

# GPU Computing

Laurea Magistrale in Informatica - AA 2019/20

Docente **G. Grossi**

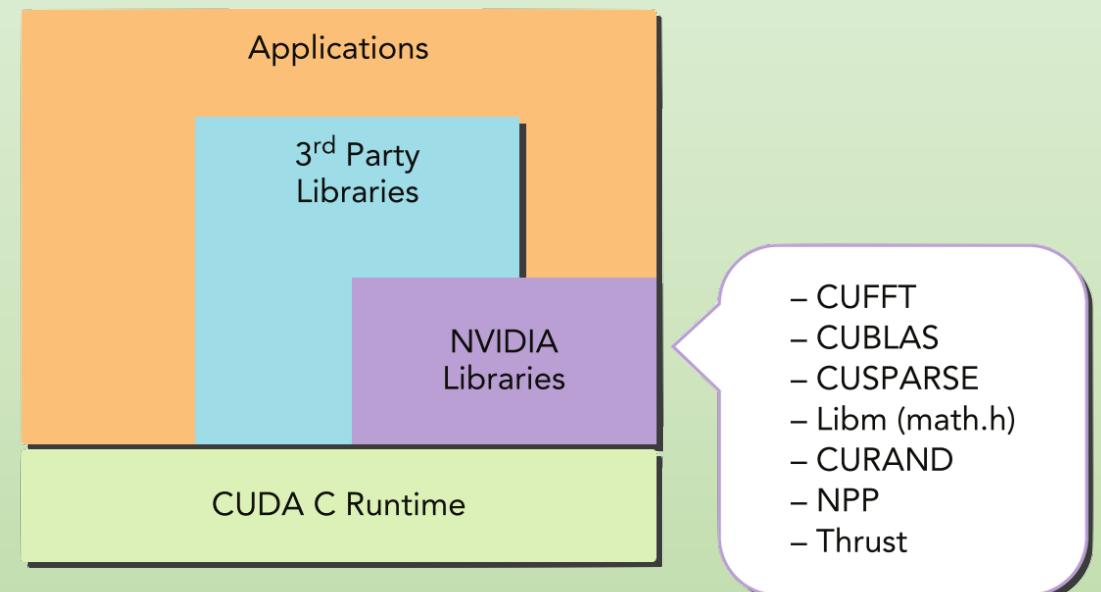
Lezione 8 – Librerie CUDA

# Sommario

- ✓ Librerie CUDA
- ✓ cuBLAS
- ✓ cuRAND
- ✓ cuFFT
- ✓ demo

# 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** nello stesso dominio
- ✓ 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
- ✓ Esistono reference guide ottimamente redatte per illustrarne uso e potenzialità



# Esempi di librerie CUDA

Esempi di librerie sviluppate in CUDA a supporto di determinati domini, scaricabili su  
[developer.nvidia.com](http://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  
(Se gli input non sono già in un formato supportato dalla libreria, convertirli al **formato specifico** di uso della libreria, 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, meaning even functions that return values do it in such a way as the value may not be available unless the programmer specifically waits for the asynchronous GPU operation to complete.



## Error status

All cuBLAS library function calls return the error status **cublasStatus\_t**

## cuBLAS context

The application must initialize the **handle** to the cuBLAS library context by calling the **cublasCreate()** function. Then, it is explicitly passed to every subsequent library function call. Once the application finishes using the library, it must call the function **cublasDestory()** to release the resources associated with the cuBLAS library context

## Thread Safety

The library is thread safe and its functions can be called from **multiple host threads**, even with the same handle... it is not recommended that multiple threads share the same CUBLAS handle

## Library support

For example, on Linux, to compile a small application using cuBLAS, against the **dynamic library**, the following command can be used:

```
nvcc myCublasApp.c -lcublas -o myCublasApp
```

Read more at: <http://docs.nvidia.com/cuda/cublas/index.html#ixzz4hR2vT2yG>

# cuBLAS lib

Implementazione delle librerie BLAS (Basic Linear Algebra Subprograms)

✓ **Livello 1:** operazioni basate su **vettori e scalari** (BLAS1)

- ES:  $y[i] = \alpha * x[i] + y[i], \quad i = 1, \dots, n$

✓ **Livello 2:** operazioni basate su **vettori e matrici** (BLAS2)

- ES:  $y = \alpha Ax + \beta y, \quad$  dove:  $\alpha, \beta$  scalari,  $A$  matrice,  $x, y$  vettori

✓ **Livello 3:** operazioni basate su **matrici e matrici** (BLAS3)

- ES:  $C = \alpha AB + \beta C, \quad$  dove:  $\alpha, \beta$  scalari,  $A, B, C$  matrici

✓ Precisione

- single: real & complex
- double: real & complex (not all functions)

✓ Non è richiesta nessuna invocazione di kernel, no uso della shared memory, etc

# Operare con cuBLAS

Passi:

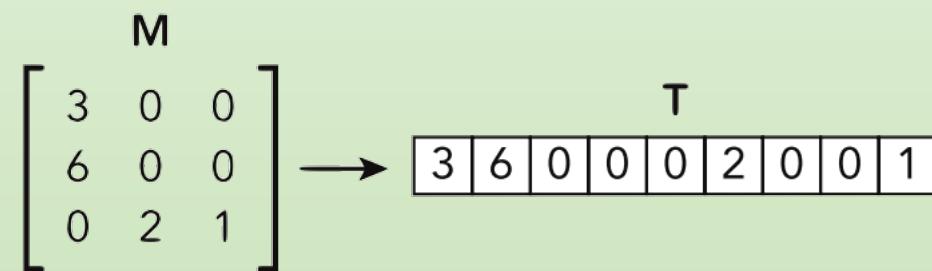
1. Create a cuBLAS handle using **cublasCreateHandle**
2. Allocate device memory for inputs and outputs using **cudaMalloc**
3. Populate the allocated device memory with inputs using **cublasSetVector** and **cublasSetMatrix**
4. Execute (e.g.) **cublasSgemv** library call to offload a matrix-vector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

