positive scale.

Source: `formalization/ClippedPrecision.lean`.

Exact lines 211–234. Excerpt `clipped_precision`. SHA-256 in Section 15.

```
theorem clipped_unique_minimizer [Nonempty E] (s c m : E → ℝ) (budget : ℝ)
    (hs : ∀ e, 0 < s e) (hc : ∀ e, 0 < c e) (hm : ∀ e, 0 < m e)
    (hb : 0 < budget) (hcap : budget ≤ ∑ e, s e * m e) :
    ∃ t, 0 < t ∧ ClippedFeasible s m budget (clippedTolerance s c m t) ∧
      ∀ eps, ClippedFeasible s m budget eps →
        ((∀ other, ClippedFeasible s m budget other →
            precisionObjective c eps ≤ precisionObjective c other) ↔
          eps = clippedTolerance s c m t) := by
  obtain ⟨t, ht, heq⟩ := clipped_scale_exists s c m budget hs hc hm hb hcap
  have hf : ClippedFeasible s m budget (clippedTolerance s c m t) :=
    ⟨⟨clipped_positive s c m t hs hc hm ht, heq⟩, fun _ => min_le_left _ _⟩
  refine ⟨t, ht, hf, ?_⟩
  intro eps heps
  constructor
  · intro hmin
    by_contra hne
    have hl := clipped_allocation_strict s c m budget t hs hc hm ht heq eps heps hne
    have hu := hmin _ hf
    linarith
  · intro h
    subst eps
    exact fun other hother => clipped_allocation_optimal s c m budget t hs hc hm ht heq other
     ↪ hother

/- The parameter is the reciprocal of the manuscript's multiplier. -/
```

Proof reading. Obtain an attaining scale from `clipped_scale_exists`. Its positivity and budget identity establish feasibility; the minimum supplies the cap inequalities. A distinct feasible vector has strictly larger objective by the supporting logarithmic inequality. This rules out every competing minimizer

and gives the equivalence with the constructed tolerance vector. At total-cap budget, the tolerance vector remains unique but the scale, and hence the reciprocal multiplier, need not be unique. The endpoint-nonuniqueness declaration records that distinction.

7.5 Discrete choices guard natural subtraction

Each edge has a finite list of pairs: quantized error weight in $\mathbb{N}$ and exact ordered-field cost. `Assignment edges` independently chooses one listed option per edge. An empty option list is infeasible; an empty edge list has the empty assignment.

Source: `formalization/DiscretePrecision.lean`.

Exact lines 38–42. Excerpt `discrete_recurrence`. SHA-256 in Section 15.

```
def discreteTable : List (List (ℕ × R)) → ℕ → Option R
  | [], b => singletonTable 0 0 b
  | opts :: edges, b =>
    bestOver (fun o => if o.1 ≤ b then
      add (discreteTable edges (b - o.1)) (some o.2) else none) opts
```

Proof reading. The recursive table records exact total weight. Its guard checks that the current option fits before using `b - o.1`. Natural subtraction truncates at zero, so an unguarded recurrence could accept an overweight option through a zero residual. The development proves exact-weight semantics by induction on the edge list, then scans weights up to the budget. Zero-error options are allowed.

Source: `formalization/DiscretePrecision.lean`.

Exact lines 110–121. Excerpt `discrete_endpoint`. SHA-256 in Section 15.

```
theorem discreteBudget_some_iff (edges : List (List (ℕ × R))) (B : ℕ) (v : R) :
    discreteBudget edges B = some v ↔
      (∃ a : Assignment edges, a.weight ≤ B ∧ a.cost = v) ∧
        ∀ a : Assignment edges, a.weight ≤ B → v ≤ a.cost := by
  rw [minimum_some_iff _ _ (discrete_precision_program edges B)]
  constructor
  · intro h
    exact ⟨h.1, fun a ha => h.2 _ ⟨a, ha, rfl⟩⟩
  · intro h
    refine ⟨h.1, ?_⟩
    rintro z ⟨a, ha, rfl⟩
    exact h.2 a ha
```

Proof reading. The returned `some v` is equivalent to an attained feasible assignment cost together with a lower bound for every feasible assignment. The helper's `IsMinimum` semantics guarantees both parts; the proof translates its value predicate back to an actual assignment. The companion `discreteBudget_none_iff` equates `none` with absence of a complete feasible assignment. Finite sentinel costs are never substituted for infeasibility.

8 Exact packet compilation and cached refinement

8.1 An independently defined feasible set

The tree label type `α` contains application-specific packet choices. Internal nodes have `arity + 1` children, so this constructor has at least one child. Costs live in a linearly ordered field with its compatible strict ring order; this permits exact rational execution as well as real-valued models.

Source: `formalization/ExactTreeCompiler.lean`.

Exact lines 21–23. Excerpt `candidate_tree`. SHA-256 in Section 15.

```
inductive CandidateTree (α : Type*) where
  | leaf (label : α)
  | node (label : α) (arity : ℕ) (children : Fin (arity + 1) → CandidateTree α)
```

Source: `formalization/ExactTreeCompiler.lean`.

Exact lines 33–36. Excerpt `cut`. SHA-256 in Section 15.

```
inductive Cut : CandidateTree α → Type _ where
  | keep (t : CandidateTree α) : Cut t
  | split {a : α} {n : ℕ} {ts : Fin (n + 1) → CandidateTree α}
      (cs : ∀ i, Cut (ts i)) : Cut (.node a n ts)
```

Proof reading. `Cut t` is indexed by the tree it prunes. Keeping a tree gives a legal cut; splitting an internal node requires one legal cut of every child. This type is defined before the optimization recurrence. Therefore correctness has an independent feasible set against which to compare the recurrence.

Source: `formalization/ExactTreeCompiler.lean`.

Exact lines 45–47. Excerpt `tree_recurrence`. SHA-256 in Section 15.

```
def optimalValue (keep overhead : α → R) : CandidateTree α → R
  | .leaf a => keep a
  | .node a _ ts => min (keep a) (overhead a + ∑ i, optimalValue keep overhead (ts i))
```

Proof reading. A leaf must be kept. At an internal node, compare the node's keep cost against the split overhead plus child optimal values. The proof `optimalValue_le_cut` proceeds by induction on the tree and cases on a legal cut. Keeping selects the left minimum bound; splitting combines the right minimum bound with all child bounds.

Source: `formalization/ExactTreeCompiler.lean`.

Exact lines 69–76. Excerpt `compile_attains`. SHA-256 in Section 15.

```
theorem compile_attains (keep overhead : α → R) (t : CandidateTree α) :
    cutValue keep overhead (compile keep overhead t) = optimalValue keep overhead t := by
  induction t with
  | leaf a => rfl
  | node a n ts ih =>
    by_cases h : keep a ≤ overhead a + ∑ i, optimalValue keep overhead (ts i)
    · simp [compile, h, cutValue, root, optimalValue]
    · simp [compile, h, cutValue, ih, optimalValue, min_eq_right (le_of_not_ge h)]
```

Proof reading. The actual `compile` returns a cut and chooses keep at ties. Induction proves its objective equals the recurrence. If the keep comparison holds, simplifying the conditional proves the keep case. Otherwise the induction hypotheses rewrite every compiled child cost and the right branch realizes the minimum.

Source: `formalization/ExactTreeCompiler.lean`.

Exact lines 79–83. Excerpt `tree_exact`. SHA-256 in Section 15.

```
theorem exact_tree_compiler (keep overhead : α → R) (t : CandidateTree α) :
    (∀ c : Cut t, optimalValue keep overhead t ≤ cutValue keep overhead c) ∧
    ∃ c : Cut t, cutValue keep overhead c = optimalValue keep overhead t := by
  exact ⟨optimalValue_le_cut keep overhead t, compile keep overhead t,
    compile_attains keep overhead t⟩
```

Proof reading. The final conjunction pairs the universal lower bound with the constructed attaining cut. Optimality is thus stronger than a recurrence-value identity. Lagrangian compilation later specializes keep scores to cost plus multiplier times error. Weighted families combine their actual cut objectives before applying this same theorem.

8.2 The Pareto statement has a boundary

The monotone-frontier theorem compares two genuinely optimal cuts at distinct nonnegative multipliers; adding the two optimality inequalities bounds their errors in opposite order to their costs. A positive-multiplier optimum is Pareto efficient. The source also includes `unsupported_pareto_example`: weighted sums need not select every efficient point of a discrete feasible set. The exact budget compiler therefore has a separate role.

8.3 An exact quantized table

Source: `formalization/QuantizedTreeCompiler.lean`.

Exact lines 27–33. Excerpt `budget_recurrence`. SHA-256 in Section 15.

```
def budgetTable (κ : α → R) (w : α → ℕ) (σ : α → R) :
    CandidateTree α → ℕ → Option R
  | .leaf a, b => BudgetDP.singletonTable (w a) (κ a) b
  | .node a _ ts, b =>
    BudgetDP.choose (BudgetDP.singletonTable (w a) (κ a) b)
      (BudgetDP.add (some (σ a))
        (BudgetDP.combine (fun i => budgetTable κ w σ (ts i)) b))
```

Proof reading. Each keep option occupies its exact weight, while splitting convolves the child exact-weight tables and adds overhead. `BudgetDP.choose` compares two attainable options. The min-plus infrastructure proves that sums and finite choices preserve the independent `IsMinimum` predicate. Induction on the actual candidate tree therefore connects every table entry to legal cuts of exactly that weight.

Source: `formalization/QuantizedTreeCompiler.lean`.

Exact lines 107–119. Excerpt `budget_endpoint`. SHA-256 in Section 15.

```
theorem compileBudget_some_iff (κ : α → R) (w : α → ℕ) (σ : α → R)
    (t : CandidateTree α) (B : ℕ) (v : R) :
    compileBudget κ w σ t B = some v ↔
      (∃ c : Cut t, cutWeight w c ≤ B ∧ cutValue κ σ c = v) ∧
      ∀ c : Cut t, cutWeight w c ≤ B → v ≤ cutValue κ σ c := by
  rw [BudgetDP.minimum_some_iff _ _ (quantized_budget_compiler κ w σ t B)]
  constructor
  · intro h
    exact ⟨h.1, fun c hc => h.2 _ ⟨c, hc, rfl⟩⟩
  · intro h
    refine ⟨h.1, ?_⟩
    rintro z ⟨c, hc, rfl⟩
    exact h.2 c hc
```

Proof reading. The budget scan ranges over weights from zero through B. The endpoint states that a finite answer is attained by a legal cut of weight at most B, and no feasible cut costs less. This is an optimum over the original tree's cuts, rather than a continuous relaxation or a weighted-sum surrogate.

Source: `formalization/QuantizedTreeCompiler.lean`.

Exact lines 122–126. Excerpt `budget_none`. SHA-256 in Section 15.

```
theorem compileBudget_none_iff (κ : α → R) (w : α → ℕ) (σ : α → R)
    (t : CandidateTree α) (B : ℕ) :
    compileBudget κ w σ t B = none ↔ ¬ ∃ c : Cut t, cutWeight w c ≤ B := by
  rw [BudgetDP.minimum_none_iff _ _ (quantized_budget_compiler κ w σ t B)]
  simp
```

Proof reading. The infeasibility branch is exactly the negation of the existence of a budget-feasible legal cut. Both directions follow from the same proved minimum semantics. This is why the option-valued model distinguishes `none` from an expensive but attainable value.

8.4 The specification and stored implementation are separate

The functional recurrence above is a convenient specification. Evaluating it independently at many weights can recompute child tables. `StoredBudgetCompiler` adds vectors of $B+1$ option-valued entries, bounded lookup, tabulation, cached child arrays, and stored min-plus merges.

Source: `formalization/StoredBudgetCompiler.lean`.

Exact lines 109–118. Excerpt `stored_tree`. SHA-256 in Section 15.

```
def storedTree (B : ℕ) (κ : α → R) (w : α → ℕ) (σ : α → R) :
    CandidateTree α → BudgetDP.StoredTable R B
  | .leaf a => BudgetDP.tabulate B (BudgetDP.singletonTable (w a) (κ a))
  | .node a n ts =>
    let cached := Vector.ofFn (fun i => storedTree B κ w σ (ts i))
    let merged := BudgetDP.combineStored (fun i : Fin (n + 1) => cached.get i)
```

```
    BudgetDP.tabulate B (fun b => BudgetDP.choose (BudgetDP.singletonTable (w a) (κ a) b)
      (BudgetDP.add (some (σ a)) (BudgetDP.lookup merged b)))

/- The implemented cached array agrees entrywise with the proved exact recurrence. -/
```

Proof reading. At a node, `Vector.ofFn` computes the child arrays before combining them. The final tabulation reads the merged array at each requested weight. Every array is finite and each child result is reused. The loop structure justifies the stored-table work model; it is not inferred from the earlier functional recurrence.

Source: `formalization/StoredBudgetCompiler.lean`.

Exact lines 120–133. Excerpt `stored_correct`. SHA-256 in Section 15.

```
theorem storedTree_correct (B : ℕ) (κ : α → R) (w : α → ℕ) (σ : α → R)
    (t : CandidateTree α) (b : ℕ) (hb : b ≤ B) :
    BudgetDP.lookup (storedTree B κ w σ t) b = budgetTable κ w σ t b := by
  induction t generalizing b with
  | leaf a => exact BudgetDP.lookup_tabulate B _ b hb
  | node a n ts ih =>
    simp only [storedTree, budgetTable, BudgetDP.lookup_tabulate _ _ _ hb]
    congr 2
    rw [BudgetDP.combineStored_correct _ b hb]
    apply BudgetDP.combine_congr_local
    intro i k hk
    simpa only [BudgetDP.vector_get_ofFn] using ih i k (hk.trans hb)

/- The executable tabulated compiler scans its stored root table. -/
```

Proof reading. Induct on the tree. The leaf is bounded lookup of a tabulated singleton. At a node, unfold the two constructions, use correctness of stored child combination, and restrict convolution dependencies to indices no larger than the queried weight. Apply each child induction hypothesis within that bound. This yields entrywise equality with the specification. The stored optimizer's attained minimum is then derived from the already-proved specification semantics.

Source: `formalization/StoredBudgetCompiler.lean`.

Exact lines 256–260. Excerpt `pseudo_polynomial`. SHA-256 in Section 15.

```
theorem tabulation_pseudopolynomial (B : ℕ) (t : CandidateTree α) :
    tabulationWork B t ≤ nodeCount t * (3 * (B + 1) + (B + 1)^2) := by
  have h := tabulation_work_bound B t
  have hm := mergeCount_add_one t
  nlinarith
```

Proof reading. The source separately defines the tabulation work and number of merges. The bound `tabulation_work_bound` supplies per-node table and merge costs; `mergeCount_add_one` relates merges to the tree size. Algebra combines them into the displayed bound. The complexity is polynomial in the numeric budget B, so pseudo-polynomial in a binary encoding. Costs are unit-cost table cells and candidate trials; arbitrary-precision bit complexity, compiler runtime, and device timing are excluded.

8.5 Stability requires controlled branch gaps

Source: `formalization/TreeCompilerUpdates.lean`.

Exact lines 45–59. Excerpt `compiler_stability`. SHA-256 in Section 15.

```
theorem compiler_stability (keep overhead keep' overhead' : α → R)
    (t : CandidateTree α) (hg : GapControlled keep overhead keep' overhead' t) :
    compile keep' overhead' t = compile keep overhead t := by
  induction t with
  | leaf a => rfl
  | node a n ts ih =>
    rcases hg with ⟨hk, hs, hc⟩
    have ho := half_gap_order _ _ _ _ hk hs
    simp only [compile]
    by_cases h : keep a ≤ overhead a + ∑ i, optimalValue keep overhead (ts i)
    · rw [ite_eq_left h, ite_eq_left (ho.mpr h)]
    · rw [ite_eq_right h, ite_eq_right (fun hp => h (ho.mp hp))]
      congr 1
      funext i
      exact ih i (hc i)
```

Proof reading. `GapControlled` recursively supplies bounds on perturbations of keep and split scores, including all child problems. `half_gap_order` proves the original and perturbed comparisons have the same truth value. The proof follows the two branches of `compile`; when both split, induction identifies every child cut. A near tie without this control is not covered by an unchanged-decision guarantee.

Source: `formalization/TreeCompilerUpdates.lean`.

Exact lines 142–150. Excerpt `local_update`. SHA-256 in Section 15.

```
theorem local_recompilation (keep overhead : α → R) (ctx : CachedContext α R)
    (hv : Valid keep overhead ctx) (newTree : CandidateTree α) :
    (∀ c : Cut (fill ctx newTree),
      updateValue keep overhead ctx (optimalValue keep overhead newTree) ≤
        cutValue keep overhead c) ∧
    ∃ c : Cut (fill ctx newTree), cutValue keep overhead c =
      updateValue keep overhead ctx (optimalValue keep overhead newTree) := by
  rw [updateValue_correct keep overhead ctx hv newTree]
  exact exact_tree_compiler keep overhead (fill ctx newTree)
```

Proof reading. A `CachedContext` stores the ancestor context around a replaced subtree, including sibling data. `Valid` asserts those cached values match their subtree optima. `updateValue_correct` identifies the refreshed root value with the optimum of the filled tree. The exact compiler theorem then supplies its lower bound and attaining witness. Separate declarations prove sibling trees and sibling values unchanged. The displayed theorem is correctness of local recomputation; a hardware time-saving claim requires its own execution evidence.

9 Output contracts, batching, and Krylov certificates

9.1 The decoding map is part of the contract

The signature for a fair timing comparison contains a case key, output contract, reference, and timing scope. Equality of those four fields defines `FairMatch`; the source proves it an equivalence relation. This theorem does not establish that historical benchmark metadata contain every field needed to instantiate the relation.

Source: `formalization/SchedulingContracts.lean`.

Exact lines 54–60. Excerpt `contract`. SHA-256 in Section 15.

```
theorem contract_conversion {X C Y : Type*} (compressed : X → C) (explicit : X → Y)
    (decode : C → Y) (runTime : X → ℝ) (decodeTime : C → ℝ)
    (hdecode : ∀ x, decode (compressed x) = explicit x) (x : X) :
    (decodedRun compressed decode runTime decodeTime x).1 = explicit x ∧
    (decodedRun compressed decode runTime decodeTime x).2 =
      runTime x + decodeTime (compressed x) := by
  exact ⟨hdecode x, rfl⟩
```

Proof reading. The hypothesis `hdecode` says the actual decoding function converts the compressed result to the requested explicit object. The modeled decoded run returns that object and adds the declared conversion time. The two conclusion components follow from the decoding identity and the definition. The times are input models, so the theorem does not independently verify their measurement. A separate declaration shows that omitting a positive decoding time understates the modeled total.

9.2 Grouping preserves all contributions

Source: `formalization/SchedulingContracts.lean`.

Exact lines 89–93. Excerpt `grouping`. SHA-256 in Section 15.

```
theorem grouped_execution_output {E Shape V : Type*} [Fintype E] [DecidableEq Shape]
    [AddCommMonoid V] (key : E → Shape) (value : E → V) :
    (∑ k ∈ activeShapeClasses key, ∑ e ∈ Finset.univ.filter (fun e => key e = k), value e) =
      ∑ e, value e := by
  exact Finset.sum_fiberwise_of_maps_to (fun e _ => each_block_has_shape key e) value
```

Proof reading. The complete kernel key maps every active event to one shape class. Summing over the image classes and their fibers reindexes the original finite sum; the library's fiberwise sum theorem proves equality. The companion work theorem repeats this identity for natural-valued work. The launch-count theorem bounds the number of classes by the number of events. The model requires that the key really include the fields needed for a common kernel; it does not verify an arbitrary runtime grouping implementation.

9.3 Strict profitability uses floor plus one

Source: `formalization/SchedulingContracts.lean`.

Exact lines 117–123. Excerpt `amortization`. SHA-256 in Section 15.

```
theorem integer_amortization_threshold (setup baseline candidate : ℝ)
    (hs : 0 ≤ setup) (h : candidate < baseline) (reuse : ℕ) :
    setup + (reuse : ℝ) * candidate < (reuse : ℝ) * baseline ↔
      minimumReuse setup baseline candidate ≤ reuse := by
  rw [amortization_threshold setup baseline candidate reuse h, minimumReuse]
  rw [← Nat.floor_lt (div_nonneg hs (le_of_lt (sub_pos.mpr h)))]
  omega
```

Proof reading. With nonnegative setup and a strictly smaller per-use candidate time, solve the strict total-cost inequality by division through the positive saving. The natural floor lemma converts a strict real threshold into the minimum integer reuse. Equality at break-even is not a profitable use, which accounts for the added one. The minimum-attainment theorem checks that threshold and all smaller reuse counts. Capture/replay and materialization have separate specializations with their own premises.

9.4 Invariant packet Krylov spaces

Contract. The Krylov development uses bounded linear maps on a normed space over a nontrivially normed field, with a finite packet family. The exact theorem assumes the operator commutes with every packet projector.

Source: `formalization/KrylovPackets.lean`.

Exact lines 89–99. Excerpt `krylov`. SHA-256 in Section 15.

```
theorem invariant_packet_krylov (P : PacketFamily (X →L[K] X) I)
    (A : X →L[K] X) (hA : ∀ i, Commute A (P.proj i)) (v : X) (m : ℕ) :
    krylovSpace A v m ≤ ⨆ i, krylovSpace (P.block A i i) (P.proj i v) m := by
  apply Submodule.span_le.mpr
  rintro x ⟨j, rfl⟩
  change (A ^ j.val) v ∈ (⨆ i, krylovSpace (P.block A i i) (P.proj i v) m : Submodule K X)
  rw [invariant_packet_power_expansion P A hA v j.val]
  apply Submodule.sum_mem
  intro i _
  exact Submodule.mem_iSup_of_mem i
    (krylov_generator_mem (P.block A i i) (P.proj i v) m j.val j.isLt)
```

Proof reading. To prove inclusion of a span, it suffices to prove membership for each generator $A^j v$. The preceding exact power-expansion theorem writes that generator as the sum of local packet powers acting on projected seeds. Each term belongs to its local Krylov subspace, hence to the supremum of those spaces; the finite sum remains inside. This endpoint is an inclusion. `local_krylov_direct` separately proves directness using the mutually annihilating packet projectors. It does not assert equality of the global seed Krylov space with the full independent sum of all local spaces.

