Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24
«Dense Linear Algebra»

Παναγιώτης Χατζηδούκας

Basic Linear Algebra Subprograms (BLAS)

- BLAS is the de-facto standard API for any dense vector and matrix operations, first published in 1979
- A reference implementation is available on netlib.org:
  - http://www.netlib.org/blas/ for the original Fortran version
  - http://www.netlib.org/clapack/ for an f2c translated C version
- BLAS is the building block of many other libraries and programs. These libraries rely on an optimized BLAS library for optimal performance
  - LAPACK and LINPACK
  - NAG (commercial)
  - IMSL (commercial)
  - Matlab
  - Python (numpy and scipy)

Basic Linear Algebra Subprograms (BLAS)

- Open-source and optimized implementations:
  - openBLAS (https://www.openblas.net/)
  - ATLAS (https://netlib.org/atlas/): self-tuned BLAS, included with many Linux distributions
- Optimized versions exist from many hardware vendors:
  - Apple: part of the Accelerate framework (-framework veclib)
  - IBM: part of the ESSL (Engineering and Scientific Subroutine Library)
  - Cray: part of libsci library
- From CPU manufacturers
  - Intel MKL library
    - https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
  - AMD ACML and AOCL libraries
    - https://en.wikipedia.org/wiki/AMD_Core_Math_Library
    - https://developer.amd.com/amd-aocl/
- GPU implementations are also available

BLAS levels 1, 2 and 3

- The BLAS functions are split into three groups (or levels)
- BLAS level 1
  - scalar and vector operations, such as dot product and vector addition ($\boldsymbol{y} \leftarrow a\boldsymbol{x} + \boldsymbol{y}$)
  - scale as O(1) or O(N)
- BLAS level 2
  - matrix-vector operations, such as matrix-vector multiplication ($\boldsymbol{y} \leftarrow aA\boldsymbol{x} + b\boldsymbol{y}$)
  - scale as $O(N^2)$
- BLAS level 3
  - matrix-matrix operations, such as matrix-matrix multiplication ($C \leftarrow aAB + bC$)
  - scale worse than $O(N^2)$, often $O(N^3)$

Calling BLAS functions

- BLAS is a Fortran library. It can be called from any language, but you must learn some facts about Fortran and calling Fortran functions.
- Function names and arguments:
  - The function names are all lowercase independent of what is written in (case insensitive) Fortran code
  - Function names on most machines add a trailing _ compared to C/C++ functions.
  - Parameter types are not mangled into the function name: use extern “C” in the function declaration
  - all arguments are passed by address (or equivalently reference in C++). The best convention is to
    - pass scalar arguments by reference
    - pass C-style arrays as pointers
  - Be careful about how integer types relate. This can depend on compiler options. Typically, a Fortran integer is a C/C++ int, but it can be a long.

Example: DDOT

- The Fortran DDOT function

```
      DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
      INTEGER INCX,INCY,N
      DOUBLE PRECISION DX(*),DY(*)
*
*     DDOT forms the dot product of two vectors.
*     uses unrolled loops for increments equal to one.
*
```

- Has the following C++ prototype

```
extern "C" double ddot_(int& n, double *x, int& incx, double *y, int& incy);
```

