std::linalg:
Linear Algebra Coming to Standard C++

MARK HOEMMEN

Cppcon
The C++ Conference
2023
October 01 - 06

NVIDIA

std::linalg: Linear Algebra Coming to Standard C++

Mark Hoemmen, NVIDIA | CppCon 2023

Agenda

- Motivating example: Matrix multiply
- Where std::linalg fits in linear algebra's layers
- std::linalg builds on the C++ Standard Library
- std::linalg builds on the long history of the BLAS
- Detailed example: Cholesky matrix factorization

Motivating example: Updating matrix multiply

$C := \beta C + \alpha A B^T$

α and β are scalars; A, B, and C are matrices

```
// BEFORE: Call optimized Fortran library

extern “C” void
dgemm_(
  const char* TRANSA, const char* TRANSB,
  const int* m, const int* n, const int* k,
  const double* alpha,
  const double A[], const int* LDA,
  const double* beta,
  const double B[], const int* LDB,
  double C[], const int* LDC);

int m   = C.extent(0); // C^T is n x m
int n   = C.extent(1);
int k   = B.extent(1); // B^T is n x k
int LDA = A.stride(0); // layout_right
int LDB = B.stride(0); // layout_right
int LDC = C.stride(0); // layout_right

dgemm_(“T”, “N”, &n, &m, &k,
  &alpha, B.data_handle(), &LDB,
  A.data_handle(), &LDA, &beta,
  C.data_handle(), &LDC);
```

```
// AFTER: Use std::linalg

#include <linalg>

matrix_product(par_unseq,
  scaled(alpha, A), transposed(B),
  scaled(beta, C), C);
```

Does a “linear algebra library” do linear algebra?

Aspirational linearity

We use math words	We mean a computer representation
Scalars α, β in a field	Fixed-length numbers
Vectors x, y, z	Rank-1 array (coordinates in some basis)
Matrices A, B, C	Rank-2 array (linear function between 2 vector spaces, assuming a basis for each)
Products, norms, solves	Computations with array inputs & {array or scalar} output
Matrices are linear functions; addition is associative $A(x + y) = Ax + Ay$ $A(\alpha x) = \alpha Ax$ $(x + y) + z = x + (y + z)$	Destroyed by rounding error, saturation, overflow, etc.
	First electronic computers’ architects cared deeply whether computing made sense, given rounding error

My impression of René Magritte’s “The Treachery of Images” (1929)

The Householder convention

A standard notation, & a standard way of talking about computations

We use math words	We mean a computer representation
Scalars α, β in a field	Fixed-length numbers
Vectors x, y, z	Rank-1 array (coordinates in some basis)
Matrices A, B, C	Rank-2 array (linear function between 2 vector spaces, assuming a basis for each)
Products, norms, solves	Computations with array inputs & {array or scalar} output
Matrices are linear functions; addition is associative $A(x+y) = Ax + Ay$ $A(\alpha x) = \alpha Ax$ $(x+y)+z = x+(y+z)$	Destroyed by rounding error, saturation, overflow, etc.
	First electronic computers' architects cared deeply whether computing made sense, given rounding error

Alston Scott Householder (1904 – 1993)

A unifier who helped define the standard problems of the field

Linear algebra comes in (abstraction) layers

What are the responsibilities of a “linear algebra library”?

Linear algebra comes in (abstraction) layers

What are the responsibilities of a “linear algebra library”?

- **Layer -1: Fundamentals**
  - Multidimensional arrays & iteration
- **Layer 0: Performance primitives**
  - Vector: dot, norm, vector sum, plane rotation
  - Matrix-vector: matrix-vector multiply, triangular solve, outer product update
  - Matrix-matrix: matrix multiply, triangular solve with multiple vectors, symmetric matrix update
- **Layer 1: Low-level math problems**
  - Linear systems $Ax = b$ (& determinants, etc.)
  - Least-squares problems $\min_x \|Ax - b\|_2$
  - Eigenvalue & singular value problems $Ax = \lambda x$
- **Layer 2: Higher-level math problems**
  - Statistical inference
  - Physics simulations

Linear algebra comes in (abstraction) layers

What are the responsibilities of a "linear algebra library"?

- **Layer -1: Fundamentals**
  - Multidimensional arrays & iteration
- **Layer 0: Performance primitives**
  - Vector: dot, norm, vector sum, plane rotation
  - Matrix-vector: matrix-vector multiply, triangular solve, outer product update
  - Matrix-matrix: matrix multiply, triangular solve with multiple vectors, symmetric matrix update
- **Layer 1: Low-level math problems**
  - Linear systems $Ax = b$ (& determinants, etc.)
  - Least-squares problems $\min_x \|Ax - b\|_2$
  - Eigenvalue & singular value problems $Ax = \lambda x$
- **Layer 2: Higher-level math problems**
  - Statistical inference
  - Physics simulations

Existing C++ features: parallel algorithms (C++17), mdspan (C++23), submdspan (C++26)

Proposals in flight: mdarray, mdspan padded layouts, SIMD

Classic domain of "numerical linear algebra," & Fortran libraries like LAPACK.

No C++ Standard proposals in flight; many third-party C++ libraries, such as NVIDIA's MatX, Eigen, & Armadillo.

Linear algebra comes in (abstraction) layers

What are the responsibilities of a “linear algebra library”?

- **Layer -1: Fundamentals**
  - Multidimensional arrays & iteration

Existing C++ features: parallel algorithms (C++17), mdspan (C++23), submdspan (C++26)

Proposals in flight: mdarray, mdspan padded layouts, SIMD

- **Layer 0: Performance primitives**
  - Vector: dot, norm, vector sum, plane rotation
  - Matrix-vector: matrix-vector multiply, triangular solve, outer product update
  - Matrix-matrix: matrix multiply, triangular solve with multiple vectors, symmetric matrix update

Our proposed library, std::linalg

- **Layer 1: Low-level math problems**
  - Linear systems $Ax = b$ (& determinants, etc.)
  - Least-squares problems $\min_x \|Ax - b\|_2$
  - Eigenvalue & singular value problems $Ax = \lambda x$

Classic domain of “numerical linear algebra,” & Fortran libraries like LAPACK.

No C++ Standard proposals in flight; many third-party C++ libraries, such as NVIDIA’s MatX, Eigen, & Armadillo.

- **Layer 2: Higher-level math problems**
  - Statistical inference
  - Physics simulations

Performance primitives

Separation of concerns, separation of expertise

- **Layer -1: Fundamentals**
  - Multidimensional arrays & iteration
- **Layer 0: Performance primitives**
  - Vector: dot, norm, vector sum, plane rotation
  - Matrix-vector: matrix-vector multiply, triangular solve, outer product update
  - Matrix-matrix: matrix multiply, triangular solve with multiple vectors, symmetric matrix update
- **Layer 1: Low-level math problems**
  - Linear systems $Ax = b$ (& determinants, etc.)
  - Least-squares problems $\min_x \|Ax - b\|_2$
  - Eigenvalue & singular value problems $Ax = \lambda x$
- **Layer 2: Higher-level math problems**
  - Statistical inference
  - Physics simulations

- Separate "performance primitives"
  - Asymptotically slow if implemented naïvely, like sort
  - For hardware experts to optimize
  - With readable, self-documenting names
- From mathematical algorithms
  - For mathematicians (e.g., experts in rounding error analysis) to implement correctly
  - Make algorithms fast by designing them to spend most of their time in std::linalg
- Layer 0 size trade-off:
  - completeness, vs.
  - what we can reasonably expect implementers to optimize
- Paraphrasing Dodson & Lewis, "Issues relating to extension of the Basic Linear Algebra Subprograms," ACM SIGNUM, 1985