Source: `formalization/KrylovPackets.lean`.

Exact lines 183–195. Excerpt `approx_krylov`. SHA-256 in Section 15.

```
theorem approximate_packet_krylov_distance (P : PacketFamily (X →L[K] X) I)
    (D E : X →L[K] X) (hD : PacketDiagonal P D) (v : X) (j : ℕ) (hj : 0 < j) :
    Metric.infDist (((D + E) ^ j) v)
      ((⨆ i, krylovSpace (P.block D i i) (P.proj i v) (j + 1) : Submodule K X) : Set X) ≤
      (j : ℝ) * (max ‖D + E‖ ‖D‖) ^ (j - 1) * ‖E‖ * ‖v‖ := by
  have hm : (D ^ j) v ∈ (⨆ i, krylovSpace (P.block D i i) (P.proj i v) (j + 1) : Submodule K X)
   ↪  :=
    invariant_packet_krylov P D (packet_diagonal_commutes P D hD) v (j + 1)
      (krylov_generator_mem D v (j + 1) j (Nat.lt_succ_self j))
  calc
    _ ≤ dist (((D + E) ^ j) v) ((D ^ j) v) := Metric.infDist_le_dist_of_mem hm
    _ = ‖((D + E) ^ j - D ^ j) v‖ := by rw [dist_eq_norm]; rfl
    _ ≤ ‖(D + E) ^ j - D ^ j‖ * ‖v‖ := ((D + E) ^ j - D ^ j).le_opNorm v
    _ ≤ _ := mul_le_mul_of_nonneg_right (approximate_packet_power_bound D E j hj) (norm_nonneg _)
```

Proof reading. Packet diagonality of `D` gives commutation, so $D^j v$ lies in the local Krylov sum of length $j+1$. The distance to that subspace is no larger than the distance to this particular vector. Bound the difference action by its operator norm, then apply the power-perturbation estimate. The displayed premise `0 < j` permits the exponent $j-1$; a zero-power identity is handled separately in the power development.

9.5 The Jacobi normalization uses the L2 operator norm

`invSqrt D` is the actual inverse of the positive matrix square root `CFC.sqrt D`. `matrixCondition A` is the product of the Euclidean operator norms of A and its inverse. The matrix index type is finite and nonempty at the condition-bound endpoint; positivity and invertibility assumptions are explicit.

Source: `formalization/BlockJacobi.lean`.

Exact lines 87–95. Excerpt `jacobi`. SHA-256 in Section 15.

```
theorem block_jacobi_condition (D K : Matrix X X ℂ)
    (hD : D.PosDef) (hA : (D+K).PosDef) (η : ℝ)
    (hE : ‖invSqrt D*K*invSqrt D‖ ≤ η) (hη : η < 1) :
    matrixCondition (invSqrt D*(D+K)*invSqrt D) ≤ (1+η)/(1-η) := by
  have hunit := ((invSqrt_isUnit D hD).mul hA.isUnit).mul (invSqrt_isUnit D hD)
  have hdet := (Matrix.isUnit_iff_isUnit_det _).mp hunit
  have hinv := Matrix.mul_nonsing_inv (invSqrt D*(D+K)*invSqrt D) hdet
  rw [normalized_decomposition D K hD] at hinv ⊢
  exact near_one_condition_bound _ _ hinv η hE hη
```

Proof reading. Positive definiteness of D proves its inverse square root is a unit; positive definiteness of $D+K$ proves the normalized matrix is a unit. Its inverse identity is therefore justified. The normalization theorem writes the matrix as $I+E$, and the near-one condition estimate uses $\|E\| \le \eta < 1$. The theorem requires that supplied norm bound; it does not compute an empirical estimator of η.

Source: `formalization/BlockJacobi.lean`.

Exact lines 134–141. Excerpt `cg_bound`. SHA-256 in Section 15.

```
theorem certified_energy_error {κ κmax e0 em : ℝ} (m : ℕ)
    (hκ : 1 ≤ κ) (hbound : κ ≤ κmax) (he0 : 0 ≤ e0)
    (hCG : em ≤ 2*(cgFactor κ)^m*e0) :
    em ≤ 2*(cgFactor κmax)^m*e0 := by
  have hp : (cgFactor κ)^m ≤ (cgFactor κmax)^m :=
    pow_le_pow_left₀ (cgFactor_nonneg hκ) (cgFactor_mono hbound) m
  exact hCG.trans (mul_le_mul_of_nonneg_right
    (mul_le_mul_of_nonneg_left hp (by norm_num)) he0)
```

Proof reading. The classical CG energy-error estimate is the hypothesis `hCG`. The proved monotonicity of its convergence factor replaces the actual condition number by a certified upper bound. Nonnegativity permits raising the factor and multiplying the initial error. This is a formal comparison of factors and bounds. It does not independently prove CG iteration correctness or the classical convergence theorem. Item 7.5 stays a separate classical input.

9.6 Discovery residuals are orthogonal in Frobenius geometry

Source: `formalization/DiscoveryCertificate.lean`.

Exact lines 126–137. Excerpt `discovery`. SHA-256 in Section 15.

```
theorem discovery_frobenius_certificate (hP : ∀ a, (P.proj a)ᴴ = P.proj a)
    (A : Matrix X X ℂ) :
    ‖A-diagonalCompression P A‖^2 =
      ∑ b, ∑ a, if b=a then 0 else ‖P.block A b a‖^2 := by
  rw [compression_residual P A,
    frobenius_sum_sq (offDiagonalTerm P A) (offDiagonal_trace_orthogonal P hP A)]
  simp only [Fintype.sum_prod_type,offDiagonalTerm]
  apply Finset.sum_congr rfl
  intro b _
  apply Finset.sum_congr rfl
  intro a _
  by_cases h : b=a <;> simp [h]
```

Proof reading. Self-adjoint packet projectors make distinct block terms trace-orthogonal. The source first expresses the compression residual as its off-diagonal block sum. The finite Pythagorean identity for those terms gives the squared Frobenius norm as the sum of squared block norms, with diagonal terms zeroed. An operator-norm certificate is then derived from the Frobenius bound. This is exact geometry of a supplied packet family; simultaneous spectral decomposition is a separately tracked classical input rather than an algorithm proved here.

10 Small-core optimal recompression from constructed ranges

10.1 The shape of the proof

The sixth block completes item 5.6 with ten modules and 79 new theorem declarations. Start with arbitrary finite complex matrices L and R of equal column type, including rank-deficient and zero factors. Construct compact orthonormal bases of their actual column spaces. Express LR^* as an isometric lift of its small core. Supply an ordered thin SVD of that core by its exact equations. Prove diagonal low-rank lower bounds, then transfer them to every ambient rank-bounded competitor. Finally construct factors with exactly $\min(d, k)$ columns.

Every stage has its own declarations. A general transfer lemma may accept core optimality as a premise; the final theorem discharges that premise with independently proved diagonal lower bounds. The supplied SVD data contain no optimality conclusion or singular-tail formula.

10.2 Compact dimensions are the actual matrix ranks

`columnSpace A` is the range of the actual Euclidean linear map associated with A. `compactDim A` is its finite dimension. `columnBasis A` chooses an orthonormal basis of that subspace; `compactQ A` represents its inclusion into the ambient space, and `compactR A` represents the range-restricted map in that basis.

Source: `formalization/CompactRangeFactors.lean`.

Exact lines 51–57. Excerpt `compact_factor`. SHA-256 in Section 15.

```
theorem compact_factorization (A : Matrix m n ℂ) : compactQ A * compactR A = A := by
  unfold compactQ compactR
  rw [←LinearMap.toMatrix_comp]
  have he : (columnSpace A).subtype ∘ₗ (Matrix.toEuclideanLin A).rangeRestrict =
      Matrix.toEuclideanLin A := by ext x i; rfl
  rw [he,Matrix.toEuclideanLin_eq_toLin_orthonormal]
  exact LinearMap.toMatrix_toLin _ _ _
```

Proof reading. Convert the product of the two coordinate matrices to the matrix of a composition. The inclusion composed with the range restriction is the original linear map, by entrywise reflexivity. Converting that map back to the original matrix proves the reconstruction. Separate theorems prove `compactDim A = A.rank`, orthonormality of `compactQ`, and full row rank of `compactR`. These are derived facts about actual matrices. The construction uses classical choice of an orthonormal basis; it is not a numerical triangular QR procedure.

The actual factor core is $\mathrm{compactR}(L)\,\mathrm{compactR}(R)^*$. Its row and column dimensions equal the two original input ranks. Its rank equals the rank of LR^*. A numerical QR prototype can keep zero directions, so its reported intermediate size need not equal these compact dimensions. No numerical-to-formal correspondence is inferred from matching array shapes alone.

10.3 A best-rank conclusion quantifies over all competitors

Source: `formalization/SmallCoreTransfer.lean`.

Exact lines 16–17. Excerpt `best_rank`. SHA-256 in Section 15.

```
def BestRank (N : Matrix m n ℂ → ℝ) (A B : Matrix m n ℂ) (k : ℕ) : Prop :=
  B.rank ≤ k ∧ ∀ C, C.rank ≤ k → N (A-B) ≤ N (A-C)
```

Proof reading. `BestRank N A B k` combines a rank bound on the constructed approximation with the inequality for every matrix `C` of rank at most `k`. Its norm functional `N` is explicit. It neither restricts competitors to a chosen factorization nor assumes they live in the input column spaces.

For isometric rectangular factors Q, V, the source defines `lift` and `compress` by

$$\mathrm{lift}(Q, V, G) = QGV^*, \qquad \mathrm{compress}(Q, V, B) = Q^*BV.$$

The norm module proves that lifting preserves spectral and Frobenius norms, compression does not increase them, and compression does not increase rank. The spectral norm is the L2 operator norm, while the Frobenius norm is selected in its separate norm scope.

Source: `formalization/SmallCoreTransfer.lean`.

Exact lines 25–35. Excerpt `ambient_transfer`. SHA-256 in Section 15.

```
theorem spectral_best_lift (Q : Matrix m r ℂ) (V : Matrix n s ℂ)
    (hQ : Qᴴ*Q=1) (hV : Vᴴ*V=1) (G H : Matrix r s ℂ) (k : ℕ)
    (hbest : BestRank spectralNorm G H k) :
    BestRank spectralNorm (lift Q V G) (lift Q V H) k := by
  refine ⟨(rank_lift_le Q V H).trans hbest.1,fun B hB => ?_⟩
  calc
    spectralNorm (lift Q V G-lift Q V H) = spectralNorm (G-H) := by
      rw [←lift_sub,spectral_lift Q V hQ hV]
    _ ≤ spectralNorm (G-compress Q V B) := hbest.2 _ ((rank_compress_le Q V B).trans hB)
    _ = spectralNorm (compress Q V (lift Q V G-B)) := by rw [compressed_error Q V hQ hV]
    _ ≤ spectralNorm (lift Q V G-B) := spectral_compress_le Q V hQ hV _
```

Proof reading. Fix an arbitrary ambient competitor `B`. Its compressed matrix still has rank at most k. Compare the core approximation with this compressed competitor using the core best-rank hypothesis. Re-express that error as compression of the ambient error and use contraction. Isometric lifting preserves the approximation error exactly. The lifted approximation also has the required rank. This proves global ambient optimality, once the core premise is independently discharged. The parallel Frobenius lemma follows the same chain with its own norm facts.

10.4 The SVD input states equations, rather than desired conclusions

Source: `formalization/SmallCoreRecompression.lean`.

Exact lines 29–37. Excerpt `ordered_svd`. SHA-256 in Section 15.

```
structure OrderedThinSVD (G : Matrix r s ℂ) (d : ℕ) where
  left : Matrix r (Fin d) ℂ
  right : Matrix s (Fin d) ℂ
  values : Fin d → ℝ
  left_isometry : left^H*left=1
  right_isometry : right^H*right=1
  nonnegative : ∀ i, 0 ≤ values i
  sorted : Antitone values
  reconstruction : G=lift left right (diag values)
```

Proof reading. The two column matrices are isometries. Values are real, nonnegative, and antitone in the finite index order. Reconstruction says the core equals the isometric lift of their diagonal. These fields supply the exact algebraic thin SVD data. They assume no rank-bound optimality, tail error, norm preservation, or lower bound. They are also not a theorem that such data exist for every input or that a floating-point routine returns exact equations.

The formal dimension d is the number of supplied thin SVD columns. Zero values are permitted. The output column type `KeptColumn d k` retains indices below k; its cardinality is proved to be $\min(d, k)$. Square roots are taken only of nonnegative retained values. No zero singular value is inverted.

10.5 The spectral lower bound has an actual kernel witness

For a competitor of rank at most $k < d$, restrict it to the first $k + 1$ coordinate directions. The restricted matrix has rank smaller than its domain dimension. Rank-nullity yields a nonzero kernel vector; the prefix embedding makes it an ambient vector supported in those coordinates.

Source: `formalization/SpectralOptimality.lean`.

Exact lines 55–82. Excerpt `kernel_witness`. SHA-256 in Section 15.

```
theorem prefix_kernel_witness (k : ℕ) (hk : k<d)
    (B : Matrix (Fin d) (Fin d) ℂ) (hB : B.rank≤k) :
    ∃ y : EuclideanSpace ℂ (Fin d), y≠0 ∧ B*ᵥ y=0 ∧
      ∀ i : Fin d, k+1 ≤ i.val → y i=0 := by
  let J := prefixEmbedding k (by omega : k+1≤d)
  have hj : J^H*J=1 := prefix_isometry k _
  have hr : (B*J).rank < Fintype.card (Fin (k+1)) := by
    have := Matrix.rank_mul_le_left B J
    simp only [Fintype.card_fin]
    omega
  obtain ⟨z,hz,hbz⟩ := rank_lt_kernel_witness (B*J) hr
  let y : EuclideanSpace ℂ (Fin d) := WithLp.toLp 2 (J*ᵥ z)
  have hy : y≠0 := by
    intro he
    have he' : J*ᵥ z=0 := congrArg WithLp.ofLp he
    have hz' : (z : Fin (k+1) → ℂ)=0 := by
      calc
```

```
        (z : Fin (k+1) → ℂ) = (Jᴴ*J)*ᵥ z := by rw [hj,Matrix.one_mulVec]
        _ = Jᴴ*ᵥ(J*ᵥ z) := (Matrix.mulVec_mulVec _ _ _).symm
        _ = 0 := by rw [he',Matrix.mulVec_zero]
    apply hz
    exact (EuclideanSpace.equiv (Fin (k+1)) ℂ).injective hz'
  refine ⟨y,hy,?_,?_⟩
  · change B*ᵥ(J*ᵥ z)=0
    rw [Matrix.mulVec_mulVec]
    exact hbz
  · intro i hi
    exact prefix_support k _ z i hi
```

Proof reading. `rank_lt_kernel_witness` constructs a nonzero vector in the restricted kernel. The prefix embedding's isometry proves its image remains nonzero. Associativity of matrix-vector multiplication proves the competitor kills that image. The prefix-support lemma proves all later coordinates vanish. Thus the lower-bound argument has a concrete witness; it does not posit the existence of a favorable test vector as an independent assumption.

The diagonal action on that vector has norm at least the first discarded value times its vector norm. The competitor contributes zero, so the operator-norm inequality and positivity of the witness norm give the lower bound. If $k \geq d$, the first discarded value is defined to be zero and norm nonnegativity suffices.

Source: `formalization/SpectralOptimality.lean`.

Exact lines 120–125. Excerpt `spectral_best`. SHA-256 in Section 15.

```
theorem diagonal_spectral_best (σ : Fin d → ℝ) (hσ : ∀ i, 0≤σ i)
    (hsort : Antitone σ) (k : ℕ) :
    BestRank spectralNorm (diag σ) (diag (kept σ k)) k := by
  refine ⟨rank_kept_le σ k,fun B hB => ?_⟩
  rw [diagonal_residual,spectral_diagonal_tail σ hσ hsort]
  exact diagonal_spectral_lower σ hσ hsort k B hB
```

Proof reading. The retained diagonal has rank at most k. Its residual norm is the proved first-discarded-value formula. The separate `diagonal_spectral_lower` applies to every competitor and supplies the final comparison. Both the attained error and the universal lower bound are proved within the diagonal development.

10.6 The Frobenius lower bound uses kernel mass

Choose an orthonormal basis Q of the competitor's actual kernel. The row weights w_i of that basis are nonnegative, at most one, and sum to its dimension. Rank-nullity makes that total at least $d - k$. Decreasing squared singular values therefore allocate their least weighted mass at the tail.

Source: `formalization/FrobeniusOptimality.lean`.

Exact lines 35–60. Excerpt `weighted_tail`. SHA-256 in Section 15.

```
theorem weighted_tail_lower (a w : Fin d → ℝ) (ha : ∀ i, 0 ≤ a i)
    (hsort : Antitone a) (hw0 : ∀ i, 0 ≤ w i) (hw1 : ∀ i, w i ≤ 1)
    (k : ℕ) (hw : (d-k : ℕ) ≤ ∑ i,w i) :
```

```
    (∑ i, if i.val<k then (0:ℝ) else a i) ≤ ∑ i, a i*w i := by
  by_cases hk : k<d
  · let μ := a ⟨k,hk⟩
    have hμ : 0 ≤ μ := ha _
    have hp : ∀ i : Fin d,
        (if i.val<k then 0 else a i) + μ*(w i-(if i.val<k then 0 else 1)) ≤ a i*w i := by
      intro i
      by_cases hi : i.val<k
      · have hle : μ ≤ a i := hsort (by show i.val ≤ k; omega)
        simp only [hi,ite_true,zero_add,sub_zero]
        exact mul_le_mul_of_nonneg_right hle (hw0 i)
      · have hle : a i ≤ μ := hsort (by show k ≤ i.val; omega)
        simp only [hi,ite_false]
        nlinarith [mul_nonneg (sub_nonneg.mpr hle) (sub_nonneg.mpr (hw1 i))]
    have hs := Finset.sum_le_sum (fun i (_ : i ∈ (Finset.univ : Finset (Fin d))) => hp i)
    simp only [Finset.sum_add_distrib] at hs
    have ht := tail_indicator_sum (d:=d) k hk.le
    have hz : 0 ≤ μ*((∑ i,w i)-(d-k : ℕ)) := mul_nonneg hμ (sub_nonneg.mpr hw)
    rw [←Finset.mul_sum,Finset.sum_sub_distrib,ht] at hs
    linarith
  · have he : ∀ i : Fin d, i.val<k := fun i => by omega
    simp only [he,ite_true,Finset.sum_const_zero]
    exact Finset.sum_nonneg (fun i _ => mul_nonneg (ha i) (hw0 i))
```

Proof reading. For $k < d$, use the threshold value a_k. In the prefix, ordering bounds a_k above by a_i and nonnegative weights preserve the inequality. In the tail, $w_i \leq 1$ supplies the corresponding bound. Summing adds a nonnegative threshold times the excess total mass; the tail indicator sum is exactly $d - k$. For $k \geq d$, the tail sum is zero. This finite inequality is proved directly and invokes no Eckart–Young optimality theorem.

Source: `formalization/FrobeniusOptimality.lean`.

Exact lines 76–104. Excerpt `frobenius_lower`. SHA-256 in Section 15.

```
theorem diagonal_frobenius_lower (σ : Fin d → ℝ) (hσ : ∀ i, 0 ≤ σ i)
    (hsort : Antitone σ) (k : ℕ) (B : Matrix (Fin d) (Fin d) ℂ) (hB : B.rank≤k) :
    tailEnergy σ k ≤ frobeniusNorm (diag σ-B)^2 := by
  let Q := kernelFrame B
  have hQ : Qᴴ*Q=1 := kernelFrame_isometry B
  have hw : (d-k : ℕ) ≤ ∑ i,rowWeight Q i := by
    rw [rowWeight_sum Q hQ]
    simp only [Fintype.card_fin]
    have hn := kernel_rank_nullity B
    simp only [Fintype.card_fin] at hn
    exact_mod_cast (by omega : d-k ≤ kernelDim B)
  have hsort2 : Antitone (fun i => σ i^2) := by
    intro i j hij
    have hle := hsort hij
    nlinarith [hσ i,hσ j]
  have hl := weighted_tail_lower (fun i => σ i^2) (rowWeight Q)
    (fun i => sq_nonneg _) hsort2 (rowWeight_nonneg Q) (rowWeight_le_one Q hQ) k hw
  have he : tailEnergy σ k = ∑ i, if i.val<k then (0:ℝ) else σ i^2 := by
    unfold tailEnergy discarded
    apply Finset.sum_congr rfl
```

```
    intro i _
    split_ifs <;> simp
  rw [←he,←diagonal_frame_energy σ Q] at hl
  have hc := frobenius_right_compress_le Q hQ (diag σ-B)
  have hbq : (diag σ-B)*Q=diag σ*Q := by
    rw [Matrix.sub_mul,mul_kernelFrame,sub_zero]
  rw [hbq] at hc
  unfold frobeniusNorm at hl ⊢
  nlinarith [norm_nonneg (diag σ*Q),norm_nonneg (diag σ-B)]
```

Proof reading. Construct the actual kernel frame and establish its row-weight mass from rank-nullity. Apply the finite weighted-tail inequality to σ_i^2. The frame-energy identity converts its right side into the squared Frobenius norm of the diagonal action on the kernel frame. The competitor kills the frame, so the same action is the residual times that frame. Frobenius compression contraction bounds it by the full residual. Norm nonnegativity permits the final squared comparison.

10.7 Combining both optimalities and the exact errors

Source: `formalization/SmallCoreRecompression.lean`.

