High Performance Linear Algebra

Lecture 5: Continuing with BLAS, BLAS Level 1: DOT and level 2: GEMV

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

November 24, 2025 — 14.00:16.00

Dipartimento di Matematica
Università di Pisa

Back from the past

- Last time we have seen:
  - Introduction to the general BLAS idea
  - BLAS Level 1: vector-vector operations
  - Performance considerations
- Today we will continue with:
  - BLAS Level 1: DOT, NRM2 and Givens rotations
  - BLAS Level 2: GEMV

Table of Contents

2 BLAS Level 1: DOT

DOT: Dot Product

Another important operation is the dot product, which is defined as:

$$c = x^\top y = \sum_{i=1}^{n} x_i y_i$$

- The dot product is a scalar product of two vectors
- Sum of the products of corresponding elements
- BLAS routine (double precision): `ddot`

```
c = ddot(n, x, incx, y, incy)
```

where `incx` and `incy` are the increments for the input vectors x and y.

Fortran Functions: Declaration

- Functions return a single value
- Must declare return type
- Two ways to declare:

Method 1: Classic Fortran style

```fortran
real(real64) function my_function(x, y)
    real(real64), intent(in) :: x, y
    my_function = x + y
end function my_function
```

Method 2: Result clause

```fortran
function my_function(x, y) result(z)
    real(real64), intent(in) :: x, y
    real(real64) :: z
    z = x + y
end function my_function
```

Fortran Functions: External Functions

- Some BLAS routines are **external** functions: compiled separately, **no** interface
- Must declare in calling program

```fortran
program main
    use iso_fortran_env, only: real64
    implicit none
    real(real64) :: ddot   ! Declaration
    real(real64) :: x(3), y(3), result
    x = [1.0, 2.0, 3.0]
    y = [4.0, 5.0, 6.0]
    result = ddot(3, x, 1, y, 1)
end program main
```

Important

Without declaration and without `implicit none`, the compiler assumes `ddot` returns `real(real32)`, possibly causing errors.

Fortran Functions vs Subroutines

Functions

- Return single value
- Used in expressions
- Example: `ddot`

```
c = ddot(n, x, 1, y, 1)
```

Subroutines

- Return via arguments
- Called with `call`
- Example: `daxpy`

```
call daxpy(n, a, x, 1, y, 1)
```

BLAS Convention

- Scalar results: functions (`ddot`, `dnrm2`)
- Vector/matrix results: subroutines (`daxpy`, `dgemv`)

Passing Functions as Arguments

- Sometimes we need to pass a function to a subroutine
- Common in numerical algorithms (integration, optimization)
- Fortran provides mechanisms for this

Example Use Cases

- Numerical integration: $\int_a^b f(x)dx$ for different functions f,
- Root finding: find x such that $f(x) = 0$ for different functions f,
- Optimization: minimize $f(x)$ for different functions f.
- Compute $f(A)\mathbf{v}$ for different functions f and matrix A.

Method 1: External Procedure

Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

Method 1: External Procedure

Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

Step 1: **Define the function**

```fortran
real(real64) function my_func(x)
    real(real64), intent(in) :: x
    my_func = x**2
end function my_func
```

Method 1: External Procedure

Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

Subroutine that accepts function:

```fortran
subroutine integrate(f, a, b, result)
    real(real64), external :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    ! Integration code using f(x)
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

Method 1: External Procedure

Classic Fortran approach (using `external`)—which is still valid today, but less safe and should be avoided in modern code.

Subroutine that accepts function:

```fortran
subroutine integrate(f, a, b, result)
    real(real64), external :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    ! Integration code using f(x)
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

- `external` declares `f` as external function
- No type checking of arguments

Method 2: Procedure Pointer (Modern Fortran)

The procedure in modern Fortran is to use **procedure interfaces** for type safety.

Method 2: Procedure Pointer (Modern Fortran)

The procedure in modern Fortran is to use **procedure interfaces** for type safety.

Define interface:

```
abstract interface
    real(real64) function func_interface(x)
        real(real64), intent(in) :: x
    end function func_interface
end interface
```

Method 2: Procedure Pointer (Modern Fortran)

The procedure in modern Fortran is to use **procedure interfaces** for type safety.