Tour of std::linalg by example: Matrix-matrix multiply

$$C := \beta C + \alpha A B^T$$

```
#include <linalg>



mdspan A{A_raw_pointer, m, k};




mdspan B{B_raw_pointer, n, k};

mdspan C{C_raw_pointer, m, n};

matrix_product(par_unseq,
  scaled(alpha, A), transposed(B),
  scaled(beta, C), C);
```

mdspan represents views of matrices & vectors

$C := \beta C + \alpha A B^T$

- mdspan (C++23)
  - “View of a multidimensional array of elements”
  - Multiple implementations, some back-ported to C++17
- mdspan is to std::linalg, as ranges (of iterators) are to the C++ Standard Algorithms
  - Algorithms that operate on views
  - Natural extension of current C++ Standard Algorithms
- CTAD (constructor template argument deduction) makes construction in common cases easy
- submdspan (C++26) implements array slicing
  - We’ll see an example later

```
#include <linalg>

const double* A_raw_pointer =
  user_owned_storage.data();

mdspan A{A_raw_pointer, m, k};

mdspan B{B_raw_pointer, n, k};

mdspan C{C_raw_pointer, m, n};

matrix_product(par_unseq,
  scaled(alpha, A), transposed(B),
  scaled(beta, C), C);
```

mdspan layouts

$C := \beta C + \alpha A B^T$

- “extents”: the array’s dimensions
  - Encapsulated as std::extents
  - Arbitrary mix of compile-time &/or run-time values
  - Can set index type (here, int, instead of default size_t)
- mdspan has 2 customization options
  - Layout (this slide)
  - Accessor (next slide)
- Layout
  - Family of mappings, parameterized by extents
  - Layout mapping maps multi-D index (i, j) → 1-D offset k
- Different layout example
  - Nondefault column major (layout_left)
  - Instead of default row major (layout_right)

```
#include <linalg>

mdspan A{A_raw_pointer,
  layout_left::mapping{
    extents<int, m, k>{}}};

mdspan B{B_raw_pointer, n, k};

mdspan C{C_raw_pointer, m, n};

matrix_product(par_unseq,
  scaled(alpha, A), transposed(B),
  scaled(beta, C), C);
```

mdspan accessors

C := $\beta C + \alpha A B^T$

- Accessor: how to "get at the element"
  - Elements could live in memory, or somewhere else
    - e.g., in accelerator memory, across the network, on disk
  - Defines type of data handle ("pointer" or something else)
  - Maps (data handle, offset) → reference to element
  - reference: element_type&, or a proxy reference
- Custom accessor example
  - aligned_accessor (C++ Standard proposal P2897)
  - Expresses byte overalignment by using std::assume_aligned (C++20)
  - Useful for vectorization or special hardware

```
#include <linalg>

constexpr size_t byte_alignment =
  4 * sizeof(double);

mdspan A{A_raw_pointer,
  layout_left::mapping{
    extents<int, m, k>{}},
  aligned_accessor<const double,
    byte_alignment>{}};

mdspan B{B_raw_pointer, n, k};

mdspan C{C_raw_pointer, m, n};

matrix_product(par_unseq,
  scaled(alpha, A), transposed(B),
  scaled(beta, C), C);
```

mdspan also encapsulates scaling & (conjugate) transpose

C := $\beta C + \alpha A B^T$

- In earlier Fortran library example: Scalars & whether to transpose are separate function arguments
- std::linalg represents both as mdspan
- scaled(alpha, A) is an mdspan with a different accessor
- transposed(B) is an mdspan with flipped extents & a different layout
  - E.g., layout_right n x k → layout_left k x n
- For matrices of complex numbers:
  - conjugated(B): complex conjugate of each element
  - conjugate_transposed(B): conjugated(transposed(B))

```
#include <linalg>


mdspan A{A_raw_pointer, m, k};

mdspan B{B_raw_pointer, n, k};

mdspan C{C_raw_pointer, m, n};


matrix_product(par_unseq,

  scaled(alpha, A),
  transposed(B),
  scaled(beta, C),

  C);
```

std::linalg adds to C++'s existing parallel algorithms

$C := \beta C + \alpha A B^T$

- C++17 adds parallel algorithms to Standard Library
- Parallel algorithms take an ExecutionPolicy
  - Expresses user's promises & intent
  - par_unseq (Standard): user promises that it is safe to
    - run on thread(s) other than the calling thread, &
    - "interleave" operations (e.g., for vectorization)
  - 4 Standard policies, & vendors can add more
- "On-ramp" to vendor-specific performance
  - NVIDIA's implementation has policies to run on a given CUDA stream, &/or run asynchronously
- std::linalg's algorithms are all parallel algorithms

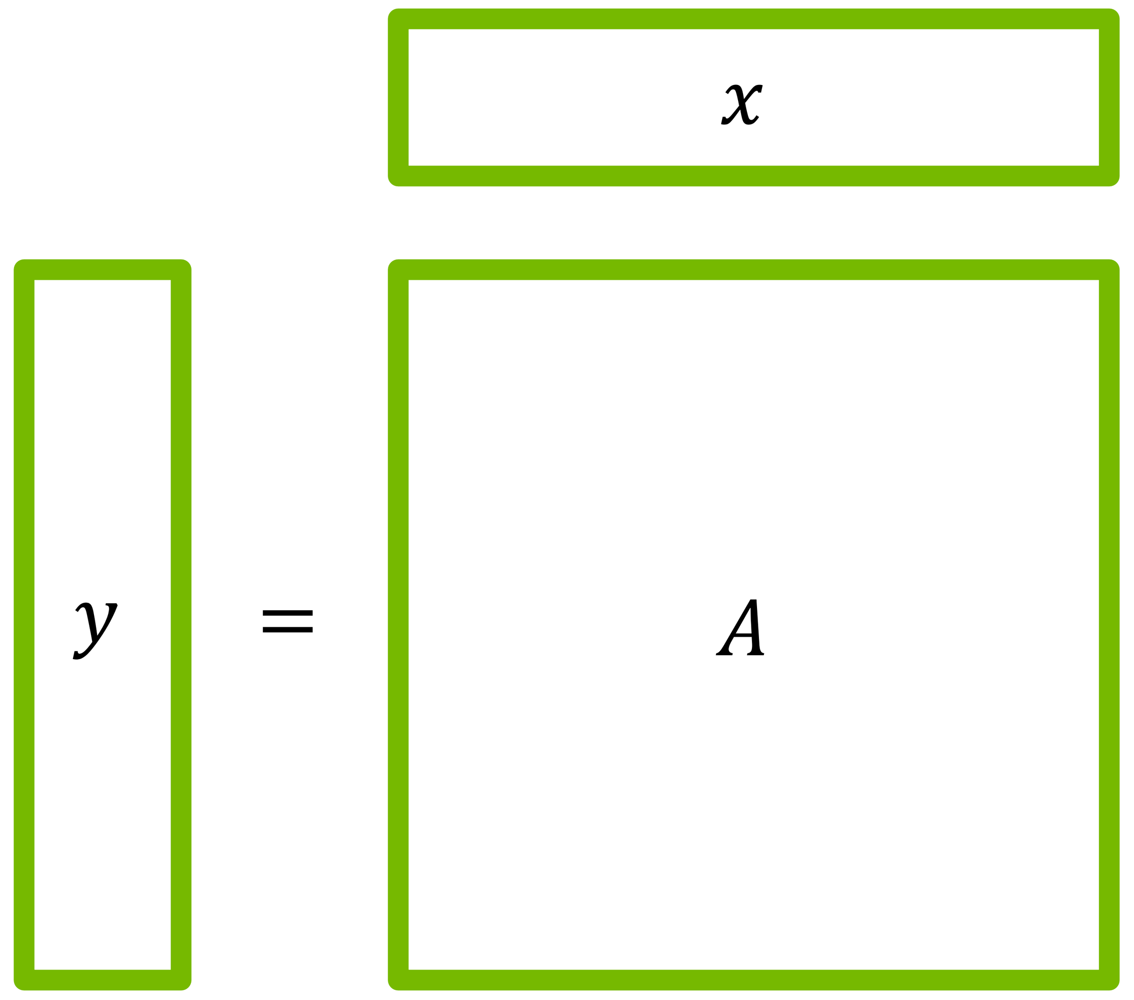
```
#include <linalg>

mdspan A{A_raw_pointer, m, k};
mdspan B{B_raw_pointer, n, k};
mdspan C{C_raw_pointer, m, n};

matrix_product(par_unseq,
  scaled(alpha, A),
  transposed(B),
  scaled(beta, C),
  C);
```