Exact lines 91–120. Excerpt `recompression_optimal`. SHA-256 in Section 15.

```
theorem recompression_optimality (Q : Matrix m r ℂ) (V : Matrix n s ℂ)
    (hQ : Qᴴ*Q=1) (hV : Vᴴ*V=1) {G : Matrix r s ℂ}
    (svd : OrderedThinSVD G d) (k : ℕ) :
    let A := lift Q V G
    let B := recompressLeft Q svd k * (recompressRight V svd k)ᴴ
    BestRank spectralNorm A B k ∧ BestRank frobeniusNorm A B k ∧
    spectralNorm (A-B)=nextSingular svd.values k ∧
    frobeniusNorm (A-B)^2=tailEnergy svd.values k ∧
    frobeniusNorm (A-B)=Real.sqrt (tailEnergy svd.values k) := by
  dsimp only
  have hG : lift Q V G=lift (Q*svd.left) (V*svd.right) (diag svd.values) :=
    (congrArg (lift Q V) svd.reconstruction).trans (lift_comp Q V _ _ _)
  rw [recompressed_product,hG]
  have hU := isometry_comp Q svd.left hQ svd.left_isometry
  have hZ := isometry_comp V svd.right hV svd.right_isometry
  have hs := spectral_best_lift (Q*svd.left) (V*svd.right) hU hZ _ _ k
    (diagonal_spectral_best svd.values svd.nonnegative svd.sorted k)
  have hf := frobenius_best_lift (Q*svd.left) (V*svd.right) hU hZ _ _ k
    (diagonal_frobenius_best svd.values svd.nonnegative svd.sorted k)
  refine ⟨hs,hf,?_,?_,?_⟩
  · rw [←lift_sub,spectral_lift _ _ hU hZ,diagonal_residual]
    exact spectral_diagonal_tail _ svd.nonnegative svd.sorted k
  · rw [←lift_sub,frobenius_lift _ _ hU hZ,diagonal_residual]
    exact frobenius_diagonal_tail _ k
  · rw [←lift_sub,frobenius_lift _ _ hU hZ,diagonal_residual]
    rw [←frobenius_diagonal_tail svd.values k,Real.sqrt_sq_eq_abs]
    have hn : 0 ≤ frobeniusNorm (diag (discarded svd.values k)) := by
      unfold frobeniusNorm
      exact norm_nonneg (diag (discarded svd.values k))
    exact (abs_of_nonneg hn).symm
```

Proof reading. Rewrite the core reconstruction as a lift through the composed isometries. Rewrite the factor product as the retained diagonal lift. Derive isometry of each composed factor. Apply the spectral and Frobenius transfer lemmas to their independently proved diagonal best-rank conclusions. For the three error formulas, use linearity of lift, norm preservation, and the diagonal tail identities. The square-root formula uses nonnegativity of the Frobenius norm. These are conclusions of the theorem, not fields of its SVD input.

Source: `formalization/SmallCoreRecompression.lean`.

Exact lines 124–141. Excerpt `small_core_endpoint`. SHA-256 in Section 15.

```
theorem small_core_optimal_recompression (L : Matrix m t ℂ) (R : Matrix n t ℂ)
    (svd : OrderedThinSVD (factorCore L R) d) (k : ℕ) :
    let Lk := recompressLeft (compactQ L) svd k
    let Rk := recompressRight (compactQ R) svd k
    BestRank spectralNorm (L*Rᴴ) (Lk*Rkᴴ) k ∧
    BestRank frobeniusNorm (L*Rᴴ) (Lk*Rkᴴ) k ∧
    spectralNorm (L*Rᴴ-Lk*Rkᴴ)=nextSingular svd.values k ∧
    frobeniusNorm (L*Rᴴ-Lk*Rkᴴ)^2=tailEnergy svd.values k ∧
    frobeniusNorm (L*Rᴴ-Lk*Rkᴴ)=Real.sqrt (tailEnergy svd.values k) ∧
    compactDim L=L.rank ∧ compactDim R=R.rank ∧
    Fintype.card (KeptColumn d k)=min d k := by
  dsimp only
  have h := recompression_optimality (compactQ L) (compactQ R)
    (compactQ_isometry L) (compactQ_isometry R) svd k
  dsimp only at h
  rw [←compact_product_reconstruction L R] at h
  exact ⟨h.1,h.2.1,h.2.2.1,h.2.2.2.1,h.2.2.2.2,
    compactDim_eq_rank L,compactDim_eq_rank R,keptColumn_card k⟩
```

Proof reading. Instantiate the preceding theorem with the compact factors constructed from the original L and R. Rewrite their lifted core as LR^*. Pair its two global optimality conclusions and exact tail errors with the proved equalities between compact dimensions and original ranks, and with the retained-column cardinality. Thus the endpoint concerns the original rectangular factor product and every ambient rank-bounded competitor.

10.8 Zero rank and full retention remain defined

Source: `formalization/RecompressionEndpoints.lean`.

Exact lines 70–74. Excerpt `full_retention`. SHA-256 in Section 15.

```
theorem recompression_full (Q : Matrix m r ℂ) (V : Matrix n s ℂ)
    {G : Matrix r s ℂ} (svd : OrderedThinSVD G d) (k : ℕ) (hk : d ≤ k) :
    recompressLeft Q svd k*(recompressRight V svd k)ᴴ = lift Q V G := by
  rw [recompressed_product,kept_all _ k hk,←lift_comp]
  exact congrArg (lift Q V) svd.reconstruction.symm
```

Proof reading. When $d \leq k$, retaining the entire value list gives the original diagonal. Rewrite the constructed product by that full-retention identity, then use the supplied SVD reconstruction under the outer lift. The separate rank-zero endpoint proves that keeping no columns gives the zero product. Empty index types avoid dummy inverse operations and support zero original factors. Repeated or zero singular values are permitted by the non-strict ordering.

10.9 The boundary after the sixth block

The ordered-SVD-data theorem proves conditional exact optimality and exact tail errors. General SVD existence, identification with the canonical Mathlib `singularValues` API, a floating-point SVD or QR algorithm, and its numerical stability are outside this formal block. Classical item 5.5 remains separately tracked, with these checked supporting results. The formal construction does not establish GPU performance, a generic dense matrix exponent, or equivalence to the historical NumPy prototype.

11 Replay, provenance, and the four classical boundaries

11.1 The accepted evidence is a cumulative source-bound run

The preserved summary `verification/sixth_final/SUMMARY.json` records a newly created accepted output directory, all 40 modules, all 392 theorem declarations, 66 construction audits, the two invalid-proof controls, and completion at `2026-09-06T22:38:55.838919+00:00`. The reported Lean version is `4.34.0-rc2`; the compiler commit and Mathlib revision are printed in Section 14.

The saved summary retains source and local dependency hashes, generated-interface hashes, commands, exit codes, and log hashes. The source release excludes local compiled interfaces. A fresh verification run regenerates them; the static checks in this guide do not recreate them or use their hashes as a substitute for compilation. The pinned Mathlib cache was reused in the inherited run, not independently rebuilt.

11.2 The status of this edition

The guide assembler resolves active namespaces, reconciles all 458 audited names, matches all 40 active source hashes to the accepted report, parses the saved compiler logs back into axiom sets, and matches all log hashes. It also verifies both rejected-control source and log identities. Every excerpt is bound to exact source lines and both source and excerpt SHA-256 values. These are fresh static documentation checks.

Lean and Lake are absent from the present host. The documentation report says `fresh_lean_kernel_run: false`. This edition makes no fresh compilation claim. The source and accepted evidence are unchanged. The original accepted build remains inherited evidence rather than a current run.

11.3 An actual kernel replay on a suitable host

Use the separately supplied portable Linux x86-64 / WSL kit for the exact recorded compiler and Mathlib versions. Python 3.10 or later is required; tests and numerical mirrors also require NumPy. Run from the extracted `original_snapshot` root:

```
python3 scripts/verify_integrity.py
python3 -m unittest discover -s scripts -p 'test_*.py' -v
python3 scripts/verify.py --kit /path/to/Komorebi_Portable_Lean_4.34.0-rc2 --out
 ↪  verification/recheck --timeout 240
python3 scripts/run_sixth_regressions.py --out verification/recheck_numerical.json
```

The verifier's exit code zero means complete pass, one means failure, and two means paused or incomplete. On a host with short command limits, use the same output directory with `--max-checks 2 --resume`. Checkpoint source, dependency, interface, and log identities govern resumption. A partial checkpoint does not establish a cumulative pass. A different compiler or dependency revision is a different replay and should be reported as such.

To rebuild only this guide, run `python3 build.py` from its directory. The deterministic isolated XeLaTeX builder needs only the shipped guide sources, generated tables, and exact excerpt files. To regenerate those tables from the edition's preserved active snapshot, run `python3 assemble.py --source /absolute/path/to/original_snapshot`, then rebuild. `python3 verify_pdf.py --source /absolute/path/to/original_snapshot` checks exact excerpt bytes, extracted declaration names, references, glyphs, page geometry, and internal links. Neither script invokes Lean.

11.4 Classical inputs are not folded into the 56 count

Item	Classical input	Precise present boundary
5.5	Eckart–Young–Mirsky	Ordered diagonal spectral and Frobenius best-rank bounds and isometric transfer are checked as support for 5.6. General SVD existence and the canonical singular-values API theorem are outside this release.
7.5	CG convergence	The classical energy-error estimate is supplied. The formal block proves monotonicity and replacement by a certified condition bound, rather than proving the CG algorithm or its classical convergence theorem.
8.2	Floating-point product model	Local error inequalities are supplied to exact normed-ring propagation. IEEE rounding semantics and a device implementation are not independently proved.
9.1	Simultaneous spectral decomposition	Exact residual and off-diagonal certificates are proved for a supplied finite self-adjoint projector family. A general simultaneous-decomposition algorithm is not independently formalized.

11.5 Historical counts, failures, and numerical mirrors

The six retained supplements document successive proof blocks. Their earlier coverage and remaining-work lists are historical; the sixth ledger supersedes those live status statements. They remain preserved as evidence of what each release actually established. The active source is the 392-declaration sixth-block branch. The earlier 298-declaration branch was not merged into it.

The sixth development record retains 31 captured checks, 21 of which failed before correction. Those diagnostics and source snapshots are separate from the accepted cumulative build. The invalid equality and admitted-proof controls were rejected in that accepted run. A failed development attempt is not a theorem; a corrected theorem is assessed through its active source and accepted evidence.

The inherited sixth numerical mirror records 182 factor pairs, 638 truncation cases, 3,190 arbitrary rank-bounded competitors, nine invalid-input controls, and two assumption-removal controls. It uses NumPy complex128 and numerical rank thresholds; it is neither extracted nor compiled Lean and does not prove universal matrix optimality. Any present numerical rerun is reported separately in the edition's verification records. The historical CPU/GPU benchmarks were not rerun here, and their recorded comparison fields do not automatically instantiate the complete formal fair-match signature.

11.6 Reproducible limits

The three companions share the original result keys and preserve the complete raw mathematical and formal sources. The paper-to-Lean map explains recorded hypotheses and scope. It is not an automatically checked semantic-equivalence theorem between prose, Lean, and prototype Python. No fresh CUDA correctness, device stability, timing advantage, full imported-library rebuild, or claims beyond the stated finite mathematics follow from the documentation checks.

12 Complete manuscript-result correspondence

The original manuscript numbering below is retained as a stable key across the companions. The preserved ledger has 85 numbered records, including definitions and expository entries. This map prints its 56 manuscript-proved results and four classical inputs. Source locations identify declarations; they do not claim that Lean certifies prose-to-code equivalence automatically.

12.1 2.3 — Exact evaluation

Traditional original item 2.3, current statement 2.3. Printed page 7, PDF page 13.

Recorded status. checked.

Recorded hypotheses and scope. Finite orthogonal idempotent family summing to one. Exact nonzero support, not thresholded support. Actual rectangular matrix batches are covered.

- `KomorebiPaperVIII.exact_active_evaluation`; `PacketIR.lean`, line 41.
- `KomorebiPaperVIII.exact_active_action`; `PacketIR.lean`, line 52.
- `KomorebiPaperVIII.exact_matrix_batch`; `PacketIR.lean`, line 145.

12.2 2.5 — Block composition

Traditional original item 2.5, current statement 2.5. Printed page 8, PDF page 14.

Recorded status. checked.

Recorded hypotheses and scope. Finite projector family over a ring. Relational composition gives an inclusion because nonzero path terms can cancel.

- `KomorebiPaperVIII.exact_block_composition`; `PacketIR.lean`, line 58.
- `KomorebiPaperVIII.product_support_inclusion`; `PacketIR.lean`, line 63.

12.3 2.6 — Adjoint and tensor semantics

Traditional original item 2.6, current statement 2.6. Printed page 9, PDF page 15.

Recorded status. checked.

Recorded hypotheses and scope. Adjoint identity explicitly assumes self-adjoint projectors in a star ring. Tensor identity uses actual Kronecker products of finite matrices.

- `KomorebiPaperVIII.exact_adjoint_block`; `PacketIR.lean`, line 95.
- `KomorebiPaperVIII.exact_tensor_block`; `PacketIR.lean`, line 135.

12.4 2.9 — Contract conversion

Traditional original item 2.9, current statement 2.9. Printed page 9, PDF page 15.

Recorded status. checked.

Recorded hypotheses and scope. Formal output conversion includes the actual decoding map and its declared time. Direct downstream composition is proved. This is a contract model, not independent validation of timings.

- `KomorebiPaperVIII.contract_conversion`; `SchedulingContracts.lean`, line 54.

- `KomorebiPaperVIII.direct_compressed_consumption`; `SchedulingContracts.lean`, line 62.
- `KomorebiPaperVIII.omitted_decode_understates_time`; `SchedulingContracts.lean`, line 69.

12.5 3.1 — Dense and packet application counts

Traditional original item 3.1, current statement 3.1. Printed page 10, PDF page 16.

Recorded status. checked.

Recorded hypotheses and scope. Finite scalar address and modeled multiply-accumulate event cardinalities. Two scalar operations per coefficient/right-hand side. No device instruction claim.

- `KomorebiPaperVIII.packet_storage_count`; `PacketCostModels.lean`, line 27.
- `KomorebiPaperVIII.packet_application_count`; `PacketCostModels.lean`, line 31.
- `KomorebiPaperVIII.dense_storage_count`; `PacketCostModels.lean`, line 36.
- `KomorebiPaperVIII.dense_application_count`; `PacketCostModels.lean`, line 39.

12.6 3.2 — Equal diagonal packets

Traditional original item 3.2, current statement 3.2. Printed page 10, PDF page 16.

Recorded status. checked.

Recorded hypotheses and scope. Equal widths, n = card(I)*b. Positive/nonzero denominators for real ratios. Work-ratio specialization follows from equal_bdlr_reduction at r=0.

- `KomorebiPaperVIII.equal_diagonal_storage`; `PacketCostModels.lean`, line 44.
- `KomorebiPaperVIII.equal_diagonal_work`; `PacketCostModels.lean`, line 48.
- `KomorebiPaperVIII.equal_block_reduction`; `PacketCostModels.lean`, line 87.

12.7 3.4 — BDLR application and storage

Traditional original item 3.4, current statement 3.4. Printed page 11, PDF page 17.

Recorded status. checked.

Recorded hypotheses and scope. Explicit diagonal and two thin factor address/event types; third kernel accumulates into the diagonal result. The actual algebraic action is the unchanged LowRankAlgebra.bdlr_apply theorem.

- `KomorebiPaperVIII.bdlr_storage_count`; `PacketCostModels.lean`, line 58.
- `KomorebiPaperVIII.bdlr_application_count`; `PacketCostModels.lean`, line 68.
- `KomorebiPaperVIII.equal_bdlr_storage`; `PacketCostModels.lean`, line 75.
- `KomorebiPaperVIII.equal_bdlr_work`; `PacketCostModels.lean`, line 81.
- `KomorebiPaperVIII.equal_bdlr_reduction`; `PacketCostModels.lean`, line 91.

12.8 3.6 — Streaming byte count

Traditional original item 3.6, current statement 3.6. Printed page 11, PDF page 17.

Recorded status. checked.

Recorded hypotheses and scope. One block read, one source read, target read and write per active edge, no interedge reuse. Count is of declared events.

- `KomorebiPaperVIII.streaming_value_count`; `PacketCostModels.lean`, line 102.
- `KomorebiPaperVIII.streaming_byte_count`; `PacketCostModels.lean`, line 111.

12.9 3.8 — Roofline lower bound

Traditional original item 3.8, current statement 3.8. Printed page 12, PDF page 18.

Recorded status. checked.

Recorded hypotheses and scope. Time bound assumes positive peak rates and actual capacity inequalities. Work-rate bound assumes nonnegative work, positive bytes and elapsed time. These hypotheses are explicit.

- `KomorebiPaperVIII.roofline_time_bound`; `PacketCostModels.lean`, line 118.
- `KomorebiPaperVIII.roofline_work_rate`; `PacketCostModels.lean`, line 126.

12.10 3.9 — Capacity threshold

Traditional original item 3.9, current statement 3.9. Printed page 12, PDF page 18.

Recorded status. checked.

Recorded hypotheses and scope. Dense scalar capacity equivalence uses n,M nonnegative and scalar byte size positive. BDLR natural-number storage inequality is exact; workspace memory is excluded.

- `KomorebiPaperVIII.dense_capacity_threshold`; `PacketCostModels.lean`, line 139.
- `KomorebiPaperVIII.bdlr_capacity_threshold`; `PacketCostModels.lean`, line 143.

12.11 3.10 — Materialization cost

Traditional original item 3.10, current statement 3.10. Printed page 13, PDF page 19.

Recorded status. checked.

Recorded hypotheses and scope. Explicit dense low-rank product followed by one scalar addition per output has 2*n^2*r+n^2 modeled operations. The remainder has the exact quadratic bound n^2, stronger than the displayed big-O term; no generic implementation lower bound.

- `KomorebiPaperVIII.materialization_work_count`; `PacketCostModels.lean`, line 151.
- `KomorebiPaperVIII.materialization_quadratic_remainder`; `PacketCostModels.lean`, line 156.
- `KomorebiPaperVIII.dense_storage_count`; `PacketCostModels.lean`, line 36.

12.12 4.1 — Equal block banded count

Traditional original item 4.1, current statement 4.1. Printed page 13, PDF page 19.

Recorded status. checked.

Recorded hypotheses and scope. Actual band address subtype in Fin p x Fin p. Count p*(2*w+1)-w*(w+1) proved from finite sums under w<p. Explicit stored-scalar and multiply/accumulate event types give b^2*N and 2*q*b^2*N. The manuscript substitution p=n/b additionally needs b>0 and exact divisibility. No hardware operation timing claim.

- `KomorebiPaperVIII.banded_block_count`; `BandedCount.lean`, line 73.
- `KomorebiPaperVIII.banded_storage_count`; `BandedCount.lean`, line 85.
- `KomorebiPaperVIII.banded_operation_count`; `BandedCount.lean`, line 90.

12.13 4.2 — Sparse plus low rank application

Traditional original item 4.2, current statement 4.2. Printed page 14, PDF page 20.

Recorded status. checked.

Recorded hypotheses and scope. Counts actual sparse-value and thin-factor address types, excluding sparse index metadata. Algebraic action follows from the upstream bdlr_apply identity, which does not require its D argument to be diagonal.

- `KomorebiPaperVIII.sparse_low_rank_storage_count`; `PacketCostModels.lean`, line 164.
- `KomorebiPaperVIII.sparse_low_rank_work_count`; `PacketCostModels.lean`, line 169.

12.14 4.3 — Circulant spectral representation

Traditional original item 4.3, current statement 4.3. Printed page 14, PDF page 20.

Recorded status. checked.

Recorded hypotheses and scope. Actual circulant and unitary-normalized Fourier matrix on ZMod n for every n>0. The exponential formula, both unitarity identities, exact diagonalization and two-transform application are checked. No FFT complexity or hardware execution is claimed.

- `KomorebiPaperVIII.fourier_matrix_exponential`; `CirculantFourier.lean`, line 39.
- `KomorebiPaperVIII.fourier_unitary`; `CirculantFourier.lean`, line 83.
- `KomorebiPaperVIII.circulant_spectral_representation`; `CirculantFourier.lean`, line 150.
- `KomorebiPaperVIII.circulant_fourier_application`; `CirculantFourier.lean`, line 155.

12.15 4.4 — Toeplitz circulant embedding

Traditional original item 4.4, current statement 4.4. Printed page 15, PDF page 21.

Recorded status. checked.

Recorded hypotheses and scope. Actual complex Toeplitz matrix, explicitly reversed generator of length n+n and Fin.addCases zero padding for n>0. The diagonal uses c[0]; r[0] is irrelevant. Neither symmetry nor Hermiticity is assumed.

- `KomorebiPaperVIII.toeplitz_circulant_block`; `ToeplitzEmbedding.lean`, line 28.

- `KomorebiPaperVIII.toeplitz_embedding_action`; `ToeplitzEmbedding.lean`, line 54.
- `KomorebiPaperVIII.toeplitz_generator_first_half`; `ToeplitzEmbedding.lean`, line 60.
- `KomorebiPaperVIII.toeplitz_generator_middle`; `ToeplitzEmbedding.lean`, line 64.

12.16 4.5 — Kronecker modewise application

Traditional original item 4.5, current statement 4.5. Printed page 15, PDF page 21.

Recorded status. checked.

Recorded hypotheses and scope. Actual column-major Matrix.vec and actual Kronecker products; the right factor uses transpose, not adjoint. Arithmetic and factor-storage counts are cardinalities of finite address/event types.

- `KomorebiPaperVIII.kronecker_modewise`; `StructuredTensor.lean`, line 19.
- `KomorebiPaperVIII.kronecker_modewise_work`; `StructuredTensor.lean`, line 27.
- `KomorebiPaperVIII.kronecker_factor_storage`; `StructuredTensor.lean`, line 32.

12.17 4.6 — Equal tensor factors

Traditional original item 4.6, current statement 4.6. Printed page 15, PDF page 21.

Recorded status. checked.

Recorded hypotheses and scope. Actual arbitrary-length recursive tensor matrix and slice-by-slice contraction. Exact event count 2*k*d^(k+1) and storage k*d^2; includes k=0 and d=0. Model charges two operations per multiply-accumulate.

- `KomorebiPaperVIII.tensor_dimension`; `StructuredTensor.lean`, line 51.
- `KomorebiPaperVIII.tensor_apply_correct`; `StructuredTensor.lean`, line 73.
- `KomorebiPaperVIII.tensor_modewise_work`; `StructuredTensor.lean`, line 112.
- `KomorebiPaperVIII.tensor_factor_storage`; `StructuredTensor.lean`, line 126.

12.18 4.7 — Walsh Hadamard recursion

Traditional original item 4.7, current statement 4.7. Printed page 16, PDF page 22.

Recorded status. checked.