- And can be easily called

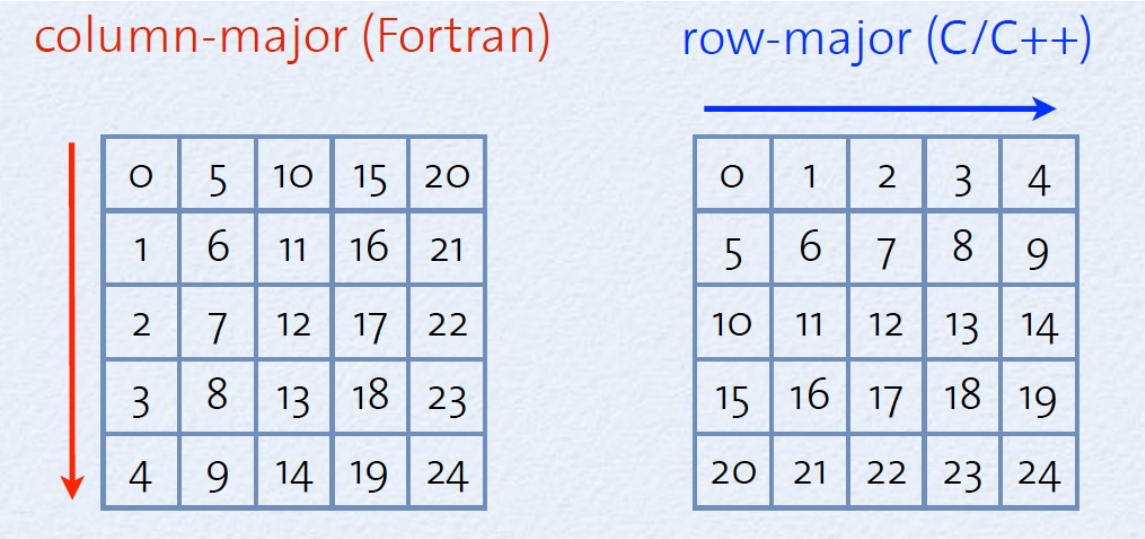
```
int main()
{
  std::vector<double> x(10, 1.); // intialize a vector with ten 1s
  std::vector<double> y(10, 2.); // intialize a vector with ten 2s

  // calculate the inner product
  int n=x.size();
  int one = 1;
  double d = ddot_(n,&x[0],one,&y[0],one);
  std:: cout << d << "\n"; // should be 20
}
```

- Do not forget to link against the BLAS library

Array storage

- Fortran indices by default start at 1, while C/C++ starts at 0
- Fortran stores arrays in column-major order, while C/C++ uses row-major order



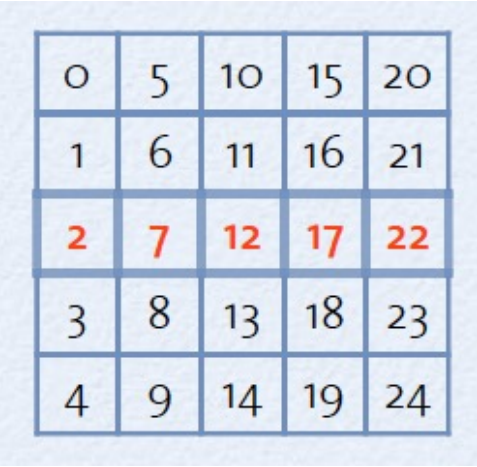
- Consequence:
  - matrices are typically transposed
  - A[i][j] in C/C++ is A(j+1,i+1) in Fortran

Another look at DDOT: increments

- The DDOT dot product function takes two pointers and two increments

```
      DOUBLE PRECISION FUNCTION DDOT(N,DX,INCX,DY,INCY)
      INTEGER INCX,INCY,N
      DOUBLE PRECISION DX(*),DY(*)
*
*     DDOT forms the dot product of two vectors.
*     uses unrolled loops for increments equal to one.
*
```

- In arrays the increments is typically 1
- The increments exist as arguments to be able to treat columns and rows in matrices as vectors




```
DX = start of storage + 2
INCX = 5
```

BLAS naming conventions

- BLAS functions always have one (or two) prefix indicating the type of the arguments and optional return value

I	int
S	float
D	double
C	std::complex<float>
Z	std::complex<double>

- Example: dot product

generic name	_DOT
float	SDOT
double	DDOT
std::complex<float>	CDOT
std::complex<double>	ZDOT

BLAS 1: vector operations

Reduction operations:				
s	$\leftarrow$	$x \cdot y$	inner product	__DOT_
s	$\leftarrow$	$max\{\lvert x_i \rvert\}$	pivot search	I_AMAX
s	$\leftarrow$	$\lVert x \rVert_2$	norm of a vector	_NRM2
s	$\leftarrow$	$\sum_i \lvert x_i \rvert$	sum of abs	_ASUM

Vector to vector transformations:				
y	$\leftarrow$	x	copy x into y	_COPY
x	$\leftrightarrow$	y	swap	_SWAP
y	$\leftarrow$	$\alpha \cdot x$	scale x	_SCAL
y	$\leftarrow$	$\alpha \cdot x + y$	saxpy	_AXPY