std::linalg generalizes existing C++ parallel algorithms

Implementable as transform + reduce, but exposes more optimizations

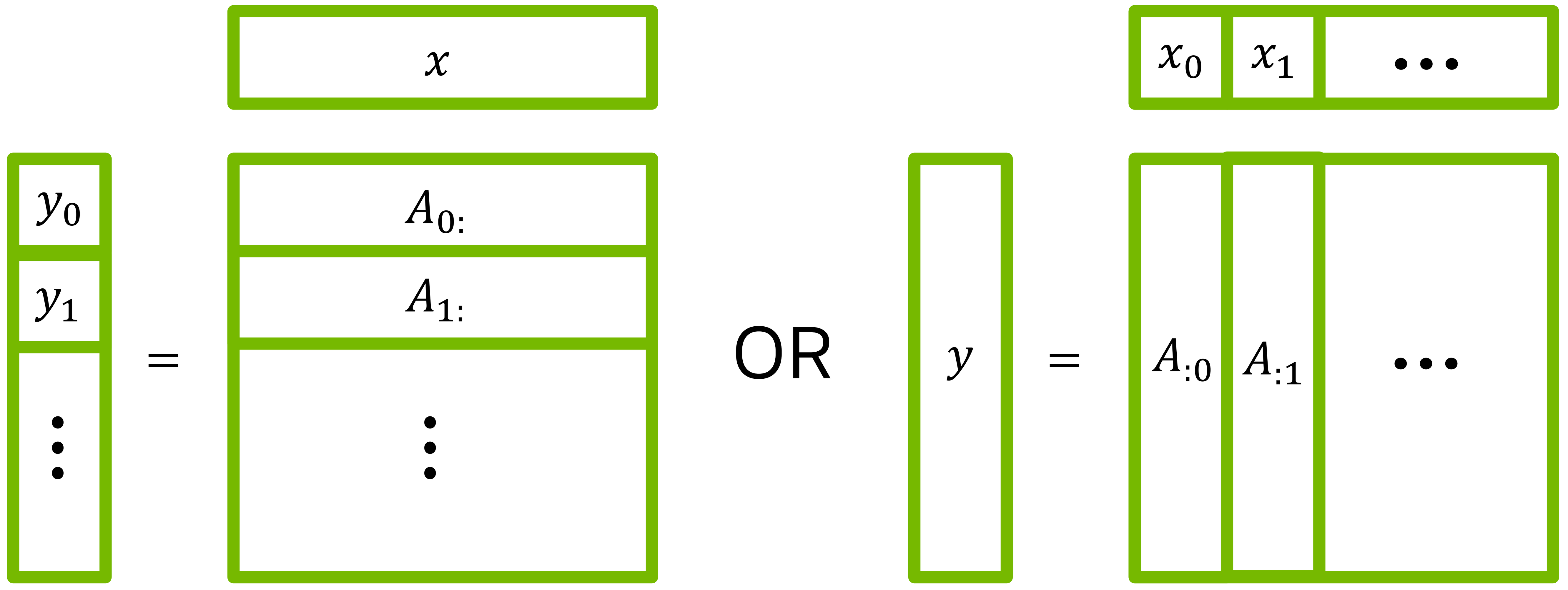
Parallel algorithms	What they do
transform (unary)	z[i] = f(x[i])
transform (binary)	z[i] = g(x[i], y[i])
reduce	x[0] + x[1] + … + x[N-1], or max(x[0], x[1], …, x[N-1])

- std::linalg's vector algorithms
  - Vector add: binary transform (+)
  - Norms: unary transform (abs), then reduce (+ or max)
  - Dot product: binary transform (*), then reduce (+)
- Matrix-vector & matrix-matrix algorithms
  - Equivalent to wrapping vector or matrix-vector algorithms in a transform over one extent
  - e.g., matrix-vector product: A.extent(0) dot products, OR A.extent(1) vector adds
- "Linearity" of linear algebra exposes optimizations
  - Loop transformations, temporary storage, even different algorithms (e.g., Strassen matrix multiply)



$y = Ax$

$$y_r = \sum_{c=0}^{N-1} A_{rc} x_c$$



y_r = dot($A_{r:}$, x)

y_r = add($A_{:c}, x_c)_r$

std::linalg is based on the BLAS

Basic Linear Algebra Subroutines

- BLAS is a standard Fortran & C library
  - Standard embraced by industry, labs, & academia
  - Many system vendors have optimized BLAS
    - e.g., AMD, ARM, Cray, IBM, Intel, NVIDIA, Xilinx
- Can implement std::linalg by calling BLAS
  - If types permit, else fall back to generic C++
- "Based on":
  - Same set of algorithms
  - Same design essence: work on views of users' data
  - But translated into C++ idioms

Fortran BLAS	C++ std::linalg
`CALL dgemm("N", "T", m, n, k, alpha, A, LDA, B, LDB, beta, C, LDC)`	`matrix_product( scaled(alpha, A), transposed(B), scaled(beta, C), C);`

- BLAS history shows its value as a design basis
  - 50 years of history
  - Codesigned with Layer 1 algorithms
  - Evolved with computer architectures
  - "Optimal" in a formal sense

1973: BLAS proposal & draft

1979: BLAS 1 paper

1988: BLAS 2 paper
1990: BLAS 3 paper

2002: BLAS Standard published

2019: std::linalg proposal (R0)

1971: Handbook for Automatic Computation: Vol II, Linear Algebra

Grandparent of the BLAS

James H. Wilkinson
(1919 – 1986)
(with his 1970 Turing Award),
Author 1

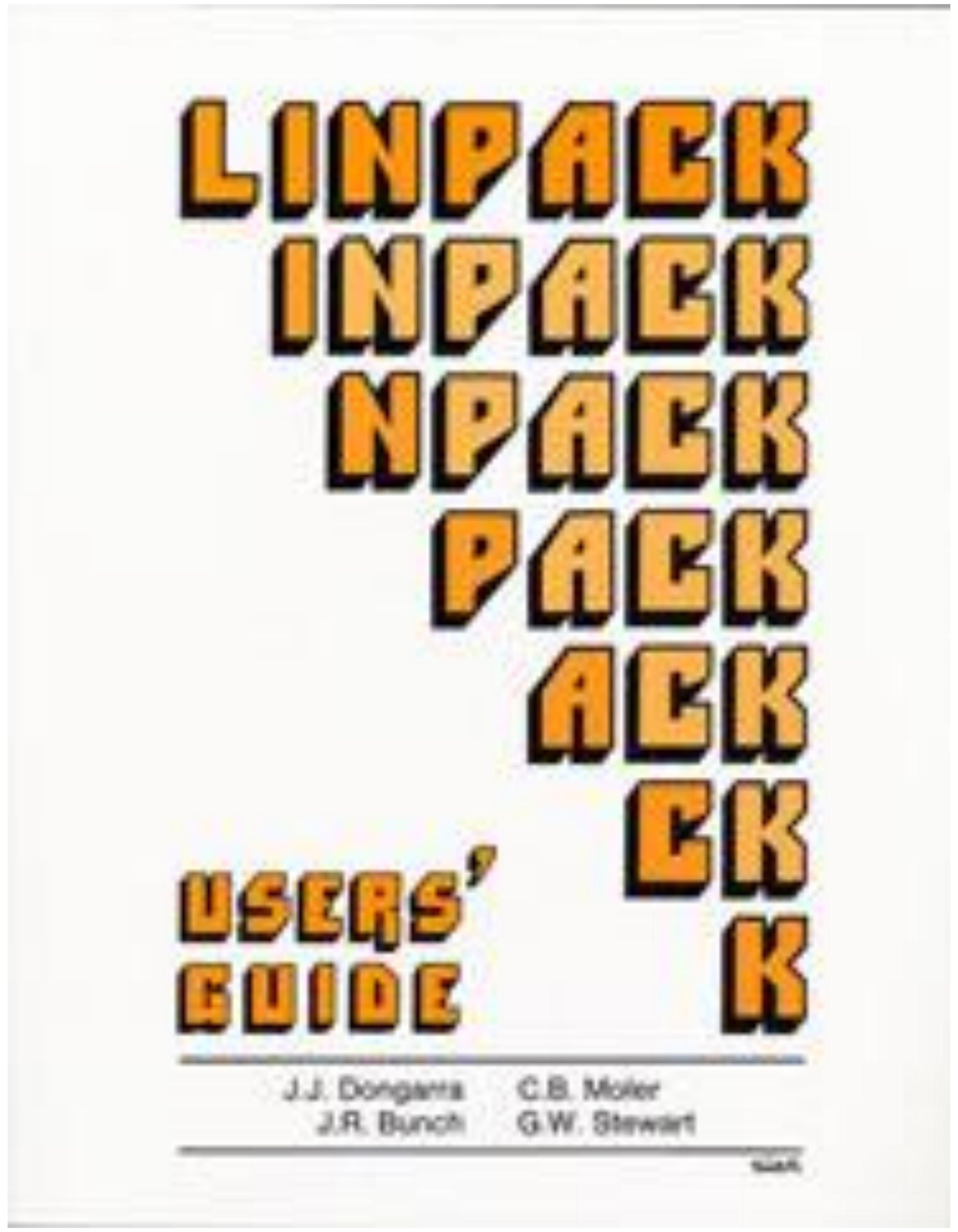
Christian Reinsch
(1934 – 2022),
Author 2

