GPU Computing

Laurea **Magistrale in Informatica** - AA 2021/22

Docente **G. Grossi**

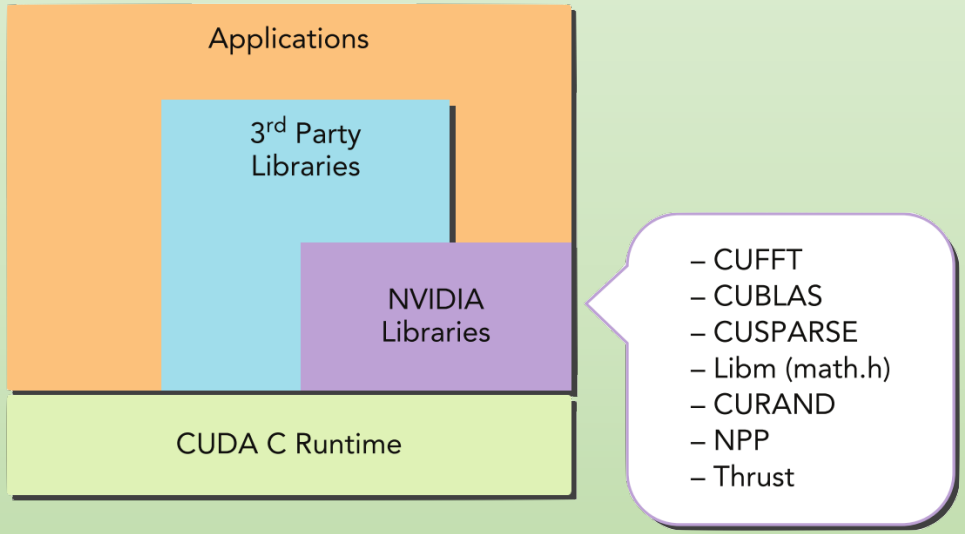
Lezione 9 – Librerie CUDA

Sommario

- ✓ Librerie CUDA
- ✓ cuBLAS
- ✓ cuRAND
- ✓ cuFFT
- ✓ Librerie CUDA in Python

GPU-accelerated CUDA library

- ✓ Tutte le computazioni implementate nelle **librerie** CUDA sono **accelerate** dalla GPU
- ✓ Per molte applicazioni, le librerie CUDA offrono un buon **bilanciamento** tra **usabilità** e **prestazioni**
- ✓ Le API di molte librerie CUDA sono volutamente **simili** a quelle della **libreria standard**
- ✓ Il vantaggio è il minimo sforzo per il **porting** del codice sequenziale a parallelo (meno tempo e complessità)
- ✓ Nessun costo di **mantenimento** che è a **carico** degli **sviluppatori** della libreria (compito svolto in genere da esperti in CUDA programming)
- ✓ Alcune sono sviluppate da **NVIDIA** direttamente altre da **terze parti**
- ✓ Tutte si appoggiano sopra il livello **runtime** al pari delle applicazioni sviluppate da utenti



Esempi di librerie CUDA

- ✓ Esempi di librerie sviluppate in CUDA a supporto di determinati domini
- ✓ Scaricabili da **developer.nvidia.com**

Libreria	Dominio
cuFFT (NVIDIA)	Fast Fourier Transforms Linear
cuBLAS (NVIDIA)	Linear Algebra (BLAS Library)
cuSPARSE (NVIDIA)	Sparse Linear Algebra
cuRAND (NVIDIA)	Random Number Generation
NPP (NVIDIA)	Image and Signal Processing
CUSP (NVIDIA)	Sparse Linear Algebra and Graph Computations
CUDA Math Library (NVIDIA)	Mathematics
Trust (terze parti)	Parallel Algorithms and Data Structures
MAGMA (terze parti)	Next generation Linear Algebra

Workflow tipico

1. Creare un **handle** specifico della libreria (per la gestione delle informazioni e relativo contesto in cui essa opera, es. uso degli stream)
2. Allocare la **device memory** per gli **input** e **output** alle funzioni della libreria (convertirli al **formato specifico** di uso della liberia, es. converti array 2D in column-major order)
3. **Popolare** con i **dati** nel formato specifico
4. **Configurare** le computazioni per l'esecuzione (es. dimensione dei dati)
5. **Eseguire** la **chiamata** della funzione di libreria che avvia la computazione sulla GPU
6. **Recuperare** i **risultati** dalla device memory
7. Se necessario, **(ri)convertire** i dati nel formato specifico o nativo dell'applicazione
8. **Rilasciare** le risorse CUDA allocate per la data libreria

cuBLAS…

Basic Linear Algebra Subprograms

Libreria cuBLAS

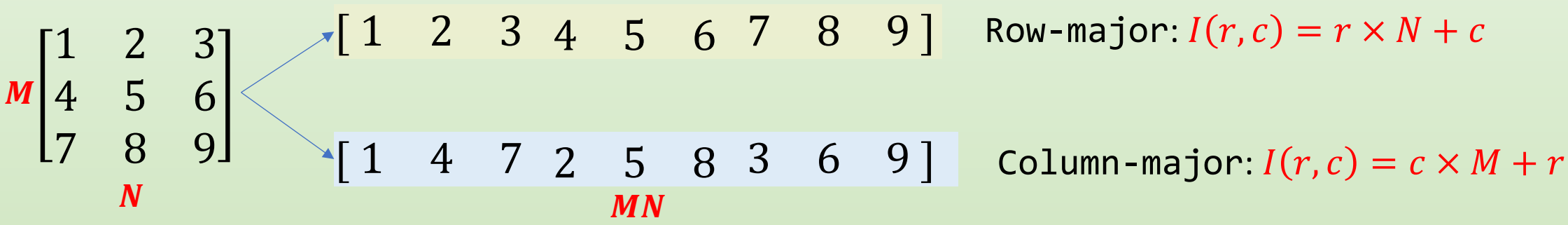
- ✓ **Basic Linear Algebra Subprograms** (**BLAS**)
- ✓ All operations are done on **dense** cuBLAS **vectors** or **matrices**
- ✓ Like BLAS, **cuBLAS** subroutines are split into **multiple classes** based on the data types on which they operate.
  - **cuBLAS Level 1** contains **vector-only** operations like vector addition.
  - **cuBLAS Level 2** contains **matrix-vector** operations like matrix-vector multiplication.
  - **cuBLAS Level 3** contains **matrix-matrix** operations like matrix-multiplication.
- ✓ Building blocks for higher level numerical linear algebra such as in **LAPACK**
- ✓ Both **BLAS** and **LAPACK** were developed in **Fortran**
- ✓ Because the original **BLAS** library was written in **fortran**, it historically uses **column-major** array storage and one-based indexing
- ✓ The new **CuBLAS interface** is entirely **asynchronous** in nature

cuBLAS lib

- ✓ **Livello 1**: operazioni basate su **vettori** e **scalari** (BLAS1)
  - ES: $y[i] = \alpha * x[i] + y[i], \qquad i = 1, \ldots, n$
- ✓ **Livello 2**: operazioni basate su **vettori** e **matrici** (BLAS2)
  - ES: $y = \alpha A x + \beta y, \qquad \text{dove: } \alpha, \beta \text{ scalari, } A \text{ matrice, } x, y \text{ vettori}$
- ✓ **Livello 3**: operazioni basate su **matrici** e **matrici** (BLAS3)
  - ES: $C = \alpha A B + \beta C, \qquad \text{dove: } \alpha, \beta \text{ scalari, } A, B, C \text{ matrici}$
- ✓ Precisione
  - **single**: `real & complex`
  - **double**: `real & complex` (not all functions)
- ✓ Non è richiesta **nessun kernel**, **no shared memory**, etc.

Column-major order in cuBLAS

- ✓ Per ragioni di **compatibilità** le cuBLAS library scelgono di usare la memorizzazione **column-major**
- ✓ Se ho una matrice di M righe e N colonne, l'entry di posto (r, c) è data da:



- ✓ **Compliazione** e **linking** di libreria dinamica **cublas**:

```
#include "cublas_v2.h"
```

```
$ nvcc -arch=sm_20 -lcublas kernel.cu -o kernel
```

cuBLAS API naming convention

- ✓ **`cublas<t>operation`** where `<t>` is one of:
  - ✓ `S` for **float** parameters
  - ✓ `D` for **double** parameters
  - ✓ `C` for **complex float** parameters
  - ✓ `Z` for **complex double** parameters
- ✓ **Example**: For the **axpy** operation ($\mathbf{y[i]} = \alpha x[i] + y[i]$)
  - Available functions are:
  - **`cublasSaxpy, cublasDaxpy, cublasCaxpy, cublasZaxpy`**

cuBLAS streams API and thread safety

- ✓ **cuBLAS** permits the use of cuda **streams** (`cudaStream_t`) for **increasing resource usage** and introduce other levels of parallelism
- ✓ **cuBLAS** is a thread **safe library**, meaning that the cuBLAS host functions can be called from multiple threads safely

- ✓ `cublasSetStream():`
  - **Sets** the **stream** to be used by cuBLAS
  - Parameters:
    - **cuBLAS handle** to set the stream
    - **cuda stream** to use
- ✓ `cublasGetStream():`
  - **Gets** the **stream** being used by cuBLAS
  - Parameters:
    - **cuBLAS handle** to get the stream
    - pointer to **cuda stream**

Operare con cuBLAS

1. **Create** a cuBLAS handle using **`cublasCreateHandle`**
2. **Allocate** device memory for input and output using **`cudaMalloc`**
3. **Populate** the device memory with inputs using **`cublasSetVector`** and **`cublasSetMatrix`**
4. **Execute** (e.g.) **`cublasSgemv`** library call to offload a matrix-vector multiplication operation to the GPU
5. **Retrieve** results from device memory using **`cublasGetVector`** and **`cublasGetMatrix`**
6. **Release** CUDA and cuBLAS resources using **`cudaFree`** and **`cublasDestroy`**