Generate and apply Givens rotations:

Compute rotation:				
$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$	$\rightarrow$	$\begin{bmatrix} r \\ 0 \end{bmatrix}$	$c, s \ni r = \sqrt{a^2 + b^2}$	_ROTG
Apply rotation:				
$\begin{bmatrix} x \\ y \end{bmatrix}$	$\leftarrow$	$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$		_ROT

BLAS matrix types and naming conventions

- BLAS 2 and BLAS 3 support various matrix types, given as two letters after the prefix

GE	general dense matrix
GB	banded matrix, stored packed
SY	symmetric, stored like a general dense matrix
SP	symmetric, stored packed
SB	symmetric banded, stored packed
HE	hermitian, stored like a general dense matrix
HP	hermitian, stored packed
HB	hermitian banded, stored packed
TR	upper or lower triangular, stored like a general dense matrix
TP	upper or lower triangular, stored packed
TB	upper or lower triangular band matrix, stored packed

- Example: DGEMV is matrix-vector multiplication for a general matrix of doubles

Packed storage formats

- Banded matrices:

Dense storage of matrix	Packed storage as a packed matrix
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{matrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{matrix}$

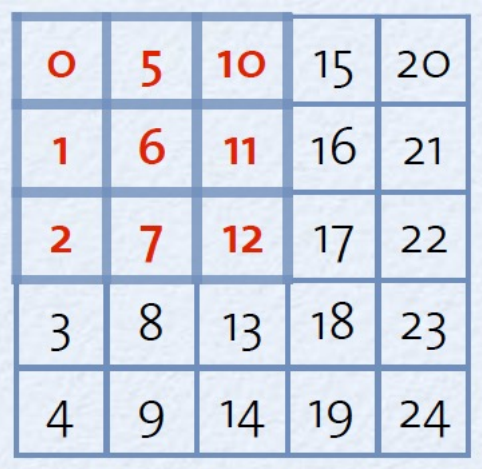
- For triangular matrices, depending on the UPLO parameter:

UPLO	Dense storage of matrix	Packed storage as array
U	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ & a_{22} & a_{23} & a_{24} \\ & & a_{33} & a_{34} \\ & & & a_{44} \end{pmatrix}$	$a_{11}\ \underbrace{a_{12}\ a_{22}}\ \underbrace{a_{13}\ a_{23}\ a_{33}}\ \underbrace{a_{14}\ a_{24}\ a_{34}\ a_{44}}$
L	$\begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$	$\underbrace{a_{11}\ a_{21}\ a_{31}\ a_{41}}\ \underbrace{a_{22}\ a_{32}\ a_{42}}\ \underbrace{a_{33}\ a_{43}}\ a_{44}$

- Symmetric and hermitian packed formats store only one triangle

Dense matrix storage

- It's a bit more complicated than you thought
  - Fortran-77 and earlier did not allow dynamical allocation
  - One might want to operate just on a submatrix
- Matrix operations accept three size arguments:
  - matrix size: rows and columns of the matrix
  - leading dimension: increment between columns



number of rows: 3
number of columns: 3
leading dimension: 5

BLAS-2: matrix-vector operations

Matrix times Vector				
x	$\leftarrow$	$\alpha Ax + \beta y$	general	_GEMV
			general band	_GBMV
			general hermitian	_HEMV
			hermitian banded	_HBMV
			hermitian packed	_HPMV
			general symmetric	_SYMV
			symmetric banded	_SBMV
			symmetric packed	_SPMV
x	$\leftarrow$	Ax	triangular	_TRMV
			triangular banded	_TBMV
			triangular packed	_TPMV

Rank one and rank two updates:				
A	$\leftarrow$	$\alpha xy^T + A$	general	_GER_
A	$\leftarrow$	$\alpha xx^* + A$	general hermitian	_HER
			hermitian packed	_HPR
A	$\leftarrow$	$\alpha(xy^* + yx^*) + A$	gen. Hermitian	_HER2
			hermitian packed	_HPR2
A	$\leftarrow$	$\alpha xx^T + A$	general symmetric	_SYR
			symmetric packed	_SPR
A	$\leftarrow$	$\alpha(xy^T + yx^T) + A$	gen. symmetric	_SYR2
			symmetric packed	_SPR2