Recorded hypotheses and scope. Recursively constructed complex Walsh matrices and actual sum/difference butterfly. Proves ordinary-transpose orthogonality, algorithm correctness and exactly N*log2(N) additions/subtractions for N=2^k, including N=1.

- `KomorebiPaperVIII.walsh_orthogonal`; `WalshTransform.lean`, line 33.
- `KomorebiPaperVIII.walsh_apply_correct`; `WalshTransform.lean`, line 60.
- `KomorebiPaperVIII.walsh_operation_count`; `WalshTransform.lean`, line 99.
- `KomorebiPaperVIII.walsh_log_operation_count`; `WalshTransform.lean`, line 115.

12.19 5.1 — Exact BDLR product

Traditional original item 5.1, current statement 5.1. Printed page 16, PDF page 22.

Recorded status. checked.

Recorded hypotheses and scope. Finite rectangular factors over a field with involution. Diagonal closure and actual correction rank are proved. No materialization or timing claim.

- `KomorebiPaperVIII.packet_diagonal_product`; `LowRankAlgebra.lean`, line 33.
- `KomorebiPaperVIII.bdlr_product`; `LowRankAlgebra.lean`, line 57.
- `KomorebiPaperVIII.bdlr_product_rank_bound`; `LowRankAlgebra.lean`, line 68.

12.20 5.2 — Rank growth in a product chain

Traditional original item 5.2, current statement 5.2. Printed page 17, PDF page 23.

Recorded status. checked.

Recorded hypotheses and scope. Actual variable-rank thin matrix factors are constructed for every length, including zero rank and the empty product. The correction matrix rank is bounded by the sum of factor widths. Packet diagonality of the base product is derived.

- `KomorebiPaperVIII.chainFactors_correct`; `LowRankChains.lean`, line 56.
- `KomorebiPaperVIII.bdlr_chain_rank_bound`; `LowRankChains.lean`, line 69.
- `KomorebiPaperVIII.packet_diagonal_chain`; `LowRankChains.lean`, line 87.

12.21 5.3 — Woodbury inversion

Traditional original item 5.3, current statement 5.3. Printed page 18, PDF page 24.

Recorded status. checked.

Recorded hypotheses and scope. Both D and I+V* D^{-1} U are units. The update is proved invertible. Ordinary matrix inverse and arbitrary right-hand-side batches. Uses Mathlib classical Woodbury identity.

- `KomorebiPaperVIII.bdlr_woodbury`; `LowRankAlgebra.lean`, line 89.
- `KomorebiPaperVIII.bdlr_update_isUnit`; `LowRankAlgebra.lean`, line 97.
- `KomorebiPaperVIII.bdlr_woodbury_solve`; `LowRankAlgebra.lean`, line 110.

12.22 5.4 — Inverse norm bound

Traditional original item 5.4, current statement 5.4. Printed page 18, PDF page 24.

Recorded status. checked.

Recorded hypotheses and scope. Real or complex matrices with the Hilbert-space L2 operator norm, including rectangular factors. Same two invertibility assumptions as 5.3.

- `KomorebiPaperVIII.bdlr_inverse_norm_bound`; `LowRankAlgebra.lean`, line 126.

12.23 5.5 — Eckart Young Mirsky theorem

Traditional original item 5.5, current statement 5.5. Printed page 18, PDF page 24.

Recorded status. external_input_with_checked_svd_data_specialization.

Recorded hypotheses and scope. Ordered diagonal spectral/Frobenius best-rank inequalities and rectangular isometric transfer are now independently checked as supporting results for 5.6. General SVD existence and a theorem phrased in the canonical Mathlib singularValues API remain outside this release. This classical entry stays separately tracked, not added to the 56 manuscript-proved count.

12.24 5.6 — Small core recompression

Traditional original item 5.6, current statement 5.6. Printed page 18, PDF page 24.

Recorded status. checked.

Recorded hypotheses and scope. Actual compact orthonormal column-space factorizations with dimensions equal to the input matrix ranks. For any supplied ordered thin SVD of the actual small core, constructs thin square-root factors with min(d,k) columns, proves rank <= k, global spectral and Frobenius optimality against every ambient competitor, and exact tail errors. No assumed optimality, tail formulas, norm invariance or rank identities. Includes zero and rank-deficient factors, k=0, k>=d and repeated singular values. SVD existence/computation, floating-point QR/SVD, and canonical Mathlib singularValues identification are not independently formalized. The supplied SVD equations are explicit hypotheses, as in the manuscript.

- `KomorebiPaperVIII.SmallCore.small_core_optimal_recompression`; `SmallCoreRecompression.lean`, line 124.
- `KomorebiPaperVIII.SmallCore.recompression_optimality`; `SmallCoreRecompression.lean`, line 91.
- `KomorebiPaperVIII.SmallCore.compact_product_reconstruction`; `SmallCoreTransfer.lean`, line 61.
- `KomorebiPaperVIII.SmallCore.diagonal_spectral_best`; `SpectralOptimality.lean`, line 120.
- `KomorebiPaperVIII.SmallCore.diagonal_frobenius_best`; `FrobeniusOptimality.lean`, line 106.

12.25 5.8 — Product chain perturbation

Traditional original item 5.8, current statement 5.8. Printed page 19, PDF page 25.

Recorded status. checked.

Recorded hypotheses and scope. Actual ordered products in a normed ring with norm-one identity. Includes the empty chain. The displayed finite sum uses perturbed suffix norms and exact prefix norms. This covers operators on nonzero normed spaces, without a commutativity assumption.

- `KomorebiPaperVIII.product_chain_perturbation`; `ProductChainError.lean`, line 148.
- `KomorebiPaperVIII.product_chain_error_bound`; `ProductChainError.lean`, line 129.

12.26 5.9 — Certified repeated recompression

Traditional original item 5.9, current statement 5.9. Printed page 20, PDF page 26.

Recorded status. checked.

Recorded hypotheses and scope. Actual left-multiplied iterates with additive post-product residuals in a normed ring. Both actual residual norms and arbitrary certified error budgets are supported, including the empty iteration.

- `KomorebiPaperVIII.repeated_recompression_bound`; `ProductChainError.lean`, line 100.
- `KomorebiPaperVIII.repeated_recompression_actual_error`; `ProductChainError.lean`, line 114.

12.27 6.1 — Path support of powers

Traditional original item 6.1, current statement 6.1. Printed page 21, PDF page 27.

Recorded status. checked.

Recorded hypotheses and scope. Recursive finite sum over active paths equals the actual block of A^n. Includes length zero explicitly.

- `KomorebiPaperVIII.power_path_expansion`; `TransferSupport.lean`, line 37.
- `KomorebiPaperVIII.power_block_zero_of_no_path`; `TransferSupport.lean`, line 74.
- `KomorebiPaperVIII.power_support_inclusion`; `TransferSupport.lean`, line 79.

12.28 6.2 — Polynomial support

Traditional original item 6.2, current statement 6.2. Printed page 21, PDF page 27.

Recorded status. checked.

Recorded hypotheses and scope. Actual finite polynomial sum in an algebra. Support contained in the union over powers, including the constant term.

- `KomorebiPaperVIII.polynomial_support_inclusion`; `TransferSupport.lean`, line 127.
- `KomorebiPaperVIII.polynomial_zero_of_no_paths`; `TransferSupport.lean`, line 142.

12.29 6.3 — Acyclic transfer nilpotence

Traditional original item 6.3, current statement 6.3. Printed page 21, PDF page 27.

Recorded status. checked.

Recorded hypotheses and scope. Absence of paths of length h+1 forces A^(h+1)=0. Also derived from a supplied strictly increasing topological level certificate. No graph-level construction algorithm is claimed.

- `KomorebiPaperVIII.nilpotent_of_path_bound`; `TransferSupport.lean`, line 86.
- `KomorebiPaperVIII.nilpotent_of_topological_levels`; `TransferSupport.lean`, line 109.

12.30 6.5 — Level schedule for a triangular solve

Traditional original item 6.5, current statement 6.5. Printed page 22, PDF page 28.

Recorded status. checked.

Recorded hypotheses and scope. Actual well-founded forward substitution and stage evaluator in dependent, possibly unequal block spaces. Finite acyclic graph, invertible local diagonal maps, and arbitrary linear edge transfers. No precomputed solution or topological schedule is assumed. Stages read only predecessors and solve the original block equations at canonical depth.

- `KomorebiPaperVIII.DAG.Graph.solve_correct`; `TriangularSolve.lean`, line 38.
- `KomorebiPaperVIII.DAG.Graph.solve_unique`; `TriangularSolve.lean`, line 46.
- `KomorebiPaperVIII.DAG.Graph.triangular_bijective`; `TriangularSolve.lean`, line 65.
- `KomorebiPaperVIII.DAG.Graph.stagedSolve_correct_through`; `TriangularSolve.lean`, line 91.
- `KomorebiPaperVIII.DAG.Graph.level_schedule_solves`; `TriangularSolve.lean`, line 112.

12.31 6.6 — Sequential depth

Traditional original item 6.6, current statement 6.6. Printed page 22, PDF page 28.

Recorded status. checked.

Recorded hypotheses and scope. Canonical predecessor-max levels are constructed from acyclicity through strict ancestor-cardinality decrease. For nonempty finite graphs, attained maximum equals one plus longest edge-path length. Empty graph has depth zero. This is an ideal unit-task schedule, not a hardware timing theorem.

- `KomorebiPaperVIII.DAG.Graph.level_attained`; `GraphSchedules.lean`, line 101.
- `KomorebiPaperVIII.DAG.Graph.path_length_le_depth`; `GraphSchedules.lean`, line 156.
- `KomorebiPaperVIII.DAG.Graph.depth_attained`; `GraphSchedules.lean`, line 164.
- `KomorebiPaperVIII.DAG.Graph.empty_depth`; `GraphSchedules.lean`, line 177.

12.32 6.7 — Inverse support lies in the transitive closure

Traditional original item 6.7, current statement 6.7. Printed page 23, PDF page 29.

Recorded status. checked.

Recorded hypotheses and scope. Actual reachable packet subspace in a finite-dimensional algebra is invariant. A left inverse makes the restricted forward map injective, hence surjective, yielding inverse invariance. Actual nonsingular matrix inverse specialization. Reachability is reflexive: zero-length paths are included. No inverse-support hypothesis is assumed.

- `KomorebiPaperVIII.inverse_block_zero_of_no_path`; `InverseSupport.lean`, line 94.
- `KomorebiPaperVIII.inverse_support_inclusion`; `InverseSupport.lean`, line 102.
- `KomorebiPaperVIII.matrix_inverse_support`; `InverseSupport.lean`, line 116.

12.33 6.8 — Schur support inclusion

Traditional original item 6.8, current statement 6.8. Printed page 23, PDF page 29.

Recorded status. checked.

Recorded hypotheses and scope. Actual rectangular retained-hidden factors and nonsingular hidden matrix inverse. Nonzero Schur block implies a direct retained edge or nonzero entrance, reachable hidden path (possibly zero length), and nonzero exit. Only support inclusion is asserted; cancellation may remove a permitted block.

- `KomorebiPaperVIII.schur_correction_expansion`; `SchurSupport.lean`, line 25.
- `KomorebiPaperVIII.schur_correction_support`; `SchurSupport.lean`, line 33.
- `KomorebiPaperVIII.schur_support_inclusion`; `SchurSupport.lean`, line 57.
- `KomorebiPaperVIII.schur_zero_of_no_route`; `SchurSupport.lean`, line 76.

12.34 7.2 — Invariant packet Krylov decomposition

Traditional original item 7.2, current statement 7.2. Printed page 24, PDF page 30.

Recorded status. checked.

Recorded hypotheses and scope. Actual bounded linear maps and spans of finite power orbits. Commutation with the complete finite projector family gives the power expansion and Krylov inclusion. Local spaces lie in the actual projector ranges and their sum is direct. No equality of the global and local Krylov spaces is asserted.

- `KomorebiPaperVIII.invariant_packet_power_expansion`; `KrylovPackets.lean`, line 76.
- `KomorebiPaperVIII.invariant_packet_krylov`; `KrylovPackets.lean`, line 89.
- `KomorebiPaperVIII.local_krylov_direct`; `KrylovPackets.lean`, line 131.

12.35 7.3 — Approximate packet Krylov bound

Traditional original item 7.3, current statement 7.3. Printed page 24, PDF page 30.

Recorded status. checked.

Recorded hypotheses and scope. Positive integer power, actual operator norm, and actual Metric.infDist to the sum of local Krylov spaces. Membership of the comparison vector D^j v is derived from packet diagonality. The power estimate also allows a trivial normed ring.

- `KomorebiPaperVIII.approximate_packet_power_bound`; `KrylovPackets.lean`, line 170.
- `KomorebiPaperVIII.approximate_packet_krylov_distance`; `KrylovPackets.lean`, line 183.

12.36 7.4 — Block Jacobi condition bound

Traditional original item 7.4, current statement 7.4. Printed page 25, PDF page 31.

Recorded status. checked.

Recorded hypotheses and scope. Nonempty finite complex matrices, D positive definite and D+K positive definite. D^(-1/2) is the actual inverse of CFC.sqrt D. With ||D^(-1/2) K D^(-1/2)||_2 <= eta <1, proves the condition product ||B||_2*||B^(-1)||_2 <=(1+eta)/(1-eta). Norms are the Euclidean operator norm. The inverse estimate does not require a separately assumed Hermitian perturbation.

- `KomorebiPaperVIII.invSqrt_isUnit`; `BlockJacobi.lean`, line 59.
- `KomorebiPaperVIII.invSqrt_normalizes`; `BlockJacobi.lean`, line 65.
- `KomorebiPaperVIII.normalized_decomposition`; `BlockJacobi.lean`, line 80.
- `KomorebiPaperVIII.block_jacobi_condition`; `BlockJacobi.lean`, line 87.

12.37 7.5 — Preconditioned conjugate gradient convergence

Traditional original item 7.5, current statement 7.5. Printed page 25, PDF page 31.

Recorded status. external_input_not_independently_formalized.

Recorded hypotheses and scope. Classical input tracked separately; see the boundary discussion.

12.38 7.6 — Certified packet preconditioning

Traditional original item 7.6, current statement 7.6. Printed page 25, PDF page 31.

Recorded status. checked.

Recorded hypotheses and scope. Monotonicity of the actual sqrt-based CG factor and its value at the normalized matrix condition number. Propagates an explicit classical energy-error estimate to the certified eta bound. Classical input 7.5 (CG convergence algorithm theorem) is not reproved or silently assumed as a new axiom.

- `KomorebiPaperVIII.matrix_condition_ge_one`; `BlockJacobi.lean`, line 97.
- `KomorebiPaperVIII.cgFactor_mono`; `BlockJacobi.lean`, line 106.
- `KomorebiPaperVIII.certified_packet_factor`; `BlockJacobi.lean`, line 126.
- `KomorebiPaperVIII.certified_energy_error`; `BlockJacobi.lean`, line 134.

12.39 8.1 — Packet perturbation certificate

Traditional original item 8.1, current statement 8.1. Printed page 26, PDF page 32.

Recorded status. checked.

Recorded hypotheses and scope. Actual continuous linear maps and their operator norm. Source support is assumed only for errors, a weaker sufficient hypothesis. Vector and uniform operator bounds. Not an IEEE/CUDA implementation verification.

- `KomorebiPaperVIII.packet_perturbation_bound`; `PacketPerturbation.lean`, line 35.
- `KomorebiPaperVIII.packet_relative_perturbation_bound`; `PacketPerturbation.lean`, line 42.
- `KomorebiPaperVIII.packet_operator_perturbation_bound`; `PacketPerturbation.lean`, line 51.
- `KomorebiPaperVIII.contractive_packet_operator_bound`; `PacketPerturbation.lean`, line 59.

12.40 8.2 — Floating point matrix product model

Traditional original item 8.2, current statement 8.2. Printed page 27, PDF page 33.

Recorded status. external_input_not_independently_formalized.

Recorded hypotheses and scope. Classical input tracked separately; see the boundary discussion.

12.41 8.3 — Mixed precision product chain

Traditional original item 8.3, current statement 8.3. Printed page 27, PDF page 33.

Recorded status. checked.

Recorded hypotheses and scope. Stagewise packet bounds are uniform operator-norm certificates on a nonzero normed space, not vector-specific bounds reused as operator norms. The exact-product perturbation theorem propagates those certificates. Vector error follows by multiplying the operator bound by the input norm.

- `KomorebiPaperVIII.mixed_precision_product_chain`; `ProductChainError.lean`, line 164.
- `KomorebiPaperVIII.chain_action_error`; `ProductChainError.lean`, line 180.

12.42 8.4 — Continuous precision allocation

Traditional original item 8.4, current statement 8.4. Printed page 27, PDF page 33.

Recorded status. checked.

Recorded hypotheses and scope. Finite nonempty edge set, strictly positive sensitivities, costs, and budget. Actual sum c*log(1/eps). Feasibility, global optimality, strict inequality away from formula, and unique minimizer. No KKT or convexity result is assumed.

- `KomorebiPaperVIII.optimal_tolerance_feasible`; `ContinuousPrecision.lean`, line 48.
- `KomorebiPaperVIII.precision_allocation_optimal`; `ContinuousPrecision.lean`, line 80.
- `KomorebiPaperVIII.precision_allocation_strict`; `ContinuousPrecision.lean`, line 100.
- `KomorebiPaperVIII.continuous_precision_unique_minimizer`; `ContinuousPrecision.lean`, line 131.

12.43 8.5 — Clipped precision allocation

Traditional original item 8.5, current statement 8.5. Printed page 28, PDF page 34.

Recorded status. checked.

Recorded hypotheses and scope. Finite nonempty set; strictly positive sensitivities, costs and caps; 0<budget<=total capped tolerance. Existence by IVT; global strict supporting-log inequality and unique minimizing vector. Unique positive multiplier only strictly below the maximum; endpoint nonuniqueness checked.

- `KomorebiPaperVIII.clipped_scale_exists`; `ClippedPrecision.lean`, line 74.
- `KomorebiPaperVIII.clipped_unique_minimizer`; `ClippedPrecision.lean`, line 211.
- `KomorebiPaperVIII.clipped_multiplier_exists_unique`; `ClippedPrecision.lean`, line 250.
- `KomorebiPaperVIII.clipped_endpoint_nonunique_multiplier`; `ClippedPrecision.lean`, line 244.

12.44 8.7 — Exact discrete precision program

Traditional original item 8.7, current statement 8.7. Printed page 29, PDF page 35.

Recorded status. checked.

Recorded hypotheses and scope. Independent legal assignments choose one listed option per edge. Nonnegative integer weights and exact ordered-field costs. Exact-weight states plus the final budget scan give the optimum with attainment or exact infeasibility. The residual-budget subtraction is

guarded. StoredBudgetCompiler proves the actual array-backed implementation equal to this specification.

- `KomorebiPaperVIII.BudgetDP.discrete_precision_program`; `DiscretePrecision.lean`, line 96.
- `KomorebiPaperVIII.BudgetDP.discreteBudget_some_iff`; `DiscretePrecision.lean`, line 110.
- `KomorebiPaperVIII.BudgetDP.discreteBudget_none_iff`; `DiscretePrecision.lean`, line 124.
- `KomorebiPaperVIII.BudgetDP.storedDiscreteBudget_correct`; `StoredBudgetCompiler.lean`, line 186.

12.45 9.1 — Simultaneous spectral decomposition

Traditional original item 9.1, current statement 9.1. Printed page 30, PDF page 36.

Recorded status. external_input_not_independently_formalized.

Recorded hypotheses and scope. Classical input tracked separately; see the boundary discussion.

12.46 9.2 — Exact block discovery certificate

Traditional original item 9.2, current statement 9.2. Printed page 30, PDF page 36.

Recorded status. checked.

Recorded hypotheses and scope. Actual finite complex matrices and arbitrary complete orthogonal projector family, not only standard coordinate projectors. Hilbert-Schmidt orthogonality via adjoint products and cyclic trace proves the squared Frobenius residual identity. Separate Euclidean operator-norm triangle inequality certificate. This certifies a supplied packet system, not autonomous discovery.

- `KomorebiPaperVIII.compression_residual`; `DiscoveryCertificate.lean`, line 49.
- `KomorebiPaperVIII.block_trace_orthogonal`; `DiscoveryCertificate.lean`, line 54.
- `KomorebiPaperVIII.discovery_frobenius_certificate`; `DiscoveryCertificate.lean`, line 126.
- `KomorebiPaperVIII.discovery_operator_certificate`; `DiscoveryCertificate.lean`, line 144.

12.47 9.4 — Exact Lagrangian packet compiler

Traditional original item 9.4, current statement 9.4. Printed page 31, PDF page 37.

Recorded status. checked.

Recorded hypotheses and scope. Arbitrary finite branching rooted trees, independent inductively defined cuts. Returned cut attains the recurrence and beats every legal cut. Generic ordered field covers rationals and reals. No universal discovery/approximation claim.

- `KomorebiPaperVIII.CandidateTree.exact_tree_compiler`; `ExactTreeCompiler.lean`, line 79.
- `KomorebiPaperVIII.CandidateTree.compile_attains`; `ExactTreeCompiler.lean`, line 69.
- `KomorebiPaperVIII.CandidateTree.exact_lagrangian_compiler`; `ExactTreeCompiler.lean`, line 102.

12.48 9.5 — Shared compilation for an operator family

Traditional original item 9.5, current statement 9.5. Printed page 31, PDF page 37.

Recorded status. checked.

Recorded hypotheses and scope. Exact weighted family aggregation at pruning level including setup and split overhead. Proved even without sign restrictions; intended application uses nonnegative weights.

- `KomorebiPaperVIII.CandidateTree.cutValue_weighted_family`; `ExactTreeCompiler.lean`, line 113.
- `KomorebiPaperVIII.CandidateTree.shared_family_compiler`; `ExactTreeCompiler.lean`, line 125.

12.49 9.6 — Monotone Lagrangian frontier

Traditional original item 9.6, current statement 9.6. Printed page 32, PDF page 38.

Recorded status. checked.

Recorded hypotheses and scope. Any two optimal cuts, 0 <= lambda1 < lambda2; error nonincreasing and cost nondecreasing. Optimality is discharged by the constructive compiler theorem for its returned cuts.