Creation and destruction of a cuBLAS environment

- ✓ cuBLAS needs an **execution context** to store internal resources
- ✓ After cuBLAS executions are finished the **context** needs to be **destroyed** to **free** resources
- ✓ **Creating** and **destroying** contexts should be considered an **expensive** operation
- ✓ Recommended that each thread and each device have its own context
- ✓ To create a context in a specific device call **`cudaSetDevice`** before the creation

- **`cublasHandle_t`**
  - **Type** used by cuBLAS to store **contexts**
- **`cublasCreate(cublasHandle_t* handle)`**
  - **Creates** a cuBLAS **context**
  - Parameters:
    - **Pointer** to **cuBLAS handle** to create
- **`cublasDestroy(cublasHandle_t handle)`**
  - **Destroys** a cuBLAS **context**
  - Parameters:
    - **cuBLAS handle** with the context to destroy
- **`cublasStatus_t`**
  - **Type** used by cuBLAS for reporting **errors**
  - Every cuBLAS returns an error **status**

CUBLAS APIs: Set & Get

✓ Copies ***n* elements** from ***cpumem*** on the CPU memory to a vector ***gpumem*** on the GPU memory

```
cublasSetVector(int n, int elemSize, const void *cpumem, int incx, void *gpumem, int incy)
```

- ***cpumem***: pointer of an array to send data
- ***gpumem***: pointer of an array to receive data
- ***elemSize***: Each element size in byte
- ***incx***, ***incy***: Storage spacing size

✓ Copies ***n* elements** from ***gpumem*** on the GPU memory to a vector ***cpumem*** on the CPU memory

```
cublasGetVector(int n, int elemSize, const void *gpumem, int incx, void *cpumem, int incy)
```

- ***cpumem***: pointer of an array to send data
- ***gpumem***: pointer of an array to receive data
- ***elemSize***: Each element size in byte
- ***incx***, ***incy***: Storage spacing size

cuBLAS memory API

`cublasGetVector()`:

- Parameters:
  - **Number** of elements to transfer
  - **Element size** in bytes
  - **Source** device pointer
  - **Stride** to use for the **source** vector
  - **Destination** host pointer
  - **Stride** to use for the **destination** vector

```
cublasGetVector(2, sizeof(float), dA, 2, y, 1);
```

Matrix A =

$A_{1,1}$	$A_{1,2}$
$A_{2,1}$	$A_{2,2}$

$A_{1,1}$	$A_{2,1}$	$A_{1,2}$	$A_{2,2}$
$A[0]$	$A[1]$	$A[2]$	$A[3]$

Column major order

$A_{1,1}$	$A_{2,1}$	$A_{1,2}$	$A_{2,2}$
$y[0]$		$y[1]$	

Transferred data to y

CUBLAS APIs: Set & Get

- ✓ This function copies a tile of **rows x cols** elements from a matrix **A** in **host** memory space to a matrix **B** in **GPU** memory space (instead of usual **cudaMalloc** and **cudaMemcpy** APIs)

```
cublasSetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)
```

  - ➢ It is assumed that each element requires storage of **elemSize** bytes and that both matrices are stored in **column-major format**, with the leading dimension of the source matrix **A** and destination matrix **B** given in **lda** and **ldb**, respectively
  - ➢ **lda, ldb**: total number of rows in A and B

- ✓ This function has the same functionality as **cublasGetMatrix()**, with the exception that the data transfer is done **asynchronously** (with respect to the host) using the given **stream** parameter

```
cublasGetMatrixAsync(int rows, int cols, int elemSize, const void *A, int lda, void *B,
                                                        int ldb, cudaStream_t stream)
```

Nvidia docs: http://http://docs.nvidia.com/cuda/cublas/index.html#cublas-helper-function-reference

Esempi: copia di matrici e vettori

- ✓ Transferring a **single column** with length **M** of a column-major matrix **A** to a vector ***d*V** could be done using:

```
cublasSetVector(M, sizeof(float), A, 1, dV, 1);
```

- ✓ To transfer a **single row** of that matrix **A** to a vector ***d*V** on the device:

```
cublasSetVector(M, sizeof(float), A, M, dV, 1);
```

- ✓ This function copies **N** elements from **A** to ***d*V**, skipping **M** elements in **A** at a time. Because **A** is column-major, this command would copy the first row of A to the device. Copying **row i** would be implemented as:

```
cublasSetVector(N, sizeof(float), A + i, M, dV, 1);
```

- ✓ If you were given a dense **two-dimensional column-major matrix A** of single-precision floating-point values on the host with M rows and N columns, you could use:

```
cublasSetMatrix(M, N, sizeof(float), A, M, dA, M);
```

- ✓ For **device-to-device transfer**: copies n elements of the vector **x** into the vector **y**, for $i = 1, \dots, n$

$$\boldsymbol{y[j] = x[k]}, \qquad \boldsymbol{k = 1 + (i - 1) * incx} \qquad \boldsymbol{j = 1 + (i - 1) * incy}$$

```
cublasScopy(handle, n, x, incx, y, incy);
```

Operazioni matriciali

$$\text{op}(A) = \begin{cases} A & \text{if transa == CUBLAS\_OP\_N} \\ A^T & \text{if transa == CUBLAS\_OP\_T} \\ A^H & \text{if transa == CUBLAS\_OP\_C} \end{cases}$$

Value	Meaning
`CUBLAS_OP_N`	the non-transpose operation is selected
`CUBLAS_OP_T`	the transpose operation is selected
`CUBLAS_OP_C`	the conjugate transpose operation is selected

CUBLAS(2) APIs: mat-vect multiplication

Prototipo

```
cublasStatus_t cublasSgemv(cublasHandle_t handle,
                           cublasOperation_t trans,
                           int m, int n,
                           const float *alpha,
                           const float *A,
                           int lda,
                           const float *x,
                           int incx,
                           const float *beta,
                           float *y,
                           int incy)
```

handle to the cuBLAS library context

operation op(A) that is non- or (conj.) transpose

number of rows, columns

<type> scalar used for multiplication

<type> array of dimension lda x n and lda x m

leading dimension used to store matrix A

<type> vector with n elements

stride between consecutive elements of x

<type> scalar used for multiplication (if beta!=0)

<type> vector with m elements

stride between consecutive elements of y

Read more at: http://docs.nvidia.com/cuda/cublas/index.html#ixzz496uNjCKh

CUBLAS(3) APIs: mat-mat multiplication

Prototipo

```
cublasStatus_t cublasSgemm(cublasHandle_t handle, cublasOperation_t transa,
                           cublasOperation_t transb,
                           int m, int n, int k,
                           const float *alpha,
                           const float *A, int lda,
                           const float *B, int ldb,
                           const float *beta,
                           float *C, int ldc)
```

Param.	Memory	In/out	Meaning
handle		input	handle to the cuBLAS library context.
transa		input	operation op(A) that is non- or (conj.) transpose.
transb		input	operation op(B) that is non- or (conj.) transpose.
m		input	number of rows of matrix op(A) and C.
n		input	number of columns of matrix op(B) and C.
k		input	number of columns of op(A) and rows of op(B).
alpha	host/dev	input	<type> scalar used for multiplication.
A	device	input	<type> array of dimensions lda x k with lda>=max(1,m) if transa == CUBLAS_OP_N and lda x m with lda>=max(1,k) otherwise.
lda		input	leading dimension of two-dimensional array used to store the matrix A.
B	device	input	<type> array of dimension ldb x n with ldb>=max(1,k) if transa == CUBLAS_OP_N and ldb x k with ldb>=max(1,n) otherwise.
ldb		input	leading dimension of two-dimensional array used to store matrix B.
beta	host/devi	input	<type> scalar used for multiplication. If beta==0, C does not have to be a valid input.
C	device	in/out	<type> array of dimensions ldc x n with ldc>=max(1,m).
ldc		input	leading dimension of a two-dimensional array used to store the matrix C

Read more at: http://docs.nvidia.com/cuda/cublas/index.html#ixzz496tehfHp

CUBLAS: esempio

inizializza cuBLAS

Alloca spazio su GPU

trasferisce il vettore e matrice sulla GPU

d_y = a*d_A*d_x + b*d_y

trasferisce dalla GPU alla CPU

```
#define m 1 << 14
#define n 1 << 14
#define p 1 << 14
CHECK_CUBLAS(cublasCreate(&handle));
CHECK(cudaMalloc((void **)&d_A, m * n * sizeof(float)));
CHECK(cudaMalloc((void **)&d_x, n * sizeof(float)));
CHECK(cudaMalloc((void **)&d_y, m * sizeof(float)));
CHECK_CUBLAS(cublasSetVector(n, sizeof(float), x, 1, d_x, 1));
CHECK_CUBLAS(cublasSetMatrix(m, n, sizeof(float), A, m, d_A, m));
CHECK_CUBLAS(cublasSgemv(handle, CUBLAS_OP_N, m, n, &a, d_A, m,
                                                    d_x, 1, &b, d_y, 1));
CHECK_CUBLAS(cublasGetVector(m, sizeof(float), d_y, 1, y, 1));
CHECK(cudaFree(d_A));
CHECK(cudaFree(d_x));
CHECK(cudaFree(d_y));
CHECK_CUBLAS(cublasDestroy(handle));
```