# 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, una entry di posto **(m,n)** nei due casi si ottiene:

Row-Major:  $f(m, n) = m \times N + n$

Column-Major:  $f(m, n) = n \times M + m$



- ✓ Compliazione e linking di lib dinamica:

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

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

# cuBLAS datatypes

Every cuBLAS API function comes in four different data types

- single precision floating point (S)
- double precision floating point (D)
- complex single precision floating point (C)
- complex double precision floating point (Z)

<type>	<t>	Meaning
<code>float</code>	's' or 'S'	real single-precision
<code>double</code>	'd' or 'D'	real double-precision
<code>cuComplex</code>	'c' or 'C'	complex single-precision
<code>cuDoubleComplex</code>	'z' or 'Z'	complex double-precision

- **SGEMM:** Single Precision General Matrix-Matrix Multiply
- **DGEMM:** Double Precision General Matrix-Matrix Multiply
- **CGEMM:** Complex Single Precision General Matrix-Matrix Multiply
- **ZGEMM:** Complex Double Precision General Matrix-Matrix Multiply

# Operazioni

<b>Value</b>	<b>Meaning</b>
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 APIs: Set & Get

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

Prototipo-> `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 (= 1 for a vector)

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

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

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

- This function copies a tile of *rows* × *cols* elements from a matrix A in host memory space to a matrix B in GPU memory space. 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

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

- 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 CUDA™ stream parameter

# Esempi: copia di matrici e vettori

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

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

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

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

- ✓ This function copies N elements from A to dV, 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);
```

# 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)
```

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

Param.	Memory	In/out	Meaning
handle		input	handle to the cuBLAS library context
trans		input	operation op(A) that is non- or (conj.) transpose
m		input	number of rows of matrix A
n		input	number of columns of matrix A
alpha	host/dev	input	<type> scalar used for multiplication
A	device	input	<type> array of dimension lda x n with lda >= max(1,m) if transa==CUBLAS_OP_N and lda x m with lda >= max(1,n) otherwise
lda		input	leading dimension of two-dimensional array used to store matrix A
x	device	input	<type> vector with n elements if transa==CUBLAS_OP_N and m elements otherwise
incx		input	stride between consecutive elements of x
beta	host/dev	input	<type> scalar used for multiplication, if beta==0 then y does not have to be a valid input
y	device	in/out	<type> vector with m elements if transa==CUBLAS_OP_N and n elements otherwise
incy		input	stride between consecutive elements of y

# 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/dev	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

trasferisce  
il vettore

$d_y =$   
 $a * d_A * d_x + b * d_y$

trasferisce  
dalla GPU

```
#define m 1 << 14
#define n 1 << 14
#define p 1 << 14

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(cublasCreate(&handle));
CHECK_CUBLAS(cublasSetVector(n, sizeof(float), x, 1, d_x, 1));
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

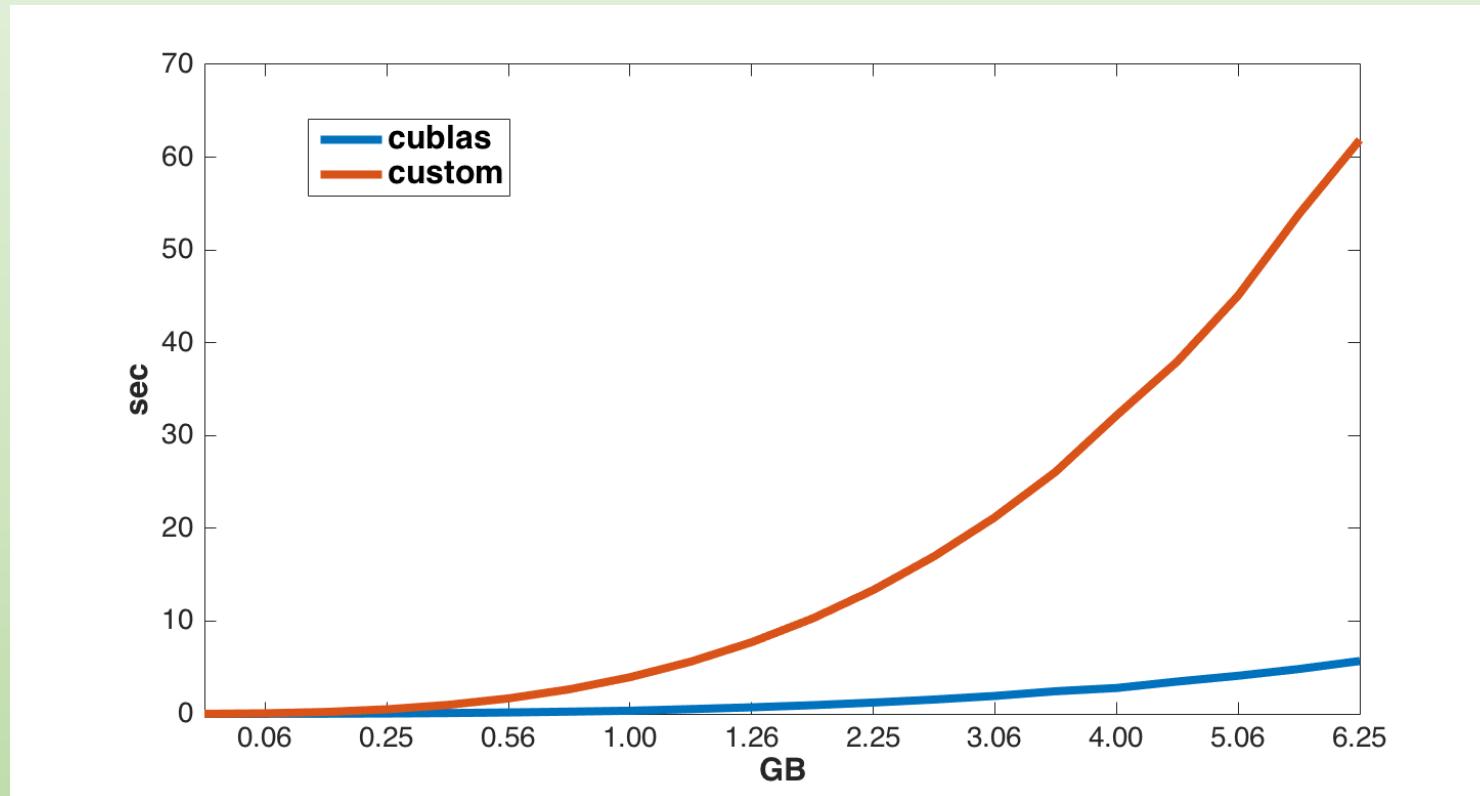
// Allocate the device matrix
float *dB = nullptr;
cudaMalloc(&dB, rowsB * colsB * sizeof(float));

// pointers and sizes
float *subA = A + (minRowA + minColA * rowsA);
cublasSetMatrix(rowsB, colsB, sizeof(float), subA, rowsA, dB, rowsB);
//          90      10          100        90    19
```