- `KomorebiPaperVIII.CandidateTree.monotone_lagrangian_frontier`; `ExactTreeCompiler.lean`, line 142.

12.50 9.7 — Supported Pareto points

Traditional original item 9.7, current statement 9.7. Printed page 32, PDF page 38.

Recorded status. checked.

Recorded hypotheses and scope. Positive scalarization parameter. Every true optimum is undominated. TreeCompilerUpdates.unsupported_pareto_example supplies the rational three-point unsupported-frontier inequality. Quantized-budget compiler remains unformalized.

- `KomorebiPaperVIII.CandidateTree.positive_lagrangian_pareto`; `ExactTreeCompiler.lean`, line 157.

12.51 9.8 — Quantized budget dynamic program

Traditional original item 9.8, current statement 9.8. Printed page 33, PDF page 39.

Recorded status. checked.

Recorded hypotheses and scope. The actual candidate tree and independent Cut type from the previous block, with arbitrary nonempty finite branching. Min-plus convolution over integer weights proves lower bounds, attainment, and infeasibility. StoredBudgetCompiler certifies the actual cached array-backed output. The implementation returns the minimum cost, not a backtracked cut.

- `KomorebiPaperVIII.CandidateTree.budgetTable_correct`; `QuantizedTreeCompiler.lean`, line 36.
- `KomorebiPaperVIII.CandidateTree.quantized_budget_compiler`; `QuantizedTreeCompiler.lean`, line 91.
- `KomorebiPaperVIII.CandidateTree.compileBudget_some_iff`; `QuantizedTreeCompiler.lean`, line 107.
- `KomorebiPaperVIII.CandidateTree.compileBudget_none_iff`; `QuantizedTreeCompiler.lean`, line 122.

- `KomorebiPaperVIII.CandidateTree.storedBudget_minimum`; `StoredBudgetCompiler.lean`, line 147.

12.52 9.9 — Pseudo polynomial complexity

Traditional original item 9.9, current statement 9.9. Printed page 33, PDF page 39.

Recorded status. checked.

Recorded hypotheses and scope. Array-backed tables cache child results. Concrete convolution candidate indices have at most (B+1)^2 elements. The declared unit-cost tabulation model is bounded by nodeCount*(3*(B+1)+(B+1)^2). Root minimum scan is an additional linear pass. This is pseudo-polynomial in numeric B and does not count exact-arithmetic bit complexity, compiler internals, or hardware instructions.

- `KomorebiPaperVIII.CandidateTree.merge_trial_count`; `StoredBudgetCompiler.lean`, line 222.
- `KomorebiPaperVIII.CandidateTree.merge_trial_bound`; `StoredBudgetCompiler.lean`, line 226.
- `KomorebiPaperVIII.CandidateTree.tabulation_pseudopolynomial`; `StoredBudgetCompiler.lean`, line 256.

12.53 9.11 — Compiler stability

Traditional original item 9.11, current statement 9.11. Printed page 33, PDF page 39.

Recorded status. checked.

Recorded hypotheses and scope. Actual solved keep/split scores satisfy strict half-gap inequalities at every internal node. Entire returned cut is unchanged. Ties cannot satisfy these strict hypotheses; no robustness at ties is claimed.

- `KomorebiPaperVIII.CandidateTree.half_gap_order`; `TreeCompilerUpdates.lean`, line 18.
- `KomorebiPaperVIII.CandidateTree.compiler_stability`; `TreeCompilerUpdates.lean`, line 45.

12.54 9.12 — Local recompilation

Traditional original item 9.12, current statement 9.12. Printed page 34, PDF page 40.

Recorded status. checked.

Recorded hypotheses and scope. Explicit one-hole context with cached sibling values. Replacing a subtree models localized data changes. Cache validity is constructed by refresh, not the new optimum. updateValue reads only new subtree score and caches on ancestor frames. Exact new global optimum and attaining cut proved.

- `KomorebiPaperVIII.CandidateTree.CachedContext.refresh_valid`; `TreeCompilerUpdates.lean`, line 95.
- `KomorebiPaperVIII.CandidateTree.CachedContext.updateValue_correct`; `TreeCompilerUpdates.lean`, line 109.
- `KomorebiPaperVIII.CandidateTree.CachedContext.sibling_tree_unchanged`; `TreeCompilerUpdates.lean`, line 128.
- `KomorebiPaperVIII.CandidateTree.CachedContext.sibling_value_unchanged`; `TreeCompilerUpdates.lean`, line 134.
- `KomorebiPaperVIII.CandidateTree.CachedContext.local_recompilation`; `TreeCompilerUpdates.lean`, line 142.

12.55 10.2 — Launch reduction by shape

Traditional original item 10.2, current statement 10.2. Printed page 35, PDF page 41.

Recorded status. checked.

Recorded hypotheses and scope. Finite grouping by complete shape signature, partition/fiberwise sum equals original output and work. One launch per active class is a backend capability assumption, not verified CUDA code.

- `KomorebiPaperVIII.shape_launch_count`; `SchedulingContracts.lean`, line 80.
- `KomorebiPaperVIII.grouped_execution_output`; `SchedulingContracts.lean`, line 89.
- `KomorebiPaperVIII.grouped_execution_work`; `SchedulingContracts.lean`, line 95.

12.56 10.3 — Acyclic schedule depth

Traditional original item 10.3, current statement 10.3. Printed page 35, PDF page 41.

Recorded status. checked.

Recorded hypotheses and scope. Finite acyclic task graph, unit positive-integer stages, dependency precedence, and unlimited concurrency. Constructed canonical level schedule is pointwise earliest and has minimal makespan; its depth is attained by an actual path when nonempty. Empty graph depth is zero. No weighted-task/resource-limited scheduling claim.

- `KomorebiPaperVIII.DAG.Graph.level_valid`; `GraphSchedules.lean`, line 126.
- `KomorebiPaperVIII.DAG.Graph.level_minimal`; `GraphSchedules.lean`, line 129.
- `KomorebiPaperVIII.DAG.Graph.optimal_depth`; `GraphSchedules.lean`, line 150.
- `KomorebiPaperVIII.DAG.Graph.depth_attained`; `GraphSchedules.lean`, line 164.
- `KomorebiPaperVIII.DAG.Graph.same_level_independent`; `GraphSchedules.lean`, line 181.
- `KomorebiPaperVIII.DAG.Graph.empty_depth`; `GraphSchedules.lean`, line 177.

12.57 10.4 — Amortization threshold

Traditional original item 10.4, current statement 10.4. Printed page 35, PDF page 41.

Recorded status. checked.

Recorded hypotheses and scope. Actual declared setup+R*candidate vs R*baseline inequality. Positive per-use advantage; setup nonnegative for smallest natural reuse floor+1. Equality is break-even, not profitability.

- `KomorebiPaperVIII.amortization_threshold`; `SchedulingContracts.lean`, line 101.
- `KomorebiPaperVIII.no_amortization`; `SchedulingContracts.lean`, line 107.
- `KomorebiPaperVIII.integer_amortization_threshold`; `SchedulingContracts.lean`, line 117.
- `KomorebiPaperVIII.minimum_reuse_attains`; `SchedulingContracts.lean`, line 125.

12.58 10.5 — Graph replay threshold

Traditional original item 10.5, current statement 10.5. Printed page 35, PDF page 41.

Recorded status. checked.

Recorded hypotheses and scope. Capture plus repeated replay vs repeated ordinary execution under replay<ordinary. Pure timing-model identity, not new GPU data.

- `KomorebiPaperVIII.graph_replay_threshold`; `SchedulingContracts.lean`, line 137.

12.59 10.6 — Materialization threshold

Traditional original item 10.6, current statement 10.6. Printed page 36, PDF page 42.

Recorded status. checked.

Recorded hypotheses and scope. Nonnegative materialization cost and reuse count. Strict profitability iff explicit per-use time is lower and reuse exceeds the ratio.

- `KomorebiPaperVIII.materialization_threshold`; `SchedulingContracts.lean`, line 142.

12.60 11.3 — Fair matching is an equivalence relation

Traditional original item 11.3, current statement 11.3. Printed page 36, PDF page 42.

Recorded status. checked.

Recorded hypotheses and scope. Equality of complete case, output, correctness reference, and timing-scope metadata. No inference of actual timing accuracy from matching metadata.

- `KomorebiPaperVIII.fair_match_iff_eq`; `SchedulingContracts.lean`, line 32.
- `KomorebiPaperVIII.fair_matching_equivalence`; `SchedulingContracts.lean`, line 40.

13 All named theorem and construction audits

The saved inventory contains 458 names: 392 theorems and 66 additional construction audits. The following catalog resolves namespaces from active source and records every audited name, its declaration kind, and its line. Theorem names include supporting lemmas and scope controls. Construction audits inspect transitive axioms of definitions, abbreviations, or structures; they are counted separately. The full source and the individual saved axiom sets are preserved with the edition.

13.1 BandedCount: 9 theorems; 4 construction audits

Source: `formalization/BandedCount.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.bandCount_card`	theorem; 24
`KomorebiPaperVIII.finite_prefix_count`	theorem; 27
`KomorebiPaperVIII.bandCount_succ`	theorem; 38
`KomorebiPaperVIII.bandCount_full`	theorem; 50
`KomorebiPaperVIII.bandCount_balance`	theorem; 58
`KomorebiPaperVIII.banded_block_count`	theorem; 73
`KomorebiPaperVIII.banded_storage_count`	theorem; 85
`KomorebiPaperVIII.banded_operation_count`	theorem; 90
`KomorebiPaperVIII.banded_diagonal_case`	theorem; 96
`KomorebiPaperVIII.BandedAddress`	abbrev; 17
`KomorebiPaperVIII.BandedStorage`	abbrev; 80
`KomorebiPaperVIII.BandedOperation`	abbrev; 83
`KomorebiPaperVIII.bandCount`	def; 20

13.2 BlockJacobi: 12 theorems; 3 construction audits

Source: `formalization/BlockJacobi.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.inverse_near_one_bound`	theorem; 20
`KomorebiPaperVIII.near_one_condition_bound`	theorem; 37
`KomorebiPaperVIII.invSqrt_isUnit`	theorem; 59
`KomorebiPaperVIII.invSqrt_normalizes`	theorem; 65
`KomorebiPaperVIII.normalized_decomposition`	theorem; 80
`KomorebiPaperVIII.block_jacobi_condition`	theorem; 87
`KomorebiPaperVIII.matrix_condition_ge_one`	theorem; 97
`KomorebiPaperVIII.cgFactor_mono`	theorem; 106
`KomorebiPaperVIII.cgFactor_nonneg`	theorem; 114
`KomorebiPaperVIII.cgFactor_lt_one`	theorem; 119
`KomorebiPaperVIII.certified_packet_factor`	theorem; 126
`KomorebiPaperVIII.certified_energy_error`	theorem; 134
`KomorebiPaperVIII.invSqrt`	def; 55

Audited declaration	Kind; line
`KomorebiPaperVIII.matrixCondition`	def; 57
`KomorebiPaperVIII.cgFactor`	def; 104

13.3 BoundaryChecks: 6 theorems; 0 construction audits

Source: `formalization/BoundaryChecks.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.cancellation_square_zero`	theorem; 29
`KomorebiPaperVIII.cancellation_block_nonzero`	theorem; 34
`KomorebiPaperVIII.paths_can_cancel`	theorem; 42
`KomorebiPaperVIII.positive_allocation_budget`	theorem; 52
`KomorebiPaperVIII.no_positive_allocation_nonpositive_budget`	theorem; 58
`KomorebiPaperVIII.one_edge_zero_budget_infeasible`	theorem; 66

13.4 BudgetConvolution: 10 theorems; 0 construction audits

Source: `formalization/BudgetConvolution.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.BudgetDP.minimum_congr`	theorem; 50
`KomorebiPaperVIII.BudgetDP.choose_correct`	theorem; 57
`KomorebiPaperVIII.BudgetDP.add_correct`	theorem; 78
`KomorebiPaperVIII.BudgetDP.bestOver_correct`	theorem; 98
`KomorebiPaperVIII.BudgetDP.convolution_correct`	theorem; 112
`KomorebiPaperVIII.BudgetDP.singletonTable_correct`	theorem; 136
`KomorebiPaperVIII.BudgetDP.combine_correct`	theorem; 148
`KomorebiPaperVIII.BudgetDP.withinBudget_correct`	theorem; 188
`KomorebiPaperVIII.BudgetDP.minimum_some_iff`	theorem; 197
`KomorebiPaperVIII.BudgetDP.minimum_none_iff`	theorem; 208

13.5 CirculantFourier: 13 theorems; 1 construction audits

Source: `formalization/CirculantFourier.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.fourier_scale_normalization`	theorem; 22
`KomorebiPaperVIII.fourier_scale_sqrt`	theorem; 33
`KomorebiPaperVIII.fourier_matrix_exponential`	theorem; 39
`KomorebiPaperVIII.fourier_action`	theorem; 49

Audited declaration	Kind; line
`KomorebiPaperVIII.fourier_adjoint_entry`	theorem; 59
`KomorebiPaperVIII.fourier_adjoint_action`	theorem; 66
`KomorebiPaperVIII.fourier_unitary`	theorem; 83
`KomorebiPaperVIII.dft_circulant`	theorem; 105
`KomorebiPaperVIII.fourier_circulant_intertwine`	theorem; 128
`KomorebiPaperVIII.circulant_spectral_dft`	theorem; 138
`KomorebiPaperVIII.normalized_fourier_eigenvalues`	theorem; 145
`KomorebiPaperVIII.circulant_spectral_representation`	theorem; 150
`KomorebiPaperVIII.circulant_fourier_application`	theorem; 155
`KomorebiPaperVIII.fourierMatrix`	def; 19

13.6 ClippedPrecision: 17 theorems; 2 construction audits

Source: `formalization/ClippedPrecision.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.clipped_positive`	theorem; 29
`KomorebiPaperVIII.clipped_total_continuous`	theorem; 34
`KomorebiPaperVIII.clipped_total_mono`	theorem; 38
`KomorebiPaperVIII.clipped_total_zero`	theorem; 47
`KomorebiPaperVIII.saturation_scale_positive`	theorem; 54
`KomorebiPaperVIII.saturation_scale_attains`	theorem; 60
`KomorebiPaperVIII.clipped_scale_exists`	theorem; 74
`KomorebiPaperVIII.clipped_total_strict_below_cap`	theorem; 95
`KomorebiPaperVIII.clipped_scale_unique`	theorem; 116
`KomorebiPaperVIII.clipped_supporting_inequality`	theorem; 131
`KomorebiPaperVIII.clipped_relative_sum_le_zero`	theorem; 149
`KomorebiPaperVIII.clipped_allocation_optimal`	theorem; 165
`KomorebiPaperVIII.clipped_allocation_strict`	theorem; 182
`KomorebiPaperVIII.clipped_unique_minimizer`	theorem; 211
`KomorebiPaperVIII.clipped_multiplier_formula`	theorem; 236
`KomorebiPaperVIII.clipped_endpoint_nonunique_multiplier`	theorem; 244
`KomorebiPaperVIII.clipped_multiplier_exists_unique`	theorem; 250
`KomorebiPaperVIII.clippedTolerance`	def; 19
`KomorebiPaperVIII.saturationScale`	def; 52

13.7 CompactRangeFactors: 7 theorems; 5 construction audits

Source: `formalization/CompactRangeFactors.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.compactDim_eq_rank`	theorem; 30

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.compactQ_apply`	theorem; 37
`KomorebiPaperVIII.SmallCore.compactQ_isometry`	theorem; 42
`KomorebiPaperVIII.SmallCore.compact_factorization`	theorem; 51
`KomorebiPaperVIII.SmallCore.compactR_is_compression`	theorem; 60
`KomorebiPaperVIII.SmallCore.compactR_full_row_rank`	theorem; 67
`KomorebiPaperVIII.SmallCore.compact_rank_zero`	theorem; 77
`KomorebiPaperVIII.SmallCore.columnBasis`	def; 19
`KomorebiPaperVIII.SmallCore.columnSpace`	def; 16
`KomorebiPaperVIII.SmallCore.compactDim`	def; 18
`KomorebiPaperVIII.SmallCore.compactQ`	def; 22
`KomorebiPaperVIII.SmallCore.compactR`	def; 25

13.8 CompactTruncation: 5 theorems; 2 construction audits

Source: `formalization/CompactTruncation.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.keptColumn_card`	theorem; 16
`KomorebiPaperVIII.SmallCore.rootKept_tail_zero`	theorem; 23
`KomorebiPaperVIII.SmallCore.drop_zero_columns`	theorem; 31
`KomorebiPaperVIII.SmallCore.rootMask_square`	theorem; 46
`KomorebiPaperVIII.SmallCore.thin_factor_product`	theorem; 53
`KomorebiPaperVIII.SmallCore.KeptColumn`	abbrev; 14
`KomorebiPaperVIII.SmallCore.rootMask`	def; 19

13.9 ContinuousPrecision: 9 theorems; 1 construction audits

Source: `formalization/ContinuousPrecision.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.total_cost_positive`	theorem; 27
`KomorebiPaperVIII.optimal_tolerance_positive`	theorem; 31
`KomorebiPaperVIII.optimal_tolerance_budget`	theorem; 36
`KomorebiPaperVIII.optimal_tolerance_feasible`	theorem; 48
`KomorebiPaperVIII.relative_budget_identity`	theorem; 54
`KomorebiPaperVIII.precision_objective_difference`	theorem; 68
`KomorebiPaperVIII.precision_allocation_optimal`	theorem; 80
`KomorebiPaperVIII.precision_allocation_strict`	theorem; 100
`KomorebiPaperVIII.continuous_precision_unique_minimizer`	theorem; 131
`KomorebiPaperVIII.optimalTolerance`	def; 21

13.10 DiagonalTruncation: 9 theorems; 6 construction audits

Source: `formalization/DiagonalTruncation.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.kept_nonneg`	theorem; 24
`KomorebiPaperVIII.SmallCore.discarded_nonneg`	theorem; 27
`KomorebiPaperVIII.SmallCore.diagonal_residual`	theorem; 30
`KomorebiPaperVIII.SmallCore.rank_kept_le`	theorem; 38
`KomorebiPaperVIII.SmallCore.rootKept_selfAdjoint`	theorem; 53
`KomorebiPaperVIII.SmallCore.rootKept_square`	theorem; 57
`KomorebiPaperVIII.SmallCore.discarded_le_next`	theorem; 66
`KomorebiPaperVIII.SmallCore.spectral_diagonal_tail`	theorem; 80
`KomorebiPaperVIII.SmallCore.frobenius_diagonal_tail`	theorem; 104
`KomorebiPaperVIII.SmallCore.diag`	def; 15
`KomorebiPaperVIII.SmallCore.discarded`	def; 14
`KomorebiPaperVIII.SmallCore.kept`	def; 13
`KomorebiPaperVIII.SmallCore.nextSingular`	def; 17
`KomorebiPaperVIII.SmallCore.rootKept`	def; 21
`KomorebiPaperVIII.SmallCore.tailEnergy`	def; 19

13.11 DiscoveryCertificate: 8 theorems; 1 construction audits

Source: `formalization/DiscoveryCertificate.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.compression_decomposition`	theorem; 30
`KomorebiPaperVIII.compression_residual`	theorem; 49
`KomorebiPaperVIII.block_trace_orthogonal`	theorem; 54
`KomorebiPaperVIII.offDiagonal_trace_orthogonal`	theorem; 83
`KomorebiPaperVIII.frobenius_sq_trace`	theorem; 97
`KomorebiPaperVIII.frobenius_sum_sq`	theorem; 108
`KomorebiPaperVIII.discovery_frobenius_certificate`	theorem; 126
`KomorebiPaperVIII.discovery_operator_certificate`	theorem; 144
`KomorebiPaperVIII.diagonalCompression`	def; 24

13.12 DiscretePrecision: 8 theorems; 2 construction audits

Source: `formalization/DiscretePrecision.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.BudgetDP.discreteTable_correct`	theorem; 45
`KomorebiPaperVIII.BudgetDP.discrete_precision_program`	theorem; 96

Audited declaration	Kind; line
`KomorebiPaperVIII.BudgetDP.discreteBudget_some_iff`	theorem; 110
`KomorebiPaperVIII.BudgetDP.discreteBudget_none_iff`	theorem; 124
`KomorebiPaperVIII.BudgetDP.over_budget_option_rejected`	theorem; 130
`KomorebiPaperVIII.BudgetDP.zero_budget_option_accepted`	theorem; 134
`KomorebiPaperVIII.BudgetDP.empty_precision_options_rejected`	theorem; 139
`KomorebiPaperVIII.BudgetDP.empty_precision_problem`	theorem; 143
`KomorebiPaperVIII.BudgetDP.discreteTable`	def; 38
`KomorebiPaperVIII.BudgetDP.discreteBudget`	def; 92

13.13 ExactTreeCompiler: 12 theorems; 2 construction audits

Source: `formalization/ExactTreeCompiler.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.CandidateTree.optimalValue_le_cut`	theorem; 57
`KomorebiPaperVIII.CandidateTree.compile_attains`	theorem; 69
`KomorebiPaperVIII.CandidateTree.exact_tree_compiler`	theorem; 79
`KomorebiPaperVIII.CandidateTree.compile_minimizes`	theorem; 85
`KomorebiPaperVIII.CandidateTree.lagrangian_decomposition`	theorem; 92
`KomorebiPaperVIII.CandidateTree.exact_lagrangian_compiler`	theorem; 102
`KomorebiPaperVIII.CandidateTree.cutValue_weighted_family`	theorem; 113
`KomorebiPaperVIII.CandidateTree.shared_family_compiler`	theorem; 125
`KomorebiPaperVIII.CandidateTree.monotone_lagrangian_frontier`	theorem; 142
`KomorebiPaperVIII.CandidateTree.positive_lagrangian_pareto`	theorem; 157
`KomorebiPaperVIII.CandidateTree.keep_decision_stable`	theorem; 170
`KomorebiPaperVIII.CandidateTree.split_decision_stable`	theorem; 177
`KomorebiPaperVIII.CandidateTree.compile`	def; 50
`KomorebiPaperVIII.CandidateTree.optimalValue`	def; 45