Subroutine with procedure argument:

```fortran
subroutine integrate(f, a, b, result)
    procedure(func_interface) :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

Method 2: Procedure Pointer (Modern Fortran)

The procedure in modern Fortran is to use **procedure interfaces** for type safety.

Subroutine with procedure argument:

```
subroutine integrate(f, a, b, result)
    procedure(func_interface) :: f
    real(real64), intent(in) :: a, b
    real(real64), intent(out) :: result
    result = (b-a) * f((a+b)/2.0)
end subroutine integrate
```

- `procedure(interface_name)` provides type safety
- Compiler checks function signature
- **Recommended** for modern code

Using the Function Argument

Calling the subroutine:

```
program main
    use iso_fortran_env, &
        only: real64
    implicit none
    real(real64) :: result
    call integrate(my_func, &
        0.0_real64, &
        1.0_real64, result)
    print *, "Result:", result
```

```
contains
    real(real64) function &
        my_func(x)
        real(real64), &
            intent(in) :: x
        my_func = x**2
    end function my_func
end program main
```

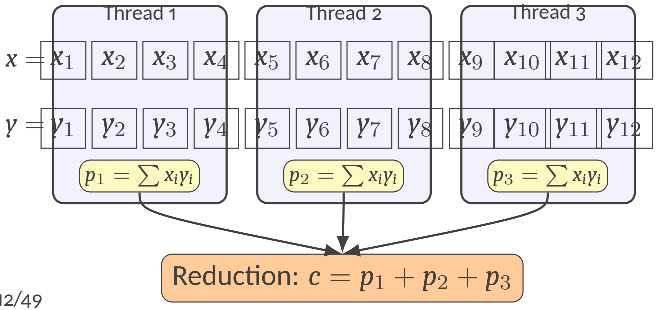
- Pass function name without parentheses
- Function **must** match expected signature, as defined by the `abstract interface`.

DOT: Parallelization Strategy

Question

What kind of parallelism can we exploit?

- The dot product is a **reduction** operation
- Strategy:



Thread 1
Thread 2
Thread 3
$x =$ x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12}
$y =$ y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 y_{10} y_{11} y_{12}
$p_1 = \sum x_i y_i$
$p_2 = \sum x_i y_i$
$p_3 = \sum x_i y_i$
Reduction: $c = p_1 + p_2 + p_3$

1. Split vectors into chunks
2. Each thread computes local dot
3. Reduce partial sums

DOT: Parallelization Strategy

Question

What kind of parallelism can we exploit?

- The dot product is a **reduction** operation
- Strategy:
  - Split vectors into chunks
  - Each thread computes local dot product
  - Reduce partial sums to get final result
- Good example for OpenMP parallelization

Reduction operations

But how do we implement reductions in OpenMP?

OpenMP Reduction Clause: Idea

Pattern: combine per-thread partial results into one final value.

- Syntax (loop form): `!$omp parallel do reduction(op:var1,var2,...)`
- Each listed variable gets:
  1. private copy (initialized),
  2. local accumulation,
  3. final merge.
- Avoid manual critical sections; scalable; fewer false sharing issues.
- Works for associative/commutative operations
  - ! floating point is *not associative*: order may changes result slightly!

Supported intrinsic operators (Fortran)

```
+ - * .and. .or. .xor. max min iand ior ieor
```

Basic Examples

2 BLAS Level 1: DOT

```
! Sum of an array
total = 0.0_real64                ! Initialize shared reduction variable; each
↪  thread gets a private copy set to 0
!$omp parallel do reduction(+:total)
do i = 1, n                       ! Iterations divided among threads
    total = total + a(i)          ! Each thread accumulates into its private
    ↪  'total'
end do                            ! Runtime combines all private totals
↪  (addition) into the shared 'total'
```

Basic Examples

2 BLAS Level 1: DOT

```
! Maximum over array
mval = -huge(mval)                    ! Initialize with very small value; private
↪  copies get same initialization
!$omp parallel do reduction(max:mval)
do i = 1, n                           ! Iterations executed in parallel
    if (a(i) > mval) mval = a(i) ! Track local maximum in each thread
end do                                ! Runtime computes global max from all
↪  thread-local maxima
```

Basic Examples

2 BLAS Level 1: DOT