# Risultati

Risultati su prodotti tra matrici quadrate su Tesla K40c:

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



# Esercitazione

**Effettuare moltiplicazioni tra matrici MQDB:**

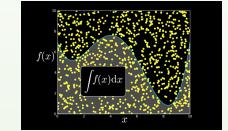
Utilizzare cuBLAS e verificare efficienza:

- Indicizzare sottomatrici
- Effettuare un loop su host senza kernel
- Provare approccio complessivo o blocco-a-blocco

# 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:

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

- ✓ **Quasirandom:** sequenza di punti **n-dimensional**i **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**)

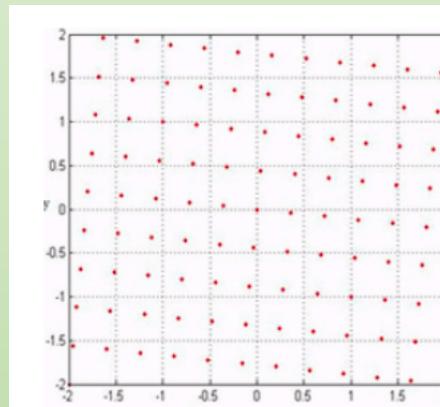


Fig-1a:Quasi-random distribution

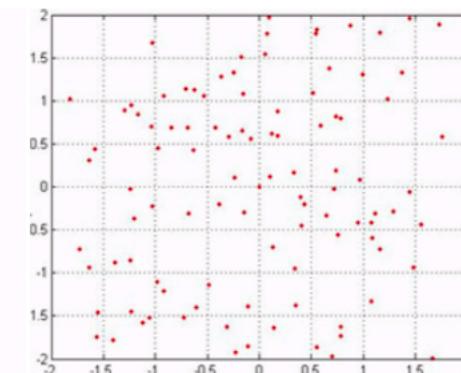


Fig-1b:Pseudo-random distribution

# 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 generation function
5. Use the results
6. If desired, 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

```
#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));
```