13.14 FrobeniusOptimality: 5 theorems; 0 construction audits

Source: `formalization/FrobeniusOptimality.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.tail_indicator_sum`	theorem; 17
`KomorebiPaperVIII.SmallCore.weighted_tail_lower`	theorem; 35
`KomorebiPaperVIII.SmallCore.diagonal_frame_energy`	theorem; 65
`KomorebiPaperVIII.SmallCore.diagonal_frobenius_lower`	theorem; 76
`KomorebiPaperVIII.SmallCore.diagonal_frobenius_best`	theorem; 106

13.15 GraphSchedules: 17 theorems; 2 construction audits

Source: `formalization/GraphSchedules.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.DAG.Graph.mem_ancestors`	theorem; 32
`KomorebiPaperVIII.DAG.Graph.ancestors_strict`	theorem; 37
`KomorebiPaperVIII.DAG.Graph.ancestors_card_lt`	theorem; 51
`KomorebiPaperVIII.DAG.Graph.wellFounded`	theorem; 55
`KomorebiPaperVIII.DAG.Graph.level_eq`	theorem; 64
`KomorebiPaperVIII.DAG.Graph.level_positive`	theorem; 69
`KomorebiPaperVIII.DAG.Graph.edge_level`	theorem; 74
`KomorebiPaperVIII.DAG.Graph.path_level_bound`	theorem; 87
`KomorebiPaperVIII.DAG.Graph.level_attained`	theorem; 101
`KomorebiPaperVIII.DAG.Graph.level_valid`	theorem; 126
`KomorebiPaperVIII.DAG.Graph.level_minimal`	theorem; 129
`KomorebiPaperVIII.DAG.Graph.level_le_depth`	theorem; 146
`KomorebiPaperVIII.DAG.Graph.optimal_depth`	theorem; 150
`KomorebiPaperVIII.DAG.Graph.path_length_le_depth`	theorem; 156
`KomorebiPaperVIII.DAG.Graph.depth_attained`	theorem; 164
`KomorebiPaperVIII.DAG.Graph.empty_depth`	theorem; 177
`KomorebiPaperVIII.DAG.Graph.same_level_independent`	theorem; 181
`KomorebiPaperVIII.DAG.Graph.level`	def; 59
`KomorebiPaperVIII.DAG.Graph.depth`	def; 144

13.16 InverseSupport: 9 theorems; 1 construction audits

Source: `formalization/InverseSupport.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.reachable_refl`	theorem; 27
`KomorebiPaperVIII.reachable_step`	theorem; 29
`KomorebiPaperVIII.seed_mem_reachableSpace`	theorem; 45
`KomorebiPaperVIII.reachableSpace_invariant`	theorem; 54
`KomorebiPaperVIII.invariant_space_inverse`	theorem; 73
`KomorebiPaperVIII.inverse_block_zero_of_no_path`	theorem; 94
`KomorebiPaperVIII.inverse_support_inclusion`	theorem; 102
`KomorebiPaperVIII.unit_inverse_support`	theorem; 108
`KomorebiPaperVIII.matrix_inverse_support`	theorem; 116
`KomorebiPaperVIII.reachableSpace`	def; 35

13.17 KernelFrame: 8 theorems; 5 construction audits

Source: `formalization/KernelFrame.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.subspaceFrame_apply`	theorem; 20
`KomorebiPaperVIII.SmallCore.subspaceFrame_isometry`	theorem; 26
`KomorebiPaperVIII.SmallCore.kernelFrame_isometry`	theorem; 40
`KomorebiPaperVIII.SmallCore.mul_kernelFrame`	theorem; 44
`KomorebiPaperVIII.SmallCore.kernel_rank_nullity`	theorem; 56
`KomorebiPaperVIII.SmallCore.rowWeight_nonneg`	theorem; 66
`KomorebiPaperVIII.SmallCore.rowWeight_sum`	theorem; 70
`KomorebiPaperVIII.SmallCore.rowWeight_le_one`	theorem; 83
`KomorebiPaperVIII.SmallCore.kernelDim`	def; 36
`KomorebiPaperVIII.SmallCore.kernelFrame`	def; 37
`KomorebiPaperVIII.SmallCore.kernelSpace`	def; 34
`KomorebiPaperVIII.SmallCore.rowWeight`	def; 63
`KomorebiPaperVIII.SmallCore.subspaceFrame`	def; 14

13.18 KrylovPackets: 11 theorems; 0 construction audits

Source: `formalization/KrylovPackets.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.packet_diagonal_commutes`	theorem; 31
`KomorebiPaperVIII.power_on_packet`	theorem; 50
`KomorebiPaperVIII.krylov_generator_mem`	theorem; 71
`KomorebiPaperVIII.invariant_packet_power_expansion`	theorem; 76
`KomorebiPaperVIII.invariant_packet_krylov`	theorem; 89
`KomorebiPaperVIII.local_krylov_le_range`	theorem; 102
`KomorebiPaperVIII.project_packet_sum`	theorem; 115
`KomorebiPaperVIII.local_krylov_direct`	theorem; 131
`KomorebiPaperVIII.power_difference_bound_succ`	theorem; 144
`KomorebiPaperVIII.approximate_packet_power_bound`	theorem; 170
`KomorebiPaperVIII.approximate_packet_krylov_distance`	theorem; 183

13.19 LowRankAlgebra: 8 theorems; 0 construction audits

Source: `formalization/LowRankAlgebra.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.packet_diagonal_product`	theorem; 33
`KomorebiPaperVIII.bdlr_product`	theorem; 57

Audited declaration	Kind; line
`KomorebiPaperVIII.bdlr_product_rank_bound`	theorem; 68
`KomorebiPaperVIII.bdlr_apply`	theorem; 79
`KomorebiPaperVIII.bdlr_woodbury`	theorem; 89
`KomorebiPaperVIII.bdlr_update_isUnit`	theorem; 97
`KomorebiPaperVIII.bdlr_woodbury_solve`	theorem; 110
`KomorebiPaperVIII.bdlr_inverse_norm_bound`	theorem; 126

13.20 LowRankChains: 5 theorems; 1 construction audits

Source: `formalization/LowRankChains.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.chainColumns_card`	theorem; 34
`KomorebiPaperVIII.chainFactors_correct`	theorem; 56
`KomorebiPaperVIII.bdlr_chain_rank_bound`	theorem; 69
`KomorebiPaperVIII.packet_diagonal_one`	theorem; 82
`KomorebiPaperVIII.packet_diagonal_chain`	theorem; 87
`KomorebiPaperVIII.chainFactors`	def; 48

13.21 PacketCostModels: 22 theorems; 0 construction audits

Source: `formalization/PacketCostModels.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.packet_storage_count`	theorem; 27
`KomorebiPaperVIII.packet_application_count`	theorem; 31
`KomorebiPaperVIII.dense_storage_count`	theorem; 36
`KomorebiPaperVIII.dense_application_count`	theorem; 39
`KomorebiPaperVIII.equal_diagonal_storage`	theorem; 44
`KomorebiPaperVIII.equal_diagonal_work`	theorem; 48
`KomorebiPaperVIII.bdlr_storage_count`	theorem; 58
`KomorebiPaperVIII.bdlr_application_count`	theorem; 68
`KomorebiPaperVIII.equal_bdlr_storage`	theorem; 75
`KomorebiPaperVIII.equal_bdlr_work`	theorem; 81
`KomorebiPaperVIII.equal_block_reduction`	theorem; 87
`KomorebiPaperVIII.equal_bdlr_reduction`	theorem; 91
`KomorebiPaperVIII.streaming_value_count`	theorem; 102
`KomorebiPaperVIII.streaming_byte_count`	theorem; 111
`KomorebiPaperVIII.roofline_time_bound`	theorem; 118
`KomorebiPaperVIII.roofline_work_rate`	theorem; 126
`KomorebiPaperVIII.dense_capacity_threshold`	theorem; 139
`KomorebiPaperVIII.bdlr_capacity_threshold`	theorem; 143

Audited declaration	Kind; line
`KomorebiPaperVIII.materialization_work_count`	theorem; 151
`KomorebiPaperVIII.materialization_quadratic_remainder`	theorem; 156
`KomorebiPaperVIII.sparse_low_rank_storage_count`	theorem; 164
`KomorebiPaperVIII.sparse_low_rank_work_count`	theorem; 169

13.22 PacketIR: 11 theorems; 1 construction audits

Source: `formalization/PacketIR.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.active_iff`	theorem; 35
`KomorebiPaperVIII.exact_active_evaluation`	theorem; 41
`KomorebiPaperVIII.exact_active_action`	theorem; 52
`KomorebiPaperVIII.exact_block_composition`	theorem; 58
`KomorebiPaperVIII.product_support_inclusion`	theorem; 63
`KomorebiPaperVIII.block_sum`	theorem; 79
`KomorebiPaperVIII.all_blocks_zero`	theorem; 83
`KomorebiPaperVIII.exact_adjoint_block`	theorem; 95
`KomorebiPaperVIII.coordinate_projector_apply`	theorem; 129
`KomorebiPaperVIII.exact_tensor_block`	theorem; 135
`KomorebiPaperVIII.exact_matrix_batch`	theorem; 145
`KomorebiPaperVIII.coordinatePackets`	def; 106

13.23 PacketPerturbation: 6 theorems; 0 construction audits

Source: `formalization/PacketPerturbation.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.packet_error_operator_identity`	theorem; 21
`KomorebiPaperVIII.packet_error_action_identity`	theorem; 28
`KomorebiPaperVIII.packet_perturbation_bound`	theorem; 35
`KomorebiPaperVIII.packet_relative_perturbation_bound`	theorem; 42
`KomorebiPaperVIII.packet_operator_perturbation_bound`	theorem; 51
`KomorebiPaperVIII.contractive_packet_operator_bound`	theorem; 59

13.24 ProductChainError: 13 theorems; 0 construction audits

Source: `formalization/ProductChainError.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.propagatedError_eq_sum`	theorem; 37
`KomorebiPaperVIII.propagatedError_mono`	theorem; 53
`KomorebiPaperVIII.norm_recurrence_bound`	theorem; 62
`KomorebiPaperVIII.product_discrepancy_step`	theorem; 75
`KomorebiPaperVIII.recompression_discrepancy_step`	theorem; 83
`KomorebiPaperVIII.chainProduct_const`	theorem; 90
`KomorebiPaperVIII.repeated_recompression_bound`	theorem; 100
`KomorebiPaperVIII.repeated_recompression_actual_error`	theorem; 114
`KomorebiPaperVIII.chainProduct_norm_bound`	theorem; 120
`KomorebiPaperVIII.product_chain_error_bound`	theorem; 129
`KomorebiPaperVIII.product_chain_perturbation`	theorem; 148
`KomorebiPaperVIII.mixed_precision_product_chain`	theorem; 164
`KomorebiPaperVIII.chain_action_error`	theorem; 180

13.25 QuantizedTreeCompiler: 5 theorems; 2 construction audits

Source: `formalization/QuantizedTreeCompiler.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.CandidateTree.budgetTable_correct`	theorem; 36
`KomorebiPaperVIII.CandidateTree.quantized_budget_compiler`	theorem; 91
`KomorebiPaperVIII.CandidateTree.compileBudget_some_iff`	theorem; 107
`KomorebiPaperVIII.CandidateTree.compileBudget_none_iff`	theorem; 122
`KomorebiPaperVIII.CandidateTree.compileBudget_monotone`	theorem; 129
`KomorebiPaperVIII.CandidateTree.budgetTable`	def; 27
`KomorebiPaperVIII.CandidateTree.compileBudget`	def; 86

13.26 QuantumPacketCore: 19 theorems; 0 construction audits

Source: `formalization/QuantumPacketCore.lean`.

Audited declaration	Kind; line
`KomorebiPaperVII.PacketFamily.idempotent`	theorem; 32
`KomorebiPaperVII.PacketFamily.reconstruct`	theorem; 38
`KomorebiPaperVII.PacketFamily.compose_blocks`	theorem; 41
`KomorebiPaperVII.PacketFamily.circuit_path_expansion`	theorem; 63
`KomorebiPaperVII.PacketFamily.circuit_reconstruction`	theorem; 70
`KomorebiPaperVII.PacketFamily.symmetry_no_transfer`	theorem; 76

Audited declaration	Kind; line
KomorebiPaperVII.PacketFamily.symmetry_diagonal	theorem; 80
KomorebiPaperVII.PacketFamily.dephase_idempotent	theorem; 90
KomorebiPaperVII.PacketFamily.adjoint_block	theorem; 114
KomorebiPaperVII.PacketFamily.unitarity_columns	theorem; 119
KomorebiPaperVII.PacketFamily.unitarity_rows	theorem; 134
KomorebiPaperVII.PacketFamily.symmetry_exponential_commutes	theorem; 156
KomorebiPaperVII.PacketFamily.symmetry_exponential_diagonal	theorem; 163
KomorebiPaperVII.PacketFamily.symmetry_exponential_no_transfer	theorem; 168
KomorebiPaperVII.matrix_unitarity_columns	theorem; 181
KomorebiPaperVII.matrix_unitarity_rows	theorem; 188
KomorebiPaperVII.dephase_trace_duality	theorem; 195
KomorebiPaperVII.measurement_invariance	theorem; 208
KomorebiPaperVII.dephase_trace_preserving	theorem; 213

13.27 RecompressionEndpoints: 6 theorems; 2 construction audits

Source: formalization/RecompressionEndpoints.lean.

Audited declaration	Kind; line
KomorebiPaperVIII.SmallCore.kept_zero	theorem; 40
KomorebiPaperVIII.SmallCore.kept_all	theorem; 44
KomorebiPaperVIII.SmallCore.tailEnergy_zero_of_full	theorem; 49
KomorebiPaperVIII.SmallCore.nextSingular_zero_of_full	theorem; 57
KomorebiPaperVIII.SmallCore.recompression_rank_zero	theorem; 62
KomorebiPaperVIII.SmallCore.recompression_full	theorem; 70
KomorebiPaperVIII.SmallCore.diagonalSVD	def; 18
KomorebiPaperVIII.SmallCore.zeroSVD	def; 30

13.28 SchedulingContracts: 17 theorems; 0 construction audits

Source: formalization/SchedulingContracts.lean.

Audited declaration	Kind; line
KomorebiPaperVIII.fair_match_iff_eq	theorem; 32
KomorebiPaperVIII.fair_matching_equivalence	theorem; 40
KomorebiPaperVIII.contract_conversion	theorem; 54
KomorebiPaperVIII.direct_compressed_consumption	theorem; 62
KomorebiPaperVIII.omitted_decode_understates_time	theorem; 69
KomorebiPaperVIII.shape_launch_count	theorem; 80
KomorebiPaperVIII.each_block_has_shape	theorem; 84
KomorebiPaperVIII.grouped_execution_output	theorem; 89
KomorebiPaperVIII.grouped_execution_work	theorem; 95

Audited declaration	Kind; line
`KomorebiPaperVIII.amortization_threshold`	theorem; 101
`KomorebiPaperVIII.no_amortization`	theorem; 107
`KomorebiPaperVIII.integer_amortization_threshold`	theorem; 117
`KomorebiPaperVIII.minimum_reuse_attains`	theorem; 125
`KomorebiPaperVIII.graph_replay_threshold`	theorem; 137
`KomorebiPaperVIII.materialization_threshold`	theorem; 142
`KomorebiPaperVIII.zero_setup_first_use`	theorem; 155
`KomorebiPaperVIII.exact_break_even_regression`	theorem; 159

13.29 SchurSupport: 4 theorems; 0 construction audits

Source: `formalization/SchurSupport.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.schur_correction_expansion`	theorem; 25
`KomorebiPaperVIII.schur_correction_support`	theorem; 33
`KomorebiPaperVIII.schur_support_inclusion`	theorem; 57
`KomorebiPaperVIII.schur_zero_of_no_route`	theorem; 76

13.30 SmallCoreNorms: 19 theorems; 4 construction audits

Source: `formalization/SmallCoreNorms.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.compress_lift`	theorem; 25
`KomorebiPaperVIII.SmallCore.lift_sub`	theorem; 33
`KomorebiPaperVIII.SmallCore.compress_sub`	theorem; 38
`KomorebiPaperVIII.SmallCore.rank_compress_le`	theorem; 43
`KomorebiPaperVIII.SmallCore.rank_lift_le`	theorem; 48
`KomorebiPaperVIII.SmallCore.rank_lift`	theorem; 53
`KomorebiPaperVIII.SmallCore.spectral_left_isometry`	theorem; 64
`KomorebiPaperVIII.SmallCore.spectral_right_isometry`	theorem; 73
`KomorebiPaperVIII.SmallCore.spectral_isometry_bound`	theorem; 82
`KomorebiPaperVIII.SmallCore.spectral_lift`	theorem; 90
`KomorebiPaperVIII.SmallCore.spectral_compress_le`	theorem; 96
`KomorebiPaperVIII.SmallCore.frobenius_rect_sq_trace`	theorem; 117
`KomorebiPaperVIII.SmallCore.frobenius_left_isometry`	theorem; 128
`KomorebiPaperVIII.SmallCore.frobenius_right_isometry`	theorem; 138
`KomorebiPaperVIII.SmallCore.frobenius_projection_pythagoras`	theorem; 148
`KomorebiPaperVIII.SmallCore.frobenius_left_compress_le`	theorem; 163
`KomorebiPaperVIII.SmallCore.frobenius_right_compress_le`	theorem; 169
`KomorebiPaperVIII.SmallCore.frobenius_lift`	theorem; 178

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.frobenius_compress_le`	theorem; 185
`KomorebiPaperVIII.SmallCore.compress`	def; 21
`KomorebiPaperVIII.SmallCore.frobeniusNorm`	def; 114
`KomorebiPaperVIII.SmallCore.lift`	def; 18
`KomorebiPaperVIII.SmallCore.spectralNorm`	def; 61

13.31 SmallCoreRecompression: 6 theorems; 3 construction audits

Source: `formalization/SmallCoreRecompression.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.isometry_comp`	theorem; 40
`KomorebiPaperVIII.SmallCore.lift_comp`	theorem; 47
`KomorebiPaperVIII.SmallCore.recompressed_product`	theorem; 64
`KomorebiPaperVIII.SmallCore.core_gram_eigenvectors`	theorem; 72
`KomorebiPaperVIII.SmallCore.recompression_optimality`	theorem; 91
`KomorebiPaperVIII.SmallCore.small_core_optimal_recompression`	theorem; 124
`KomorebiPaperVIII.SmallCore.OrderedThinSVD`	structure; 29
`KomorebiPaperVIII.SmallCore.recompressLeft`	def; 54
`KomorebiPaperVIII.SmallCore.recompressRight`	def; 59

13.32 SmallCoreTransfer: 7 theorems; 2 construction audits

Source: `formalization/SmallCoreTransfer.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.compressed_error`	theorem; 20
`KomorebiPaperVIII.SmallCore.spectral_best_lift`	theorem; 25
`KomorebiPaperVIII.SmallCore.frobenius_best_lift`	theorem; 38
`KomorebiPaperVIII.SmallCore.exact_factor_core`	theorem; 51
`KomorebiPaperVIII.SmallCore.compact_product_reconstruction`	theorem; 61
`KomorebiPaperVIII.SmallCore.compact_core_rank`	theorem; 69
`KomorebiPaperVIII.SmallCore.factorized_truncation`	theorem; 77
`KomorebiPaperVIII.SmallCore.BestRank`	def; 16
`KomorebiPaperVIII.SmallCore.factorCore`	def; 57

13.33 SpectralOptimality: 7 theorems; 1 construction audits

Source: `formalization/SpectralOptimality.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.SmallCore.prefix_isometry`	theorem; 20
`KomorebiPaperVIII.SmallCore.prefix_support`	theorem; 25
`KomorebiPaperVIII.SmallCore.rank_lt_kernel_witness`	theorem; 38
`KomorebiPaperVIII.SmallCore.prefix_kernel_witness`	theorem; 55
`KomorebiPaperVIII.SmallCore.prefix_diagonal_lower`	theorem; 87
`KomorebiPaperVIII.SmallCore.diagonal_spectral_lower`	theorem; 103
`KomorebiPaperVIII.SmallCore.diagonal_spectral_best`	theorem; 120
`KomorebiPaperVIII.SmallCore.prefixEmbedding`	def; 16

13.34 StoredBudgetCompiler: 16 theorems; 2 construction audits

Source: `formalization/StoredBudgetCompiler.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.BudgetDP.vector_get_ofFn`	theorem; 34
`KomorebiPaperVIII.BudgetDP.lookup_tabulate`	theorem; 45
`KomorebiPaperVIII.BudgetDP.bestOver_congr`	theorem; 50
`KomorebiPaperVIII.BudgetDP.convolution_congr_local`	theorem; 60
`KomorebiPaperVIII.BudgetDP.combine_congr_local`	theorem; 69
`KomorebiPaperVIII.BudgetDP.combineStored_correct`	theorem; 92
`KomorebiPaperVIII.CandidateTree.storedTree_correct`	theorem; 120
`KomorebiPaperVIII.CandidateTree.storedBudget_correct`	theorem; 140
`KomorebiPaperVIII.CandidateTree.storedBudget_minimum`	theorem; 147
`KomorebiPaperVIII.BudgetDP.storedDiscrete_correct`	theorem; 167
`KomorebiPaperVIII.BudgetDP.storedDiscreteBudget_correct`	theorem; 186
`KomorebiPaperVIII.CandidateTree.mergeCount_add_one`	theorem; 207
`KomorebiPaperVIII.CandidateTree.merge_trial_count`	theorem; 222
`KomorebiPaperVIII.CandidateTree.merge_trial_bound`	theorem; 226
`KomorebiPaperVIII.CandidateTree.tabulation_work_bound`	theorem; 242
`KomorebiPaperVIII.CandidateTree.tabulation_pseudopolynomial`	theorem; 256
`KomorebiPaperVIII.CandidateTree.storedTree`	def; 109
`KomorebiPaperVIII.BudgetDP.storedDiscrete`	def; 159

