CUBLAS linear algebra library

Presentation of CUBLAS

Some matrix technicalities

Some helper functions

Some core functions

Example: matrix multiplication

Presentation of CUBLAS

BLAS library: Basic Linear Algebra Subprograms

traditional library

highly optimized operations on vectors and matrices

RAM bottleneck → vectorial approach to the handling of data within cache memory and registers

initially used in Fortran → CUBLAS assumes *column-major* storage of matrices

but *row-major* storage of matrices in C

→ use two accessors:

row-major storage:

```
#define Mij(M , srow , i , j)  (*((M) + (i) * (srow) + (j)))
```

column-major storage:

```
#define cMij(M , scol , i , j)  (*((M) + (i) + (j) * (scol)))
```

and write some (basic) conversion functions between *row-major* and *column-major* storage

CUBLAS library: BLAS parallelized on CUDA

matrices and vectors must be allocated and loaded on the device global memory

- *helper functions:*

  initialize - *shutdown* CUBLAS mode

  allocate - *free* memory in device global memory (analogous to **cudaMalloc** - **cudaFree**)

  copy vectors and matrices between host memory and device global memory

- *core functions:*

  traditional operations of BLAS organized in 3 groups:

  - *BLAS1*: operations on *vectors*
  - *BLAS2*: operations on both *vectors and matrices*
  - *BLAS3*: operations on *matrices* (matrix multiplication, triangular system resolution, ...)

  name: **cublasDxxxx**
  - **D** for double-precision floating point arithmetic
  - **xxxx** name of operation

template of a CUBLAS code:

```
#include <cublas.h>
.......
cublasInit();
cublasAlloc(...);
.......
cublasSetVector(...);
cublasSetMatrix(...);
.......
sequence of core functions
.......
cublasGetVector(...);
cublasGetMatrix(...);
.......
cublasFree(...);
cublasShutdown();
```

handling of errors:

encapsulate a call to a *helper function* within our function **CUBLAS_HELPER_CALL** :

```
void CUBLAS_HELPER_CALL(char *call_name , cublasStatus status)
{   if (status != CUBLAS_STATUS_SUCCESS)
    {  printf("\n ERROR for cublas helper call:   %s" , call_name);    exit(0); }  }
```

follow a call to a *core function* with a call to our function **CUBLAS_CALL** :

```
void CUBLAS_CALL(char *call_name)
{   if (cublasGetError() != CUBLAS_STATUS_SUCCESS)
    {  printf("\n ERROR for cublas call:   %s" , call_name);    exit(0); }  }
```

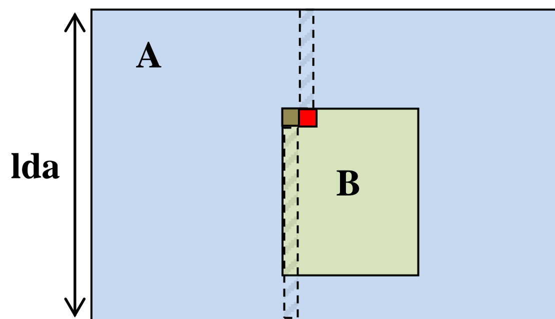
Some matrix technicalities

- *leading dimension of a matrix:*

= number of rows = size of a column (since *column-major storage*)

notation: matrix **A** → leading dimension **lda**

assume **B** *partial matrix* of **A** → the leading dimension of **B** is *still* **lda**



- *vector increment:*

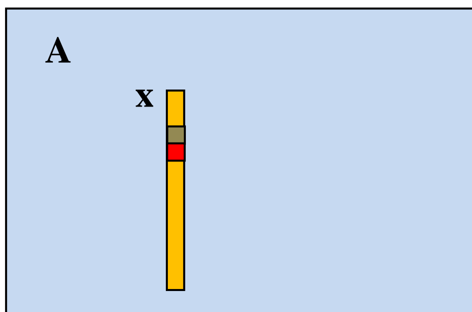
= distance (as a number of elements) between two consecutive elements of the vector

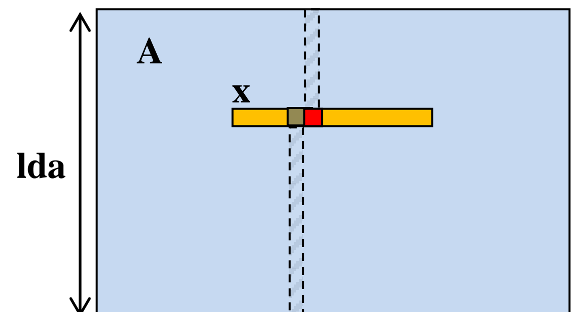
notation: vector **x** → increment **incx**

assume vector **x** is part of matrix **A** (stored in *column-major* order):

incx = 1 → **x** is a (partial) column of **A**

incx = lda → **x** is a (partial) row of **A**





Some helper functions

cublasStatus cublasInit();

cublasStatus cublasShutdown();

cublasStatus cublasAlloc(int n , int el_size , void **d_p) → use cast **(void **)**

allocate **n** elements of size **el_size** at address **d_p** in *device*

cublasStatus cublasFree(void *d_p)

cublasStatus cublasSetVector(int n , int el_size , void *x , int incx , void *y , int incy)

copy **n** elements of size **el_size** from **x** in *host* to **y** in *device*

cublasStatus cublasGetVector(int n , int el_size , void *x , int incx , void *y , int incy)

copy **n** elements of size **el_size** from **x** in *device* to **y** in *host*

cublasStatus cublasSetMatrix(int rows , int cols , int el_size , void *A , int lda , void *B , int ldb)

copy **rows×cols** elements of size **el_size** from **A** in *host* to **B** in *device*

cublasStatus cublasGetMatrix(int rows , int cols , int el_size , void *A , int lda , void *B , int ldb)

copy **rows×cols** elements of size **el_size** from **A** in *device* to **B** in *host*