Copiare una sottomatrice

Estrazione di sottomatrice da una matrice [**100 x 100**] di dimensioni: **rows [10:99], cols [90:99]**

allocazione sul device di **rowsB * colsB (90 x 10)** float

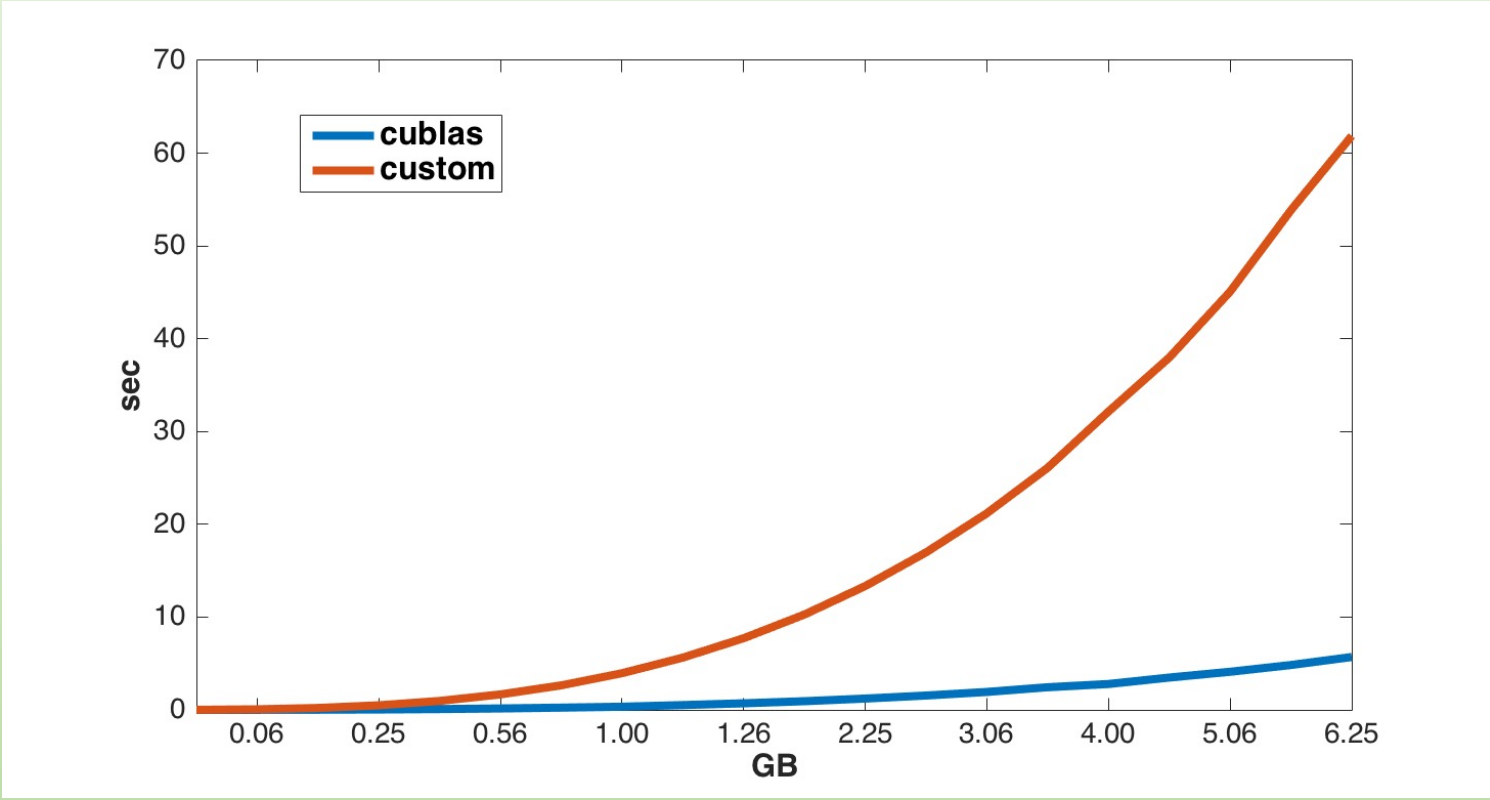
salta 10 righe e 90 colonne

```
// The matrix in host memory
int rowsA = 100;
int colsA = 100;
float *A = (float*) malloc(rowsA * colsA * sizeof(float ));
...
// The sub-matrix that should be copied to the device.
// The minimum index is INCLUSIVE
// The maximum index is EXCLUSIVE
int minRowA = 10;
int maxRowA = 100;
int minColA = 90;
int maxColA = 100;
int rowsB = maxRowA-minRowA; // = 90
int colsB = maxColA-minColA; // = 10

// Allocate the device matrix
float *dB = nullptr;
cudaMalloc(&dB, rowsB * colsB * sizeof(float));

// pointers and sizes
float *subA = A + (minRowA + minColA * rowsA);
cublasSetMatrix(rowsB, colsB, sizeof(float), subA, rowsA, dB, rowsB);
//                 90    10                      100         90
```

Risultati

Risultati su prodotti tra matrici quadrate su Tesla K40c:

- Dimensione delle matrici: da **0.06 GByte** a **6.25 Gbyte** (size `N x N = 20480 x 20480`)
- **cublas**: funzione di libreria cuBLAS
- **custom**: kernel sviluppato a lezione



Conjugate Gradient

✓The **Conjugate Gradient** method is an **iterative method** to compute an approximation for the **solution** of the linear algebraic **system** $\mathrm{Ax} = b$

✓Assumptions:

- Let A be a $n * n$ **symmetric** and **positive definite** matrix (all the eigenvalues are positive)
- Let b be a n-dimensional vector

- x_0 = *vettore iniziale a caso*
- $r_0 = b - A * x_0$
- $p_0 = r_0$
- For $k = 0,1, \dots, n$
  1. $\alpha_k = \frac{p_k^T * r_k}{p_k^T * A * p_k}$
  2. $x_{k+1} = x_k + \alpha_k p_k$
  3. $r_{k+1} = b - A * x_{k+1}$
  4. $\beta_k = \frac{p_k^T * A * r_{k+1}}{p_k^T * A * p_k}$
  5. $p_{k+1} = r_{k+1} - \beta_k p_k$
  6. $k = k + 1$
- return x_{k+1}

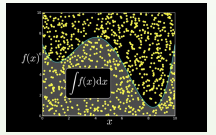
→ lab9

Soluzione approssimata $Ax = b$ basata su gradiente coniugato

cuRAND…

Pseudorandom sequence

Libreria cuRAND



La libreria cuRAND fornisce semplici ed efficienti generatori di numeri

- ✓ **Pseudorandom**: **sequenza** di numeri che **soddisfa** molte delle **proprietà statistiche** di una vera sequenza casuale… es. la seq. generata da un algoritmo deterministico:

$$A_{n+1} = (Z * A_n + I) MOD (M)$$

- ✓ **Quasirandom**: sequenza di punti **n-dimensionali uniformemente generati** da un algoritmo deterministico (non si formano cluster nello spazio di generazione)

- ✓ **cuRAND** si compone di due parti: una libreria per l'host (**curand.h**) e una per il device (**curand_kernel.h**)

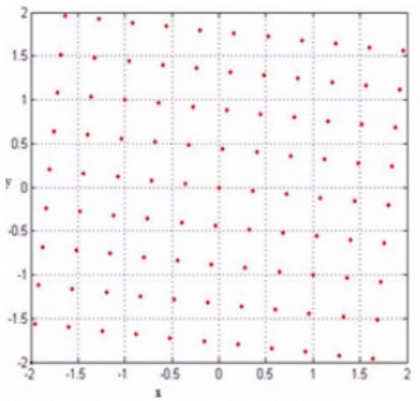


Fig-1a:Quasi-random distribution

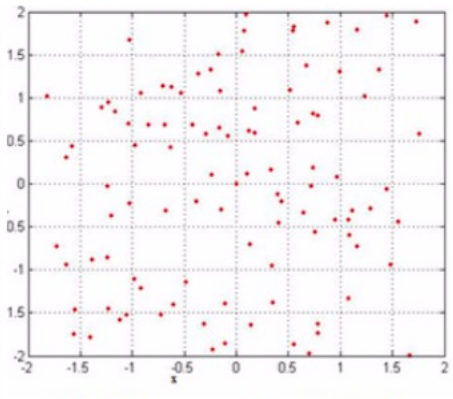


Fig-1b:Pseudo-random distribution

Host API: passi (da documentazione)

1. Create a **new generator** of the desired type (see Generator Types)

   with **`curandCreateGenerator`**( )
2. Set the generator options (see Generator Options) - for example

   use **`curandSetPseudoRandomGeneratorSeed`**() to set the seed
3. Allocate memory on the device with **`cudaMalloc`**( )
4. Generate random numbers with **`curandGenerate`**() or another function
5. Use the results
6. Generate more random numbers with more calls to **`curandGenerate`**()
7. Clean up with **`curandDestroyGenerator`**()

Host API

- ✓ **`curandCreateGenerator(&g, GEN_TYPE)`**
  - ✓ `GEN_TYPE = CURAND_RNG_PSEUDO_DEFAULT, CURAND_RNG_PSEUDO_XORWOW`
  - ✓ Doesn't particularly matter, differences are small
- ✓ **`curandSetRandomGeneratorSeed(g, SEED)`**
  - ✓ Again, `SEED` doesn't matter too much, just pick one (ex.: time(NULL))
- ✓ **`curandGenerate_____(…)`**
  - ✓ Depends on distribution
  - ✓ Ex.: **`curandGenerateUniform`**`(g, src, n)`, **`curandGenerateNormal`**`(g, src, n, mean, stddev)`
- ✓ **`curandDestroyGenerator(g)`**

Host: uso generatore CURAND…

- ✓ This program uses the **host CURAND API** to generate 100 pseudorandom floats

include API host

new generator

```
#include <curand.h>


int main(int argc, char *argv[]) {
  size_t n = 100;
  size_t i;
  curandGenerator_t gen;
  float *devData, *hostData;


  /* Allocate n floats on host */
  hostData = (float *) calloc(n, sizeof(float));


  /* Allocate n floats on device */
  CUDA_CALL(cudaMalloc((void ** )&devData, n*sizeof(float)));


  /* Create pseudo-random number generator */
  CURAND_CALL(curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT));
. . .
```