13.35 StructuredTensor: 7 theorems; 3 construction audits

Source: `formalization/StructuredTensor.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.kronecker_modewise`	theorem; 19
`KomorebiPaperVIII.kronecker_modewise_work`	theorem; 27
`KomorebiPaperVIII.kronecker_factor_storage`	theorem; 32
`KomorebiPaperVIII.tensor_dimension`	theorem; 51
`KomorebiPaperVIII.tensor_apply_correct`	theorem; 73
`KomorebiPaperVIII.tensor_modewise_work`	theorem; 112
`KomorebiPaperVIII.tensor_factor_storage`	theorem; 126
`KomorebiPaperVIII.tensorMatrix`	def; 59
`KomorebiPaperVIII.tensorApply`	def; 66
`KomorebiPaperVIII.TensorOperation`	def; 99

13.36 ToeplitzEmbedding: 6 theorems; 2 construction audits

Source: `formalization/ToeplitzEmbedding.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.toeplitz_circulant_block`	theorem; 28
`KomorebiPaperVIII.zero_pad_left`	theorem; 45
`KomorebiPaperVIII.zero_pad_right`	theorem; 50
`KomorebiPaperVIII.toeplitz_embedding_action`	theorem; 54
`KomorebiPaperVIII.toeplitz_generator_first_half`	theorem; 60
`KomorebiPaperVIII.toeplitz_generator_middle`	theorem; 64
`KomorebiPaperVIII.toeplitzGenerator`	def; 22
`KomorebiPaperVIII.zeroPad`	def; 26

13.37 TransferSupport: 10 theorems; 0 construction audits

Source: `formalization/TransferSupport.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.power_path_expansion`	theorem; 37
`KomorebiPaperVIII.active_path_sum_zero`	theorem; 55
`KomorebiPaperVIII.power_block_zero_of_no_path`	theorem; 74
`KomorebiPaperVIII.power_support_inclusion`	theorem; 79
`KomorebiPaperVIII.nilpotent_of_path_bound`	theorem; 86
`KomorebiPaperVIII.path_level_bound`	theorem; 93
`KomorebiPaperVIII.nilpotent_of_topological_levels`	theorem; 109
`KomorebiPaperVIII.block_smul`	theorem; 121

Audited declaration	Kind; line
`KomorebiPaperVIII.polynomial_support_inclusion`	theorem; 127
`KomorebiPaperVIII.polynomial_zero_of_no_paths`	theorem; 142

13.38 TreeCompilerUpdates: 9 theorems; 1 construction audits

Source: `formalization/TreeCompilerUpdates.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.CandidateTree.half_gap_order`	theorem; 18
`KomorebiPaperVIII.CandidateTree.compiler_stability`	theorem; 45
`KomorebiPaperVIII.CandidateTree.CachedContext.refresh_valid`	theorem; 95
`KomorebiPaperVIII.CandidateTree.CachedContext.refresh_fill`	theorem; 102
`KomorebiPaperVIII.CandidateTree.CachedContext.updateValue_correct`	theorem; 109
`KomorebiPaperVIII.CandidateTree.CachedContext.sibling_tree_unchanged`	theorem; 128
`KomorebiPaperVIII.CandidateTree.CachedContext.sibling_value_unchanged`	theorem; 134
`KomorebiPaperVIII.CandidateTree.CachedContext.local_recompilation`	theorem; 142
`KomorebiPaperVIII.CandidateTree.unsupported_pareto_example`	theorem; 155
`KomorebiPaperVIII.CandidateTree.CachedContext.updateValue`	def; 75

13.39 TriangularSolve: 7 theorems; 3 construction audits

Source: `formalization/TriangularSolve.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.DAG.Graph.solve_eq`	theorem; 32
`KomorebiPaperVIII.DAG.Graph.solve_correct`	theorem; 38
`KomorebiPaperVIII.DAG.Graph.solve_unique`	theorem; 46
`KomorebiPaperVIII.DAG.Graph.triangular_bijective`	theorem; 65
`KomorebiPaperVIII.DAG.Graph.local_solve_independent`	theorem; 75
`KomorebiPaperVIII.DAG.Graph.stagedSolve_correct_through`	theorem; 91
`KomorebiPaperVIII.DAG.Graph.level_schedule_solves`	theorem; 112
`KomorebiPaperVIII.DAG.Graph.triangularAction`	def; 24
`KomorebiPaperVIII.DAG.Graph.solve`	def; 28
`KomorebiPaperVIII.DAG.Graph.stagedSolve`	def; 85

13.40 WalshTransform: 7 theorems; 2 construction audits

Source: `formalization/WalshTransform.lean`.

Audited declaration	Kind; line
`KomorebiPaperVIII.walsh_two_orthogonal`	theorem; 21
`KomorebiPaperVIII.walsh_recursion`	theorem; 29
`KomorebiPaperVIII.walsh_orthogonal`	theorem; 33
`KomorebiPaperVIII.walsh_apply_correct`	theorem; 60
`KomorebiPaperVIII.walsh_operation_count`	theorem; 99
`KomorebiPaperVIII.walsh_inverse_action`	theorem; 110
`KomorebiPaperVIII.walsh_log_operation_count`	theorem; 115
`KomorebiPaperVIII.walshMatrix`	def; 25
`KomorebiPaperVIII.walshApply`	def; 54

14 Dependencies and source identities

Both public imports and import-all commands contribute to this graph. Import-all exposes implementation details needed by downstream proofs in this module edition. Each identity below is checked against the corresponding saved compiler record and log. The runtime and imported Mathlib corpus are dependencies of the inherited evidence, rather than newly rechecked results.

14.1 Direct local dependencies

Module	Direct local imports
`BandedCount`	Mathlib / Init only
`BlockJacobi`	Mathlib / Init only
`BoundaryChecks`	`PacketIR`; `QuantumPacketCore`; `TransferSupport`
`BudgetConvolution`	Mathlib / Init only
`CirculantFourier`	Mathlib / Init only
`ClippedPrecision`	`ContinuousPrecision`
`CompactRangeFactors`	`SmallCoreNorms`
`CompactTruncation`	`DiagonalTruncation`; `SmallCoreNorms`
`ContinuousPrecision`	Mathlib / Init only
`DiagonalTruncation`	`SmallCoreNorms`; `SmallCoreTransfer`
`DiscoveryCertificate`	`PacketIR`; `QuantumPacketCore`
`DiscretePrecision`	`BudgetConvolution`
`ExactTreeCompiler`	Mathlib / Init only
`FrobeniusOptimality`	`DiagonalTruncation`; `KernelFrame`; `SmallCoreNorms`; `SmallCoreTransfer`
`GraphSchedules`	Mathlib / Init only
`InverseSupport`	`PacketIR`; `QuantumPacketCore`; `TransferSupport`
`KernelFrame`	`SmallCoreNorms`
`KrylovPackets`	`LowRankAlgebra`; `LowRankChains`; `ProductChainError`; `QuantumPacketCore`
`LowRankAlgebra`	`PacketIR`; `QuantumPacketCore`
`LowRankChains`	`LowRankAlgebra`; `PacketIR`; `ProductChainError`; `QuantumPacketCore`
`PacketCostModels`	Mathlib / Init only
`PacketIR`	`QuantumPacketCore`
`PacketPerturbation`	Mathlib / Init only
`ProductChainError`	`PacketPerturbation`
`QuantizedTreeCompiler`	`BudgetConvolution`; `ExactTreeCompiler`
`QuantumPacketCore`	Mathlib / Init only
`RecompressionEndpoints`	`CompactTruncation`; `DiagonalTruncation`; `SmallCoreNorms`; `SmallCoreRecompression`; `SmallCoreTransfer`
`SchedulingContracts`	Mathlib / Init only
`SchurSupport`	`InverseSupport`; `PacketIR`; `QuantumPacketCore`
`SmallCoreNorms`	Mathlib / Init only
`SmallCoreRecompression`	`CompactRangeFactors`; `CompactTruncation`; `DiagonalTruncation`; `FrobeniusOptimality`; `KernelFrame`; `SmallCoreNorms`; `SmallCoreTransfer`; `SpectralOptimality`

Module	Direct local imports
`SmallCoreTransfer`	`CompactRangeFactors`; `SmallCoreNorms`
`SpectralOptimality`	`DiagonalTruncation`; `KernelFrame`; `SmallCoreNorms`; `SmallCoreTransfer`
`StoredBudgetCompiler`	`BudgetConvolution`; `DiscretePrecision`; `ExactTreeCompiler`; `QuantizedTreeCompiler`
`StructuredTensor`	Mathlib / Init only
`ToeplitzEmbedding`	Mathlib / Init only
`TransferSupport`	`PacketIR`; `QuantumPacketCore`
`TreeCompilerUpdates`	`ExactTreeCompiler`
`TriangularSolve`	`GraphSchedules`
`WalshTransform`	`StructuredTensor`

14.2 Source and compiler-log identities

BandedCount

Source SHA-256: `b13124a6d12cdd0589895d3d12c07da364fd47976e28b2c9541b8828bb291a0d`.

Log SHA-256: `8fe4bd93789d7d03432b770afeb52e31f2fe4802b84a7b1ac096df3f76fa1721`.

BlockJacobi

Source SHA-256: `0b5b13491150bdab13be5cc72e6fb0040f39f06cc06398095c360bcb632ced71`.

Log SHA-256: `208d359ecf6a62cc2e0ea02c60f340ab7acae41cbf8caf710e12582627d9f404`.

BoundaryChecks

Source SHA-256: `18a64ef9bdd427697b5fc70417ff123dc9aed37a74be2fcc8cfc30b509ce32b9`.

Log SHA-256: `952e994cd8e9e81a93cdb61b2d8d8b2c87407c23274f97fb9b96357e9ae82969`.

BudgetConvolution

Source SHA-256: `e6421dc70c6050e77e5d6c8987fa7dd515ef5d2e1c0ef10c3a98b2ea18344fd3`.

Log SHA-256: `d1fe718701677b79398cb32bcda5af5b3f1eb49456a0e65c45fd13f993747765`.

CirculantFourier

Source SHA-256: `8db2cde5eec1565ff44c496d4e73a49a2ef3d28a5afc2e1ca0874ad25af3b02c`.

Log SHA-256: `14c85764e294596e7ad47896ed2baf8a0e1d45d9c21f2b4385996b32ba21e722`.

ClippedPrecision

Source SHA-256: `aa86e409a530f0916da245bfe3cf81b7fba26e97c12f8fd7a7690351a08dffc9`.

Log SHA-256: `35d8dd8e973deb2043deaaa73ad1c463477131541231f6c9ae3867b41e0a16f5`.

CompactRangeFactors

Source SHA-256: `3ae44a0f80503e8455cc2cd4ad7de140e631c59e32a109cd5fae2366b34d2e1c`.

Log SHA-256: `6840ecbef5562873065f69d71efdfbd8d1dd63e6c70dccb3bb1aff04e1413ed9`.

CompactTruncation

Source SHA-256: `e387ce9da3cf16879225207523cc0923ae5adaaa1c3e317c447b4923ec40b6a2`.

Log SHA-256: `0162357f46179fd8d1c70d3ed0488c1cff7cf9290e6ac12be299e38e322def53`.

ContinuousPrecision

Source SHA-256: `5a737f5aa8ed23525c225b48fc6cef46a623fe85b7ddf8ee4bf9115976667230`.

Log SHA-256: `1473e51a2a09550739223cfe7c2ce4576daa3abf4ff700e9bc347ce7e4ae04ab`.

DiagonalTruncation

Source SHA-256: `21594e839e131b8ae6a5ef4fe06a1640fdf2ef03e5b745985c3e5f21429809ad`.

Log SHA-256: `f689d98f40296f56dad426161e3636e287d3cfa624a4c885398a916a58009afb`.

DiscoveryCertificate

Source SHA-256: `ae8a69d0022cb8258a75d91b5e7e98c9a4f0f60d4d67f2d976b093d8a5987e8f`.

Log SHA-256: `5fa99bbe29375b3e7ee48d27a08362201829200e888dcd37829334d7b74b2ec8`.

DiscretePrecision

Source SHA-256: `0b12bf6ecf1c9bc36bc735ee737c06db5a37651d12dca3ffcad230493bb7a654`.

Log SHA-256: `0a45c7a80f06fe0f55002efd9abc600a469d78d6242283eb14b6ee54dd293316`.

ExactTreeCompiler

Source SHA-256: `680d0a063af7cba93ce7b29ac9bfa1ba65dea1b9cf42309c55550b394c3a3f5f`.

Log SHA-256: `8f4156453fc9298a64518da7dba7178c606d2269a7c3f47c2db53fa534c4c66d`.

FrobeniusOptimality

Source SHA-256: `3d28205f0088a2ebc9ca78141cefb73d4d14bdb4f1feb9cb860968cfa9b29f87`.

Log SHA-256: `576c50d4d646d0a75e382953b4e4c9cd1fcca2aa0763abe6bcf8125080698785`.

GraphSchedules

Source SHA-256: `1b845790a59fdf87562a2afa5dd56cda8ee0d65ae4cfbb8a1259364d4a01d4cf`.

Log SHA-256: `a8ce351f7fabb514b6a3de6cda7498541ca7491294274f6756f252b6abd46a6d`.

InverseSupport

Source SHA-256: `e8214096eab227677cdd7977840b02844bd0ec706df57dca405bcfae2f6e2063`.

Log SHA-256: `104d9d3d784e307a6e864540a07f4f60259b60b5d8b3765bee1919e38c7b0d53`.

KernelFrame

Source SHA-256: `dc4fd08f29d82779453ad49007b38339423f6c09aec0393eca5b56f0674af90c`.

Log SHA-256: `b5c78b4e160d3f33b1b1bb59ab2a222f6642f8cb034b163d3d483f46b7ba8548`.

KrylovPackets

Source SHA-256: `12fd73f08368ec528953e66e165ef4281f1a978052c21e0c61dbde6d12c508b5`.

Log SHA-256: `4b7a9affc66c399b35a54e9aca5483e78b99acf970cbb9435a029098fe6d5044`.

LowRankAlgebra

Source SHA-256: `61e29902b761b36930587c604cee5740c16fbb076fbde070df7ba80ce0c7984d`.

Log SHA-256: `d274abfbfd96bb5ece7a700a42eeb3bf7b0fbfbdcb9a54f569860b8b144653db`.

LowRankChains

Source SHA-256: `970c35e0219c27dedaf97e215a9868ebd4e6015b7f736b9dc0aea385b592212b`.

Log SHA-256: `c72865e5ff284eaba62955b9daa8f9da87a7589e81b6097de6af72978e2c8da4`.

PacketCostModels

Source SHA-256: `77aa5594485ec62c259e1a1d9493874481e72ceaecd6c2b6f654f42474f999b5`.

Log SHA-256: `2d53dec907e3286a529bb2de82d6cbc3f800a0ca31e01f91fb89505962623a88`.

PacketIR

Source SHA-256: `67e2e088b17bb95162585592da91c1ed506c5faf7f8d6574cc8b3af8ff0e47c7`.

Log SHA-256: `ab2bc3c6cbc4cce4f38ea6a4c285c596f496fb5759bb3f32c8ce7bb2db83b746`.

PacketPerturbation

Source SHA-256: `5fbb1026da706b46afaf2793e138fe503e00ed37a470c193388ad6e79a37bd12`.

Log SHA-256: `649c81ea6be13008f2b5d21194eaac3e67ac8915b270a5b3278298417d1b6411`.

ProductChainError

Source SHA-256: `5bad0d7ed04be1a94bb19e54c9371e3aea0dc69ff90e4afc33c6cdc0e4686a2c`.

Log SHA-256: `0634b1ccd6522518ee87aa2a00d71b1bcd9e44723ae646645c186c5d03b9121b`.

QuantizedTreeCompiler

Source SHA-256: `564ddda43334a4cd506e7bf990704b4fc604eeccb891084a223cae53b96268e4`.

Log SHA-256: `f88e40d19ee915fd044cc719c48d746bfb2b7b12c25a58c8f4d338d2aa1c5b2f`.

QuantumPacketCore

Source SHA-256: `92e49bb023877ed1f1c2bb34968aa81ee7724fa3a8e1801a831cfc70f1de3595`.

Log SHA-256: `cde10a754cab43da66a12045d11df455573b5cc3f647497d29a54359e39f5798`.

RecompressionEndpoints

Source SHA-256: `d8e8dc3290e938198c524607904db67869659461b852964e46297de4690a3789`.

Log SHA-256: `f08cd5b88cff2be28ff88b1fb9923c8dbe6a3003e0a92c82bd0919cfc1c67e19`.

SchedulingContracts

Source SHA-256: `ea646fab14df1c6f6b4855745035b64835920a75fe2e66d168cb515f74854160`.

Log SHA-256: `79f97910adb6bce9043ffe0717877e663b87c479bcd07a35ee6f8f990e7b082a`.

SchurSupport

Source SHA-256: `36b368041215ef6857fb3d5809525c1f78436601ff1897e04de9ff1992acfceb`.

Log SHA-256: `fe9fbac8b5b20877f8cabf76b6d71488b1317fdec24ce9c4b395f6ef267530ca`.

SmallCoreNorms

Source SHA-256: `dd397aad6a2b83f9c292d50d201fea3719fc5831b0d982f4ab0036144c28d3e9`.

Log SHA-256: `7ae75ddba072bee8bddb8ed1d858f0f86f54bbeed4c9e5b082e716bb76600d1b`.

SmallCoreRecompression

Source SHA-256: `4a75c726f933e466bbbf14bdfa227def9509b6b891701e61e860f9702f9829f1`.

Log SHA-256: `e9fd9e09de7b3a24cac21183a4c82dd9e0461d70db55f46c3d534e82a77b53b1`.

SmallCoreTransfer

Source SHA-256: `d6040b4f192abcd15ee0f7aebdf81054208e2dae8c0a8a77a40ad473f2efff1b`.

Log SHA-256: `d0fc63e9344310c4d09db11a8697ffdd018300e7463ec80c1678faa7afa08e66`.

SpectralOptimality

Source SHA-256: `b7ae843f3c3dc73462b8761e69c6532225fe0e73040d029218ff65887cf6dba2`.

Log SHA-256: `780db5d54a9e37cce1ed34fb8c16e5d08e1db1403a25f400708e5005ed588383`.

StoredBudgetCompiler

Source SHA-256: `1cf96ed6762939a89b434e7ed6c07f5f696cc794461725bedf590f5eff4d9a89`.

Log SHA-256: `b10e398878b0630b8f7412cbcac6ca0676e06e52a673fc4b1b86e692c2f0155c`.

StructuredTensor

Source SHA-256: `659f010a9ad19e3470f98825b20934147c807aa7582e445d8735e82ff80b71e0`.

Log SHA-256: `50e13efc7deed747d3422504c03a6b830c4373376e5dafd07bffc953bcf97f37`.

ToeplitzEmbedding

Source SHA-256: `b40f87ab523e8fe12299eef50fb822839d13d4635e257b9a644aae397b83e8c5`.

Log SHA-256: `37c02a27a35fa9d66f4a4b8e86dc0e3aeac6a23d07bae989e54a766e17f6f58f`.

TransferSupport

Source SHA-256: `f984a44d1fd1f0b4d6b937ff6e6c4b69f58f7e8a4f29ccf61715252eec5f6cfe`.

Log SHA-256: `e6a96bcb9010fee9eca5e3dde81892729ec623ef4a85de6ba6e8a4146f5df194`.

TreeCompilerUpdates

Source SHA-256: `7d7f8569bef69bc8938e1e883789d72a9e9b8a82def59b79cb637b85274f5001`.

Log SHA-256: `8076a73777b0870b8e5640a9ae56c213685996814570283d00ab6729559c6e54`.

TriangularSolve

Source SHA-256: `0f5939c4982d65b5b74c10455503686adc32923d5211561a17a291eae1e33a0e`.

Log SHA-256: `55275e231567634b6c3001a7f24c0d567986691c74a9a3933d62fb39dadcd61a`.

WalshTransform

Source SHA-256: `fd74891c276e65c3bdade75cd37768243b5b1d3f61a6a4c3a8b03ed6236551e7`.

Log SHA-256: `841deef0ed05655c97bf09eb407dd2832ab396bf4fded35f4174aa102bc28af6`.

14.3 Pinned inherited run

Lean version: `4.34.0-rc2`; saved compiler commit: `6a10ac8c22beadecabdbb0919c2b50214762f91d`; Mathlib revision: `aeb72f1e9981b6d35b426b16dca7759be3e7ba06`. The source archive does not include the separately supplied runtime kit. A new replay must obtain that kit and validate its actual revisions.

Summary SHA-256: `768b48a902e571c8f5e7dacc8424aea447add7df03bbd9d79dd892fe27cae629`.

Inventory SHA-256: `bee2888bbff71d8d4b120b1ca34de2d3b4a6a8023c5208844c6435a205c6811a`.

15 Exact excerpt register

Each selection is a contiguous block of the unchanged active source, retaining the declaration and proof. Module-level variables and type classes remain in force; the teaching chapters state the relevant context. The printed code applies a font fallback for the one unavailable monospaced glyph. The accompanying `.lean.txt` files preserve the raw UTF-8 bytes without substitutions.

packet_family

Source: `formalization/QuantumPacketCore.lean`.

Lines 22–25; `PacketFamily`.

Source SHA-256: `92e49bb023877ed1f1c2bb34968aa81ee7724fa3a8e1801a831cfc70f1de3595`.

Excerpt SHA-256: `94b93d351a97d973bf990a4a8d1fcd4a8d04e12ae60192ab2eaf58fd05722bde`.

active_evaluation

Source: `formalization/PacketIR.lean`.

Lines 41–49; `exact_active_evaluation`.

Source SHA-256: `67e2e088b17bb95162585592da91c1ed506c5faf7f8d6574cc8b3af8ff0e47c7`.

Excerpt SHA-256: `b8db812ffae61d1a5628229653be4d010699845fb7b5588e5997d9407d3f2d85`.

support_inclusion

Source: `formalization/PacketIR.lean`.

Lines 63–77; `product_support_inclusion`.

Source SHA-256: `67e2e088b17bb95162585592da91c1ed506c5faf7f8d6574cc8b3af8ff0e47c7`.

Excerpt SHA-256: `bd4107318d9f6faefcc4896c60f1995b1a410de7cb48159c39f0ba4c0acd41af`.

adjoint

Source: `formalization/PacketIR.lean`.

Lines 95–97; `exact_adjoint_block`.

Source SHA-256: `67e2e088b17bb95162585592da91c1ed506c5faf7f8d6574cc8b3af8ff0e47c7`.