Some core functions

```
void cublasDcopy (int n , double *x , int incx
                         , double *y , int incy)
```

$y \leftarrow x$ for **n** elements

```
void cublasDscal (int n , double alpha , double *x , int incx)
```

$x \leftarrow alpha \times x$ for **n** elements

```
void cublasDaxpy (int n , double alpha , double *x , int incx
                                       , double *y , int incy)
```

$y \leftarrow alpha \times x + y$ for **n** elements

```
void cublasDgemm ( char transa , char transb
                 , int n , int m , int p
                 , double alpha , double *A , int lda , double *B , int ldb
                 , double beta  , double *C , int ldc)
```

$$C \leftarrow alpha \times A^{(t)} \times B^{(t)} + beta \times C$$

C: n×m ; A: n×p ; B: p×m

- **A** if **transa** = **'N'** ; A^t if **transa** = **'T'**
- **B** if **transb** = **'N'** ; B^t if **transb** = **'T'**

```
void cublasDsyrk (  char uplo , char trans
                  , int n , int p
                  , double alpha  ,  double *A , int lda
                  , double beta   ,  double *C , int ldc)
```

$\mathbf{C} \leftarrow \mathbf{alpha} \times \mathbf{A} \times \mathbf{A}^t + \mathbf{beta} \times \mathbf{C}$ if **trans** = **'N'**

or

$\mathbf{C} \leftarrow \mathbf{alpha} \times \mathbf{A}^t \times \mathbf{A} + \mathbf{beta} \times \mathbf{C}$ if **trans** = **'T'**

n×n **n×p** **p×n**

C symmetric →
- if **uplo** = **'U'**, only the upper triangle of **C** is updated
- if **uplo** = **'L'**, only the lower triangle of **C** is updated

triangular system resolution:

```
void cublasDtrsm ( char side , char uplo , char transa , char diag
                 , int n , int m
                 , double alpha , double *A , int lda , double *B , int ldb)
```

n×n

$A^{(t)} \times X = \text{alpha} \times B$ if **side** = **'L'**

or → solution **X** is put into **B**

$X \times A^{(t)} = \text{alpha} \times B$ if **side** = **'R'**

m×m n×m

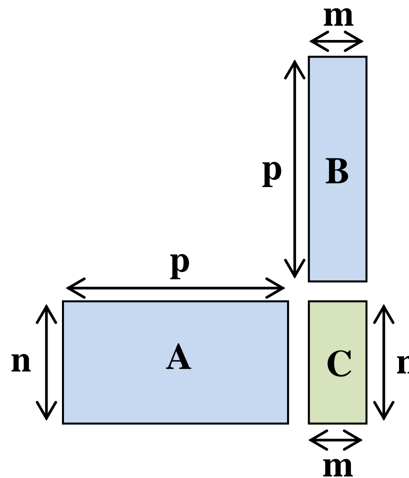
- if **uplo** = **'U'**, **A** is an upper triangular matrix
- if **uplo** = **'L'**, **A** is a lower triangular matrix

A if **transa** = **'N'** ; A^t if **transa** = **'T'**

- if **diag** = **'U'**, **A** is assumed to be unit triangular
- if **diag** = **'N'**, **A** is not assumed to be unit triangular

Example: matrix multiplication

$C \leftarrow A \times B$




```
void mul_mat(double *C , double *A , double *B , int n , int m , int p)
{
   double *d_A , *d_B , *d_C;

   CUBLAS_HELPER_CALL("init" , cublasInit());

   CUBLAS_HELPER_CALL("alloc d_A" , cublasAlloc(n * p  , sizeof(double) , (void **)&d_A));
   CUBLAS_HELPER_CALL("alloc d_B" , cublasAlloc(p * m , sizeof(double) , (void **)&d_B));
   CUBLAS_HELPER_CALL("alloc d_C" , cublasAlloc(n * m , sizeof(double) , (void **)&d_C));

   CUBLAS_HELPER_CALL("set d_A" , cublasSetMatrix(n , p , sizeof(double) , A , n , d_A , n));
   CUBLAS_HELPER_CALL("set d_B" , cublasSetMatrix(p , m , sizeof(double) , B , p , d_B , p));

   cublasDgemm(  'N' , 'N' , n , m , p
                  , 1.0
                  , d_A , n
                  , d_B , p
                  , 0.0
                  , d_C , n);
   CUBLAS_CALL("C = A * B + 0 * C");

   CUBLAS_HELPER_CALL("get d_C" , cublasGetMatrix(n , m , sizeof(double) , d_C , n , C , n));

   CUBLAS_HELPER_CALL("dealloc d_A" , cublasFree(d_A));
   CUBLAS_HELPER_CALL("dealloc d_B" , cublasFree(d_B));
   CUBLAS_HELPER_CALL("dealloc d_C" , cublasFree(d_C));

   CUBLAS_HELPER_CALL("end" , cublasShutdown());
}
```

up to 220 Gflops