Friedrich L. Bauer
(1924 – 2015),
Chief Editor

- "…continuous efforts by acknowledged experts over more than ten years" (SIAM Review, 14 (4), 1972)
- Established what problems "linear algebra libraries" solve
  - Linear systems
  - Linear least squares
  - Eigenvalue & singular value problems
- Algorithms in ALGOL 60
- Chief editor, F. L. Bauer, contributed to ALGOL 60 & chaired 1968 NATO software engineering conference

1970s: Implementing the Handbook

EISPACK birthed the BLAS; LINPACK put it to the test



LINPACK Users' Guide (1979)
Dongarra, Moler, Bunch, & Stewart

- EISPACK: solves eigenvalue & singular value problems
  - 1971-72: EISPACK 1
- 1973: BLAS proposed & drafted
- BLAS revised in 2 meetings
  - 1974 (many changes)
  - 1975 (fewer changes)
- LINPACK: solves linear systems & linear least-squares problems
  - Specifically written to use BLAS
  - 1976 – 79: LINPACK 1
  - 1979: "BLAS 1" paper
  - Includes a benchmark that solves a linear system, using algorithms that spend most of their time in BLAS

Jack Dongarra (born 1950), 2021 Turing Award winner, coauthored the BLAS 2 & 3 papers & the LINPACK benchmark (associated with the TOP500 list of supercomputers).

1980’s: Evolving computer architectures expanded BLAS

Architecture	Mainframe	Vector	Parallel / memory hierarchy (“cache-based”)
Representative system	 Univac 1108 (1970)	 Cray 1 (1976)	 CM-2 (1986-87)
Code optimization strategy	Hand-optimize key loops in assembly	Fuse loops to amortize latency & maximize instruction-level parallelism	Maximize data reuse & minimize communication via algorithms with low “surface-to-volume” ratio
BLAS level	1	2	3
BLAS operations	Dot products, vector norms, vector sum, plane rotations	Matrix-vector & outer products, triangular solves	Matrix-matrix multiply, multiple {triangular solves, outer products}
Development years	1973 - 79	1984 - 88	1987 - 1990

1980's: Evolving computer architectures expanded BLAS

Architecture	Mainframe	Vector	Parallel / memory hierarchy ("cache-based")
Representative system	 Univac 1108 (1970)	 Cray 1 (1976)	 CM-2 (1986-87)
Code optimization strategy	Hand-optimize key loops in assembly	Fuse loops to amortize latency & maximize instruction-level parallelism	Maximize data reuse & minimize communication via algorithms with low "surface-to-volume" ratio
BLAS level	1	2	3
BLAS operations	Dot products, vector norms, vector sum, plane rotations	Matrix-vector & outer products, triangular solves	Matrix-matrix multiply, multiple {triangular solves, outer products}
Development years	1973 - 79	1984 - 88	1987 - 1990

BLAS level reflects:

1. **Time published**
2. **Number of nested loops in a textbook sequential implementation**
3. **Increasing potential data reuse, loop fusion, & parallelism**

1980’s: Evolving computer architectures expanded BLAS

Architecture	Mainframe	Vector	Parallel / memory hierarchy (“cache-based”)
Representative system	 Univac 1108 (1970)	 Cray 1 (1976)	 CM-2 (1986-87)
Code optimization strategy	Hand-optimize key loops in assembly	Fuse loops to amortize latency & maximize instruction-level parallelism	Maximize data reuse & minimize communication via algorithms with low “surface-to-volume” ratio
BLAS level	1	2	3
BLAS operations	Dot products, vector norms, vector sum, plane rotations	Matrix-vector & outer products, triangular solves	Matrix-matrix multiply, multiple {triangular solves, outer products}
Development years	1973 - 79	1984 - 88	1987 - 1990

BLAS level reflects:

1. **Time published**
2. **Number of nested loops in a textbook sequential implementation**
3. **Increasing potential data reuse, loop fusion, & parallelism**

Competed in ‘80s & early ‘90s; see 1991 New York Times article “Killer Micros”

1980’s: Evolving computer architectures expanded BLAS

Architecture	Mainframe	Vector	Parallel / memory hierarchy (“cache-based”)
Representative system	 Univac 1108 (1970)	 Cray 1 (1976)	 CM-2 (1986-87)
Code optimization strategy	Hand-optimize key loops in assembly	Fuse loops to amortize latency & maximize instruction-level parallelism	Maximize data reuse & minimize communication via algorithms with low “surface-to-volume” ratio
BLAS level	1	2	3
BLAS operations	Dot products, vector norms, vector sum, plane rotations	Matrix-vector & outer products, triangular solves	Matrix-matrix multiply, multiple {triangular solves, outer products}
Development years	1973 - 79	1984 - 88	1987 - 1990

Competed in ‘80s & early ‘90s; see 1991 New York Times article “Killer Micros”