new generator

# 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
```

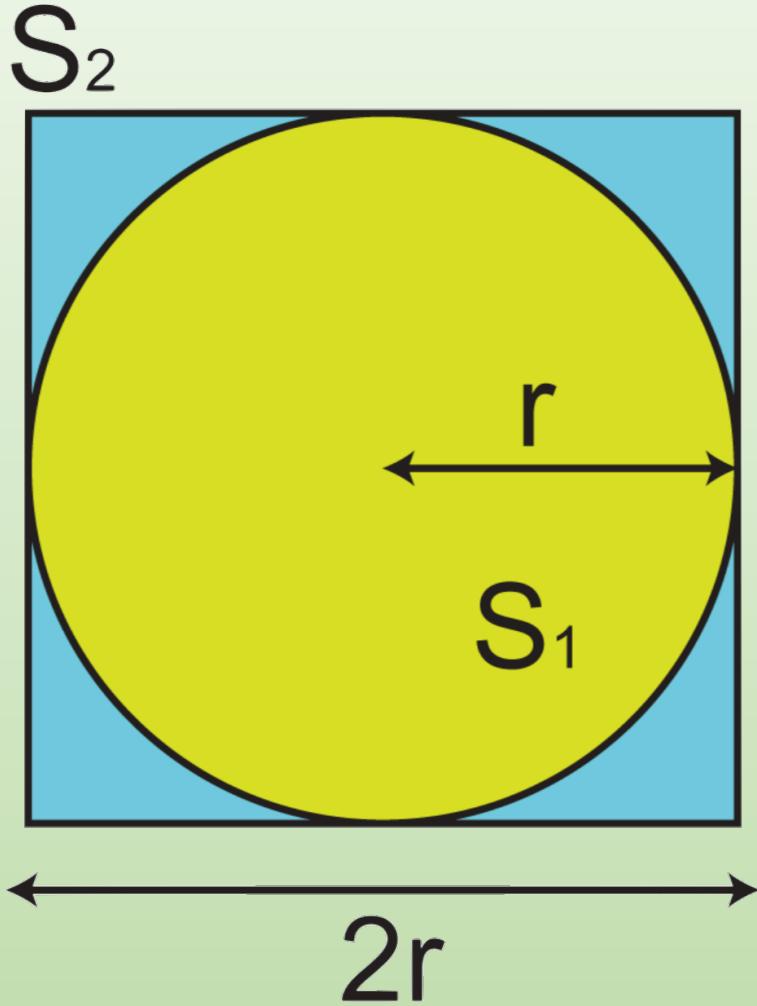
# Host API: svantaggi

- ✓ **curandGenerate()** launches asynchronously
  - ✓ Much faster than serial CPU generation
- ✓ However, we still need to copy data to GPU
  - `src` in `curandGenerate()` is host pointer, not device pointer!
  - Introduces some undesired overhead
  - Might need more memory than we can pass in one go
- ✓ Solution: cuRAND device API

# 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 CUDA kernel call
4. Executing a CUDA 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 $\pi$ con il metodo Monte Carlo



$$\text{Circle: } S_1 = \pi r^2$$

$$\text{Square: } S_2 = (2r)^2 = 4r^2$$

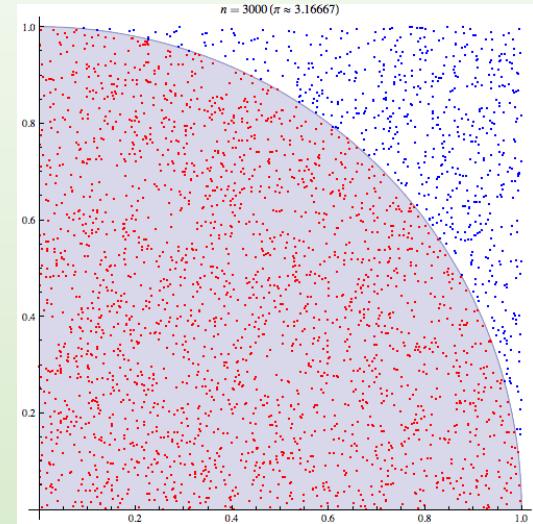
$$\frac{S_1}{S_2} = \frac{\pi r^2}{4r^2} \rightarrow \pi = \frac{4S_1}{S_2}$$

# Stima su CPU

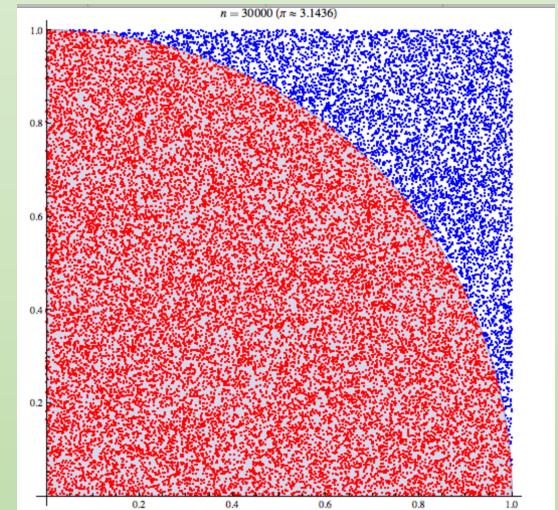
Tecnica numerica usando i numeri casuali:

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

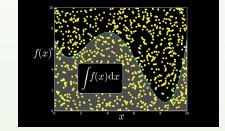
```
float pi_mc_CPU(long trials) {  
    float x, y;  
    long points_in_circle=0;  
    for(long i = 0; i < trials; i++) {  
        x = rand() / (float) RAND_MAX;  
        y = rand() / (float) RAND_MAX;  
        points_in_circle += (x*x + y*y <= 1.0f);  
    }  
    return 4.0f * points_in_circle/trials;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



$$\pi \approx 3.1415926535897932$$



# cuRAND device API



**Genera numeri casuali nella memoria**

**GPU** Passi per la generazione:

1. Includere **curand\_kernel.h**
2. Allocare spazio di memoria sul device
3. Inizializzare lo stato con un “seed”
4. Generare la sequenza di numeri random