Host: uso generatore CURAND

2. definizione seme innesco

generazione dei numeri secondo modello di distribuzione

pulizia…

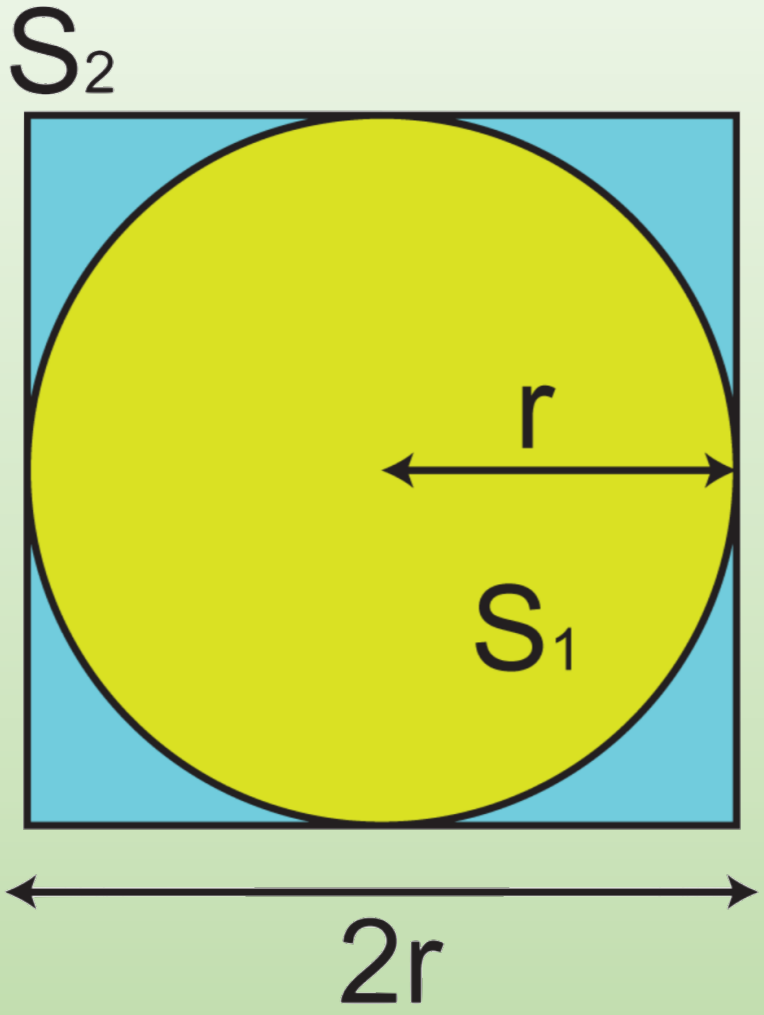
```
/* Set seed */
CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen, 1234ULL));


/* Generate n floats on device */
CURAND_CALL(curandGenerateUniform(gen, devData, n));


/* Copy device memory to host */
CUDA_CALL(cudaMemcpy(hostData, devData, n * sizeof(float),
                                          cudaMemcpyDeviceToHost));


/* Show result */
for (i = 0; i < n; i++)
   printf("%1.4f ", hostData[i]);


/* Cleanup */
CURAND_CALL(curandDestroyGenerator(gen));
CUDA_CALL(cudaFree(devData));
```

```
$ nvcc myCurandApp.c -lcurand -o myCurandApp
```

Device API: generazione su device

1. **Pre-allocating** a set of **cuRAND state objects** in device memory for **each thread** to manage its RNG's state
2. Optionally, **pre-allocating device memory** to store the **random values** generated by cuRAND (if they are intended to be copied back to the host or must be persisted for later kernels)
3. **Initializing** the **state** of all cuRAND state objects in **device memory** with a kernel call
4. **Executing** a **kernel** that calls a cuRAND device function (for example., **`curand_uniform`**) and generates random values using the pre-allocated cuRAND state objects
5. Optionally, **transferring random values back** to the host if device memory was pre-allocated in step 2 for retrieving random values

Calcolo di π con il metodo Monte Carlo



Circle: $S_1 = \pi r^2$

Square: $S_2 = (2r)^2 = 4r^2$

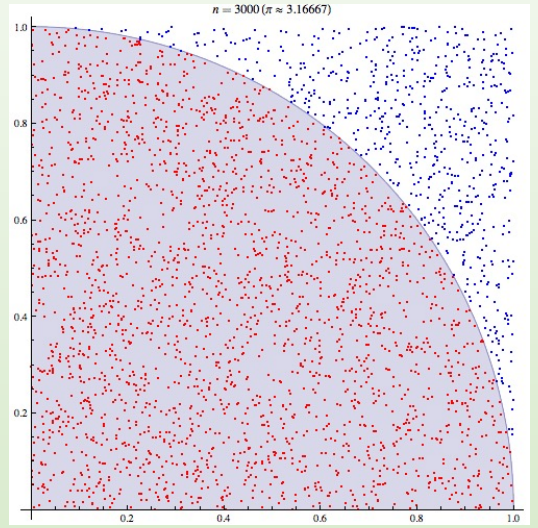
$$\frac{S_1}{S_2} = \frac{\pi r^2}{4r^2} \rightarrow \pi = \frac{4S_1}{S_2}$$

Stima su CPU

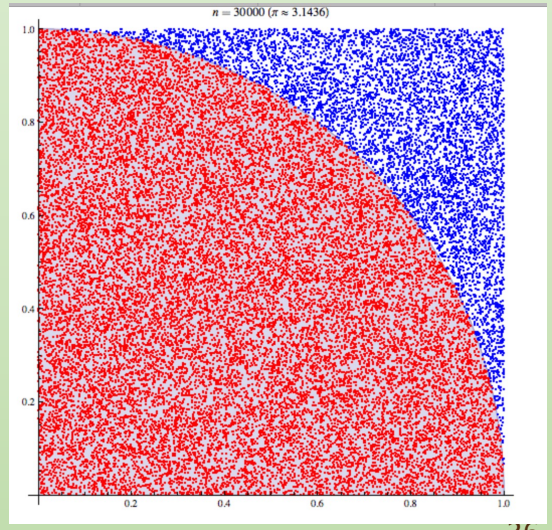
Tecnica numerica usando i numeri casuali:

- ✓ Generare un **elevato** numero di **punti random** all'interno del **quadrato** ([0,1] x [0,1])
- ✓ **L'area** del **cerchio** può essere ottenuta come il **rapporto** tra i numero di **punti all'interno** del cerchio e il numero **totale** di punti
- ✓ Esempio di codice in CPU:

```
float pi_mc_CPU(long trials) {
        float x, y;
        long points_in_circle = 0;
        for(long i = 0; i < trials; i++) {
                x = rand() / (float) RAND_MAX;
                y = rand() / (float) RAND_MAX;
                points_in_circle += (x*x + y*y <= 1.0f);
        }
        return 4.0f * points_in_circle/trials;
}
```



$\pi \approx 3.1415926535897932$



cuRAND device API

✓ Passi per la generazione di **numeri casuali nella memoria GPU**

API device per l'inizializzazione del «contesto» di generazione casuale

```
__device__ void curand_init ( unsigned long long seed,
                              unsigned long long sequence,
                              unsigned long long offset,
                              curandState_t *state)
```

Allocare spazio di memoria sul device: un **curandState** per ogni thread

```
// host
curandState *devStates;
cudaMalloc( (void **)&devStates, B*T*sizeof(curandState));
```

Inizializzare lo stato con un "seed"

Generare la sequenza di numeri random

```
// kernel
int tid = threadIdx.x + blockDim.x*blockIdx.x;
curand_init(tid, 0, 0, &states[tid]);
x = curand_uniform(&states[tid]);
y = curand_uniform(&states[tid]);
```

Esempio: MonteCarlo

include API dev

num trial x thread

num threads legato a numero prove indipend.

inizializz. dello stato

kernel Monte Carlo

```
#include <curand_kernel.h>
#define TRIALS_PER_THREAD 10000
#define BLOCKS  264
#define THREADS 264
#define PI 3.1415926535 // known value of pi

float host[BLOCKS * THREADS];
float *dev;

// GPU procedure
curandState *devStates;
cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
cudaMalloc((void **) &devStates, BLOCKS * THREADS * sizeof(curandState));
cudaEventRecord(start);
pi_mc_GPU<<<BLOCKS, THREADS>>>(dev, devStates);
```

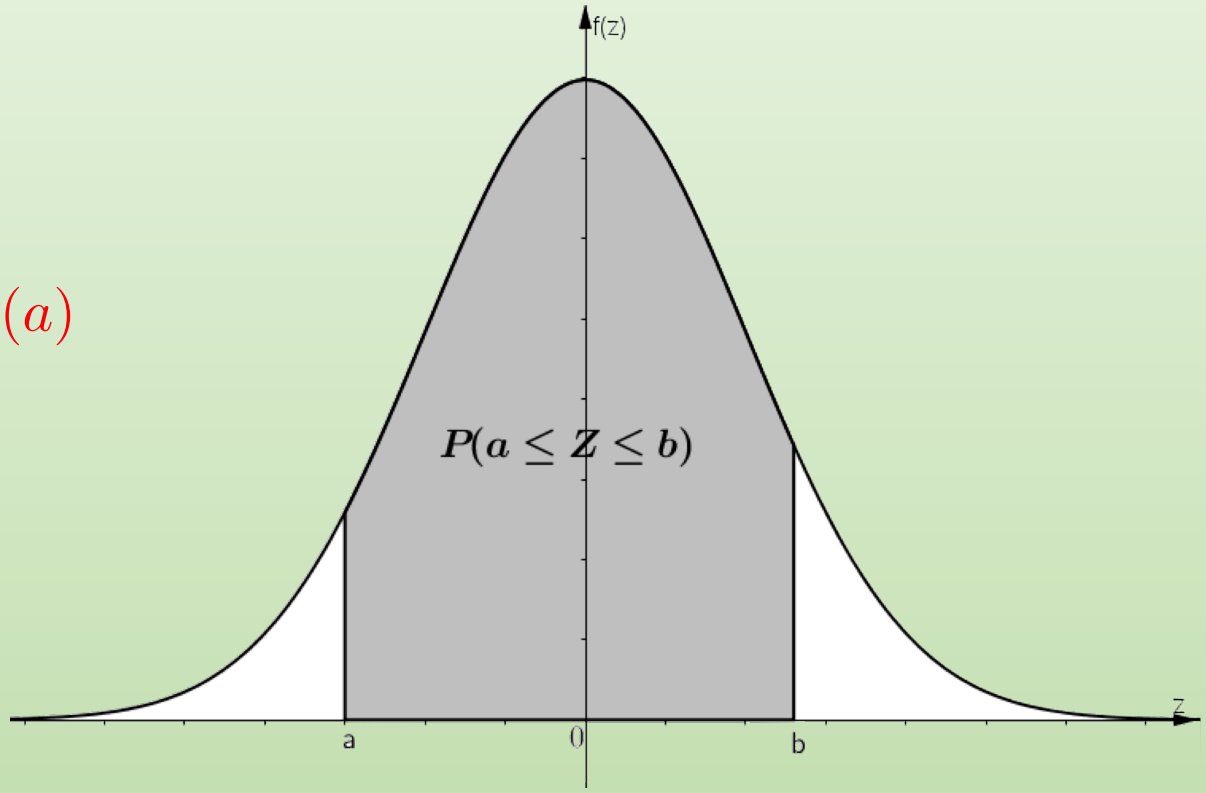
Esempio: kernel MonteCarlo

```
__global__ void pi_mc_GPU(float *estimate, curandState *states) {
        unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
        int points_in_circle = 0;
        curand_init(tid, 0, 0, &states[tid]);
        for (int i = 0; i < TRIALS_PER_THREAD; i++) {
                float x = curand_uniform(&states[tid]);
                float y = curand_uniform(&states[tid]);
                points_in_circle += (x * x + y * y <= 1.0f);
        }
        estimate[tid] = 4.0f * points_in_circle / (float) TRIAL_x_TH;
}
```