```
! Logical OR over flags
any_flag = .false.                  ! Identity for .or.; private copies start as
↪  .false.
!$omp parallel do reduction(.or.:any_flag)
do i = 1, n                         ! Parallel traversal of flags
    any_flag = any_flag .or. flags(i)  ! Accumulate local logical OR
end do                              ! Final any_flag is OR of all thread-local
↪  results
```

What the Runtime Does

1. Creates one private copy per thread (initialized suitably).
2. Executes loop chunks independently, updating private copies.
3. At implicit barrier: combines privates into the original variable: *in unspecified order*.

Initialization Rules

- +: zero; $*$: one; logical: identity; `max`/`min`: extreme values.
- Ensure you **do NOT re-initialize** inside the loop.

Numerical Note

Floating point reductions are order-dependent; expect tiny round-off differences vs serial.

Common Pitfalls

- Forgetting initialization before the directive (needed for clarity; runtime overwrites).
- Writing to the reduction variable outside the loop: creates race conditions.
- Using non-associative custom operations without care (order not guaranteed).
- Large objects: reduction copies can be expensive; consider **manual chunking** or **atomic updates** if contention low.

Atomic Updates

Atomic updates protect a single read-modify-write operation on one scalar or array element so that threads do not interleave it. In OpenMP add `!$omp atomic` before the assignment; only that memory operation is serialized, not the whole loop.

When to use:

- Few conflicting updates (low contention).
- Irregular indices (e.g. histogram, sparse gather).
- Operation not available as a reduction.

Drawbacks: High contention degrades scalability; for dense loops prefer a reduction (private copies + final combine).

Rules: Single assignment only; limited set of operators; protects exactly one memory location.

Example:

```fortran
total = 0.0_real64
!$omp parallel do shared(total)
do i = 1, n
   !$omp atomic
   total = total + a(i)
end do
```

Atomic with function calls

Example:

```
!$omp parallel do shared(total)
do i = 1, n
    !$omp atomic
    total = total + my_func(a(i))
end do
```

Function:

```
real(real64) function my_func(x)
    real(real64), intent(in) :: x
    my_func = x**2
end function my_func
```

- Function calls inside atomic regions can lead to undefined behavior if the function itself is not thread-safe.
- Ensure that any function called within an atomic region does not modify shared state or rely on non-thread-safe operations.
- It is only the update to the memory location of the variable `total` that will occur atomically.

Atomic with function calls

Example:

```
!$omp parallel do shared(total)
do i = 1, n
    !$omp atomic
    total = total + my_func(a(i))
end do
```

Function:

```
real(real64) function my_func(x)
    real(real64), intent(in) :: x
!$omp critical (my_func_lock)
    my_func = x**2
!$omp end critical (my_func_lock)
end function my_func
```

- If the application developer does not intend to permit the threads to execute `my_func` at the same time, then the *`!$omp critical`* construct must be used instead,
- the critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously,
- When a thread encounters a critical construct, it waits until **no other thread is executing a critical region with the same name**.

DOT: OpenMP Implementation

```fortran
program dot_omp
    use iso_fortran_env, only: output_unit, real64
    use omp_lib
    implicit none
    integer :: i
    integer, parameter :: n = 10000
    integer :: nthreads
    real(real64) :: x(n), y(n)
    real(real64) :: sum, c
    real(real64) :: start_time, end_time
    real(real64) :: ddot
    !$omp parallel
    !$omp single
    nthreads = omp_get_num_threads()
```

DOT: OpenMP Implementation

```fortran
!$omp end single
!$omp end parallel
write(output_unit,'("Number of threads: ",I0)') nthreads
! Initialize arrays
x = 1.0
y = 2.0
c = 0.0
start_time = omp_get_wtime()
!$omp parallel do private(i) shared(x,y) reduction(+:c)
do i = 1, n
    c = c + x(i) * y(i)
end do
!$omp end parallel do
end_time = omp_get_wtime()
```

DOT: OpenMP Implementation