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

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

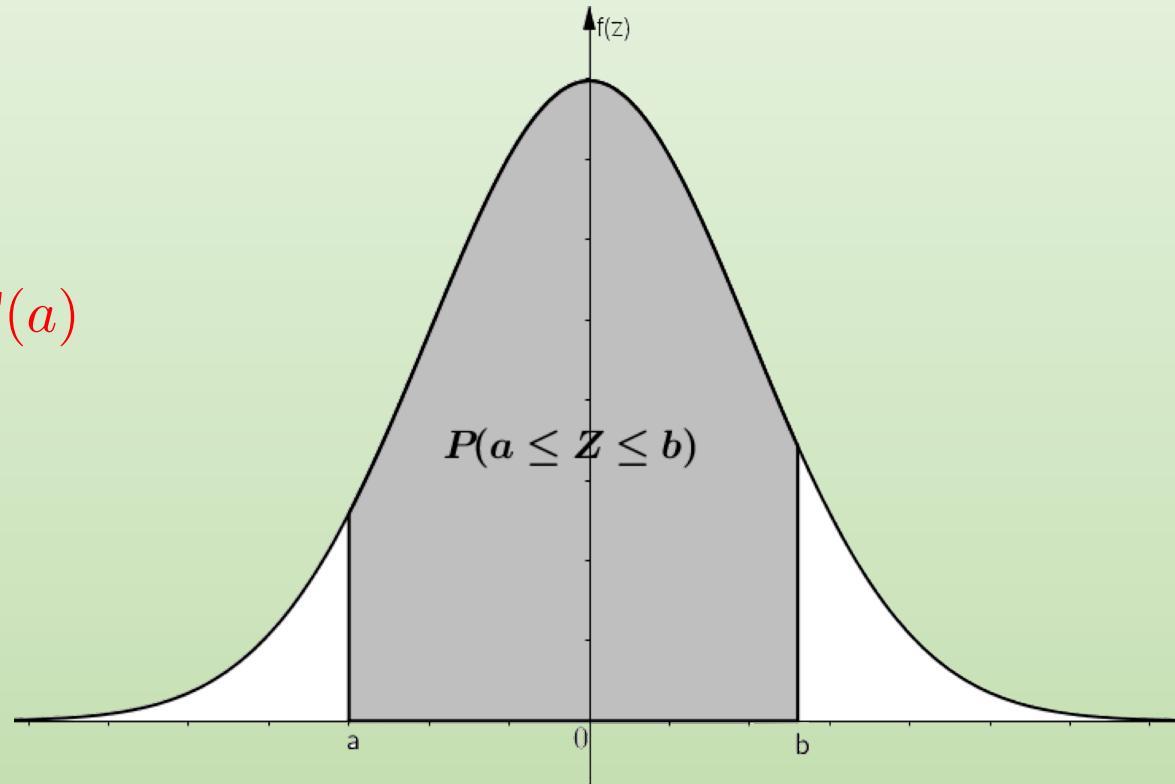
# Esempio: kernel MonteCarlo

```
__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

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

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



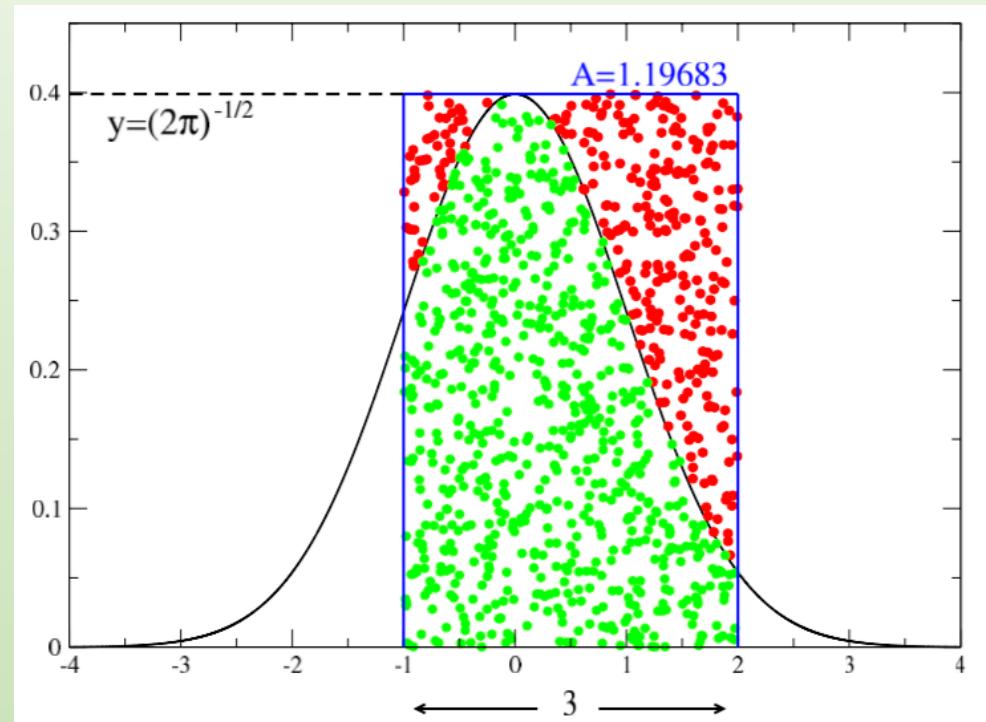
# Calcolo di un'area con Monte Carlo

- ✓ 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:  $\int_a^b f(x)dx \approx \frac{A \cdot s}{n}$



# Esercitazione

## MONTE CARLO SU GPU:

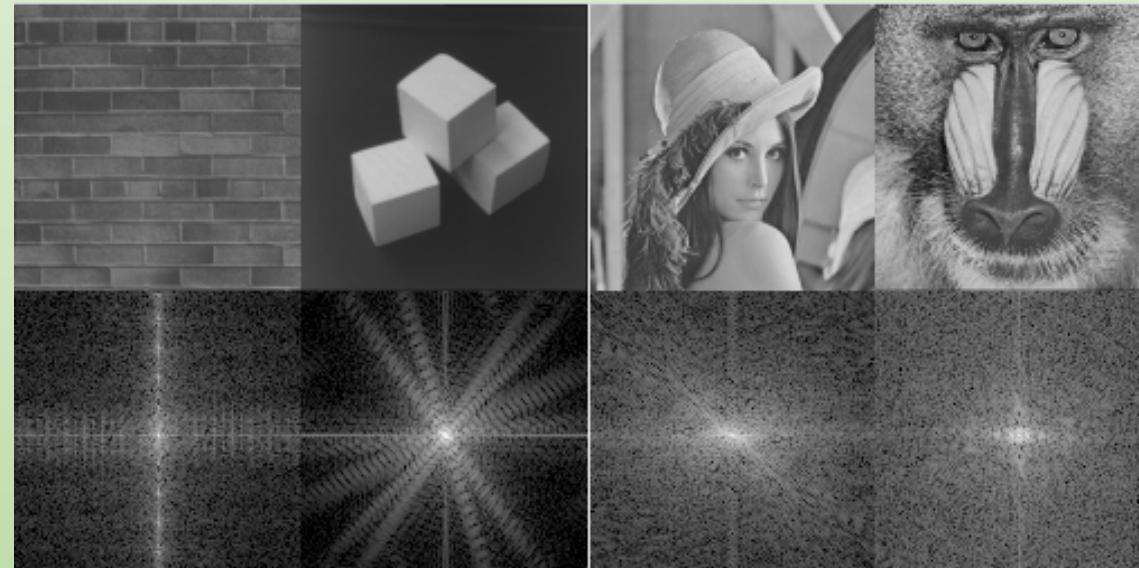
Disegnare un kernel per la stima dell'area della campana di gauss con il metodo Monte Carlo usando la libreria curand

# cuFFT...

Fast Fourier Transform algotirithms

# Fourier Transform (cuFFT)

- ✓ 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)



# cuFFT lib

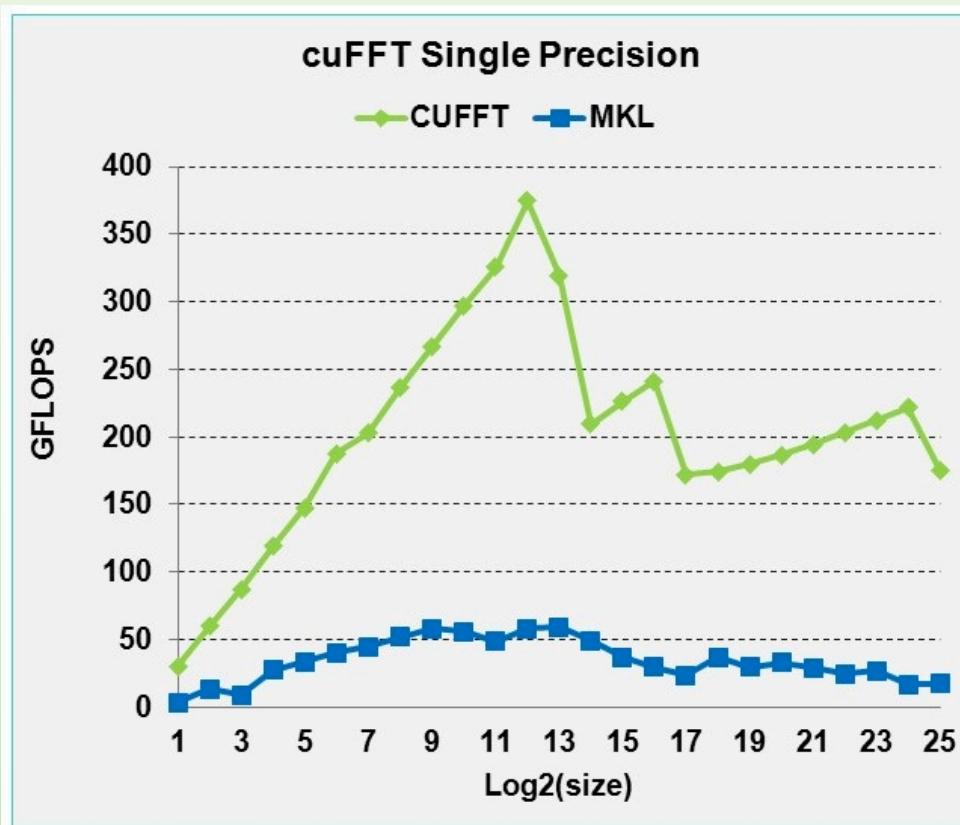
- ✓ The **cuFFT** library provides an optimized, CUDA-based implementation of the fast Fourier transform (**FFT**)
- ✓ An **FFT** is a **transformation** in signal processing that converts a signal from the **time domain** to the **frequency domain**