Calcolo area campana di Gauss

$$P(a \leq X \leq b) = \int_a^b f(x)\, dx = \int_a^b \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}\, dx$$

$$P(a \leq X \leq b) = F(b) - F(a)$$

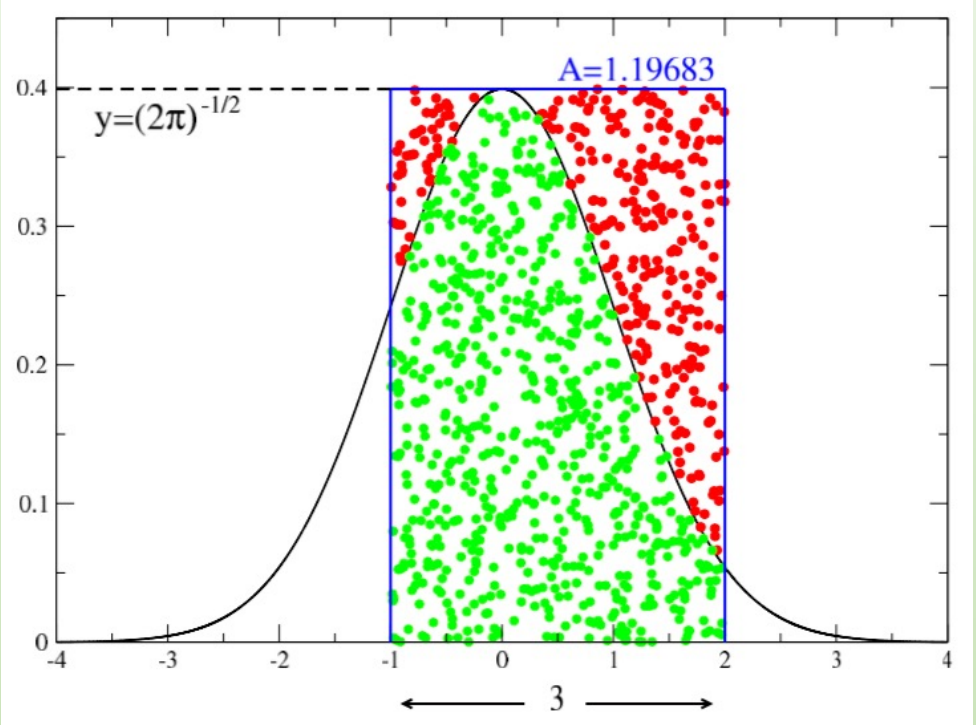


https://www.geogebra.org/m/tz6dzuRh

Calcolo di un'area con Monte Carlo

- ✓ Supponiamo di avere una funzione *g* definita su $[a, b]$ a valori in $[c, d], c, d \geq 0$, e di volerne calcolare l'integrale

$$P = \int_{-1}^{2} \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} dx$$



Metodo Monte Carlo:

1. poniamo $s = 0,\ A = (b - a) * (d - c)$
2. si estrare un numero casuale uniforme $u \in [a, b]$
3. si estrare un numero casuale uniforme $v \in [c, d]$
4. Se $v < f(u)$ allora si pone $s = s + 1$
5. si itera la procedura n volte
6. Il valore dell'integrale è circa: $\int_a^g f(x)dx \approx \frac{A \cdot s}{n}$

→ lab9

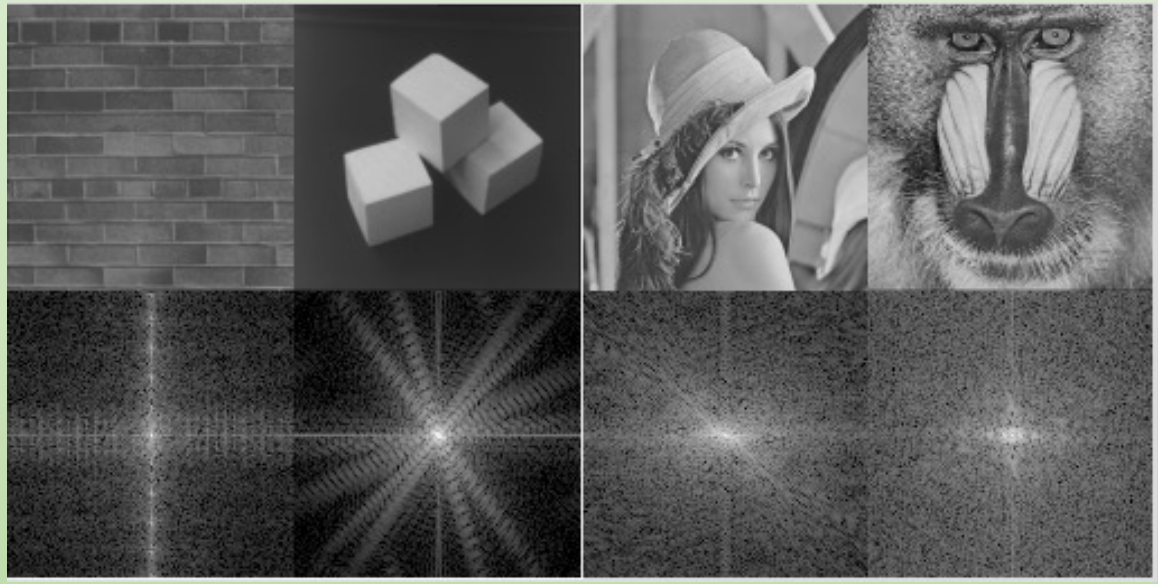
Calcolo campana di Gauss con metodo MC

cuFFT…

Fast Fourier Transform algorithms

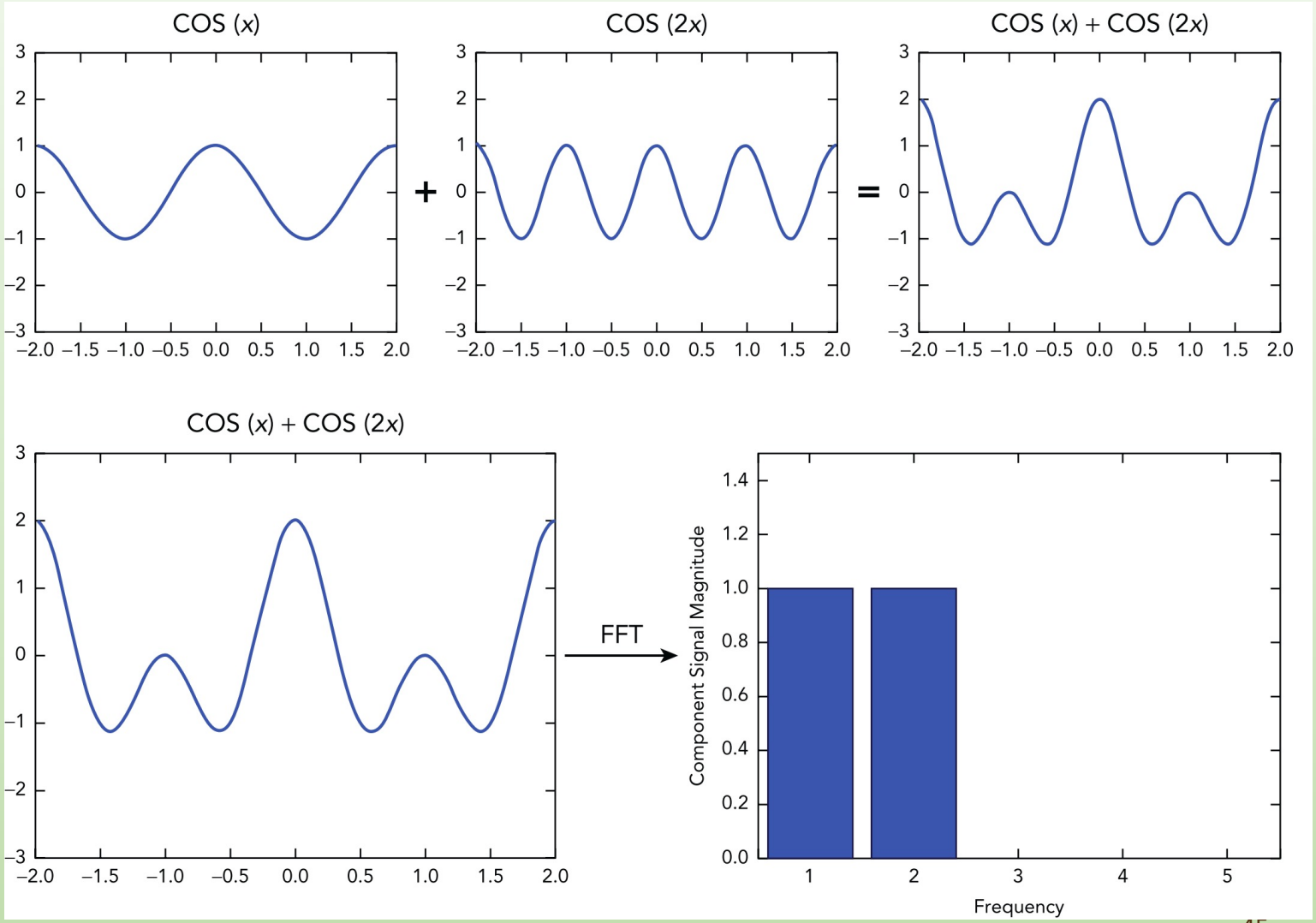
Fourier Transform of images

- ✓Another concept with lots of application, scalability, and parallelizability: Fourier Transformation
- ✓Commonly used in physics, signal processing, etc.
- ✓Oftentimes needs to be real-time (makes great use of GPU)