```
    write(output_unit,'("Dot product: ",F0.2)') sum
    write(output_unit,'("Time taken: ",E0.2)') end_time - start_time
    ! Check the result with blas
    call cpu_time(start_time)
    sum = ddot(n, x, 1, y, 1)
    call cpu_time(end_time)
    if (abs(c - sum) > 1.0e-12) then
        write(output_unit,'("Abs. Error: ",F0.2)') abs(c - sum)
    else
        write(output_unit,'("Result is correct")')
    end if
    write(output_unit,'("BLAS time: ",E0.2)') end_time - start_time
end program dot_omp
```

DOT: OpenMP Implementation

- *`!$omp parallel do`* creates parallel region
- `reduction(+:c)` clause for reduction variable
- Each thread has private copy of `c`
- Values combined at end of parallel region

Exercise

A possible exercise is to implement the `ddot` function using, instead of the reduction clause, `atomic` or `critical` sections, and then compare the performance with the reduction version.

Back to the roofline model

- DOT operation: $c = \sum_{i=1}^{n} x_i y_i$
- Arithmetic: $2n$ flops (multiply + add per element)
- Memory traffic: $2n \times 8$ bytes (read x_i and y_i)
- Arithmetic intensity:

$$AI = \frac{2n}{16n} = \frac{1}{8} \text{ flops/byte}$$

Memory Bound

DOT is **memory bound**: performance limited by **memory bandwidth**, not compute capability.

- Peak performance: $P = \min(\pi, \beta \times AI) = \beta \times {}^{1}/_{8}$
- where β is memory bandwidth (GB/s) and π is peak compute (GFLOPS/s)

Table of Contents

3 BLAS Level 1: NRM2

NRM2: 2-Norm of a Vector

The 2-norm is defined as:

$$c = \|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

BLAS routine: `dnrm2`

```
c = dnrm2(n, x, incx)
```

where `incx` is the increment for the input vector *x*.

NRM2: OpenMP Implementation

Similar implementation to dot product:

```
c = 0.0
!$omp parallel do reduction(+:c) shared(x) private(i)
do i = 1, n
        c = c + x(i)**2
end do
!$omp end parallel do
c = sqrt(c)
```

- Same reduction pattern
- Square root applied after parallel region
- Same declaration note applies to `dnrm2`

Back to the roofline model

- NRM2 operation: $c = \sqrt{\sum_{i=1}^{n} x_i^2}$
- Arithmetic: $2n$ flops (square + add per element)
- Memory traffic: $n \times 8$ bytes (read x_i)
- Arithmetic intensity:

$$AI = \frac{2n}{8n} = \frac{1}{4} \text{ flops/byte}$$

Memory Bound

NRM2 is also **memory bound**: performance limited by **memory bandwidth**, not compute capability.

- Peak performance: $P = \min(\pi, \beta \times AI) = \beta \times {}^{1}/{}_{4}$
- where β is memory bandwidth (GB/s) and π is peak compute (GFLOPS/s)

Table of Contents

4 All the other Level 1 BLAS

Other Level 1 BLAS Routines

- **SCAL**: Scale vector by a constant
$$y = \alpha x,$$
- **COPY**: Copy vector x to y,
- **SWAP**: Swap vectors x and y,
- **ASUM**: 1-norm of a vector,
- **IAMAX**: finds the index of the first element having maximum absolute value.

Exercise

Implement these routines using OpenMP parallelization, the OpenMP instruction seen so far are sufficient for this purpose.

Givens rotations

Another important class of Level 1 BLAS routines are those implementing Givens rotations.

- Givens rotation zeroes elements in vectors/matrices.
- Used in QR factorization, least squares problems.
- Basic operation:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

  where $c = \cos(\theta)$, $s = \sin(\theta)$.

BLAS routines: `drotg` (generate), `drot` (apply).

```
call drotg(a, b, c, s)
```

Givens rotations

Another important class of Level 1 BLAS routines are those implementing Givens rotations.

- Givens rotation zeroes elements in vectors/matrices.
- Used in QR factorization, least squares problems.
- Basic operation:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

  where $c = \cos(\theta)$, $s = \sin(\theta)$.

BLAS routines: `drotg` (generate), `drot` (apply).

```
call drot(n, x, incx, y, incy, c, s)
```

This applies the rotation to vectors x and y.

Givens rotations: example

```
program givens_example
    use iso_fortran_env, only: real64
    implicit none
    real(real64) :: a, b, c, s
    a = 3.0_real64
    b = 4.0_real64
    call drotg(a, b, c, s)
    print *, "Givens rotation parameters:"
    print *, "c =", c, ", s =", s
end program givens_example
```

- This program computes the Givens rotation parameters for the vector $(3, 4)$.
- The output will show the cosine and sine values used in the rotation.
- These parameters can then be used to zero out one of the elements in a vector or matrix.

Using Givens rotations to zero out elements

Example: Zeroing out elements in a vector

```
subroutine zero_givens(x, c, s)
    use iso_fortran_env, only: real64
    real(real64), intent(inout) :: x(:)
    real(real64), allocatable,
    ↪  intent(out) :: c(:), s(:)
    integer :: n, i
    real(real64) :: a, b, cc, ss
    n = size(x)
    if (n <= 1) then
        allocate(c(0), s(0))
        return
    end if
    allocate(c(n-1), s(n-1))
```

- This subroutine takes a vector `x` and computes Givens rotations to zero out all elements except the first one.
- It allocates arrays `c` and `s` to store the cosine and sine values of the rotations.

Using Givens rotations to zero out elements

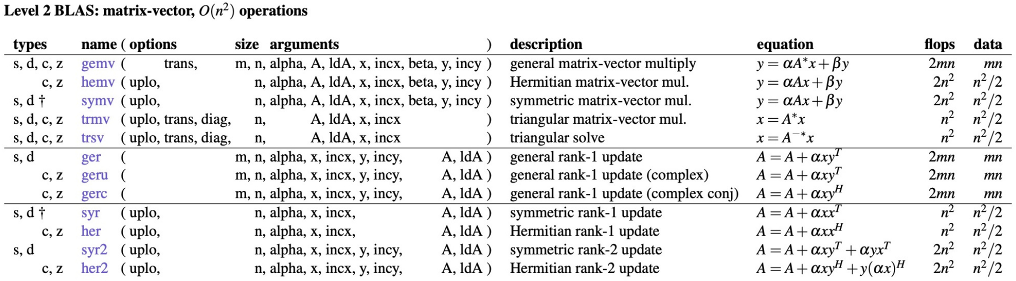
Example: Zeroing out elements in a vector

```
    do i = 2, n
        a = x(1)
        b = x(i)
        ! build Givens for (a,b)
        call drotg(a, b, cc, ss)
        c(i-1) = cc
        s(i-1) = ss
        ! Apply rotation to the 2-vector
        ↪   [x(1); x(i)] using BLAS drot
        call drot(1, x(1:1), 1, x(i:i),
        ↪  1, cc, ss)
    end do
end subroutine zero_givens
```

- This subroutine takes a vector `x` and computes Givens rotations to zero out all elements except the first one.
- It allocates arrays `c` and `s` to store the cosine and sine values of the rotations.
- For each element in the vector (from the second to the last), it computes the Givens rotation parameters using `drotg`.
- It then applies the rotation to the pair of elements $x(1)$ and $x(i)$ using `drot`.

Level 2 BLAS Overview

- Level 2 BLAS routines perform matrix-vector operations.



Level 2 BLAS: matrix-vector, $O(n^2)$ operations

types	name	(options	size	arguments)	description	equation	flops	data
s, d, c, z	gemv	(trans,	m, n,	alpha, A, ldA, x, incx, beta, y, incy)	general matrix-vector multiply	$y = \alpha A^* x + \beta y$	$2mn$	mn
c, z	hemv	(uplo,	n,	alpha, A, ldA, x, incx, beta, y, incy)	Hermitian matrix-vector mul.	$y = \alpha A x + \beta y$	$2n^2$	$n^2/2$
s, d †	symv	(uplo,	n,	alpha, A, ldA, x, incx, beta, y, incy)	symmetric matrix-vector mul.	$y = \alpha A x + \beta y$	$2n^2$	$n^2/2$
s, d, c, z	trmv	(uplo, trans, diag,	n,	A, ldA, x, incx)	triangular matrix-vector mul.	$x = A^* x$	n^2	$n^2/2$
s, d, c, z	trsv	(uplo, trans, diag,	n,	A, ldA, x, incx)	triangular solve	$x = A^{-*} x$	n^2	$n^2/2$
s, d	ger	(	m, n,	alpha, x, incx, y, incy, A, ldA)	general rank-1 update	$A = A + \alpha x y^T$	$2mn$	mn
c, z	geru	(	m, n,	alpha, x, incx, y, incy, A, ldA)	general rank-1 update (complex)	$A = A + \alpha x y^T$	$2mn$	mn
c, z	gerc	(	m, n,	alpha, x, incx, y, incy, A, ldA)	general rank-1 update (complex conj)	$A = A + \alpha x y^H$	$2mn$	mn
s, d †	syr	(uplo,	n,	alpha, x, incx, A, ldA)	symmetric rank-1 update	$A = A + \alpha x x^T$	n^2	$n^2/2$
c, z	her	(uplo,	n,	alpha, x, incx, A, ldA)	Hermitian rank-1 update	$A = A + \alpha x x^H$	n^2	$n^2/2$
s, d	syr2	(uplo,	n,	alpha, x, incx, y, incy, A, ldA)	symmetric rank-2 update	$A = A + \alpha x y^T + \alpha y x^T$	$2n^2$	$n^2/2$
c, z	her2	(uplo,	n,	alpha, x, incx, y, incy, A, ldA)	Hermitian rank-2 update	$A = A + \alpha x y^H + y(\alpha x)^H$	$2n^2$	$n^2/2$

- These routines are essential for many numerical algorithms, including solving linear systems and eigenvalue problems.

The GEMV Routine

- One of the most important Level 2 BLAS routines is **GEMV** (General Matrix-Vector multiplication).
- It computes the operation:

  $$y = \alpha Ax + \beta y$$

  where A is a matrix, x and y are vectors, and α and β are scalars.
- GEMV is widely used in various applications, including solving linear systems and performing transformations.

GEMV: BLAS Interface

$$y = \alpha A x + \beta y$$

- Routine: `dgemv` (double precision)
- Prototype:

```
call dgemv(trans, m, n, alpha, A, lda, x, incx, beta, y, incy)
! trans = 'N','T','C'; lda = leading dimension of A
```

- Column-major storage; lda = first dimension of A as declared.
- Increments allow strided access (usually 1).

GEMV: Example Program

```fortran
program gemv_blas
    use iso_fortran_env, only: real64, output_unit, error_unit
    implicit none
    integer :: m, n, lda
    real(real64) :: alpha, beta
    real(real64), allocatable :: A(:,:), x(:), y(:)
    character(len=100) :: m_str, n_str
    real(real64) :: start_time, end_time, elapsed_time
    integer :: i,j,info
    ! Read m and n from command line arguments
    if (command_argument_count() < 2) then
        write(error_unit, '("Usage: gemv_blas <m> <n>")')
        stop
```

GEMV: Example Program

```
end if
call get_command_argument(1, m_str, status=info)
call get_command_argument(2, n_str, status=info)
if (info /= 0) then
    write(error_unit, '("Error reading command line arguments")')
    stop
end if
read(m_str, *) m
read(n_str, *) n
! set parameters
lda = m
alpha = 1.0d0
beta = 1.0d0
allocate(A(lda, n), x(n), y(m),stat=info)
```

GEMV: Example Program

```fortran
if (info /= 0) then
    write(error_unit, '("Error allocating memory")')
    stop
end if
! Initialize matrix A and vectors x and y
do i = 1, m
    do j = 1, n
        A(i, j) = real(i + j, kind=real64)
    end do
end do
do i = 1, n
    x(i) = real(i, kind=real64)
end do
do i = 1, m
```

GEMV: Example Program

```
        y(i) = real(i, kind=real64)
    end do
    ! Compute the matrix-vector product using BLAS gemv
    call cpu_time(start_time)
    call dgemv('N', m, n, alpha, A, lda, x, 1, beta, y, 1)
    call cpu_time(end_time)
    elapsed_time = end_time - start_time
    write(output_unit, '("BLAS dgemv time: ", E0.6)') elapsed_time
    ! Free allocated memory
    deallocate(A, x, y, stat=info)
    if (info /= 0) then
        write(error_unit, '("Error deallocating memory")')
        stop
```

GEMV: Example Program

```
    end if
end program gemv_blas
```

Organizing Implementations

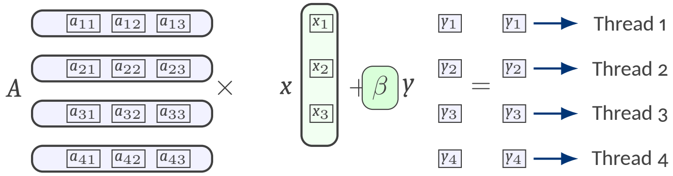
- Place multiple GEMV variants in a module for reuse.

```
module gemvmod
    use iso_fortran_env
    use omp_lib
    implicit none
    private
    public :: gemv_openmp_n, gemv_openmp_n_block
contains
    ! Implementations follow
    ⟨ see next slides ⟩
end module gemvmod
```

Parallel Formulation

$$y_i = \alpha \sum_{j=1}^{n} A_{ij} x_j + \beta y_i, \quad i = 1, \ldots, m$$

- Natural outer-loop parallelism over rows i (independent updates).
- Uses dot products: reuse optimized `ddot`.



Variant 1: Parallel Over Rows

```fortran
subroutine gemv_openmp_n(m, n, alpha, A, lda, x, beta, y)
    use iso_fortran_env, only: real64
    use omp_lib
    implicit none
    integer, intent(in) :: m, n, lda
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda,*), x(*)
    real(real64), intent(inout) :: y(*)
    real(real64) :: ddot
    integer :: i
    real(real64) :: temp
    !$omp parallel do private(i,temp) shared(m,n,A,x,y,alpha,beta)
    do i = 1, m
```

Variant 1: Parallel Over Rows

```
          temp = ddot(n, A(i,1:n), 1, x, 1)
          y(i) = alpha*temp + beta*y(i)
      end do
      !$omp end parallel do
  end subroutine
```

⚠ Strided row access in column-major layout: **less cache-friendly**.

Memory Access Patterns: Cache Misses

Accessing data in a non-contiguous manner can lead to cache misses, as the CPU cache is optimized for spatial locality. When data is accessed sequentially, it is more likely to be present in the cache, leading to faster access times. However, when data is accessed in a strided or non-contiguous manner, it can result in **cache misses**, as the required data may not be present in the cache, leading to **slower access** times due to **fetching data from main memory**.

Variant 1b: Dynamic Scheduling

```fortran
subroutine gemv_openmp_n_block(m, n, alpha, A, lda, x, beta, y)
    use iso_fortran_env, only: real64
    use omp_lib
    implicit none
    integer, intent(in) :: m, n, lda
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda,*), x(*)
    real(real64), intent(inout) :: y(*)
    real(real64) :: ddot, temp
    integer :: i
    !$omp parallel do schedule(dynamic,32) private(i,temp) &
    !$omp     shared(m,n,A,x,y,alpha,beta)
    do i = 1, m
```

Variant 1b: Dynamic Scheduling

```
        temp = ddot(n, A(i,1:n), 1, x, 1)
        y(i) = alpha*temp + beta*y(i)
    end do
    !$omp end parallel do
end subroutine
```

- Dynamic chunks mitigate load imbalance; chunk size tunable.

OpenMP schedule clause: chunk size reminder

- Syntax: `!$omp do schedule(kind[,chunk])`
- `chunk` = max iterations handed to a thread each time it receives work.
- `dynamic[,c]`: Threads pull blocks of size c from a queue until done. Good for irregular work; more overhead. Default c often = 1 if omitted.

OpenMP schedule clause: chunk size effects

- Rule of thumb: pick c so that per-chunk work dominates scheduling cost.
  - Too small: higher scheduling overhead, more contention.
  - Too large: potential load imbalance (idle threads at end).

```
!$omp parallel do
↪   schedule(static,64)
do i = 1, n
    work(i)
end do
```

- Static scheduling with large chunks.
- Low scheduling overhead, but potential load imbalance.

```
!$omp parallel do
↪   schedule(dynamic,8)
do i = 1, n
    work(i)
end do
```

- Dynamic scheduling with small chunks.
- Better load balance, but higher overhead.

```
!$omp parallel do
↪   schedule(guided,4)
do i = 1, n
    work(i)
end do
```

- Guided scheduling with minimum chunk size.
- Balances load and overhead adaptively.

OpenMP schedule clause: chunk size effects

- Rule of thumb: pick c so that per-chunk work dominates scheduling cost.
  - Too small: higher scheduling overhead, more contention.
  - Too large: potential load imbalance (idle threads at end).

Tip

Benchmark several chunk sizes; optimal values depend on loop body cost variability and hardware.

Table of Contents

5 Summary and next lecture

Summary and Next Lecture

- OpenMP reductions simplify parallel accumulation patterns.
- **Atomic updates** provide fine-grained synchronization for low-contention cases.
- Level 1 BLAS routines (DOT, NRM2) are **memory-bound**.
- Level 2 BLAS routines (GEMV) involve matrix-vector operations; parallelism can be exploited over rows.
- Memory access patterns **significantly impact performance**; consider data layout and access order.

Next lecture

- Investigate better memory access patterns for GEMV.
- Level 3 BLAS: Matrix-Matrix operations (GEMM).
- Blocking techniques for cache efficiency.