Excerpt SHA-256: `be55df55acfc0c920766f0848b6325d49a2b00dfd58b78fe36897c8217786a5b`.

batch

Source: `formalization/PacketIR.lean`.

Lines 145–149; `exact_matrix_batch`.

Source SHA-256: `67e2e088b17bb95162585592da91c1ed506c5faf7f8d6574cc8b3af8ff0e47c7`.

Excerpt SHA-256: `55c9fd65ca205c50da1c090e08ba5220eb54d38cddebcad392e005fc5946e62a`.

bdlr_count

Source: `formalization/PacketCostModels.lean`.

Lines 68–73; `bdlr_application_count`.

Source SHA-256: `77aa5594485ec62c259e1a1d9493874481e72ceaecd6c2b6f654f42474f999b5`.

Excerpt SHA-256: `c8640596a7f0bb7aef34ac7b91d5efc23cfec34f8d2166bd00dd5537df37ced7`.

roofline

Source: `formalization/PacketCostModels.lean`.

Lines 118–124; `roofline_time_bound`.

Source SHA-256: `77aa5594485ec62c259e1a1d9493874481e72ceaecd6c2b6f654f42474f999b5`.

Excerpt SHA-256: `ea76d943ec216b343c478c484ee260e81281349adf817f422945f448d29454a6`.

band_count

Source: `formalization/BandedCount.lean`.

Lines 73–77; `banded_block_count`.

Source SHA-256: `b13124a6d12cdd0589895d3d12c07da364fd47976e28b2c9541b8828bb291a0d`.

Excerpt SHA-256: `215d38825da4198d4d652e7cc7460819c457b1ffee9139397540fe88b1236d7b`.

bdlr_product

Source: `formalization/LowRankAlgebra.lean`.

Lines 57–65; `bdlr_product`.

Source SHA-256: `61e29902b761b36930587c604cee5740c16fbb076fbde070df7ba80ce0c7984d`.

Excerpt SHA-256: `9264a6332bf3610bc332de96452aa893026990771d125af2c65fdabe90fe735e`.

chain_factors

Source: `formalization/LowRankChains.lean`.

Lines 56–66; `chainFactors_correct`.

Source SHA-256: `970c35e0219c27dedaf97e215a9868ebd4e6015b7f736b9dc0aea385b592212b`.

Excerpt SHA-256: `65b33e5cf658798a4651f3e8a83c1a60af9af6c303b799e3275b202c6a0fe756`.

woodbury

Source: `formalization/LowRankAlgebra.lean`.

Lines 89–94; `bdlr_woodbury`.

Source SHA-256: `61e29902b761b36930587c604cee5740c16fbb076fbde070df7ba80ce0c7984d`.

Excerpt SHA-256: `b6e65a998993dadf81b79f505d629ab076210bc137f29932b27b6a62421f7ee4`.

kronecker

Source: `formalization/StructuredTensor.lean`.

Lines 19–22; `kronecker_modewise`.

Source SHA-256: `659f010a9ad19e3470f98825b20934147c807aa7582e445d8735e82ff80b71e0`.

Excerpt SHA-256: `42d94ff97bf545b6b0abb297f76db13f7caec1ef8bcd501a96e2d8a19f6fc967`.

tensor_apply

Source: `formalization/StructuredTensor.lean`.

Lines 73–96; `tensor_apply_correct`.

Source SHA-256: `659f010a9ad19e3470f98825b20934147c807aa7582e445d8735e82ff80b71e0`.

Excerpt SHA-256: `87c9fee83936f6c3056c31d7bbdeede737219cb900935b1921dbdf777527e4a8`.

walsh_apply

Source: `formalization/WalshTransform.lean`.

Lines 60–84; `walsh_apply_correct`.

Source SHA-256: `fd74891c276e65c3bdade75cd37768243b5b1d3f61a6a4c3a8b03ed6236551e7`.

Excerpt SHA-256: `13305b25799d85ceee84f430c17c47158717349a71a5eef9aa90e34d5ee93431`.

toeplitz_apply

Source: `formalization/ToeplitzEmbedding.lean`.

Lines 54–58; `toeplitz_embedding_action`.

Source SHA-256: `b40f87ab523e8fe12299eef50fb822839d13d4635e257b9a644aae397b83e8c5`.

Excerpt SHA-256: `45b0c8e014eba5778271dc6127c731cf5b76fdccbe470570482a9f20ef341441`.

fourier

Source: `formalization/CirculantFourier.lean`.

Lines 150–153; `circulant_spectral_representation`.

Source SHA-256: `8db2cde5eec1565ff44c496d4e73a49a2ef3d28a5afc2e1ca0874ad25af3b02c`.

Excerpt SHA-256: `4d96344b6bfd07b20e24ed96afb28abf92fffc5f654d261025c97ec8c3018275`.

power_paths

Source: `formalization/TransferSupport.lean`.

Lines 37–52; `power_path_expansion`.

Source SHA-256: `f984a44d1fd1f0b4d6b937ff6e6c4b69f58f7e8a4f29ccf61715252eec5f6cfe`.

Excerpt SHA-256: `872dd779a4e39a29bd50c57cb1326c95870477e966e459372538ecbaf2d0938c`.

nilpotent

Source: `formalization/TransferSupport.lean`.

Lines 86–90; `nilpotent_of_path_bound`.

Source SHA-256: `f984a44d1fd1f0b4d6b937ff6e6c4b69f58f7e8a4f29ccf61715252eec5f6cfe`.

Excerpt SHA-256: `6874427d6bcf4444acc78d23c0f6b7ae953896e30607aa5b37b6641e90e08e77`.

inverse_support

Source: `formalization/InverseSupport.lean`.

Lines 94–100; `inverse_block_zero_of_no_path`.

Source SHA-256: `e8214096eab227677cdd7977840b02844bd0ec706df57dca405bcfae2f6e2063`.

Excerpt SHA-256: `49f48da9ac56e4b9f42a3624291f367fecb5ed88da26b521a5882edbc3895fb6`.

schur_support

Source: `formalization/SchurSupport.lean`.

Lines 57–73; `schur_support_inclusion`.

Source SHA-256: `36b368041215ef6857fb3d5809525c1f78436601ff1897e04de9ff1992acfceb`.

Excerpt SHA-256: `eb2070911268dd8d6f00ab3c496f5fa0f3983d3525362018d17dab441083e6df`.

solve_correct

Source: `formalization/TriangularSolve.lean`.

Lines 38–43; `solve_correct`.

Source SHA-256: `0f5939c4982d65b5b74c10455503686adc32923d5211561a17a291eae1e33a0e`.

Excerpt SHA-256: `16f88870ae9e77b3046abcd6f1986ba4992f59ff2254bffb42825467c9fcd4e4`.

schedule_depth

Source: `formalization/GraphSchedules.lean`.

Lines 150–154; `optimal_depth`.

Source SHA-256: `1b845790a59fdf87562a2afa5dd56cda8ee0d65ae4cfbb8a1259364d4a01d4cf`.

Excerpt SHA-256: `48fea4f397c61d44b2d4ce4ecb71402f7d3947c83b2338ed5624499b09eded57`.

packet_error

Source: `formalization/PacketPerturbation.lean`.

Lines 35–39; `packet_perturbation_bound`.

Source SHA-256: `5fbb1026da706b46afaf2793e138fe503e00ed37a470c193388ad6e79a37bd12`.

Excerpt SHA-256: `f9be290eddd13b0e1e78d379a60f35f7d7aa6f6b25d5b8e19ea8c158b24c27c5`.

chain_error

Source: `formalization/ProductChainError.lean`.

Lines 129–145; `product_chain_error_bound`.

Source SHA-256: `5bad0d7ed04be1a94bb19e54c9371e3aea0dc69ff90e4afc33c6cdc0e4686a2c`.

Excerpt SHA-256: `d5ce5f5abbf53f974cdaae1745fbbc526424537fc51b88c495f58fa6fec5a798`.

recompression_error

Source: `formalization/ProductChainError.lean`.

Lines 100–111; `repeated_recompression_bound`.

Source SHA-256: `5bad0d7ed04be1a94bb19e54c9371e3aea0dc69ff90e4afc33c6cdc0e4686a2c`.

Excerpt SHA-256: `1d5d7a1765f121c95ac8c2571665e1ccb9429065c71075df02f552ccb9816965`.

continuous_precision

Source: `formalization/ContinuousPrecision.lean`.

Lines 80–97; `precision_allocation_optimal`.

Source SHA-256: `5a737f5aa8ed23525c225b48fc6cef46a623fe85b7ddf8ee4bf9115976667230`.

Excerpt SHA-256: `56e7ccbc5bc16b61893e32284ae60ea67fd8f4241dbe4de8ce75f39d2e94b544`.

clipped_precision

Source: `formalization/ClippedPrecision.lean`.

Lines 211–234; `clipped_unique_minimizer`.

Source SHA-256: `aa86e409a530f0916da245bfe3cf81b7fba26e97c12f8fd7a7690351a08dffc9`.

Excerpt SHA-256: `c92e6134eb561e19b80ed28b63c178e1893a6a2f905b5628a7c1c633d45be8f3`.

discrete_recurrence

Source: `formalization/DiscretePrecision.lean`.

Lines 38–42; `discreteTable`.

Source SHA-256: `0b12bf6ecf1c9bc36bc735ee737c06db5a37651d12dca3ffcad230493bb7a654`.

Excerpt SHA-256: `6228e94e6ca74427d363ac3f162d2ad71b2aa3bb7dfdf857d23338416d090413`.

discrete_endpoint

Source: `formalization/DiscretePrecision.lean`.

Lines 110–121; `discreteBudget_some_iff`.

Source SHA-256: `0b12bf6ecf1c9bc36bc735ee737c06db5a37651d12dca3ffcad230493bb7a654`.

Excerpt SHA-256: `12a6e2e0aac33aa266dcebe1a641bf422a65954cd6cf9caaca44d92c07576adf`.

candidate_tree

Source: `formalization/ExactTreeCompiler.lean`.

Lines 21–23; `CandidateTree`.

Source SHA-256: `680d0a063af7cba93ce7b29ac9bfa1ba65dea1b9cf42309c55550b394c3a3f5f`.

Excerpt SHA-256: `6042f121d7e093d71bf88604481295fc3a44039b07719a96e678fb01847dfe08`.

cut

Source: `formalization/ExactTreeCompiler.lean`.

Lines 33–36; `Cut`.

Source SHA-256: `680d0a063af7cba93ce7b29ac9bfa1ba65dea1b9cf42309c55550b394c3a3f5f`.

Excerpt SHA-256: `c39d677c2ec68517663c4047337d620a3fd2e9f7a76c3b6014e90d714b448125`.

tree_recurrence

Source: `formalization/ExactTreeCompiler.lean`.

Lines 45–47; `optimalValue`.

Source SHA-256: `680d0a063af7cba93ce7b29ac9bfa1ba65dea1b9cf42309c55550b394c3a3f5f`.

Excerpt SHA-256: `bdd0f6e304db9f408f6700b56f5c28afe3b8e723b8e07f89b176e676535151a7`.

compile_attains

Source: `formalization/ExactTreeCompiler.lean`.

Lines 69–76; `compile_attains`.

Source SHA-256: `680d0a063af7cba93ce7b29ac9bfa1ba65dea1b9cf42309c55550b394c3a3f5f`.

Excerpt SHA-256: `758756e03cc7b28620f65cfb1246578a05b283dbe8c19abe503e4c7bf140e9b8`.

tree_exact

Source: `formalization/ExactTreeCompiler.lean`.

Lines 79–83; `exact_tree_compiler`.

Source SHA-256: `680d0a063af7cba93ce7b29ac9bfa1ba65dea1b9cf42309c55550b394c3a3f5f`.

Excerpt SHA-256: `7638f7a2a4eb00eb9e8d12462200118215899cf68b16346b667fa72c94ebfc39`.

budget_recurrence

Source: `formalization/QuantizedTreeCompiler.lean`.

Lines 27–33; `budgetTable`.

Source SHA-256: `564ddda43334a4cd506e7bf990704b4fc604eeccb891084a223cae53b96268e4`.

Excerpt SHA-256: `c6e79eb95cd127303ec581bedf142a2d6fd275df9d5463d932e9e8e0813dc748`.

budget_endpoint

Source: `formalization/QuantizedTreeCompiler.lean`.

Lines 107–119; `compileBudget_some_iff`.

Source SHA-256: `564ddda43334a4cd506e7bf990704b4fc604eeccb891084a223cae53b96268e4`.

Excerpt SHA-256: `36aa0f165da0f22adee78e1a2a896121e4aa17d33cbde46f6f4e79da2a53fed1`.

budget_none

Source: `formalization/QuantizedTreeCompiler.lean`.

Lines 122–126; `compileBudget_none_iff`.

Source SHA-256: `564ddda43334a4cd506e7bf990704b4fc604eeccb891084a223cae53b96268e4`.

Excerpt SHA-256: `b3fc8f236ff026cdd83f7caf79190b0ad2c15d99c14ff943bf9aa6bf554fe6ba`.

stored_tree

Source: `formalization/StoredBudgetCompiler.lean`.

Lines 109–118; `storedTree`.

Source SHA-256: `1cf96ed6762939a89b434e7ed6c07f5f696cc794461725bedf590f5eff4d9a89`.

Excerpt SHA-256: `c584d5dc39f3557a80946b4b00d454f949b0a032083c1bead34a2b3c7eaae365`.

stored_correct

Source: `formalization/StoredBudgetCompiler.lean`.

Lines 120–133; `storedTree_correct`.

Source SHA-256: `1cf96ed6762939a89b434e7ed6c07f5f696cc794461725bedf590f5eff4d9a89`.

Excerpt SHA-256: `d0fb967877905890a693b75b63146bd23243caa565a1ab2640781370abfe278c`.

pseudo_polynomial

Source: `formalization/StoredBudgetCompiler.lean`.

Lines 256–260; `tabulation_pseudopolynomial`.

Source SHA-256: `1cf96ed6762939a89b434e7ed6c07f5f696cc794461725bedf590f5eff4d9a89`.

Excerpt SHA-256: `694efc5328fd56ccb8f37633c3f507172d6de874811ff601b23cee7eb9dbefe7`.

compiler_stability

Source: `formalization/TreeCompilerUpdates.lean`.

Lines 45–59; `compiler_stability`.

Source SHA-256: `7d7f8569bef69bc8938e1e883789d72a9e9b8a82def59b79cb637b85274f5001`.

Excerpt SHA-256: `20d720b1679d7fa385f677c73a0070be0fc0e6ec4d78100b1c96d206c426bf3d`.

local_update

Source: `formalization/TreeCompilerUpdates.lean`.

Lines 142–150; `local_recompilation`.

Source SHA-256: `7d7f8569bef69bc8938e1e883789d72a9e9b8a82def59b79cb637b85274f5001`.

Excerpt SHA-256: `2f57ee7d82d42de016997621fdd1564a1ad6a29730d07461ed424e24fc9027e3`.

contract

Source: `formalization/SchedulingContracts.lean`.

Lines 54–60; `contract_conversion`.

Source SHA-256: `ea646fab14df1c6f6b4855745035b64835920a75fe2e66d168cb515f74854160`.

Excerpt SHA-256: `e9796ff3b4bd8914b18d8c74a03dc9d332f6c25c6e2e05d63e395be81feb5dda`.

grouping

Source: `formalization/SchedulingContracts.lean`.

Lines 89–93; `grouped_execution_output`.

Source SHA-256: `ea646fab14df1c6f6b4855745035b64835920a75fe2e66d168cb515f74854160`.

Excerpt SHA-256: `f5631a2725315cbf2b0f719019445a0049b43d600322868ac7f91353192d2243`.

amortization

Source: `formalization/SchedulingContracts.lean`.

Lines 117–123; `integer_amortization_threshold`.

Source SHA-256: `ea646fab14df1c6f6b4855745035b64835920a75fe2e66d168cb515f74854160`.

Excerpt SHA-256: `3228ffc9dda59871a65a0470e3d4c06a738ad5934a73f1e3bff9fad957aa1da5`.

krylov

Source: `formalization/KrylovPackets.lean`.

Lines 89–99; `invariant_packet_krylov`.

Source SHA-256: `12fd73f08368ec528953e66e165ef4281f1a978052c21e0c61dbde6d12c508b5`.

Excerpt SHA-256: `5493ac684d15f1329599b43e94cf1a98c0e78bf41c1af8a47e980968c09bb759`.

approx_krylov

Source: `formalization/KrylovPackets.lean`.

Lines 183–195; `approximate_packet_krylov_distance`.

Source SHA-256: `12fd73f08368ec528953e66e165ef4281f1a978052c21e0c61dbde6d12c508b5`.

Excerpt SHA-256: `f5b2dc70dee98e47baf485eef18ba8a584d181afc886b22d5b47022d1c6f4cb8`.

jacobi

Source: `formalization/BlockJacobi.lean`.

Lines 87–95; `block_jacobi_condition`.

Source SHA-256: `0b5b13491150bdab13be5cc72e6fb0040f39f06cc06398095c360bcb632ced71`.

Excerpt SHA-256: `f7dedba3028e599a34f27ec89a2847a54f3e3fc76babd443c4464d1dc5c9add1`.

cg_bound

Source: `formalization/BlockJacobi.lean`.

Lines 134–141; `certified_energy_error`.

Source SHA-256: `0b5b13491150bdab13be5cc72e6fb0040f39f06cc06398095c360bcb632ced71`.

Excerpt SHA-256: `9206b8018309d6b6fd6c47ce89cbaa62b10910c8912755a3eeb92d1bad64d930`.

discovery

Source: `formalization/DiscoveryCertificate.lean`.

Lines 126–137; `discovery_frobenius_certificate`.

Source SHA-256: `ae8a69d0022cb8258a75d91b5e7e98c9a4f0f60d4d67f2d976b093d8a5987e8f`.

Excerpt SHA-256: `ceb4c4d0d9e6d4cae8f5a47105676a6ea536874e27575476646e7ce647dcfc90`.

compact_factor

Source: `formalization/CompactRangeFactors.lean`.

Lines 51–57; `compact_factorization`.

Source SHA-256: `3ae44a0f80503e8455cc2cd4ad7de140e631c59e32a109cd5fae2366b34d2e1c`.

Excerpt SHA-256: `3c31204f3c15ded1aefab7a77dace1ccfa12800618b4d57bb0e25094518e2fe1`.

best_rank

Source: `formalization/SmallCoreTransfer.lean`.

Lines 16–17; `BestRank`.

Source SHA-256: `d6040b4f192abcd15ee0f7aebdf81054208e2dae8c0a8a77a40ad473f2efff1b`.

Excerpt SHA-256: `152df4ae4bf5c843eba1d574336f437b77a7dc6482f91f2d68b5487f553dffe4`.

ambient_transfer

Source: `formalization/SmallCoreTransfer.lean`.

Lines 25–35; `spectral_best_lift`.

Source SHA-256: `d6040b4f192abcd15ee0f7aebdf81054208e2dae8c0a8a77a40ad473f2efff1b`.

Excerpt SHA-256: `10fd13227a608516e1d94fe05a23e19a264c25dd3587d80d9b8c8f20bd5d88fc`.

ordered_svd

Source: `formalization/SmallCoreRecompression.lean`.

Lines 29–37; `OrderedThinSVD`.

Source SHA-256: `4a75c726f933e466bbbf14bdfa227def9509b6b891701e61e860f9702f9829f1`.

Excerpt SHA-256: `7ffa30b8be7d8fcc025e5d89b1d85745eee3f58847d9ff10150482633ae170f7`.

kernel_witness

Source: `formalization/SpectralOptimality.lean`.

Lines 55–82; `prefix_kernel_witness`.

Source SHA-256: `b7ae843f3c3dc73462b8761e69c6532225fe0e73040d029218ff65887cf6dba2`.

Excerpt SHA-256: `b244da5b0dbb939ab440aea1192e76e0d87213a86ba223d011ca0057183c1143`.

spectral_best

Source: `formalization/SpectralOptimality.lean`.

Lines 120–125; `diagonal_spectral_best`.

Source SHA-256: `b7ae843f3c3dc73462b8761e69c6532225fe0e73040d029218ff65887cf6dba2`.

Excerpt SHA-256: `845fbd554f691fa869be101e0439cdf3a57b77d4cecae9db0f51d0ea0b26ab3a`.

weighted_tail

Source: `formalization/FrobeniusOptimality.lean`.

Lines 35–60; `weighted_tail_lower`.

Source SHA-256: `3d28205f0088a2ebc9ca78141cefb73d4d14bdb4f1feb9cb860968cfa9b29f87`.

Excerpt SHA-256: `06a99100943ba24f123308c7cbc9be31d6358ac68c4bf96738b538ef6270faac`.

frobenius_lower

Source: `formalization/FrobeniusOptimality.lean`.

Lines 76–104; `diagonal_frobenius_lower`.

Source SHA-256: `3d28205f0088a2ebc9ca78141cefb73d4d14bdb4f1feb9cb860968cfa9b29f87`.

Excerpt SHA-256: `ec9c2efbd1402edb36371b9234ce5986fd1c8d8912d9b117a4b720ca95e0806b`.

recompression_optimal

Source: `formalization/SmallCoreRecompression.lean`.

Lines 91–120; `recompression_optimality`.

Source SHA-256: `4a75c726f933e466bbbf14bdfa227def9509b6b891701e61e860f9702f9829f1`.

Excerpt SHA-256: `89293b779e3b624a8aa564e3650b986204d99e06c2633e0be689bdfbd8b0d7e3`.

small_core_endpoint

Source: `formalization/SmallCoreRecompression.lean`.

Lines 124–141; `small_core_optimal_recompression`.

Source SHA-256: `4a75c726f933e466bbbf14bdfa227def9509b6b891701e61e860f9702f9829f1`.

Excerpt SHA-256: `30ffd8ab1cfe1b7c7fd7c3867247600a5946e08d57e3b4c17e77f2d6df1c883c`.

full_retention

Source: `formalization/RecompressionEndpoints.lean`.

Lines 70–74; `recompression_full`.

Source SHA-256: `d8e8dc3290e938198c524607904db67869659461b852964e46297de4690a3789`.

Excerpt SHA-256: `e8689561b90aaef3b937ea6d2bd9c95e0bea3d1c1c44a0bfd5579e31fb3c3219`.