Fourier Transform on signals

Sum of two signals to form the signal cos(x)+cos(2x) and its decomposition by FFT into frequencies of 1.0 and 2.0



Trasformata di Fourier (DTFT)

- ✓ Dal processo di **campionamento**, con freq espressa in [cicli/campione]:

$$X(\nu) = \sum_{n=-\infty}^{+\infty} x(n) e^{-i2\pi\nu n}, \qquad 0 \leq \nu < 1$$

$$x(n) = \int_0^1 X(\nu) e^{i2\pi\nu n} d\nu$$

- ✓ Dalla **risposta in frequenza**, con freq espressa in [rad/campione]:

$$X(e^{i\omega}) = \sum_{n=-\infty}^{+\infty} x(n) e^{-i\omega n}, \qquad 0 \leq \omega < 2\pi$$

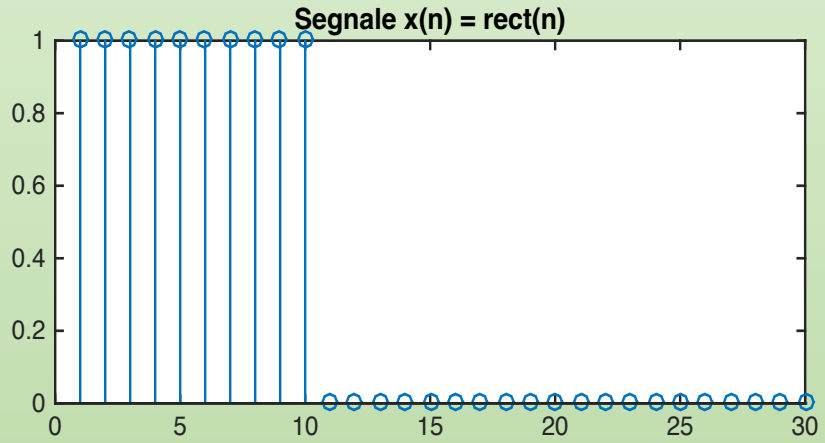
$$x(n) = \frac{1}{2\pi} \int_0^{2\pi} X(e^{i\omega}) e^{i\omega n} d\omega$$

DTFT della funzione rettangolo discreto

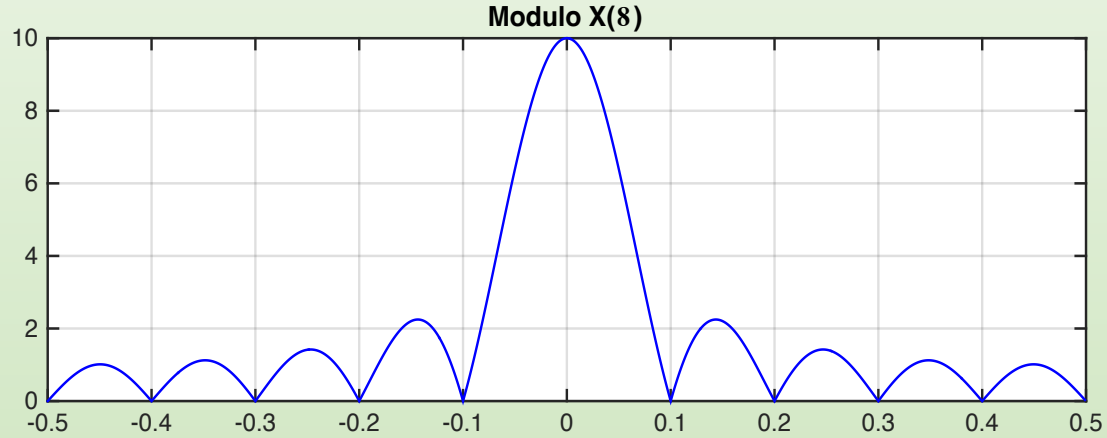
Tempo

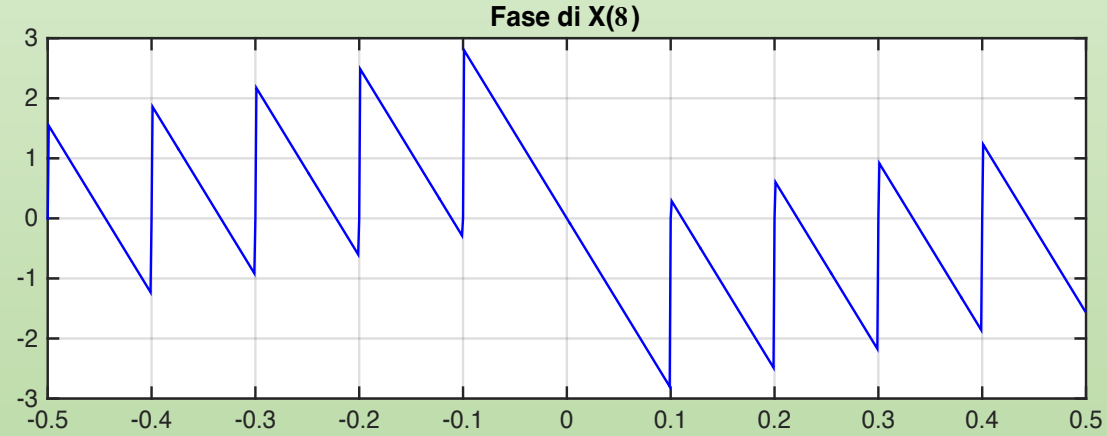
$$x(n) = \mathrm{rect}(n)$$

$$X(\nu) = \frac{\sin(\pi N \nu)}{\sin(\pi \nu)}$$



Frequenza





Trasformata di Fourier discreta - DFT

- ✓ Se *x(n)* è un segnale periodico di periodo *NT*, trasformata e antitrasformata di Fourier sono

$$X(kF) = T \sum_{n=0}^{N-1} x(nT) e^{-j2\pi \frac{nk}{N}}, \quad f \in \{0, F, \ldots, F(N-1)\}, \quad F = \frac{1}{NT}$$

$$x(nT) = F \sum_{k=0}^{N-1} X(kF) e^{j2\pi \frac{nk}{N}}, \quad x = [x(0), x(T), x(2T) \ldots, x((N-1)T)\}$$

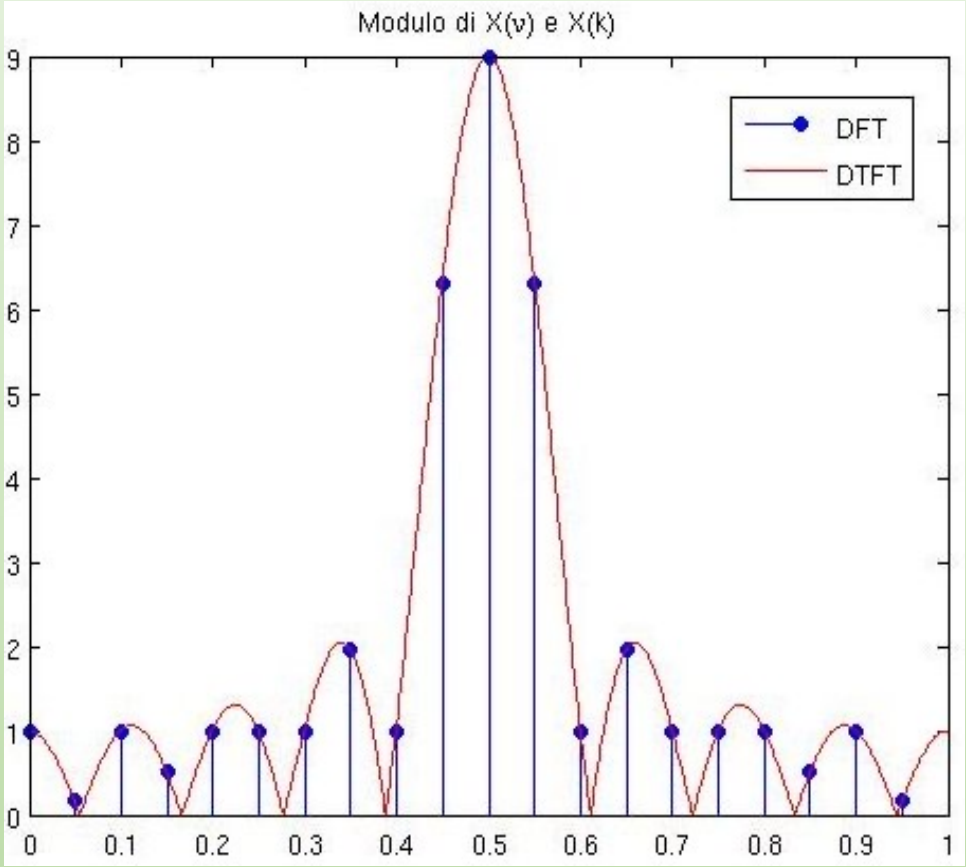
- ✓ Ponendo *T = 1* si ha rispettivamente:

$$X(k+1) = \sum_{n=0}^{N-1} x(n+1) e^{-j2\pi \frac{nk}{N}}, \qquad x(n+1) = \frac{1}{N} \sum_{k=0}^{N-1} X(k+1) e^{j2\pi \frac{nk}{N}}$$