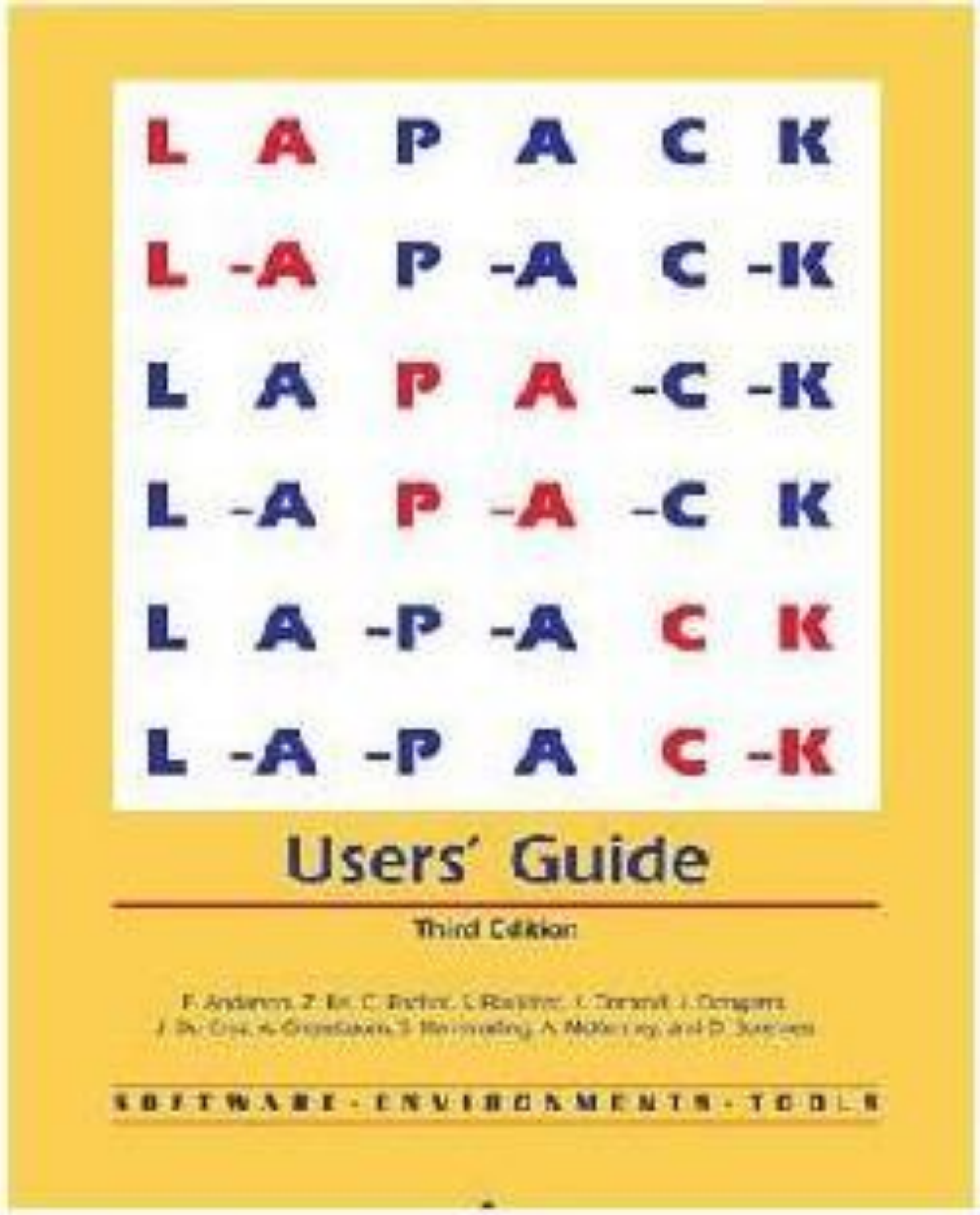
Describes modern fast computers!

“Flops are free, bandwidth is money, latency is physics” – Prof. Kathy Yelick, UC Berkeley

This is why BLAS, especially BLAS 3, remains relevant.

BLAS is a stable interface

Form building blocks for provably efficient algorithms



- LAPACK: Successor to EISPACK & LINPACK
- Codesigned with BLAS 3
- Proposed 1987, released 1992
- Current release: 3.11 (2022)

- Lasted through 2 algorithm "waves"
  - Each effectively "updated the Handbook"
  - 1980's – 90's: LAPACK's block algorithms
  - 2000's – 10's: "Communication-avoiding algorithms" in various libraries (e.g., MAGMA)
- Algorithms proven optimal [1]
  - Minimize data movement
  - Maximize parallelism
  - Given constraints on rounding error
- Thus, we do not foresee the need for a future "BLAS 4" of radically different algorithms [2]

[1] "Communication lower bounds & optimal algorithms for numerical linear algebra" (Acta Numerica 2014).

[2] Ask me about Batched BLAS & P2901 afterwards!

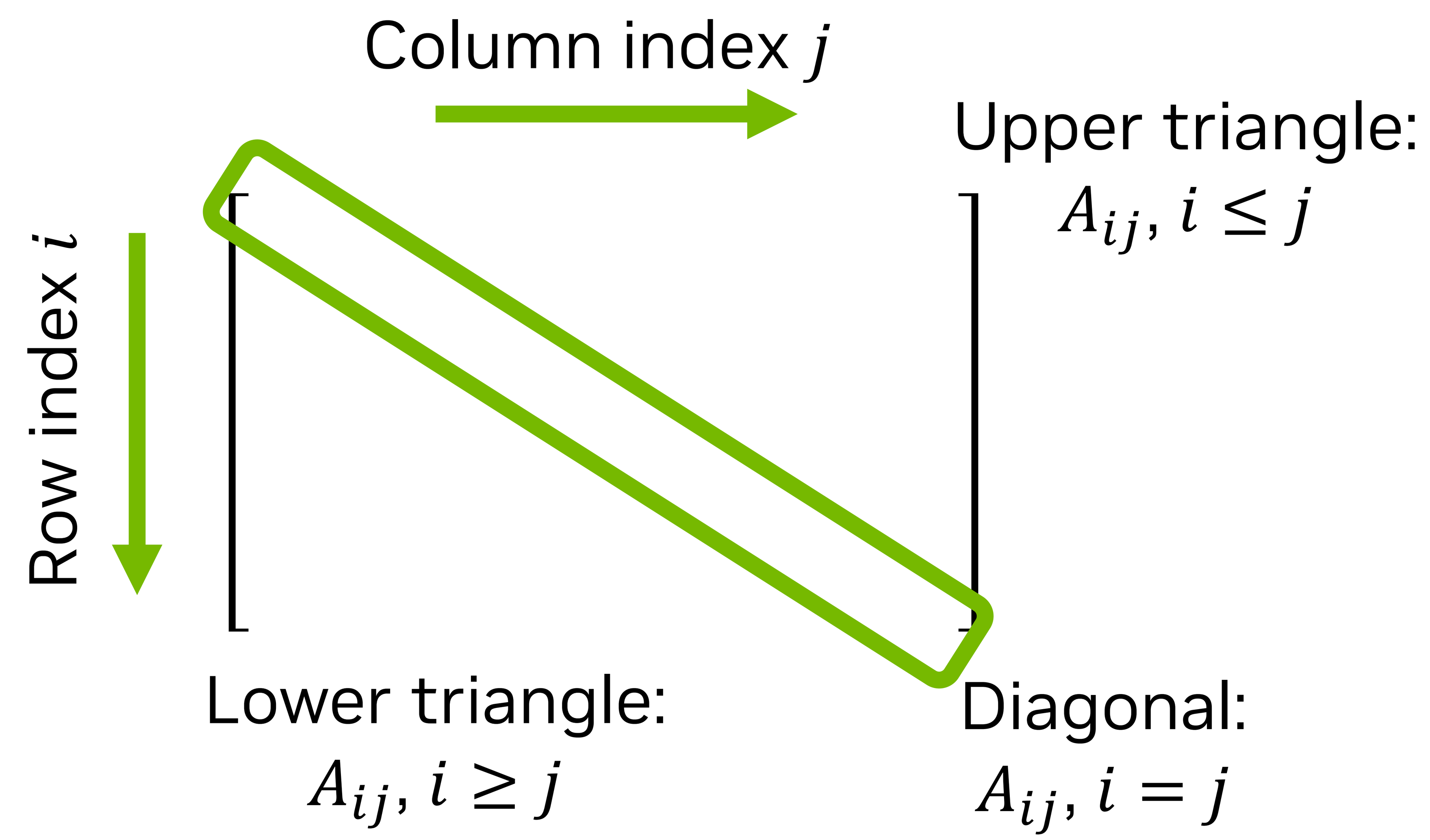
Detailed example: Cholesky matrix factorization

Solve a symmetric positive definite (SPD) linear system Ax = b

André-Louis Cholesky (1875 - 1918), French artillery officer, geodesist, & mathematician

