High Performance Linear Algebra

Lecture 6: Continuing with BLAS, BLAS Level 2: GEMV and level 3: GEMM

Ph.D. program in High Performance Scientific Computing

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

November 26, 2025 — 14.00:16.00

Dipartimento di Matematica
Università di Pisa

Back from the past

- Last time we have seen:
  - BLAS Level 1: DOT, NRM2 and Givens rotations
  - BLAS Level 2: GEMV a first look
- Today we will continue with:
  - BLAS Level 2: GEMV: better memory access patterns
  - BLAS Level 3: GEMM

Table of Contents

2 BLAS Level 2: GEMV continued

- BLAS Level 2: GEMV continued
  - Looking at the performance: roofline model
  - BLAS Level 2: TRSV

- BLAS Level 3
  - DGEMM with OpenMP
  - Tiled version
  - Tiled version with OpenMP

Last lecture

2 BLAS Level 2: GEMV continued

We have implemented two variants of the GEMV kernel, in the following `module`:

```fortran
module gemvmod
    use iso_fortran_env
    use omp_lib
    implicit none
    private
    public :: gemv_openmp_n, gemv_openmp_n_block
contains
    ! Implementations in the last lecture
end module gemvmod
```

- `gemv_openmp_n`: simple OpenMP parallelization over rows,
- `gemv_openmp_n_block`: blocked version with OpenMP parallelization over blocks of rows.

GEMV: limitations of the previous implementations

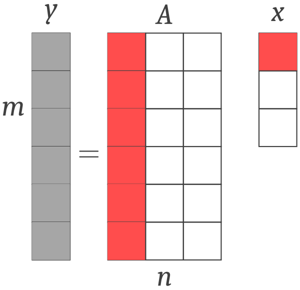
- Both implementations have a limitation: they access the matrix A in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.

GEMV: limitations of the previous implementations

- Both implementations have a limitation: they access the matrix A in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.

GEMV: limitations of the previous implementations

- Both implementations have a limitation: they access the matrix A in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.



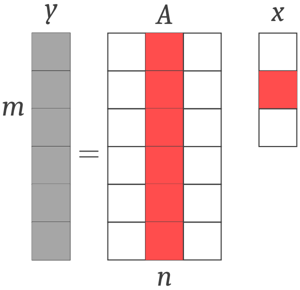
We express the matrix-vector product as a linear combination of the columns of A:

$$\mathbf{y} = x_1\mathbf{a}_{:,1} + x_2\mathbf{a}_{:,2} + \cdots + x_n\mathbf{a}_{:,n}$$

- From an operative point of view, this means swapping the two loops in the naïve implementation.

GEMV: limitations of the previous implementations

- Both implementations have a limitation: they access the matrix A in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.



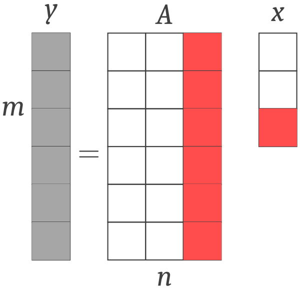
We express the matrix-vector product as a linear combination of the columns of A:

$$\mathbf{y} = x_1\mathbf{a}_{:,1} + x_2\mathbf{a}_{:,2} + \cdots + x_n\mathbf{a}_{:,n}$$

- From an operative point of view, this means swapping the two loops in the naïve implementation.

GEMV: limitations of the previous implementations

- Both implementations have a limitation: they access the matrix A in column-major order, which is not cache friendly.
- We can improve the memory access pattern by changing the way we parallelize the computation.
- Instead of parallelizing over rows, we can **parallelize over columns**.



We express the matrix-vector product as a linear combination of the columns of A:

$$\mathbf{y} = x_1\mathbf{a}_{:,1} + x_2\mathbf{a}_{:,2} + \cdots + x_n\mathbf{a}_{:,n}$$

- From an operative point of view, this means swapping the two loops in the naïve implementation.

GEMV: parallelization over columns

- The matrix A is stored in column-major order, this allows A to be read sequentially, optimizing memory access.
- Each element of the vector x is loaded into registers and reused efficiently.
- The vector y is accessed in every iteration, but if its size is smaller than the cache capacity, it can be reused at the cache level.

```fortran
y = beta * y ! Update y with beta * y
!$omp parallel do private(i) shared(A, x, alpha) reduction(+:y)
do i = 1, n
    call daxpy(m, alpha*x(i), A(1:m,i), 1, y, 1)
end do
!$omp end parallel do
```

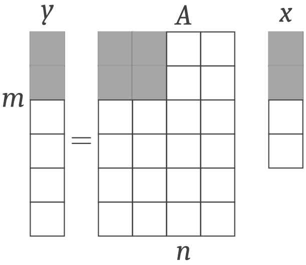
Limitation of the column-wise parallelization

The column-wise parallelization has a limitation:

- The number of columns n may be small compared to the number of available threads,
- Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
- For large m, maintaining multiple copies of y can exceed cache capacity

Limitation of the column-wise parallelization

- The column-wise parallelization has a limitation:
  - The number of columns n may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
  - For large m, maintaining multiple copies of y can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.



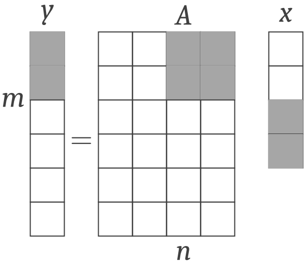
- This is a good approach if the matrix A is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix A.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

Limitation of the column-wise parallelization

The column-wise parallelization has a limitation:

- The number of columns n may be small compared to the number of available threads,
- Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
- For large m, maintaining multiple copies of y can exceed cache capacity

To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.



- This is a good approach if the matrix A is large enough, and it allows us to take advantage of the cache hierarchy.

The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix A.

This is an example of divide-and-conquer approach:

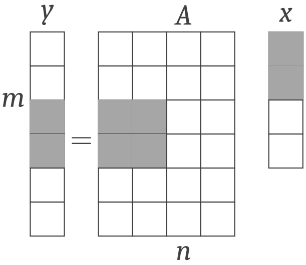
- we divide the problem into smaller subproblems,
- solve each subproblem by a sequential GEMV operation.

Limitation of the column-wise parallelization

The column-wise parallelization has a limitation:

- The number of columns n may be small compared to the number of available threads,
- Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
- For large m, maintaining multiple copies of y can exceed cache capacity

To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.



- This is a good approach if the matrix A is large enough, and it allows us to take advantage of the cache hierarchy.

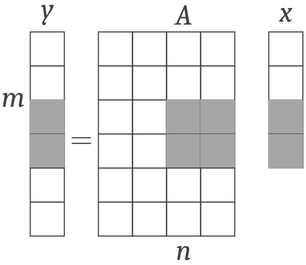
The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix A.

This is an example of divide-and-conquer approach:

- we divide the problem into smaller subproblems,
- solve each subproblem by a sequential GEMV operation.

Limitation of the column-wise parallelization

- The column-wise parallelization has a limitation:
  - The number of columns n may be small compared to the number of available threads,
  - Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
  - For large m, maintaining multiple copies of y can exceed cache capacity
- To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.