- ✓ An **inverse FFT** does the opposite
- ✓ FFT receives 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 superimposed to create the signal that **generated the input samples**

# Lib. features

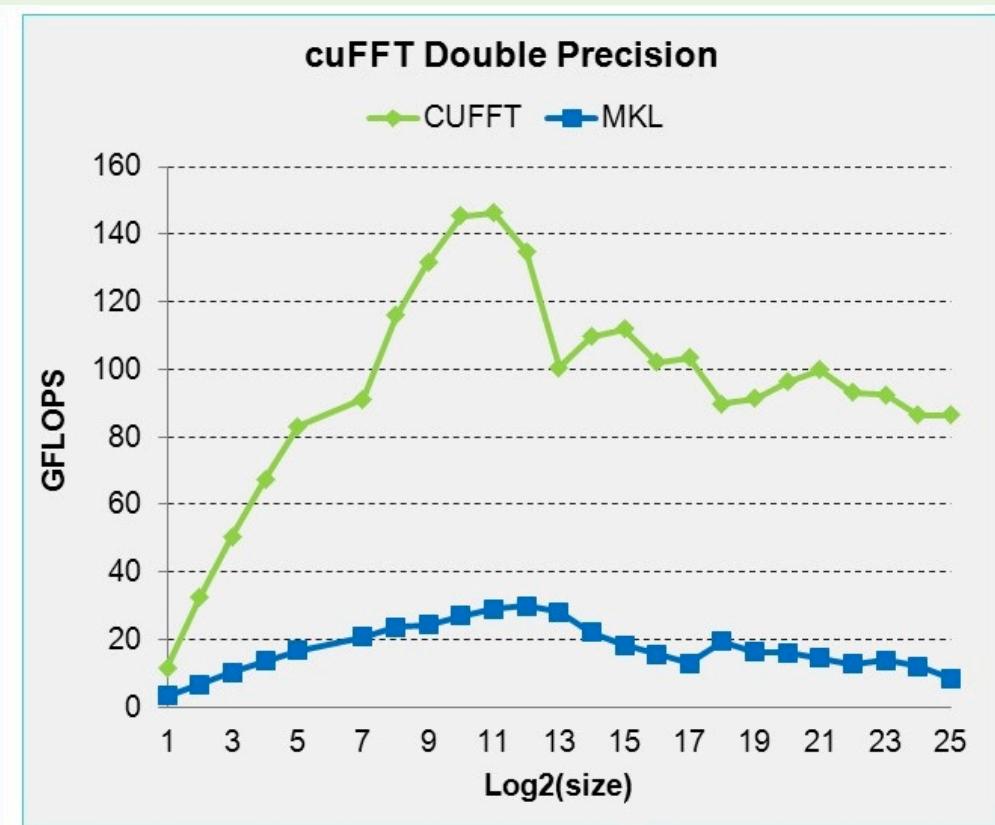
- ✓ Supports 1D, 2D, or 3D Fourier Transforms
- ✓ 1D transforms can have up to 128 million elements
- ✓ Based on Cooley-Tukey and Bluestein FFT algorithms
- ✓ Similar API to FFTW, if familiar
- ✓ Thread-safe, streamed, asynchronous execution
- ✓ Supports both in-place and out-of-place transforms
- ✓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

# cuFFT vs MKL

Tesla M2090 vs Xeon x5680 3.3GHz



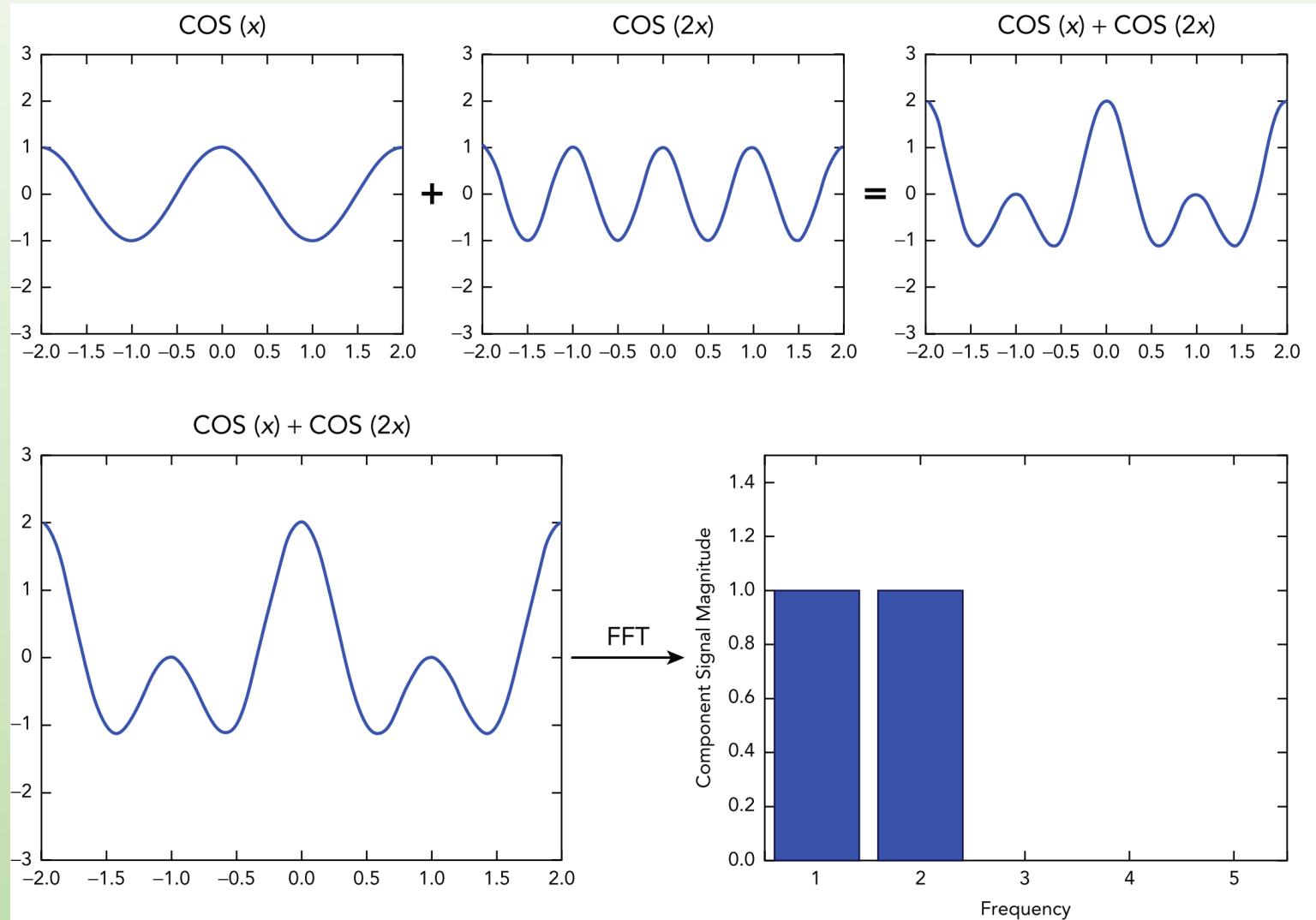
- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz



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

# Example

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]:

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

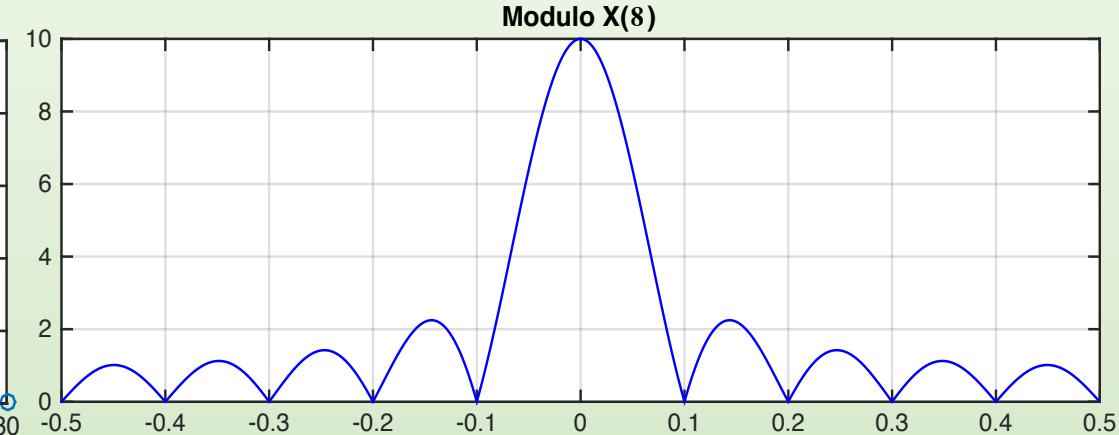
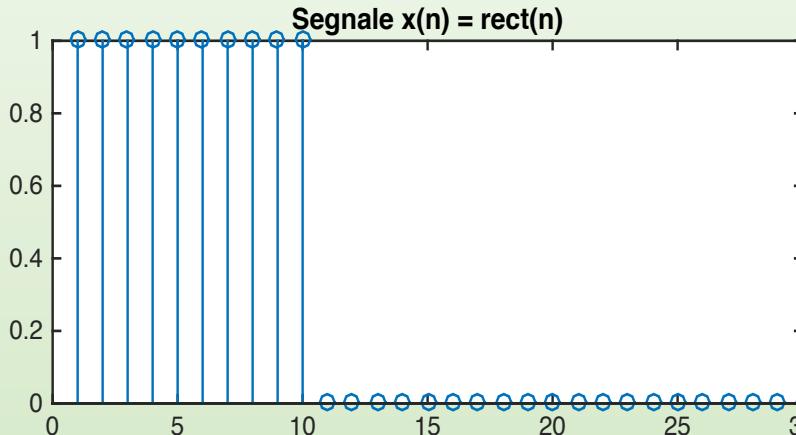
$$x(n) = \int_0^1 X(\nu)e^{i2\pi\nu n} d\nu$$

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

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

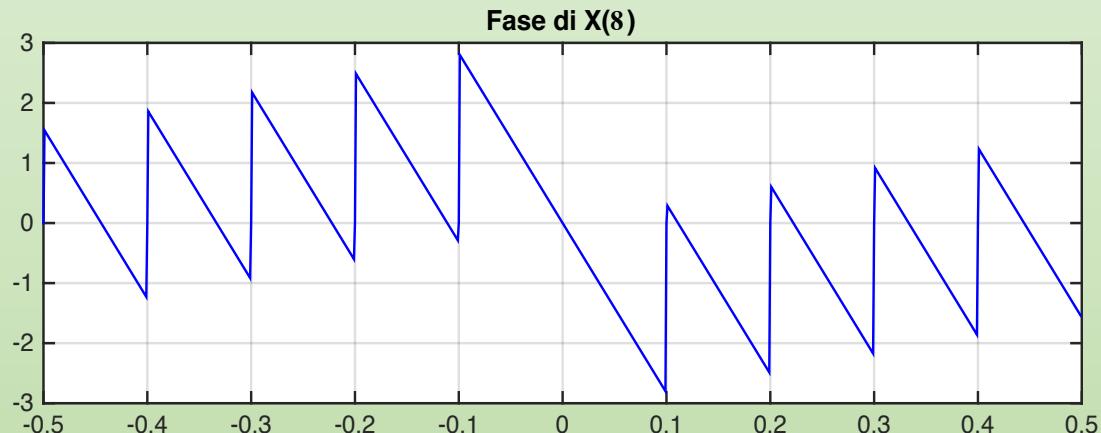
$$x(n) = \frac{1}{2\pi} \int_0^{2\pi} X(e^{i\omega})e^{i\omega n} d\omega$$

# Esempio: DTFT di $\text{rect}(n)$



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

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



# Trasformata di Fourier discreta - DFT

Se  $x(n)$  è un segnale periodico di periodo  $per = NT$ , trasformata e antitrasformata di Fourier sono date da:

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

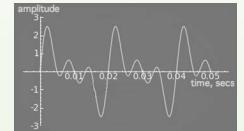
$$x(nT) = F \sum_{k=0}^{N-1} X(kF) e^{j2\pi \frac{nk}{N}}, \quad x = [x(0), x(T), x(2T), \dots, x((N-1)T)]$$

Ponendo  $T = 1$ , si ha rispettivamente:

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

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

# 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

**D2Z**: double to double complex

**C2R**: Complex to real

**Z2D**: double complex to double

**C2C**: complex to complex

**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);
```

# 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);

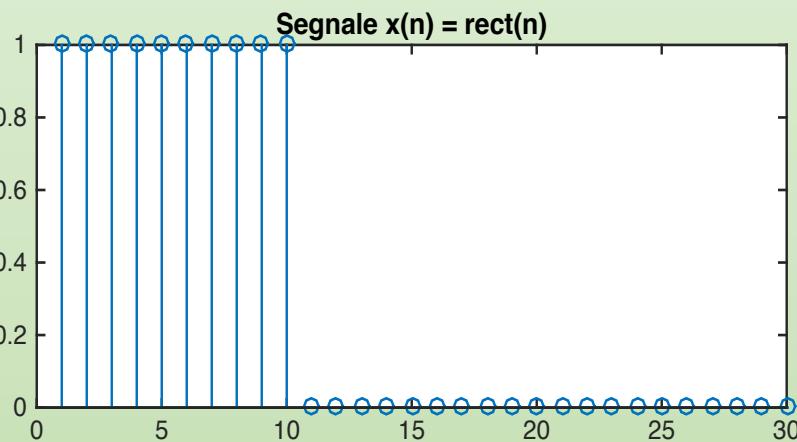
cudaFree(dComplexSamples);
cufftDestroy(plan);
```

# Esercitazione

## FFT di un impulso rettangolare:

Effettuare il calcolo della trasformata e della trasformata inversa verificando che  $\text{IFFT}(\text{FFT}(x)) = x$ :

- Allocare il vettore con impulso di durata pari a  $\frac{1}{4}, \frac{1}{2}, \frac{3}{4}$  del segnale (di lunghezza **N**)
- Verificare l'efficienza usando tipi reali e complessi



$$\leftarrow x = \text{IFFT}(\text{FFT}(x)) \rightarrow$$