- Solve $Ax = b$, where matrix A is
  - Symmetric: $A_{ij} = A_{ji}$, and
  - Positive definite: $x^T Ax > 0$ for all nonzero x
  - A common matrix structure that linear algebra is good at exploiting
- Factor A into LL^T, where L is lower triangular
  - Reduces solving $Ax = b$ to 2 triangular systems
  - First $Lc = b$, then $L^T x = c$
  - Can reuse for different b

Parts of a matrix



$$\begin{bmatrix} 8 & 0 & 0 \\ -2 & 16 & 0 \\ 1 & -4 & 32 \end{bmatrix} \times \begin{bmatrix} 8 & -2 & 1 \\ 0 & 16 & -4 \\ 0 & 0 & 32 \end{bmatrix} = \begin{bmatrix} 64 & -16 & 8 \\ -16 & 252 & -66 \\ 8 & -66 & 1009 \end{bmatrix}$$

Example of a Cholesky factorization $LL^T = A$

Detailed example: Cholesky matrix factorization

Solve a symmetric positive definite linear system Ax = b

André-Louis Cholesky (1875 - 1918), French artillery officer, geodesist, & mathematician

- Solve $Ax = b$, where matrix A is
  - Symmetric: $A_{ij} = A_{ji}$, and
  - Positive definite: $x^T Ax > 0$ for all nonzero x
  - A common matrix structure that linear algebra is good at exploiting
- Factor A into LL^T, where L is lower triangular
  - Reduces solving $Ax = b$ to 2 triangular systems
  - First $Lc = b$, then $L^T x = c$
  - Can reuse for different b

- Key std::linalg feature: Symmetry is an algorithm, not a data structure
  - No special “symmetric matrix type”
  - Only read lower or upper triangle
  - Interpretation of the other triangle
- Different named algorithms for different matrix structures
  - symmetric_*
  - hermitian_*
  - triangular_*

$$\begin{bmatrix} 8 & 0 & 0 \\ -2 & 16 & 0 \\ 1 & -4 & 32 \end{bmatrix} \times \begin{bmatrix} 8 & -2 & 1 \\ 0 & 16 & -4 \\ 0 & 0 & 32 \end{bmatrix} = \begin{bmatrix} 64 & -16 & 8 \\ -16 & 252 & -66 \\ 8 & -66 & 1009 \end{bmatrix}$$

Example of a Cholesky factorization $LL^T = A$

Cholesky factorization function signature

Illustrates std::linalg idioms

```
// Index of first zero or NaN pivot (bad),
// else nullopt (good).
template<class ValueType,
  size_t Ext0, size_t Ext1,
  class Layout,
  class Accessor>

requires(
  ! std::is_const_v<ValueType> &&
  Layout::is_always_unique())

std::optional<size_t>

cholesky_factor(
  mdspan<
    ValueType,
    extents<size_t, Ext0, Ext1>,
    Layout,
    Accessor> A);
```

Cholesky factorization function signature

Illustrates std::linalg idioms

```
// Index of first zero or NaN pivot (bad),
// else nullopt (good).
template<class ValueType,
  size_t Ext0, size_t Ext1,
  class Layout,
  class Accessor>

requires(
  ! std::is_const_v<ValueType> &&
  Layout::is_always_unique())

std::optional<size_t>

cholesky_factor(
  mdspan<
    ValueType,
    extents<size_t, Ext0, Ext1>,
    Layout,
    Accessor> A);
```

Cholesky can fail if the matrix is not positive definite. Return nullopt on success, else return least index k where the "pivot" A_{kk} (possibly after changes to A) is zero or NaN.

mdspan represents a "view of a matrix's elements." Views: no container, no copies, no allocation. Factorizations idiomatically modify the data in place.

Cholesky factorization function signature

Illustrates std::linalg idioms

```
// Index of first zero or NaN pivot (bad),
// else nullopt (good).
template<class ValueType,
  size_t Ext0, size_t Ext1,
  class Layout,
  class Accessor>

requires(
  ! std::is_const_v<ValueType> &&
  Layout::is_always_unique())

std::optional<size_t>

cholesky_factor(
  mdspan<
    ValueType,
    extents<size_t, Ext0, Ext1>,
    Layout,
    Accessor> A);
```

ValueType: type for which A[r, c] is a reference.
std::linalg algorithms work for any "number-y" types.
mdspan<const T, …> is a view-of-const; not writeable.

Ext0, Ext1: Extents (dimensions). Either or both can be dynamic_extent (run-time value) or compile-time values.

Layout, Accessor: std::linalg algorithms are generic on how we arrange & store the matrix's elements.
Layout must be unique, though, else we don't know how to write to it.

Nonunique layout example: "constant matrix" (every (r, c) maps to offset 0).

Cholesky factorization function signature

Illustrates std::linalg idioms

```
// Index of first zero or NaN pivot (bad),
// else nullopt (good).
template<class ValueType,
  size_t Ext0, size_t Ext1,
  class Layout,
  class Accessor>

requires(
  ! std::is_const_v<ValueType> &&
  Layout::is_always_unique())

std::optional<size_t>

cholesky_factor(
  mdspan<
    ValueType,
    extents<size_t, Ext0, Ext1>,
    Layout,
    Accessor> A);
```

mdspan represents a "view of a matrix's elements."

There is no "symmetric matrix type" or layout.
Symmetry is just an interpretation of the data.
Access only the "lower triangle" A_{rc} with $r \geq c$.

Factorization reinterprets A on output
as a lower triangular matrix L.

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

Base case:
For a 1 x 1 matrix, $A = \sqrt{A}\sqrt{A}$ is a valid Cholesky factorization as long as A is positive.

Recursion down to n = 1 is slower than optimal ($\log_2 B$ recursion steps use less than optimal block size B for A21 and A22 updates), but easier to explain.

Base case:
0 x 0 matrix: Trivial success

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

Start recursion by partitioning the matrix into equal (+/- 1) submatrices.

$$A \rightarrow \begin{bmatrix} A_{11} & \mathbf{0} \\ A_{21} & A_{22} \end{bmatrix}$$

Don't access the upper triangle.

Use submdspan (C++26) to create subviews (slices).

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

$$\begin{bmatrix} \boldsymbol{A_{11}} & \boldsymbol{0} \\ \boldsymbol{A_{21}} & \boldsymbol{A_{22}} \end{bmatrix} \rightarrow \begin{bmatrix} \boldsymbol{L_{11}} & \boldsymbol{0} \\ \boldsymbol{A_{21}} & \boldsymbol{A_{22}} \end{bmatrix}$$

Recursively factor $\boldsymbol{A_{11} \rightarrow L_{11} L_{11}^T}$ in place.

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

$$\begin{bmatrix} L_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} & 0 \\ L_{21} = L_{11}^{-T} A_{21} & A_{22} \end{bmatrix}$$

Solve ${L_{11}}^T L_{21} = A_{21}$ for L_{21} in place.

transposed(A11): reinterpret L_{11} as ${L_{11}}^T$.
Result is upper triangular & layout_right.

"explicit_diagonal": opposite of "implicit_unit_diagonal," used for other factorizations.

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

$$\begin{bmatrix} L_{11} & 0 \\ L_{21} & A_{22} \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} = A_{22} - L_{21}L_{21}^T \end{bmatrix}$$

Overwrite A_{22} with $L_{22} = A_{22} - L_{21}L_{21}^T$.

This generalizes a symmetric outer product.
We call this a "symmetric rank-k update" of A_{22} with the "rank k" (= n1) matrix A_{21}.