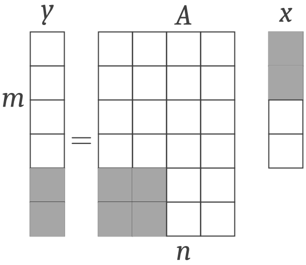
- This is a good approach if the matrix A is large enough, and it allows us to take advantage of the cache hierarchy.
- The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix A.
- This is an example of divide-and-conquer approach:
  - we divide the problem into smaller subproblems,
  - solve each subproblem by a sequential GEMV operation.

Limitation of the column-wise parallelization

The column-wise parallelization has a limitation:

- The number of columns n may be small compared to the number of available threads,
- Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
- For large m, maintaining multiple copies of y can exceed cache capacity

To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.



- This is a good approach if the matrix A is large enough, and it allows us to take advantage of the cache hierarchy.

The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix A.

This is an example of divide-and-conquer approach:

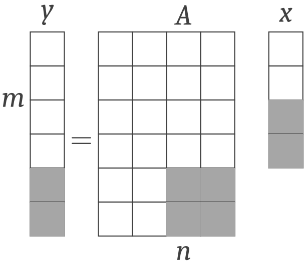
- we divide the problem into smaller subproblems,
- solve each subproblem by a sequential GEMV operation.

Limitation of the column-wise parallelization

The column-wise parallelization has a limitation:

- The number of columns n may be small compared to the number of available threads,
- Array reductions are expensive: each thread needs to maintain a private copy of the output vector y and then reduce them at the end of the computation,
- For large m, maintaining multiple copies of y can exceed cache capacity

To overcome this limitation, we can use a **blocked version** of the column-wise parallelization.



- This is a good approach if the matrix A is large enough, and it allows us to take advantage of the cache hierarchy.

The code is similar, but we need to add an outer loop that iterates over the blocks of the matrix A.

This is an example of divide-and-conquer approach:

- we divide the problem into smaller subproblems,
- solve each subproblem by a sequential GEMV operation.

Divide and conquer GEMV code

```fortran
! scale y by beta
y = beta * y
!$omp parallel default(none) &
!$omp    shared(A, x, y, m, n, lda, alpha, n_x_, n_y_) &
!$omp    private(i,j,ti,mb,nb,yloc)
allocate(yloc(m))
yloc = 0.0_real64
! Tile the i-j loops; collapse for better load balance
!$omp do collapse(2) schedule(static)
do i = 1, m, n_x_
    do j = 1, n, n_y_
        mb = min(n_x_, m - i + 1) ! handle edge tiles
```

Divide and conquer GEMV code

2 BLAS Level 2: GEMV continued

```fortran
        nb = min(n_y_, n - j + 1)
! perform the small GEMV into the thread--local yloc
        call dgemv('N', mb, nb, alpha, &
                    A(i, j), lda, &
                    x(j), 1, &
                    1.0_real64, yloc(i), 1)
    end do
end do
!$omp end do
! Safely accumulate thread--local yloc into global y
do ti = 1, m
    !$omp atomic
```

Divide and conquer GEMV code

```
    y(ti) = y(ti) + yloc(ti)
end do
deallocate(yloc)
!$omp end parallel
```

- We need to choose **appropriate block sizes** `n_x_` and `n_y_`.
- We need to handle edge tiles when the matrix dimensions are not multiples of the block sizes.

Description of the code

- We allocate a private copy of the output vector `yloc` for each thread.
- We tile the i-j loops, and we use the *`!collapse(2)`* clause to parallelize over the tiles.
- For each tile, we call a sequential `dgemv` to compute the partial result into the private `yloc`.
- Finally, we safely accumulate the private copies into the global output vector `y` using an *`!atomic`* operation.

Compile

We can add the module to our main project `objblas`:

```
add_library(objblas src/blas.f90 src/blasOMP.f90 src/gemvmod.f90)
target_link_libraries(objblas PUBLIC BLAS::BLAS OpenMP::OpenMP_Fortran)
```

Roofline Model Refresher

2 BLAS Level 2: GEMV continued

We analyze GEMV

$$y = \alpha A x + \beta y, \quad A \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^{n},\ y \in \mathbb{R}^{m}$$

- Floating-point ops (precise):

$$\underbrace{2mn}_{Ax} + \underbrace{m}_{\beta y\text{ scale}} + \underbrace{m}_{\text{final add}} = 2mn + 2m \approx 2mn \quad (mn \gg m).$$

- Bytes moved (no reuse):

$$\underbrace{8mn}_{A} + \underbrace{8n}_{x} + \underbrace{16m}_{y\text{ read+write}} = 8mn + 8n + 16m.$$

- Operational intensity:

$$I = \frac{2mn + 2m}{8mn + 8n + 16m} \approx \frac{2mn}{8mn + 8n + 16m}.$$

Roofline Model Refresher

We analyze GEMV

$$y = \alpha Ax + \beta y, \quad A \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^{n},\ y \in \mathbb{R}^{m}$$

Platform (Intel® Core™ i9-14900HX, approximate):

- Peak DP FLOPs ≈ 0.8–0.9 TFLOP/s
- Sustained memory bandwidth ≈ 89.6 GB/s
- Cache sizes:

```
L1d:                    896 KiB (24 instances)
L1i:                    1,3 MiB (24 instances)
L2:                     32 MiB (12 instances)
L3:                     36 MiB (1 instance)
```

Use roofline: Attainable GFLOP/s = $\min(\text{Peak, Bandwidth} \times I)$.

Cache-Aware Problem Size Selection

Goal: choose (m,n) to exercise cache vs. memory bandwidth. Working set (no temporal reuse):

$$W(m,n) = 8mn + 8n + 16m \text{ bytes.}$$

Hierarchy (per core / shared, simplified):

- L1d: 32–48 KiB
- L2 (per P-core): 2 MiB; E-core cluster: 4 MiB
- LLC (L3): 36 MiB shared
- DRAM: 89.6 GB/s sustained

We classify regimes using full matrix footprint vs. cache levels (or effective tiled working set).

Representative Matrix Sizes

L1-working-set (fully fits when tiled):

- Example global size: m = n = 64
- Full A: $64^2 * 8B = 32KiB; x + yoverhead \approx$ 2 KiB
- Entire working set $\approx$ 34 KiB (fits in 48 KiB L1d)

L2/LLC-resident (fits in LLC, not L1):

- m = n = 2000
- A: 32 MB; x + y $\approx$ 0.032 MB
- Fits in 36 MB L3; stresses LLC bandwidth / latency

DRAM-bound:

- m = n = 20000
- A: 3.2 GB; exceeds LLC; compulsory DRAM traffic

Tile Size Selection (b$_m$, b_n)
$2\,BLASLevel2 : GEMVcontinued$

Tile footprint (data needed per tile GEMV assuming contiguous columns):

$$F(b_m, b_n) = 8\,b_m b_n + 8\,b_n + 16\,b_m \text{ bytes.}$$

Guidelines:

- Minimize capacity misses for A; keep (portion of) y hot.
- Reuse x entries across all rows of the tile.