Triangular solve:				
x	$\leftarrow$	$A^{-1}x$	triangular	_TRSV
			triangular banded	_TBSV
			triangular packed	_TPSV

BLAS-3: matrix-matrix operations

Matrix product:				
C	$\leftarrow$	$\alpha A \cdot B + \beta C$	general	_GEMM
			symmetric	_SYMM
			hermitian	_HEMM
B	$\leftarrow$	$\alpha A \cdot B$	triangular	_TRMM
Rank k update:				
C	$\leftarrow$	$\alpha A \cdot A^T + \beta C$		_SYRK
C	$\leftarrow$	$\alpha A \cdot A^H + \beta C$		_HERK
C	$\leftarrow$	$\alpha(A \cdot B^T + B \cdot A^T) + \beta C$		_SYRK2
C	$\leftarrow$	$\alpha(A \cdot B^H + B \cdot A^H) + \beta C$		_HERK2
Triangular solve for multiple r.h.s.:				
B	$\leftarrow$	$\alpha A^{-1} \cdot B$	triangular	_TRSM

Transpose arguments

- _GEMV, _GBMV, _T_MV, and _T_SV take arguments indicating whether the matrix should be transposed

TRANS	Real matrix S, D	Complex matrix C, Z
'N' or 'n'	no transpose	no transpose
'T' or 't'	transposed	transposed
'C' or 'c'	transposed	transposed and complex conjugated

- Similarly, some of the BLAS-3 calls take one or two transpose arguments:
  - _GEMM,_TRMM
  - _SYRK, _HERK, _SY2RK,
  - _TRSM

C/C++ interface

- Available in cblas.h

```
typedef enum CBLAS_ORDER {CblasRowMajor=101, CblasColMajor=102} CBLAS_ORDER;
typedef enum CBLAS_TRANSPOSE {CblasNoTrans=111, CblasTrans=112, CblasConjTrans=113,
                              CblasConjNoTrans=114} CBLAS_TRANSPOSE;
typedef enum CBLAS_UPLO {CblasUpper=121, CblasLower=122} CBLAS_UPLO;



typedef int blasint;
#define OPENBLAS_CONST const
#define CBLAS_INDEX size_t

float cblas_sdot(const int n, const float *x, const int incx,
                              const float *y, const int incy);

double cblas_ddot(const int n, const double *x, const int incx,
                               const double *y, const int incy);

void cblas_saxpy(const int n, const float alpha, const float *x, const int incx,
                                                 float *y, const int incy);

void cblas_sgemv(const enum CBLAS_ORDER order, const enum CBLAS_TRANSPOSE trans,
                 const int m, const int n,
                 const float alpha, const float *a, const int lda,
                 const float *x, const int incx,
                 const float beta, float *y, const int incy);
```

Optimizing linear algebra operations

- BLAS-1 is best optimized by SIMD vectorization
  - examples: _DOT and _SCAL
- BLAS-2 and BLAS-3 build on top of BLAS-1
  - reuse all optimizations done for BLAS-1
  - potential for further optimization by multithreading
  - examples: _GEMV and _GEMM
- Other libraries, like LAPACK, are built on top of BLAS
  - reuse all optimizations done for BLAS-1, 2 and 3
  - further parallelization may be possible
  - example: Gaussian elimination (_GEFA)