Cholesky factorization function body

Reads like a description of the algorithm

```
const size_t n = min(A.extent(0), A.extent(1));
if (n == 1) {
  if (A[0,0] <= ValueType{} || std::isnan(A[0,0])) {
    return {size_t(0)};
  }
  A[0,0] = std::sqrt(A[0,0]);
}
else if (n != 0) {
  const size_t n1 = n / 2;  // [A11, ___]
                            // [A21, A22]
  auto A11 = submdspan(A, tuple{0, n1}, tuple{0, n1});
  auto A21 = submdspan(A, tuple{n1, n}, tuple{0, n1});
  auto A22 = submdspan(A, tuple{n1, n}, tuple{n1, n});

  const auto info1 = cholesky_factor(A11);
  if (info1.has_value()) { return info1; }

  triangular_matrix_matrix_right_solve(transposed(A11),
    upper_triangle, explicit_diagonal, A21);
  symmetric_matrix_rank_k_update(-ValueType(1.0),
    A21, A22, lower_triangle);

  const auto info2 = cholesky_factor(A22);
  if (info2.has_value()) { return {info2.value() + n1}; }
}
return std::nullopt;
```

$$\begin{bmatrix} L_{11} & 0 \\ L_{11}^{-T} A_{21} & A_{22} - L_{11}^{-T} A_{21} A_{21}^T L_{11}^T \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}$$

Recursively factor A_{22}.

Solve linear system Ax = b with factorization result

2 lines!

```
triangular_matrix_vector_solve(A,
  lower_triangle, explicit_diagonal, b, x);

triangular_matrix_vector_solve(transposed(A),
  upper_triangle, explicit_diagonal, x);
```

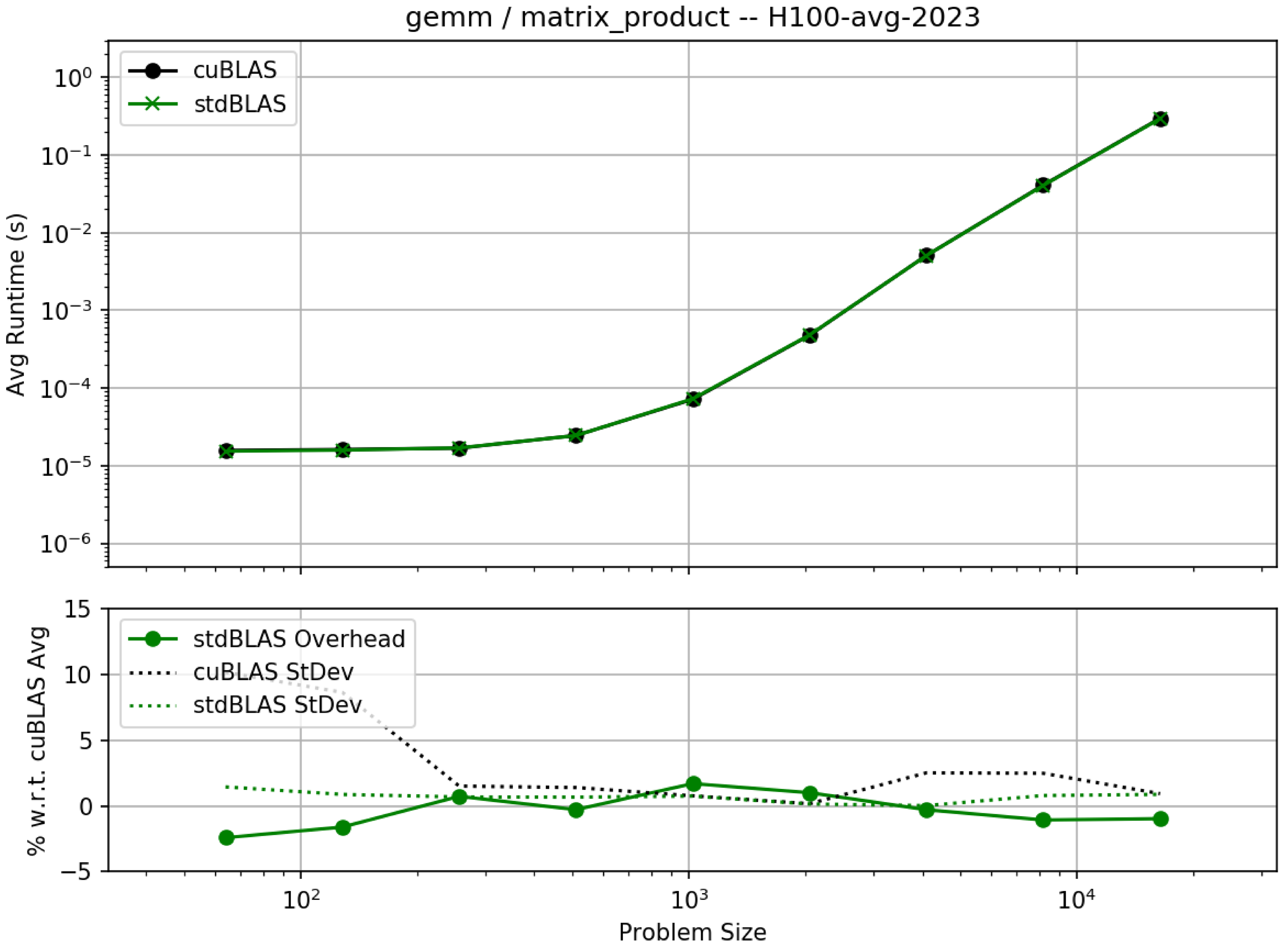
Solve $Lc = b$ for c, using x to store c

Solve $L^T x = c$ for x

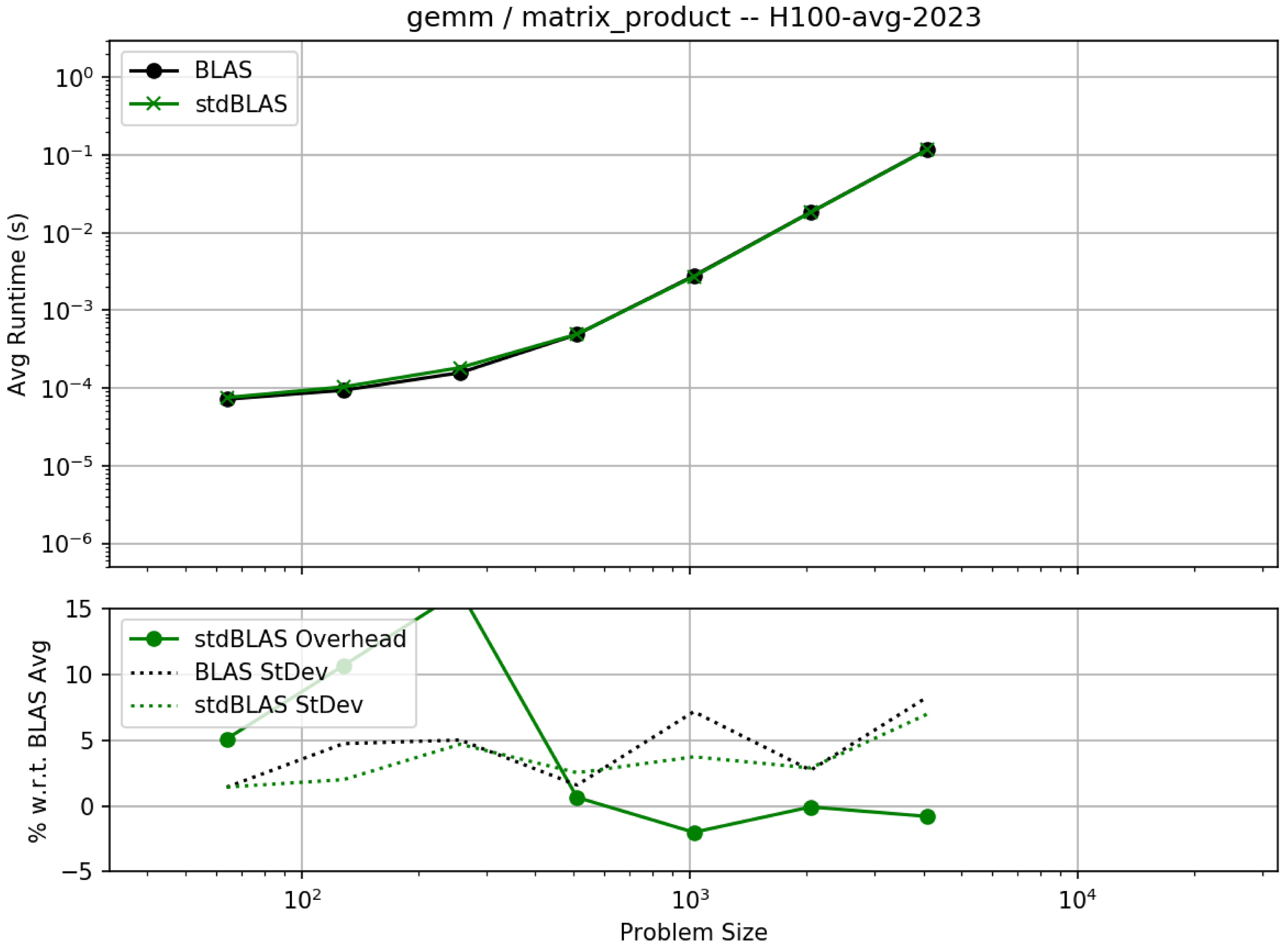
$A = LL^T$, so $Ax = b$ reduces to $LL^T x = b$.

Performance results: std::linalg vs. “raw” BLAS

Double-precision real matrix product



NVIDIA H100,
cuBLAS 12.2.1.6,
HPC SDK 23.7



Intel Xeon Gold 6338,
OpenBLAS 0.3.23

Summary

- std::linalg is a C++ linear algebra library
- Performance primitives
  - Encapsulate hardware-specific optimizations
  - Let mathematicians focus on algorithm development
- Idiomatic C++ interface via
  - mdspan (a multidimensional array view)
  - C++ parallel algorithms
- Based on the BLAS (Basic Linear Algebra Subroutines)
  - Fortran & C standard library
  - Codesigned with algorithms
  - Many optimized implementations
  - Over 50 years of history & practice

- std::linalg implementations
  - Reference: https://github.com/kokkos/stdBLAS
  - NVIDIA's: in the HPC SDK (free download)
- To learn more:
  - C++ Standard Library proposal P1673: https://wg21.link/p1673
  - How we designed std::linalg as the minimal idiomatic C++ interface wrapping the BLAS: https://wg21.link/p1674
  - Contact me at Mark Hoemmen <mhoemmen@nvidia.com>

NVIDIA

BLAS functions & their std::linalg equivalents

BLAS1 name(s)	std::linalg name(s)	BLAS 2 name(s)	std::linalg name(s)	BLAS 3 name(s)	std::linalg name(s)
xLARTG	givens_rotation_setup	xGEMV	matrix_vector_product	xGEMM	matrix_product
xROT	givens_rotation_apply	xSYMV	symmetric_matrix_vector_product	xSYMM	symmetric_matrix_product
xSWAP	swap_elements	xHEMV	hermitian_matrix_vector_product	xHEMM	hermitian_matrix_product
xSCAL	scale, scaled	xTRMV	triangular_matrix_vector_product	xTRMM	triangular_matrix_product
xCOPY	copy	xGER(U)	matrix_rank_1_update	xSYRK	symmetric_matrix_rank_k_update
xAXPY	add, scaled	xGERC	matrix_rank_1_update_c	xHERK	hermitian_matrix_rank_k_update
xDOT(U)	dot	xSYR	symmetric_matrix_rank_1_update	xSYR2K	symmetric_matrix_rank_2k_update
xDOTC	dotc	xHER	hermitian_matrix_rank_1_update	xHER2K	hermitian_matrix_rank_2k_update
(xLASSQ)	vector_sum_of_squares	xSYR2	symmetric_matrix_rank_2_update	xTRSM	triangular_matrix_matrix_left_solve, triangular_matrix_matrix_right_solve
xNRM2	vector_two_norms	xHER2	hermitian_matrix_rank_2_update	xHER2	hermitian_matrix_rank_2_update
xASUM	vector_abs_sum				
xIAMAX	idx_abs_max				
N/A	matrix_frob_norm, matrix_one_norm, matrix_inf_norm				

References

Most of them freely available online

G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwarz, "Communication lower bounds and optimal algorithms for numerical linear algebra," Acta Numerica, Vol. 23, May 2014, pp. 1 – 155. Available online: https://doi.org/10.1017/S0962492914000038 [last accessed 2023/09/12].

David S. Dodson and John G. Lewis, "Issues relating to extension of the Basic Linear Algebra Subprograms," ACM SIGNUM Newsletter, Vol. 20, No. 1, Jan. 1985, pp. 19 – 22. Available online: https://doi.org/10.1145/1057935.1057937 [last accessed 2023/09/14].

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," ACM Transactions on Mathematical Software (TOMS), Vol. 14, No. 1, pp. 1 – 17, Mar. 1988. Available online: https://dl.acm.org/doi/10.1145/42288.42291 [last accessed 2023/09/12].

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," ACM TOMS, Vol. 16, No. 1, pp. 1 – 17, Mar. 1990. Available online: https://dl.acm.org/doi/10.1145/77626.79170 [last accessed 2023/09/12].

James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen, "Prospectus for the Development of a Linear Algebra Library for High-Performance Computers," Technical Memorandum No. 97, Mathematics and Computer Science Division, Argonne National Laboratory, Sep. 1987.

R. J. Hanson, F. T. Krogh, and C. L. Lawson, "Improving the efficiency of portable software for linear algebra," ACM SIGNUM Newsletter, Vol. 8, No. 4, Oct. 1973, p. 16. Available online: https://doi.org/10.1145/1052646.1052653 [last accessed 2023/09/12].

Ibid., "A Proposal for Standard Linear Algebra Subprograms," Technical Memorandum 33-660, Jet Propulsion Laboratory, Nov. 1973. Available online: https://ntrs.nasa.gov/api/citations/19740005175/downloads/19740005175.pdf [last accessed 2023/09/12].

A. S. Householder, The Theory of Matrices in Numerical Analysis, Dover, 1964.

A. M. Turing, "Proposals for Development in the Mathematics Division of an Automatic Computing Engine (ACE)," Report E.882, The National Physical Laboratory, Feb. 1946. Available online: https://www.npl.co.uk/getattachment/about-us/History/Famous-faces/Alan-Turing/turing-proposal-Alan-LR.pdf?lang=en-GB [last accessed 2023/09/14].

J. H. Wilkinson and C. Reinsch, Handbook for Automatic Computation, Vol. II, Linear Algebra, Springer, 1971.

Image credits

- Slide 5 (pipe)
  - Mark Hoemmen, taken 2023/09/21
- Slide 6 (Householder)
  - Oak Ridge National Laboratory
  - https://en.wikipedia.org/wiki/Alston_Scott_Householder#/media/File:Alston_Householder.jpg
- Slide 7 (onions)
  - Lali Masriera
  - https://en.wikipedia.org/wiki/File:Cortando_cebolla.jpg
- Slide 20 (Handbook authors & editor)
  - https://en.wikipedia.org/wiki/James_H._Wilkinson#/media/File:James_H._Wilkinson.jpg (ACM)
  - https://blogs.mathworks.com/cleve/2022/10/23/christian-reinsch-roland-bulirsch-and-the-svd/ (Christoph Zenger, via Clever Moler)
  - https://inf.ethz.ch/de/news-und-veranstaltungen/spotlights/infk-news-channel/2015/04/nachruf-bauer.html (ETH Zürich)
- Slide 21 (LINPACK)
  - https://awards.acm.org/award-recipients/dongarra_3406337 (ACM)
- Slide 22 (BLAS 1,2,3 computer examples)
  - https://upload.wikimedia.org/wikipedia/commons/8/85/Univac_1108_Census_Bureau.jpg (US Census Bureau)
  - https://www.computerhistory.org/revolution/supercomputers/10/7/3 (Computer History Museum)
  - https://tamikothiel.com/cm/cm-image.html (Steve Grohe)
- Slide 27 (Cholesky)
  - https://en.wikipedia.org/wiki/File:Andre_Cholesky.jpg (public domain)