Tile Size Selection (b$_m, b_n)$

$2\ BLASLevel2 : GEMVcontinued$

L1 tile:

$$b_m = b_n = 32 \Rightarrow F \approx 8 \cdot 1024 + 8 \cdot 32 + 16 \cdot 32 \approx 8192 + 256 + 512 \approx 8.96 \text{ KiB}.$$

L2 tile:

$$b_m = b_n = 256 \Rightarrow F \approx 512 \text{ KiB} + \text{vector overhead} \approx 513 \text{ KiB}.$$

Large / DRAM-stress tile:

$$b_m = b_n = 512 \Rightarrow F \approx 2 \text{ MiB}.$$

Pick (b_m,b_n) so multiple thread–private y tiles fit without eviction.

Operational Intensity Examples

For square case $m = n$:

$$I(n) = \frac{2n^2 + 2n}{8n^2 + 8n + 16n} = \frac{2n^2 + 2n}{8n^2 + 24n}.$$

As $n \to \infty$: $I \to \frac{2}{8} = 0.25$ FLOP/Byte.
Examples (rounded):

- $n = 64$: $I \approx 0.24$
- $n = 2000$: $I \approx 0.25$
- $n = 20000$: $I \approx 0.25$

Conclusion: GEMV remains memory-bound on modern CPUs (low intensity). Optimization focuses on:

- Reducing data traffic (tiling, avoiding redundant y copies)
- Prefetch-friendly sequential column access
- Minimizing reduction overhead

Performance Expectation vs. Roofline

Given peak P and bandwidth B:

$$\text{Bound}_{\text{memory}} = B \cdot I, \quad \text{Bound}_{\text{compute}} = P.$$

With $I \approx 0.25$, $B = 89.6$ GB/s:

$$\text{Memory bound} \approx 22.4 \text{ GFLOP/s} \ll P.$$

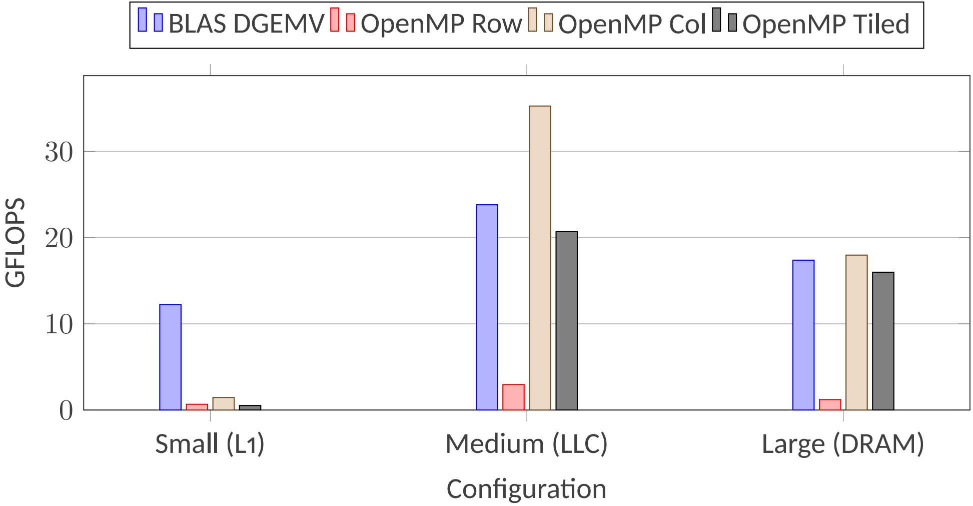
Thus:

- Optimized GEMV should approach 20–22 GFLOP/s.
- Large deviations imply poor locality or bandwidth saturation issues.
- Parallel scaling limited once bandwidth saturated.

Use roofline to validate improvement of column-wise and tiled implementations.

Results and Observations

2 BLAS Level 2: GEMV continued



Measured performance: consistency check

Matrix sizes used (matches earlier size slide):

$$\text{Small: } m = n = 64, \quad \text{Medium: } m = n = 2000, \quad \text{Large: } m = n = 20000.$$

Arithmetic intensity (AI) for square case

$$I(n) = \frac{2n^2 + 2n}{8n^2 + 24n} \xrightarrow[n\to\infty]{} 0.25 \text{ FLOP/Byte.}$$

Concrete values:

$$I(64) \approx 0.243, \; I(2000) \approx 0.250, \; I(20000) \approx 0.250.$$

DRAM roofline (bandwidth B = 89.6 GB/s):

$$R_{\text{mem}} = B \cdot I \approx 21.8\text{–}22.4 \text{ GFLOP/s (all sizes).}$$

Observed (from bar chart):

Measured performance: consistency check

- Small (64): BLAS 13.3 GF/s (≈61% of DRAM roof); OMP row 0.68; OMP col 1.71; OMP tiled 0.48.
- Medium (2000): BLAS 23.4; OMP col 33.4; OMP tiled 20.5 (some values exceed DRAM roof).
- Large (20000): BLAS 17.1; OMP col 17.8; OMP tiled 16.2 (all below DRAM roof).

Flags:

- Column-wise medium case exceeds simple DRAM roof $\Rightarrow$ model underestimates attainable due to cache reuse.
- Small case far below roof $\Rightarrow$ kernel overhead $+$ underutilization dominate.
- Large case memory-bound as expected.

Why some results exceed the DRAM roofline

The 22 GF/s bound assumes pure DRAM streaming. It is not a universal ceiling.

1. Cache residency blocking:
   - Medium matrix (2000×2000): A = 32 MB fits in LLC (36 MB) $\Rightarrow$ many accesses served from L3 after first pass.
   - Effective bandwidth becomes L3 (hundreds GB/s) not DRAM $\Rightarrow$ higher feasible GFLOP/s.
2. Reuse pattern (column-wise outer-product):
   - Reuses each column of A sequentially with daxpy; x(j) stays in registers; y streamed once per column.
3. Simplified bytes-moved model overcounts:
   - Counts full read of A, x, y each repetition; ignores temporal locality of x and partial y residency.

Why some results exceed the DRAM roofline

4. Library optimizations:
   - Vendor BLAS uses micro-kernels, software prefetching, packing improving cache-line reuse.
5. Timing / FLOP accounting alignment:
   - FLOPs formula correct (2mn + 2m), but if beta=0 path or fused operations shorten memory traffic, effective intensity rises.

Conclusion: Use a multi-ceiling roofline (L1/L2/L3/DRAM) to interpret results; DRAM roof alone is insufficient for cached regimes.

Behavior of implementation variants

- Row-wise (ddot per row):
  - Very small per-thread work; function-call overhead; poor vector length; limited ILP → low GFLOP/s.
- Column-wise (daxpy per column / outer-product form):
  - Long contiguous daxpy operations → good SIMD utilization; favorable sequential column access; high cache reuse → best performance.
- Tiled version (current):
  - Private yloc per thread then atomic add for each element → m atomics per thread → heavy serialization.
  - Tile scheduling + allocation overhead further reduces throughput.
- BLAS:
  - Hand-tuned kernels, packing, minimized write-allocate misses, balanced threading.

Key bottleneck now: reduction scheme in tiled kernel (atomics) rather than GEMV arithmetic.

OpenMP SIMD Directive

2 BLAS Level 2: GEMV continued

The *!$omp simd* directive instructs the compiler to vectorize the following loop using SIMD (Single Instruction, Multiple Data) instructions.

```fortran
!$omp simd
do i = istart, iend
    y(i) = y(i) + alpha * A(i,j) * xj
end do
```

Key features:

- Enables automatic vectorization: multiple operations executed simultaneously
- Utilizes CPU vector registers (e.g., AVX-512 on modern Intel/AMD)
- No thread creation overhead — purely instruction-level parallelism
- Can combine with thread parallelism: *!$omp parallel do simd*

Performance impact: 4×–8× speedup typical with AVX2/AVX-512 for suitable loops.

OpenMP SIMD Directive

The *!$omp simd* directive instructs the compiler to vectorize the following loop using SIMD (Single Instruction, Multiple Data) instructions.

```
!$omp simd
do i = istart, iend
    y(i) = y(i) + alpha * A(i,j) * xj
end do
```

Requirements for effective SIMD:

- Contiguous memory access (unit stride)
- No loop-carried dependencies
- Simple loop body (FMA-friendly operations)
- Alignment helps but not strictly required

Performance impact: 4×–8× speedup typical with AVX2/AVX-512 for suitable loops.

Improved GEMV (row-partitioned, column-streaming, no atomics)

Goals: (1) column-major streaming of A, (2) reuse x(j) in registers, (3) avoid per-thread full y copies + atomic reduction.

```
subroutine gemv_openmp_blocked(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(*)
    integer :: i, j, tid, nth, istart, iend, base
    real(real64) :: xj
```

Improved GEMV (row-partitioned, column-streaming, no atomics)

```
! Parallel scale y by beta (write each element once)
!$omp parallel default(none) shared(m,y,beta) private(i)
!$omp do schedule(static)
do i = 1, m
    y(i) = beta * y(i)
end do
!$omp end do
!$omp end parallel

!$omp parallel default(none) shared(m,n,A,lda,x,y,alpha) &
!$omp         private(tid,nth,istart,iend,j,i,xj,base)
tid = omp_get_thread_num()
nth = omp_get_num_threads()
```

Improved GEMV (row-partitioned, column-streaming, no atomics)

```
! Contiguous row partition among threads
base   = (tid * m) / nth
istart = base + 1
iend   = ((tid + 1) * m) / nth

do j = 1, n
    xj = x(j)
    !$omp simd
    do i = istart, iend
        y(i) = y(i) + alpha * A(i,j) * xj
    end do
end do
```

Improved GEMV (row-partitioned, column-streaming, no atomics)

```
    !$omp end parallel
end subroutine gemv_openmp_blocked
```

Key differences vs previous tiled version:

- Eliminates thread-private full copies (`yloc`) and expensive atomic accumulation.
- Each thread updates a disjoint contiguous slice of `y`: no write-sharing, no reduction.
- Outer loop over columns preserves sequential (contiguous) access to `A(:,j)` in column-major layout.
- Reuses scalar `x(j)` across whole row block (likely register-resident).

Improved GEMV (row-partitioned, column-streaming, no atomics)

Why this helps:

- Reduction overhead removed → lower synchronization cost.
- Contiguous row blocks reduce TLB pressure and improve prefetch.
- `!$omp simd` on inner loop encourages vectorization across rows.

Trade-offs / limits:

- Parallelism tied to `m` (number of rows). If `m < threads` utilization poor.
- Load balance assumes uniform cost per row; acceptable for dense `A`.
- Still memory-bound; each `A(i,j)` touched exactly once; intensity capped at 0.25 FLOP/Byte.

Improved GEMV (row-partitioned, column-streaming, no atomics)

Possible refinements:

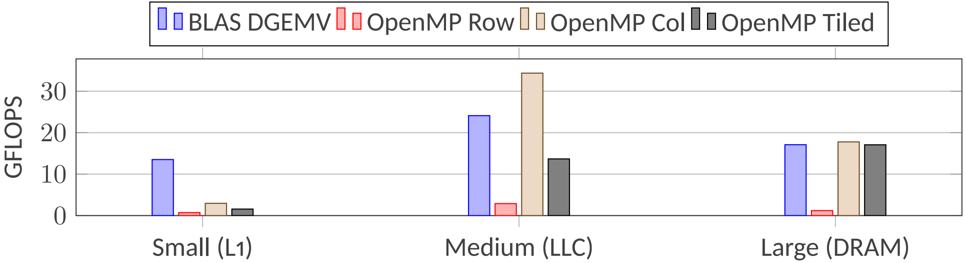
- Hybrid blocking: partition rows, then process columns in chunks to keep `y` slice hot in cache.
- Use micro-kernel (unroll + FMA) for inner loop; block rows in multiples of SIMD width.
- If `n` very large, manual software prefetch on upcoming `A(i,j+pf)`.
- Replace scalar update with small packed panel (register block of `A`).

When to prefer earlier tiled + reduction approach:

- When `n` (columns) is huge and `m` moderately small: tiling over columns can improve `x` reuse granularity.
- When fusing multiple GEMVs sharing the same `y` (batch / multi-right-hand-side emulation).

Improved GEMV (row-partitioned, column-streaming, no atomics)

Summary: This variant removes the dominant bottleneck (atomic reduction) while retaining cache-friendly column streaming over `A(:,j)` and register reuse of `x(j)`, approaching vendor `dgemv` behavior when `m` is large enough for thread scaling.



TRSV: Triangular Solve with Vector

⚠ **Not all Level 2 BLAS operations parallelize well!**

- TRSV solves triangular systems $Ax = b$ where A is:

Lower triangular

$$A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Upper triangular

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

🚫 Sequential dependency: must solve for x_i before x_{i+1} (forward) or x_{i-1} (backward)

💡 Options exist: iterative methods for sparse matrices, specialized parallel algorithms (covered later)

✓ **Design principle:** Avoid triangular solves in parallel algorithms when possible!

TRSV: Triangular Solve with Vector

```fortran
subroutine fwd_subs(A, b, x)
implicit none
integer :: i, j, n
real(real64), intent(in) :: A(:,:)
real(real64), intent(in) :: b(:)
real(real64), intent(out) :: x(size(b))
n = size(b)
x(1) = b(1)/A(1,1)
do i = 2, n
    x(i) = b(i)
    do j = 1, i-1
        x(i) = x(i) - A(i,j)*x(j)
    end do
    x(i) = x(i)/A(i,i)
end do
end subroutine fwd_subs
```

Backward substitution algorithm.

TRSV: Triangular Solve with Vector

```
subroutine bwd_subs(A, b, x)
implicit none
integer :: i, j, n
real(real64), intent(in) :: A(:,:)
real(real64), intent(in) :: b(:)
real(real64), intent(out) :: x(size(b))
n = size(b)
x(n) = b(n)/A(n,n)
do i = n-1, 1, -1
    x(i) = b(i)
    do j = i+1, n
        x(i) = x(i) - A(i,j)*x(j)
    end do
    x(i) = x(i)/A(i,i)
end do
end subroutine bwd_subs
```

Forward substitution algorithm.

Table of Contents

3 BLAS Level 3

- BLAS Level 2: GEMV continued
  - Looking at the performance: roofline model
  - BLAS Level 2: TRSV

- BLAS Level 3
  - DGEMM with OpenMP
  - Tiled version
  - Tiled version with OpenMP

BLAS Level 3: Overview

- Level 3 BLAS define operators that involve matrices
- Key operations:

  GEMM Computes $C = \alpha AB + \beta C$

  SYR2K Computes symmetric rank-2 update $C = \alpha AB^\top + \alpha BA^\top + \beta C$
- For $n \times n$ matrices: $O(n^3)$ arithmetic operations with $O(n^2)$ data accesses

💡 Excellent arithmetic intensity compared to Level 1 and 2!

GEMM: General Matrix Multiply

Mathematical formulation

$$C = \alpha AB + \beta C$$

- A and B can optionally be transposed or conjugated
- All three matrices may be strided
- Standard matrix multiplication: $\alpha = 1.0$, $\beta = 0.0$

Element-wise formulation:

$$C_{ij} = \alpha \sum_{l=1}^{k} A_{il} B_{lj} + \beta C_{ij}, \quad i = 1, \ldots, m,\ j = 1, \ldots, n$$

GEMM: BLAS Interface

```
call dgemm(transa, transb, m, n, k, alpha, A, lda,
           B, ldb, beta, C, ldc)
```

Parameters:

- `transa`, `transb`: transposition options for A and B
- `m`, `n`, `k`: matrix dimensions
- `alpha`, `beta`: scalar multipliers
- `lda`, `ldb`, `ldc`: leading dimensions

DGEMM: Example Usage

```fortran
program gemm_blass
    use iso_fortran_env, only: real64, output_unit, error_unit
    implicit none
    character(len=100) :: n_str, m_str, k_str
    integer :: n, m, k, info
    real(real64), allocatable :: a(:,:), b(:,:), c(:,:)
    ! Read from command line arguments n, m, k
    if (command_argument_count() < 3) then
        write(error_unit, *) "Usage: gemm_blass n m k"
        stop
    end if
    call get_command_argument(1, n_str)
    call get_command_argument(2, m_str)
    call get_command_argument(3, k_str)
    read(n_str, *) n
```

DGEMM: Example Usage

```fortran
read(m_str, *) m
read(k_str, *) k
! Check if n, m, k are positive integers
if (n <= 0 .or. m <= 0 .or. k <= 0) then
    write(error_unit, '("n = ",I0,", m = ",I0,", k = ",I0," must be positive
    ↪  integers")') n,m,k
    stop
else
    write(output_unit, '("n = ",I0,", m = ",I0,", k = ",I0)') n,m,k
end if
! Allocate matrices
allocate(a(n,k), b(k,m), c(n,m), stat=info)
if (info /= 0) then
    write(error_unit, *) "Error allocating matrices"
    stop
```

DGEMM: Example Usage

```
    end if
    ! Initialize matrices
    call random_number(a)
    call random_number(b)
    call random_number(c)
    ! Perform matrix multiplication using BLAS
    call dgemm('N', 'N', n, m, k, 1.0d0, a, n, b, k, 1.0d0, c, n)
    ! Free matrices
    deallocate(a, b, c, stat=info)
    if (info /= 0) then
        write(error_unit, *) "Error deallocating matrices"
        stop
    end if
end program gemm_blass
```

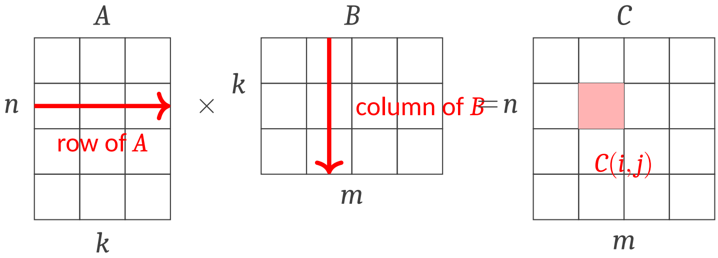
Naïve Implementation: (i, j, l) ordering

```
subroutine matmul_ijl(n,m,k,alpha,A,B,beta,C)
    use iso_fortran_env, only: real64
    implicit none
    integer, intent(in) :: n, m, k
    real(real64), intent(in) :: alpha, A(n,k), B(k,m), beta
    real(real64), intent(inout) :: C(n,m)
    integer :: i, j, l
    do i = 1, m
        do j = 1, n
            C(i,j) = beta * C(i,j)
            do l = 1, k
                C(i,j) = C(i,j) + alpha * A(i,l) * B(l,j)
            end do
        end do
    end do
end subroutine matmul_ijl
```

Naïve Implementation: (i, j, l) ordering

3 BLAS Level 3

- Direct formula-to-code translation
- Poor memory access pattern for column-major storage

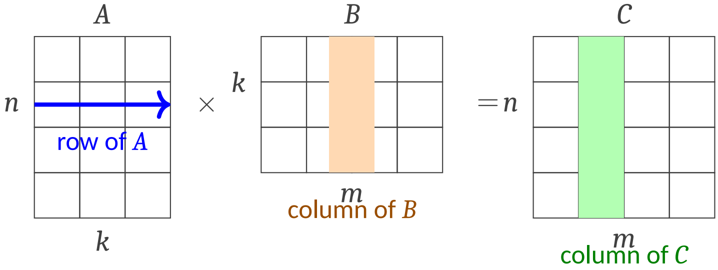


Loop Reordering: (j, i, l) ordering

```
subroutine matmul_jil(n,m,k,alpha,A,B,beta,C)
    use iso_fortran_env, only: real64
    implicit none
    integer, intent(in) :: n, m, k
    real(real64), intent(in) :: alpha, A(n,k), B(k,m), beta
    real(real64), intent(inout) :: C(n,m)
    integer :: i, j, l
    do j = 1, m
        do i = 1, n
            C(i,j) = beta * C(i,j)
            do l = 1, k
                C(i,j) = C(i,j) + alpha * A(i,l) * B(l,j)
            end do
        end do
    end do
end subroutine matmul_jil
```

Loop Reordering: (j, i, l) ordering

- ✓ Better: accesses C in column-major order
- $\sim 2.6\times$ faster than (i, j, l) ordering



Optimal Loop Ordering: (j, l, i)

Why (j, l, i) is best for Fortran column-major layout:

- ✓ **Innermost loop (i):**
  - Reads $A(i, l)$ and $C(i, j)$ contiguously (unit stride)
  - Maximizes cache-line utilization
- ✓ **Middle loop (l):**
  - $B(l, j)$ constant, kept in register
  - Sequential access to columns of A and B
- ✓ **Outer loop (j):**
  - Computes each column of C in turn
  - Good spatial locality

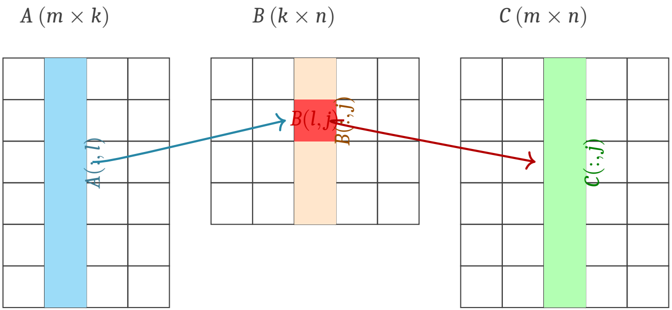
Key principle: Loop order matters! Memory access pattern dominates performance.

Optimal Implementation: (j, l, i) ordering

```fortran
subroutine matmul_jli(n,m,k,alpha,A,B,beta,C)
    use iso_fortran_env, only: real64
    implicit none
    integer, intent(in) :: n, m, k
    real(real64), intent(in) :: alpha, A(n,k), B(k,m), beta
    real(real64), intent(inout) :: C(n,m)
    integer :: i, j, l
    C = beta * C
    do j = 1, n
        do l = 1, k
            do i = 1, m
                C(i,j) = C(i,j) + alpha * A(i,l) * B(l,j)
            end do
        end do
    end do
end subroutine matmul_jli
```

Optimal Implementation: (j, l, i) ordering

- Unit-stride access on all arrays
- $\sim$1.5× faster than (j, i, l), $\sim$4× faster than (i, j, l)



Outer: j

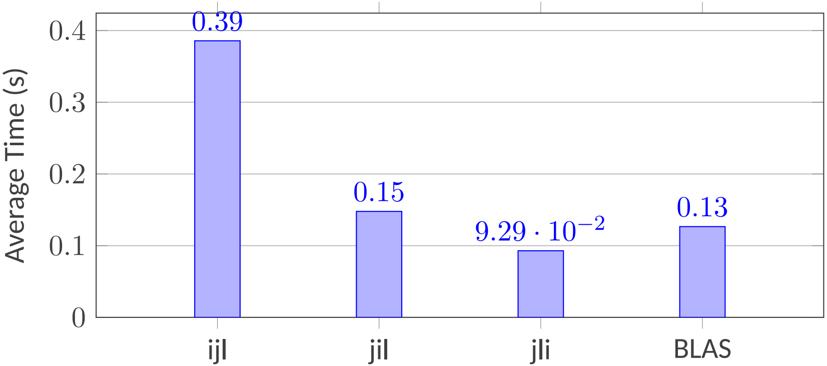
Middle: l

Inner: i

$$C(:,j) \mathrel{+}= \alpha\, A(:,l)\, B(l,j)$$

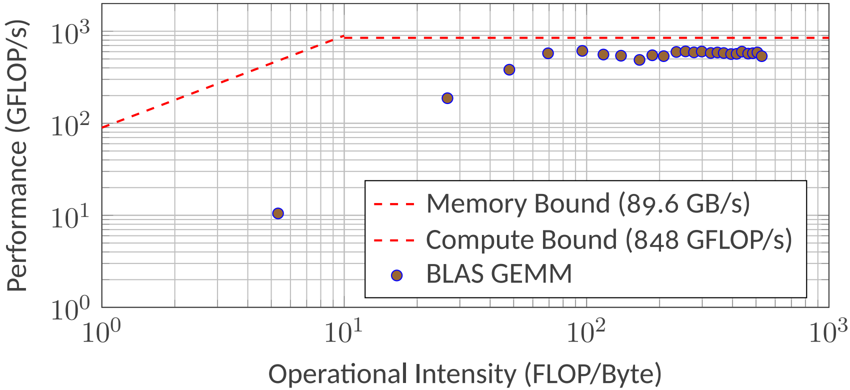
🟦 $A(:,l)$ ▨ $B(:,j)$ 🟥 $B(l,j)$ 🟩 $C(:,j)$ updated

Performance Comparison ($n = 10^3$, avg. over 100 runs)



- (j, l, i) achieves 73% of OpenBLAS performances
- ! BLAS still faster: uses blocking, packing, micro-kernels

Performance Comparison ($n = 10^3$, avg. over 100 runs)



- ✓ High operational intensity (500 FLOP/Byte)
- ✓ Performance approaches compute-bound regime

Parallel DGEMM with OpenMP

We now want to parallelize our DGEMM operation using OpenMP.

- A good starting point is to **start from** our **optimal sequential** implementation (loop order (j, l, i))

Parallel DGEMM with OpenMP

We now want to parallelize our DGEMM operation using OpenMP.

- A good starting point is to **start from** our **optimal sequential** implementation (loop order (j, l, i))
- We just write the triple loop as before, and add OpenMP directives to parallelize the outer loops:

```
!$omp parallel default(none) shared(C,beta,m,n) private(i,j)
!$omp do schedule(static)
do j = 1, n
    !$omp simd
    do i = 1, m
        C(i,j) = beta * C(i,j)
    end do
end do
!$omp end do
```

⟨ Continues on the next slide ⟩

Parallel DGEMM with OpenMP

We now want to parallelize our DGEMM operation using OpenMP.

- ! We just write the triple loop as before, and add OpenMP directives to parallelize the outer loops:

```
!$omp do collapse(2) schedule(static) default(none) &
!$omp   shared(A,B,C,alpha,m,n,k) private(i,j,l,blj)
do j = 1, n
    do l = 1, k
        blj = alpha * B(l,j)
        !$omp simd
        do i = 1, m
            C(i,j) = C(i,j) + A(i,l) * blj
        end do
    end do
end do
!$omp end do
!$omp end parallel
```

OpenMP SIMD Pragma

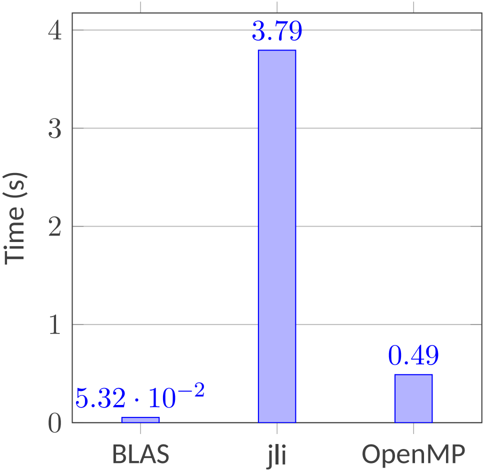
- The *!$omp simd* directive is used to instruct the compiler to vectorize the loop that follows it.
- This pragma allows the compiler to generate SIMD (Single Instruction, Multiple Data) instructions, which can process multiple data points in parallel.
- It is particularly useful in loops where iterations are independent, allowing for significant performance improvements on modern processors.
- Example usage:

```
!$omp simd
do i = 1, m
    C(i,j) = C(i,j) + A(i,l) * blj
end do
```

- In this example, the loop iterations can be executed simultaneously, leveraging the capabilities of the CPU’s vector units.

What did we gain?

3 BLAS Level 3



We used OpenMP to parallelize our optimal sequential DGEMM implementation

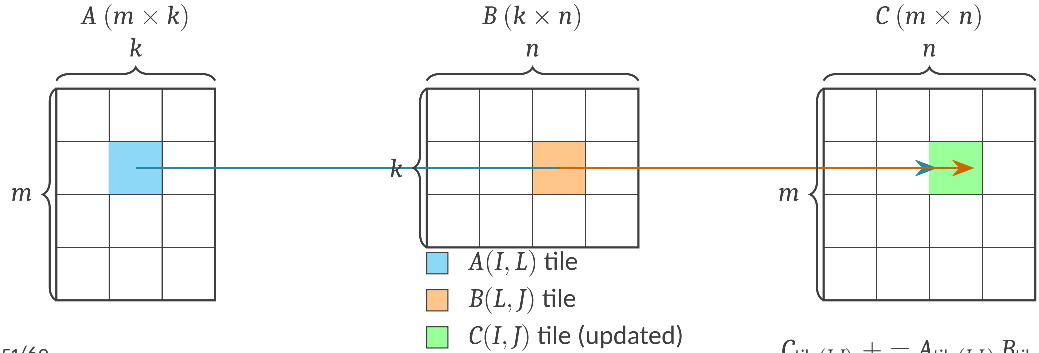
- Tested on matrices of size $n = 2560$, averaged over 20 runs, using 32 threads.
- ✓ Achieved a speedup of $\sim$7.75$\times$ over the sequential version.
- ✗ Still slower than vendor BLAS implementation.

Tiled DGEMM

3 BLAS Level 3

To **improve cache utilization**, we can implement a tiled version of DGEMM.

- Divide the matrices into smaller sub-matrices (tiles) that fit into the cache.
- ! This reduces cache misses and improves data locality.



$$C_{\text{tile}(I,J)} + = A_{\text{tile}(I,L)}\, B_{\text{tile}(L,J)}$$

Tiled DGEMM Implementation

```fortran
subroutine dgemm_tiled(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc, tile_m,
↪  tile_n, tile_k)
   use iso_fortran_env, only: real64
   implicit none
   integer, intent(in) :: m, n, k, lda, ldb, ldc
   integer, intent(in), optional :: tile_m, tile_n, tile_k
   real(real64), intent(in) :: alpha, beta
   real(real64), intent(in) :: A(lda, *)
   real(real64), intent(in) :: B(ldb, *)
   real(real64), intent(inout) :: C(ldc, *)
   ! Local variables
   integer :: i, j, l, ii, jj, ll
   integer :: ts_m, ts_n, ts_k
   integer :: i_end, j_end, l_end
   real(real64) :: temp
```

Tiled DGEMM Implementation

```fortran
! Set tile sizes (default 64)
ts_m = 64
ts_n = 64
ts_k = 64
if (present(tile_m)) ts_m = tile_m
if (present(tile_n)) ts_n = tile_n
if (present(tile_k)) ts_k = tile_k

! Scale C by beta
do j = 1, n
   do i = 1, m
      C(i,j) = beta * C(i,j)
   end do
end do
```

Tiled DGEMM Implementation

```
! Tiled matrix multiplication with non-square tiles
do jj = 1, n, ts_n
   j_end = min(jj + ts_n - 1, n)
   do ll = 1, k, ts_k
      l_end = min(ll + ts_k - 1, k)
      do ii = 1, m, ts_m
         i_end = min(ii + ts_m - 1, m)

         ! Multiply tile
         do j = jj, j_end
            do l = ll, l_end
               temp = alpha * B(l,j)
               do i = ii, i_end
                  C(i,j) = C(i,j) + A(i,l) * temp
```

Tiled DGEMM Implementation

```
                  end do
               end do
            end do

         end do
      end do
   end do

end subroutine dgemm_tiled
```

Tiled DGEMM: Implementation Notes

Key design choices:

- **Tile size selection:** Default 64×64×64 tiles
  - Balances L1/L2 cache capacity vs. parallelism granularity
  - Overridable via optional arguments for tuning
- **Edge handling:** `min()` ensures correct partial tiles at boundaries
- **Loop nest structure:**
  - Outer 3 loops (`jj, ll, ii`): tile iteration
  - Inner 3 loops (`j, l, i`): computation within tile
  - Maintains optimal `(j,l,i)` ordering for cache-friendly access
- **Beta scaling:** Applied once before tiling to avoid redundant operations
- **Temporal reuse:** Each tile of C accumulates contributions from multiple A/B tile pairs, improving cache hit rate

Parallelization opportunity: Outer tile loops are independent.

🏠 Exercise: tuning tiled DGEMM

An exercise for you to try at home!

1. Write a driver program to test the performance of the tiled DGEMM implementation.
2. Experiment with different tile sizes (e.g., 32, 64, 128) to see how they affect performance.
3. Measure execution time and compute performance (GFLOP/s) for various matrix sizes (e.g., 512, 1024, 2048).
4. Compare the performance of your tiled DGEMM with the non-tiled version and with a vendor BLAS implementation.
5. Analyze the results and determine the optimal tile size for your specific hardware.

Tiled DGEMM with OpenMP

We can further enhance our tiled DGEMM implementation by adding OpenMP directives to parallelize the outer tile loops.

- 💡 This allows multiple tiles to be computed simultaneously, leveraging multi-core processors.
- ❗ We add OpenMP pragmas to the outer loops iterating over tiles.

  To hope for good performance, make sure to choose

  - tile sizes that provide enough work per thread to amortize threading overhead,
  - tile sizes that divide the matrix dimensions exactly to avoid load imbalance.
  - **If not**, we would need to implement dynamic scheduling or handle edge cases carefully to avoid loss of performance.

Tiled DGEMM with OpenMP: implementation

```
subroutine dgemm_tiled_openmp(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc, &
    tile_m, tile_n, tile_k)
    use iso_fortran_env, only: real64
    implicit none

    integer, intent(in) :: m, n, k, lda, ldb, ldc
    integer, intent(in), optional :: tile_m, tile_n, tile_k
    real(real64), intent(in) :: alpha, beta
    real(real64), intent(in) :: A(lda, *)
    real(real64), intent(in) :: B(ldb, *)
    real(real64), intent(inout) :: C(ldc, *)

    integer :: ts_m, ts_n, ts_k
    integer :: ii, jj, ll
    integer :: i, j, l
```

Tiled DGEMM with OpenMP: implementation

```
integer :: i_end, j_end, l_end
integer :: ib, jb
real(real64) :: tmp

! ---- MAXIMUM tile sizes (adjust safely for your CPU cache) ----
integer, parameter :: MAX_TS_M = 128
integer, parameter :: MAX_TS_N = 128

! Local tile buffer, fixed size (thread-private due to OpenMP)
real(real64) :: Cbuf(MAX_TS_M, MAX_TS_N)

! Default tile sizes
ts_m = 64
ts_n = 64
ts_k = 64
```

Tiled DGEMM with OpenMP: implementation

```
if (present(tile_m)) ts_m = min(tile_m, MAX_TS_M)
if (present(tile_n)) ts_n = min(tile_n, MAX_TS_N)
if (present(tile_k)) ts_k = tile_k

!$omp parallel default(none) &
!$omp shared(m,n,k,ts_m,ts_n,ts_k,A,B,C,alpha,beta,lda,ldb,ldc) &
!$omp private(ii,jj,ll,i,j,l,i_end,j_end,l_end,ib,jb,Cbuf,tmp)
!$omp do collapse(2) schedule(static)
do jj = 1, n, ts_n
   do ii = 1, m, ts_m
      ! Work tile bounds
      i_end = min(ii + ts_m - 1, m)
      j_end = min(jj + ts_n - 1, n)

      ib = i_end - ii + 1   ! actual tile height
```

Tiled DGEMM with OpenMP: implementation

```
jb = j_end - jj + 1   ! actual tile width
! -------------------------------
! Load and scale C tile: Cbuf = beta * C
! -------------------------------
do j = 1, jb
   do i = 1, ib
      Cbuf(i, j) = beta * C(ii + i - 1, jj + j - 1)
   end do
end do
! -------------------------------
! Accumulate over all K tiles
! -------------------------------
do ll = 1, k, ts_k
   l_end = min(ll + ts_k - 1, k)
```

Tiled DGEMM with OpenMP: implementation

```
   do l = ll, l_end
      do j = 1, jb
         ! scalar needed for whole column
         tmp = alpha * B(l, jj + j - 1)

         !$omp simd
         do i = 1, ib
            Cbuf(i, j) = Cbuf(i, j) + A(ii + i - 1, l) * tmp
         end do
      end do
   end do
end do
! ------------------------------
! Write tile back to C
! ------------------------------
```

Tiled DGEMM with OpenMP: implementation

```fortran
         do j = 1, jb
            do i = 1, ib
               C(ii + i - 1, jj + j - 1) = Cbuf(i, j)
            end do
         end do

      end do
   end do
   !$omp end do
   !$omp end parallel

end subroutine dgemm_tiled_openmp
```

Tiled DGEMM with OpenMP: Implementation Notes

Key improvements over sequential tiled version:

- **Thread-private tile buffer:** Each thread allocates `Cbuf(MAX_TS_M, MAX_TS_N)` on its stack
  - Eliminates write conflicts to shared `C` during accumulation
  - Local buffer has better cache affinity than scattered `C` updates
- **Collapsed parallelization:** *`!$omp do collapse(2)`* over `(jj, ii)` tile indices
  - Increases parallel grain count: `(n/ts_n) * (m/ts_m)` independent tasks
  - Better load balance when `n` or `m` is small relative to thread count
- **Three-stage tile computation:**
  1. **Load & scale:** `Cbuf = beta * C(tile)`
  2. **Accumulate:** Loop over `ll` (K-tiles), perform `Cbuf += alpha * A(tile) * B(tile)`
  3. **Write-back:** `C(tile) = Cbuf`

  Minimizes memory traffic to global `C`: two passes instead of `O(k/ts_k)` read-modify-writes
- **Innermost SIMD:** *`!$omp simd`* on row loop within tile maximizes ILP

Tiled DGEMM with OpenMP: Trade-offs

Memory considerations:

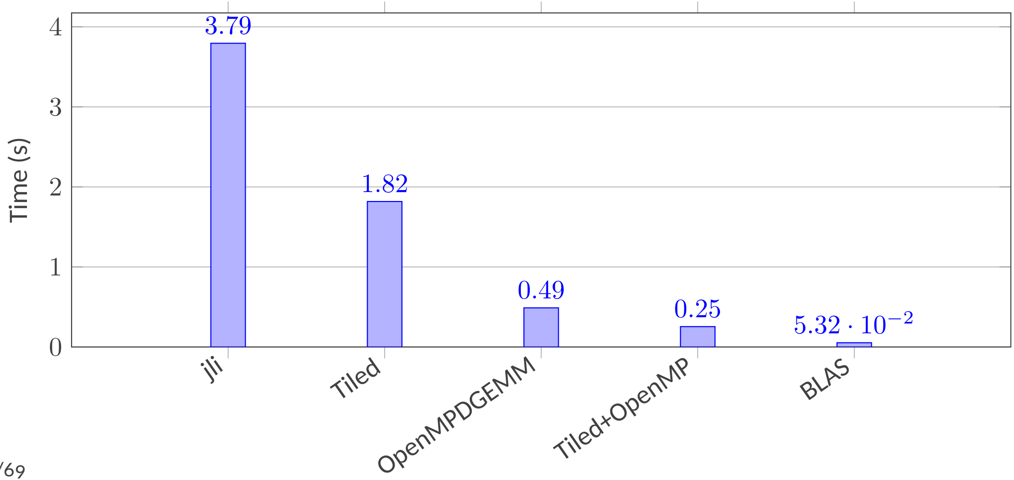
- ! **Stack pressure:** Each thread needs `8 * MAX_TS_M * MAX_TS_N` bytes
  - Example: `MAX_TS_M = MAX_TS_N = 128` ⇒ 128 KB/thread
  - 32 threads ⇒ 4 MB total (acceptable on modern systems)
  - May need `ulimit -s unlimited` or adjust stack size limits
- 💡 **Cache optimization:** `Cbuf` stays hot in L1/L2 during K-loop accumulation
  - Temporal reuse: each `Cbuf` element updated `k/ts_k` times without eviction
  - Reduces `C` memory traffic by factor of `k/ts_k`

Performance tuning:

- 🔧 Choose `ts_m`, `ts_n` to balance:
  - Tile buffer fits in L2 cache (`ts_m * ts_n * 8` bytes $\lesssim$ L2 size)
  - Enough tiles for good thread utilization: `(m/ts_m)*(n/ts_n) >= num_threads * 4`
- 🔧 `ts_k` primarily affects `A`/`B` reuse, less critical than `ts_m`/`ts_n`

Performance Comparison: Tiled vs. Tiled+OpenMP

Test configuration: `m = n = k = 2560`, 32 threads, tile sizes $64 \times 64 \times 64$



Final remarks and conclusions

Observations:

- ✓ OpenMP tiling achieves ∼7.1× speedup over sequential tiled version
- ✓ Reaches ∼21% of vendor BLAS performance (reasonable for educational implementation)
- ! Remaining gap due to:

Register blocking: further subdividing tiles to fit in CPU registers

Micro-kernels: hand-optimized inner loops using assembly or intrinsics

Prefetching: software prefetch instructions to hide memory latency

Packing: reorganizing data in contiguous buffers to improve cache access patterns

Summary of Lecture 6

- We completed our study of the DGEMV operation.
- We explored the implementation of DGEMM from basic triple-loop to optimized tiled and parallel versions.
- We analyzed performance using the Roofline model, highlighting the importance of operational intensity.
- We discussed key optimization techniques such as loop ordering, tiling, and OpenMP parallelization.
- We provided a foundation for further exploration into high-performance computing and numerical linear algebra.

Next up: start pushing outside the frontier of a single CPU: distributed memory parallelism with MPI!