Paralleling _GEMV

- We have two loops in _GEMV over i and j

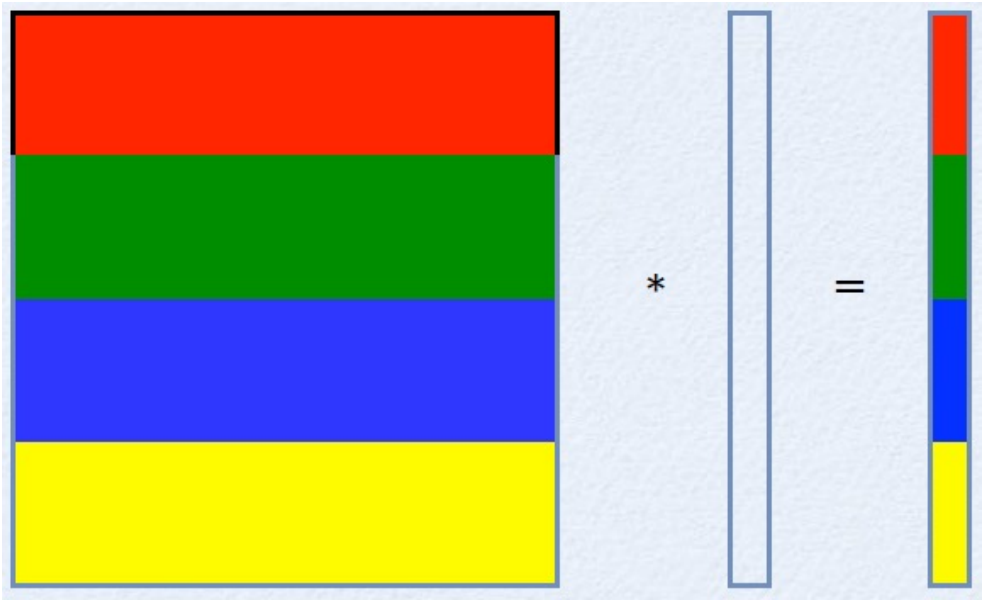
$$y_i = \sum_j A_{ij} x_j$$

- Four versions
  - Loop order can be i, j or j, i
  - either the inner or the outer loop can be parallelized
- Two more versions:
  - split the matrix into blocks and use a single-threaded BLAS _GEMV for each block
  - hope for a parallel BLAS and just call _GEMV

Case 1: i,j, parallelizing outer loop

- Parallelize the outer loop over i

```
void dgemv_omp1(int m, int n, int lda, const double* A, const double* x, double* y)
{
  #pragma omp parallel for
  for (int i = 0; i < m; ++i) {
    y[i] = 0;
    for (int j = 0; j < n; ++j)
      y[i] += A[i * lda + j] * x[j];
  }
}
```



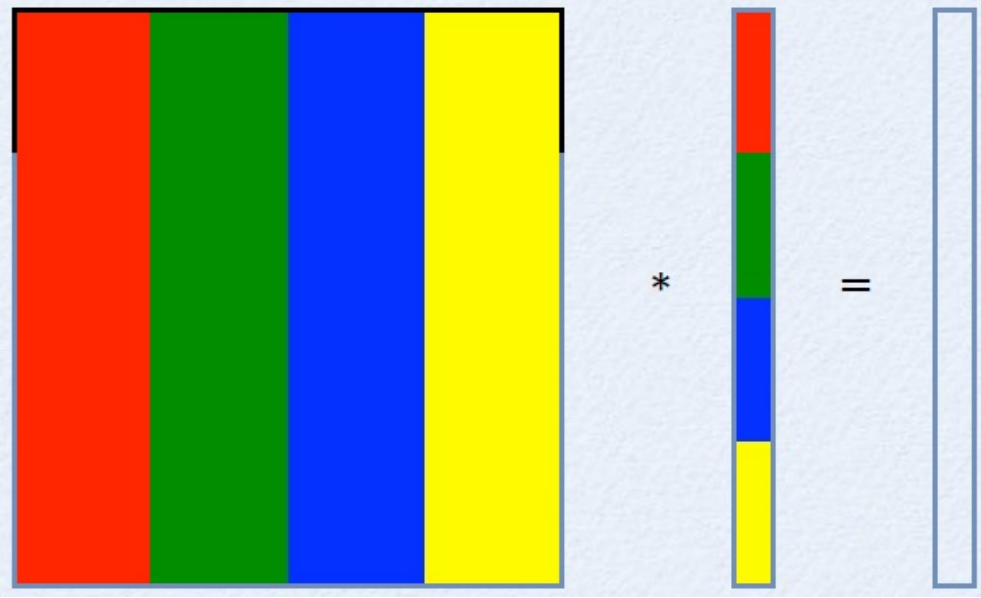
Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 2: i,j, parallelizing inner loop

- Parallelize the inner loop over j

```
void dgemv_omp2(int m, int n, int lda, const double* A, const double* x, double* y)
{
  for (int i = 0; i < m; ++i) {
    double tmp = 0.0;
    #pragma omp parallel for reduction(+:tmp)
    for (int j = 0; j < n; ++j)
      tmp += A[i * lda + j] * x[j];

    y[i] = tmp;
  }
}
```



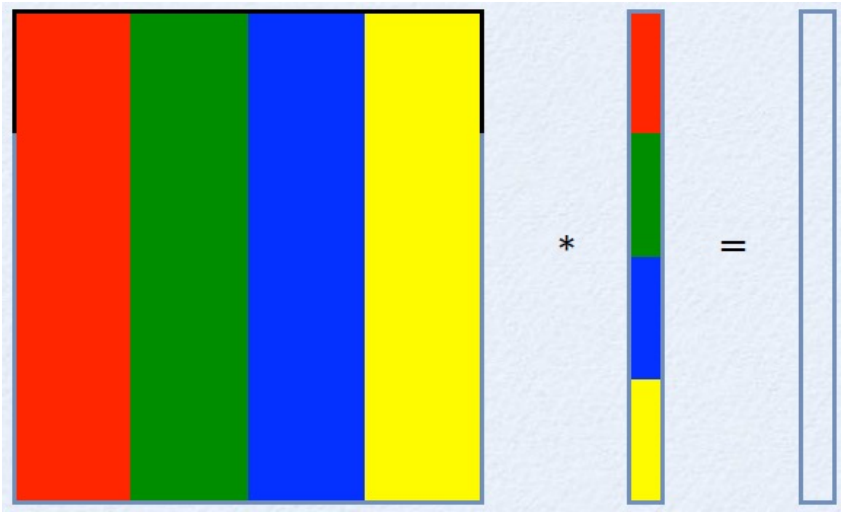
Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 3: j,i parallelizing outer loop

- Parallelize the outer loop over j, followed by vector reduction

```
void dgemv_omp3(int m, int n, int lda, const double* A, const double* x, double* y)
{
  memset(y, 0, m*sizeof(double));
  #pragma omp parallel
  {
    double z[m];
    memset(z, 0, m*sizeof(double));
    #pragma omp for
    for (int j = 0; j < n; ++j)
      for (int i = 0; i < m; ++i)
        z[i] += A[i * lda + j] * x[j];

    #pragma omp critical
    for (int i = 0; i < m; ++i) y[i] += z[i];
  }
}
```



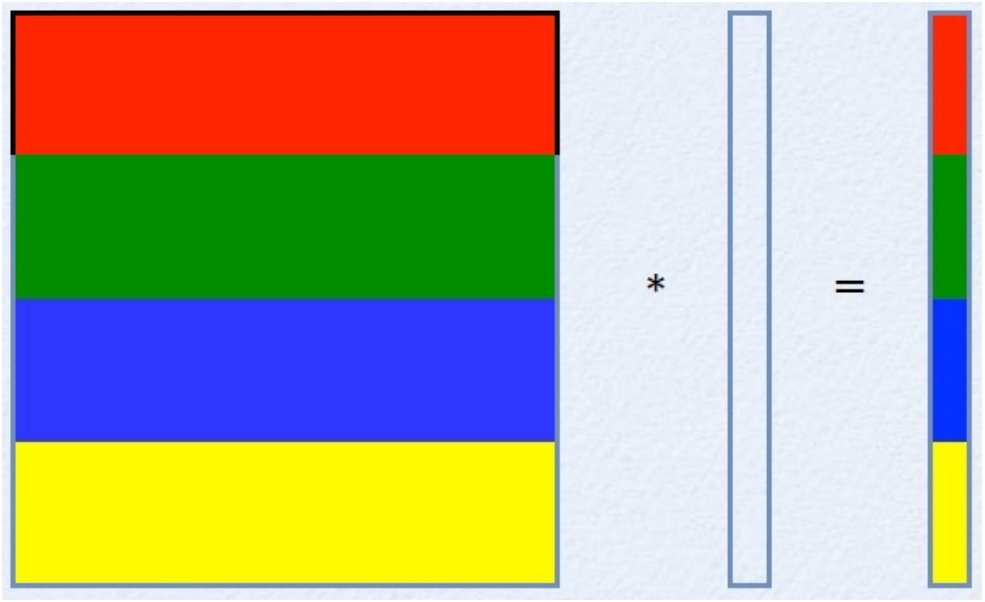
Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 4: j,i parallelizing inner loop

- Parallelize the inner loop over i

```
void dgemv_omp4(int m, int n, int lda, const double* A, const double* x, double* y)
{
  for (int i = 0; i < m; ++i) y[i] = 0;

  for (int j = 0; j < n; ++j)
    #pragma omp parallel for
    for (int i = 0; i < m; ++i) {
      y[i] += A[i * lda + j] * x[j];
    }
}
```



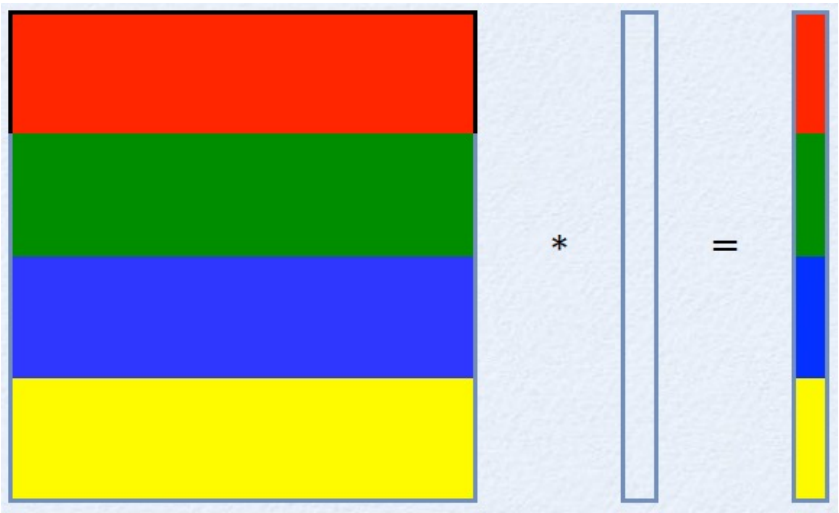
Colors indicate splitting of the matrix over threads for the case of 4 threads

Case 5: calling BLAS from every thread

```
void dgemv_omp5(int m, int n, int lda, const double* A, const double* x, double* y)
{
  openblas_set_num_threads(1);

  #pragma omp parallel
  {
    int i = omp_get_thread_num();
    int p = omp_get_num_threads();
    int chunk = (m + p - 1)/p;
    int m0 = chunk;
    int m1 = min(m0, m-i*m0);
    //int lda = m;

    cblas_dgemv(CblasRowMajor, CblasNoTrans, m1, n, 1.0, &A[i*m0],
                lda, x, 1.0, 0.0, &y[i*m0], 1);
  }
}
```



Colors indicate splitting of the matrix over threads for the case of 4 threads

Indicative performance results

```
minmac:dgemv phadjido$ export OMP_NUM_THREADS=4
minmac:dgemv phadjido$ ./dgemv1
dgemv_naive [n=1024] time= 1.738 ms -> 1.207 GFLOPs
dgemv_omp1  [n=1024] time= 0.707 ms -> 2.968 GFLOPs
dgemv_omp2  [n=1024] time=42.262 ms -> 0.050 GFLOPs
dgemv_omp3  [n=1024] time= 3.532 ms -> 0.594 GFLOPs
dgemv_omp4  [n=1024] time=46.589 ms -> 0.045 GFLOPs
dgemv_omp5  [n=1024] time= 0.603 ms -> 3.479 GFLOPs
dgemv_blas  [n=1024] time= 0.593 ms -> 3.537 GFLOPs
```

References

- Prof. M. Troyer, HPCSE I, ETH Zurich