Se *T ≠ 1*, basta porre : $X(k+1) = X(kF)/T$

DFT impulso rettangolare

- ✓ Impulso rettangolare: $x(n) = rect(n), \qquad X(\nu) = \frac{\sin(\pi N \nu)}{\sin(\pi \nu)}$
- ✓ Inviluppo: $\mathrm{sinc}(\nu)$



cuFFT lib

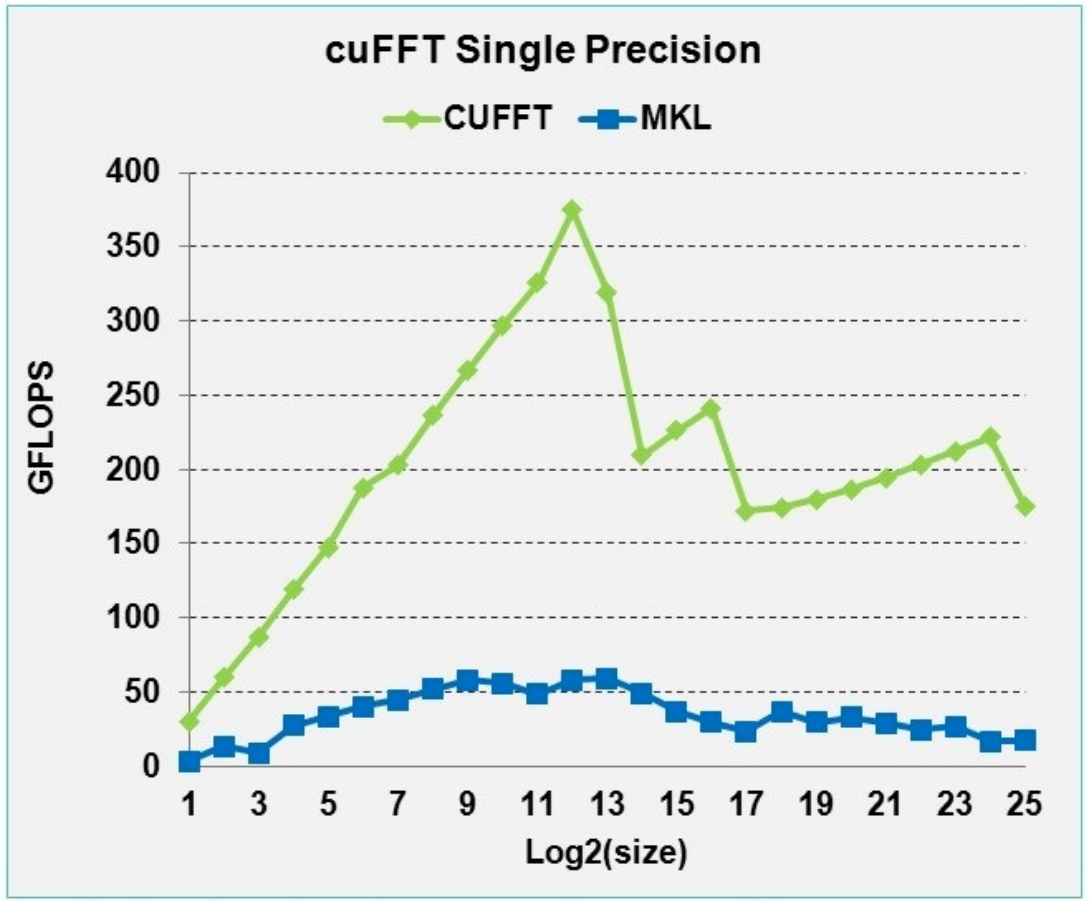
- ✓ The **cuFFT** library provides an optimized, CUDA-based implementation of the fast Fourier transform (**FFT**)
- ✓ An **FFT** is a **transformation** in signal processing that converts a signal from the **time domain** to the **frequency domain**
- ✓ An **inverse FFT** does the opposite
- ✓ FFT receives as input a **sequence** of **samples** taken **from** a **signal** at regular time intervals
- ✓ It uses those samples to **generate** a **set** of **component frequencies** that are combined to create the signal that **generated** the **input samples**

Lib. features

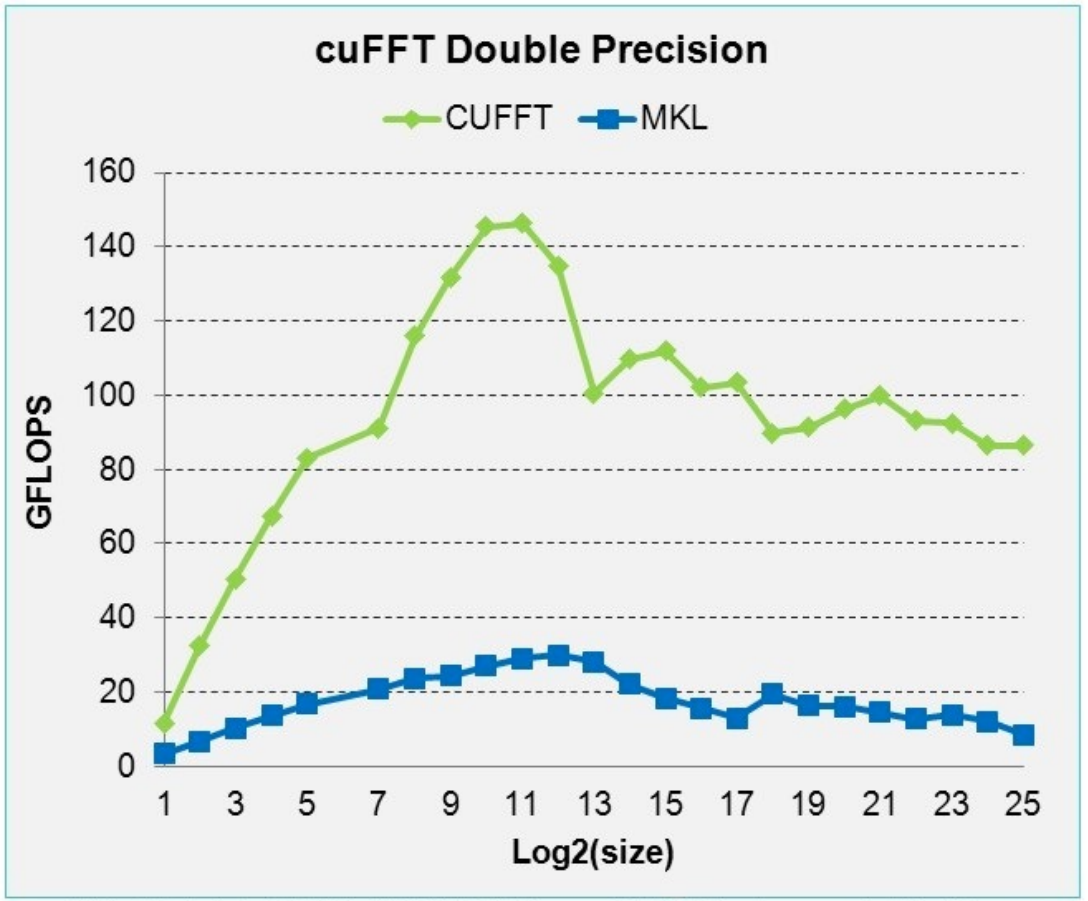
- ✓Supports 1D, 2D, or 3D Fourier Transforms
- ✓1D transforms can have up to 128 million elements
- ✓Based on Cooley-Tukey and Bluestein FFT algorithms
- ✓Similar API to FFTW, if familiar
- ✓Streamed asynchronous execution
- ✓Supports both in-place and out-of-place transforms
- ✓Supports real, complex, float, double data

cuFFT vs MKL

Tesla M2090 vs Xeon x5680 3.3GHz

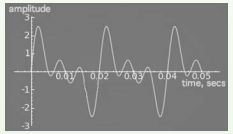


- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz



- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
- Performance may vary based on OS version and motherboard configuration

Libreria cuFFT



- ✓ **Half-precision** (16-bit floating point), **single-precision** (32-bit) and **double-precision** (64-bit) input and output
- ✓ **Types** supported are:
  - ✓ **C2C** - Complex input to complex output
  - ✓ **R2C** - Real input to complex output
  - ✓ **C2R** - Symmetric complex input to real output
- ✓ Execution of multiple **1D**, **2D** and **3D** transforms simultaneously
- ✓ **higher performance** than single transforms. In-place and out-of-place transforms
- ✓ Arbitrary intra- and inter-dimension element **strides** (strided layout)
- ✓ **FFTW** compatible data layout
- ✓ Execution of transforms across multiple GPUs
- ✓ **Streamed** execution, enabling **asynchronous** computation and data movement

Read more at: http://docs.nvidia.com/cuda/cufft/index.html#ixzz4hV8LC4Sc

cuFFt types

- ✓ `cufftHandle`

  Handle type to store CUFFT plans

- ✓ `cufftResult`

  Return values, like CUFFT_SUCCESS, CUFFT_INVALID_PLAN, CUFFT_ALLOC_FAILED,

  CUFFT_INVALID_TYPE, etc

- ✓ Types:

  `cufftReal, cufftDoubleReal, cufftComplex, cufftDoubleComplex`

- ✓ Transform types:

- ✓ **R2C**: real to complex

  C2R: Complex to real

  C2C: complex to complex

  D2Z: double to double complex

  Z2D: double complex to double

  Z2Z: double complex to double complex

- ✓ Plans:

  `cufftPlan1d(), cufftPlan2d(), cufftPlan3d(), cufftPlanMany()`

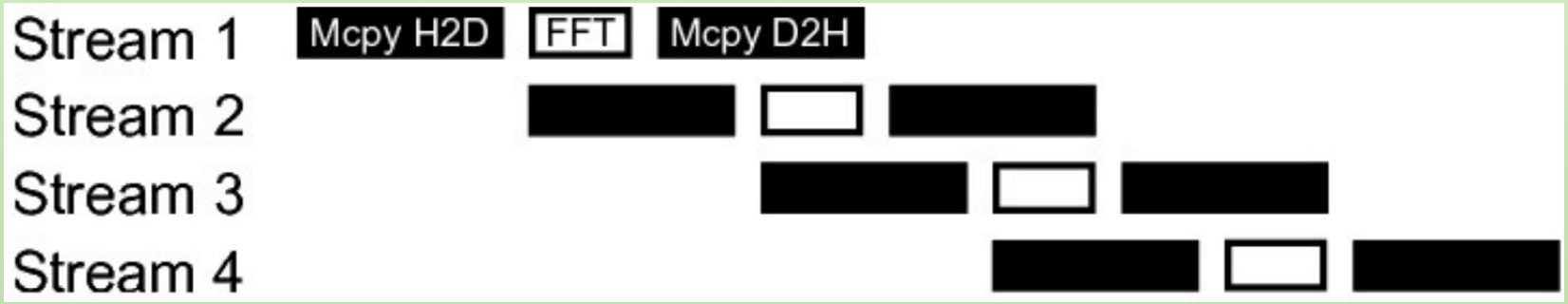
cuFFT plans

- ✓ **Configuration** of the cuFFT library is done with FFT **plans** (the terminology that cuFFT uses to refer to its handles)
- ✓ A **plan** defines a **single transform operation** to be performed
- ✓ cuFFT uses plans to **derive** the **internal memory allocations**, **transfers**, and kernel **launches** that must occur to perform the requested transformation

```
cufftResult cufftPlan1d(cufftHandle *plan, int nx, cufftType type, int batch);

cufftResult cufftPlan2d(cufftHandle *plan, int nx, int ny, cufftType type);

cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz, cufftType type);
```

Batched executions

- ✓ Schematic of CUDA stream use to **overlap data transfer** from the host to the device (H2D) with **cuFFT computation**
- ✓ While the batch streamed approach provides little benefit with NVLink, we see a **performance gain of 30%** when tuned for the optimal number of streams on a PCIe-based system
- ✓ Black "**Mcpy H2D**" ("D2H") boxes indicate copies of the data from the host to the device (or vice versa),
- ✓ while white "**FFT**" boxes correspond to the computation of the FFTs on the data available in that stream.



Schema

1. Create and configure a **cuFFT plan**
2. Allocate device memory to store the input samples and output frequencies from cuFFT using **`cudaMalloc`**
3. Populate that device memory with the input signal samples using **`cudaMemcpy`**
4. Execute the plan using a **`cufftExec`*** function
5. Retrieve the result from device memory using **`cudaMemcpy`**
6. Release CUDA and cuFFT resources using cudaFree and **`cufftDestroy`**

✓ Compliazione e linking di lib dinamica:

```
#include "cufft.h"
```

```
$ nvcc -arch=sm_20 -lcufft kernel.cu -o kernel
```

Conversione a valori complessi

tipo complesso della libreria cuFFT

```
/*
 * Convert a real-valued vector r of length N to a complex-valued
  * vector.
 */
void real_to_complex(float *r, cufftComplex **complx, int N) {

   (*complx) = (cufftComplex *) malloc(sizeof(cufftComplex) * N);

   for (int  i = 0; i < N; i++) {
      (*complx)[i].x = r[i];
      (*complx)[i].y = 0;
   }
}
```

Esempio

```
int main(int argc, char **argv) {

int i;
int N = 2048;
float *samples;
cufftHandle plan = 0;
cufftComplex *dComplexSamples, *complexSamples, *complexFreq;

// Input Generation
generate_fake_samples(N, &samples);
real_to_complex(samples, &complexSamples, N);
complexFreq = (cufftComplex *) malloc(sizeof(cufftComplex) * N);
printf("Initial Samples:\n");
```

Esempio

```
// Setup the cuFFT plan
cufftPlan1d(&plan, N, CUFFT_C2C, 1);

// Allocate device memory
cudaMalloc((void ** )&dComplexSamples, sizeof(cufftComplex) * N);

// Transfer inputs into device memory
cudaMemcpy(dComplexSamples, complexSamples, sizeof(cufftComplex) * N,
                                                cudaMemcpyHostToDevice);

// Execute a complex-to-complex 1D FFT
cufftExecC2C(plan, dComplexSamples, dComplexSamples, CUFFT_FORWARD);

// Retrieve the results into host memory
cudaMemcpy(complexFreq, dComplexSamples, sizeof(cufftComplex) * N,
                                                cudaMemcpyDeviceToHost);

cudaFree(dComplexSamples);
cufftDestroy(plan);
```

Python CUDA libs

CUDA libraries with high level Python

CUDA Libs

- **NumPy -> CuPy, PyTorch, TensorFlow**
  - Array computing
  - Mature due to deep learning boom
  - Also useful for other domains
  - Obvious fit for GPUs
- **Pandas -> cuDF**
  - Tabular computing
  - New development
  - Parsing, joins, groupbys
  - **Not** an obvious fit for GPUs
- **Scikit-Learn -> cuML**
  - Traditional machine learning
  - Somewhere in between

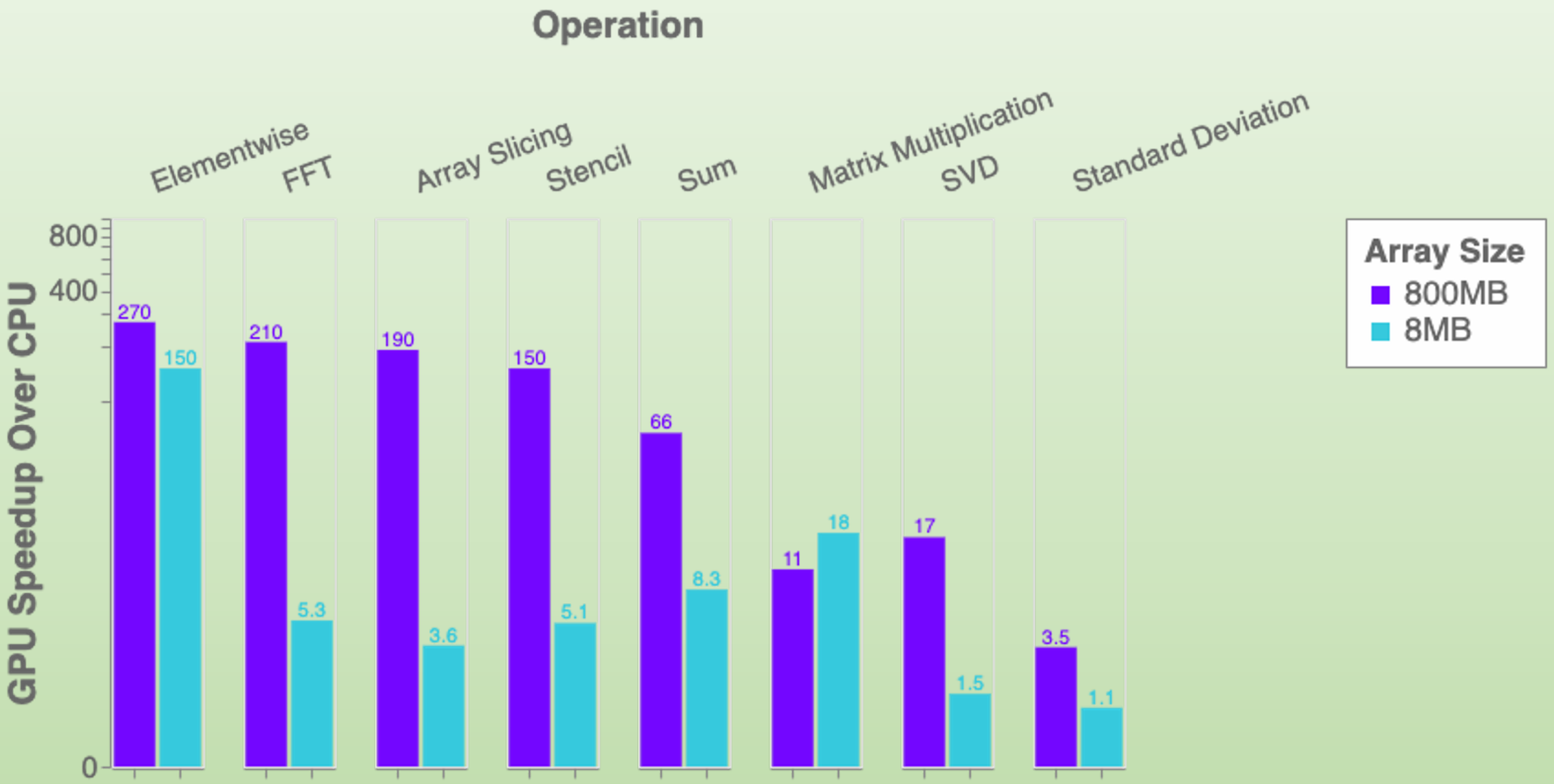
cupy

```
[1]: import cupy, numpy

[2]: c = numpy.random.random((20000, 20000))
     g = cupy.random.random((20000, 20000))

[3]: %timeit float(c.dot(c).sum())  # Ten CPU cores

     40.1 s ± 313 ms per loop (mean ± std. dev. of 7 runs,

[4]: %timeit float(g.dot(g).sum())  # One V100

     2.6 s ± 3.58 ms per loop (mean ± std. dev. of 7 runs,
```

Pandas

```
1]: import pandas, cudf

2]: %time len(pandas.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))

CPU times: user 25.9 s, sys: 3.26 s, total: 29.2 s
Wall time: 29.2 s

2]: 12748986

3]: %time len(cudf.read_csv('data/nyc/yellow_tripdata_2015-01.csv'))

CPU times: user 1.59 s, sys: 372 ms, total: 1.96 s
Wall time: 2.12 s

3]: 12748986

4]: !du -hs data/nyc/yellow_tripdata_2015-01.csv

1.9G    data/nyc/yellow_tripdata_2015-01.csv
```

CuPy Performance Comparison



Riferimenti bibliografici

Testi generali:

1. J. Cheng, M. Grossman, T. Mckercher, Professional Cuda C Programming, Wrox Pr Inc. (1^ ed), 2014 (cap 6)

2. D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors, Morgan Kaufmann (3^ ed), 2013 (cap 20)

NVIDIA docs:

1. CUDA Programming: *http://docs.nvidia.com/cuda/cuda-c-programming-guide/*

2. CUDA C Best Practices Guide: *http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/*

3. Specifiche tecniche (pag wiki): https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications