



Librerie CUDA e Applicazioni

Sistemi di Elaborazione Accelerata, Modulo 2

A.A. 2025/2026

Fabio Tosi, Università di Bologna

Librerie CUDA e Applicazioni

➤ Introduzione alle Librerie CUDA

- Cos'è una Libreria
- Principali Librerie CUDA

➤ Utilizzo delle Librerie CUDA

- Workflow di Importazione e Utilizzo
- Esempi Pratici con cuBLAS e cuRAND

➤ CUDA in Python

- PyCUDA
- CuPy
- Accenni ad altri Framework

➤ Applicazioni

- AI e Computer Vision
- Opportunità di Tesi

Introduzione alle Librerie CUDA

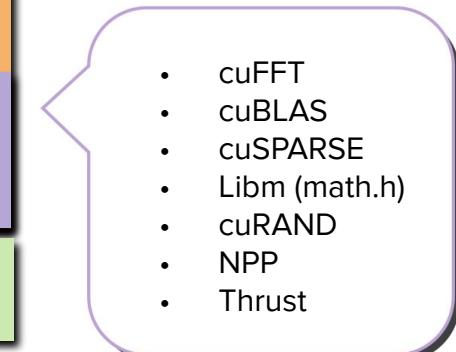
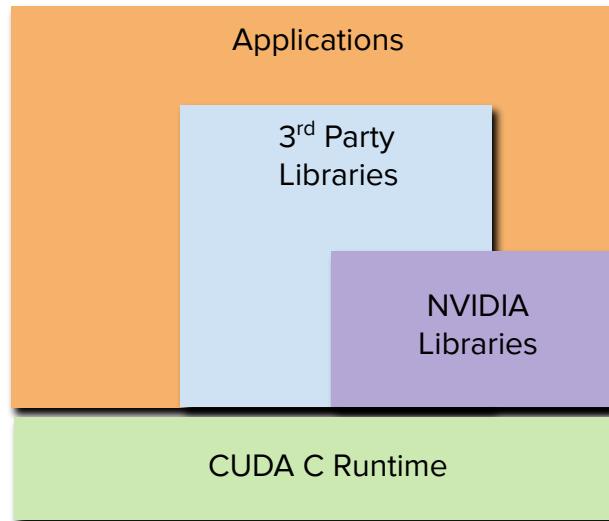
Cosa sono le librerie CUDA?

- Le librerie CUDA sono collezioni di **funzioni** e **routine** ottimizzate per l'esecuzione su GPU NVIDIA.
- **Parte integrante dell'ecosistema CUDA.**
- Progettate per sfruttare l'architettura parallela massiccia delle GPU.
- Simili alle librerie di sistema o personalizzate, ma **accelerate da GPU**.

Astrazione e Funzionalità Specifiche

Le librerie CUDA si basano sul runtime CUDA e forniscono:

- **Interfaccia Semplice:** Facilità di utilizzo per sviluppatori di applicazioni host e di librerie di terze parti.
- **Funzionalità Specifiche per Dominio:** Accesso ottimizzato a funzioni e algoritmi per settori specifici.



Introduzione alle Librerie CUDA

Caratteristiche Principali e Vantaggi

- **Ottimizzazione per GPU:**
 - Implementazioni altamente efficienti di algoritmi comuni (es. algebra lineare, trasformate di Fourier).
 - Sfruttano appieno le caratteristiche hardware specifiche delle GPU NVIDIA.
- **Astrazione della Complessità**
 - Offrono il miglior equilibrio tra facilità d'uso e prestazioni per molte applicazioni.
 - Nascondono i dettagli di basso livello della programmazione CUDA.
 - Forniscono API di alto livello per operazioni complesse.
- **Prestazioni Elevate**
 - Le librerie CUDA offrono prestazioni migliori rispetto alle librerie esclusive per host e spesso anche rispetto alle implementazioni custom CUDA.
 - Scalano efficacemente su diverse generazioni di GPU, garantendo compatibilità futura.
- **Versatilità**
 - Coprono una vasta gamma di domini applicativi, dalla matematica di base, all'elaborazione dei segnali digitali, all'intelligenza artificiale e molto altro.
- **Basso Carico di Manutenzione**
 - Le librerie CUDA riducono il carico di manutenzione per gli sviluppatori di software, poiché le implementazioni esistenti e mature sono già state testate e gestite da NVIDIA e dai suoi partner.

Librerie Matematiche in CUDA

- Le librerie matematiche accelerate da GPU pongono le basi per applicazioni ad alta intensità di calcolo in aree come la dinamica molecolare, la fluidodinamica computazionale, la chimica computazionale, l'imaging medico, l'esplorazione sismica e così via.



cuBLAS

Libreria di algebra lineare di base (BLAS) accelerata da GPU.

[Documentazione](#)



cuFFT

Libreria accelerata da GPU per implementazioni della FFT

[Documentazione](#)



cuRAND

Generazione di numeri casuali accelerata da GPU.

[Documentazione](#)



cuSOLVER

Risolutori diretti densi e sparsi accelerati da GPU.

[Documentazione](#)



cuSPARSE

BLAS accelerato da GPU per matrici sparse.

[Documentazione](#)



cuDSS

Libreria di risolutori diretti sparsi accelerata da GPU.

[Documentazione](#)



CUDA Math API

API di funzioni matematiche standard accelerate da GPU.

[Documentazione](#)



cuTENSOR

Libreria di algebra lineare per tensori accelerata da GPU.

[Documentazione](#)

Librerie Quantistiche in CUDA

- Consentono di accelerare simulazioni e applicazioni avanzate nel **calcolo quantistico**.

cuQuantum

NVIDIA cuQuantum è un insieme di librerie altamente ottimizzate per accelerare le simulazioni di calcolo quantistico.

[Documentazione](#)

cuPQC

SDK di librerie ottimizzate per accelerare i flussi di lavoro di crittografia post-quantistica (PQC).

[Documentazione](#)

Librerie per Immagini e Video in CUDA

- Librerie accelerate da GPU per la **decodifica, codifica ed elaborazione di immagini e video** che utilizzano CUDA e componenti hardware specializzati delle GPU.

RAPIDS cuCIM

Accelerà l'input/output (IO), la visione artificiale e l'elaborazione delle immagini di immagini n-dimensionalì, in particolare immagini biomediche.

[Documentazione](#)

CV-CUDA

Libreria open-source per pre- e post-elaborazione ad alte prestazioni, accelerata da GPU, nei pipeline di visione AI.

[Documentazione](#)

NVIDIA DALI

Libreria portatile e open-source per la decodifica e l'augmented imaging di immagini e video per accelerare le applicazioni di deep learning.

[Documentazione](#)

nvJPEG

Libreria ad alte prestazioni accelerata da GPU per la decodifica JPEG.

[Documentazione](#)

NVIDIA Performance Primitives

Funzioni per l'elaborazione di immagini, video e segnali accelerate da GPU.

[Documentazione](#)

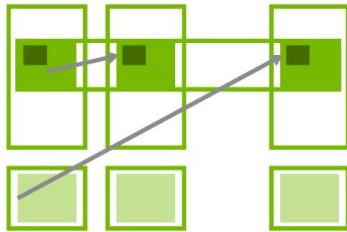
NVIDIA Video Codec

Codifica e decodifica video accelerate dall'hardware su Windows e Linux.

[Documentazione](#)

Librerie di Comunicazione in CUDA

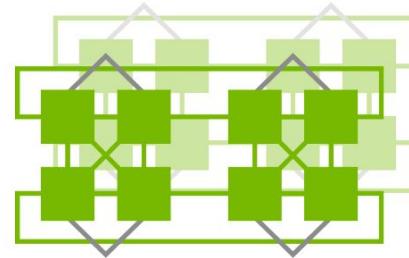
- Primitive di comunicazione ottimizzate per le prestazioni su **multi-GPU e multi-nodo**.



NVSHMEM

Standard OpenSHMEM per la memoria GPU, con estensioni per migliorare le prestazioni sulle GPU.

[Documentazione](#)



NCCL

Libreria open-source per comunicazione rapida tra multi-GPU e multi-nodo che massimizza la larghezza di banda mantenendo bassa la latenza.

[Documentazione](#)

Librerie di Algoritmi Paralleli e di Litografia Computazionale

- **Librerie di Algoritmi Paralleli:** GPU-accelerate, queste librerie offrono algoritmi paralleli efficienti per operazioni in C++ e analisi di grafi in vari campi, come scienze naturali, logistica e pianificazione viaggi.
- **Libreria di Litografia Computazionale:** Progettata per affrontare le sfide della litografia computazionale su scala nanometrica.



Thrust

Libreria accelerata da GPU di algoritmi paralleli e strutture dati in C++.

[Documentazione](#)



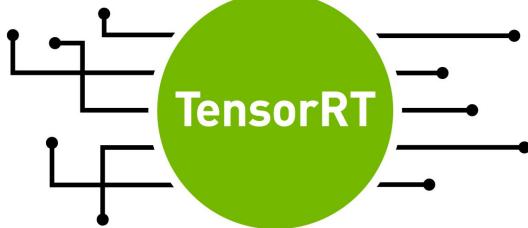
cuLitho

Libreria con strumenti e algoritmi ottimizzati per accelerare la litografia computazionale e la produzione di semiconduttori utilizzando le GPU.

[Documentazione](#)

Librerie di Deep Learning in CUDA

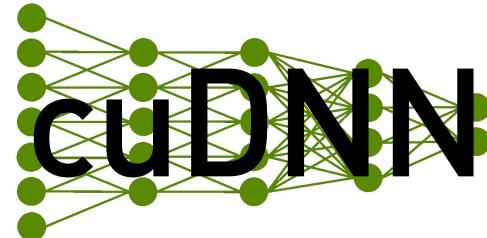
- Librerie accelerate da GPU per applicazioni di deep learning che utilizzano CUDA e componenti hardware specializzati delle GPU.



TensorRT

Ottimizzatore e runtime ad alte prestazioni per l'inferenza di deep learning, pensato per il deployment in produzione.

[Documentazione](#)



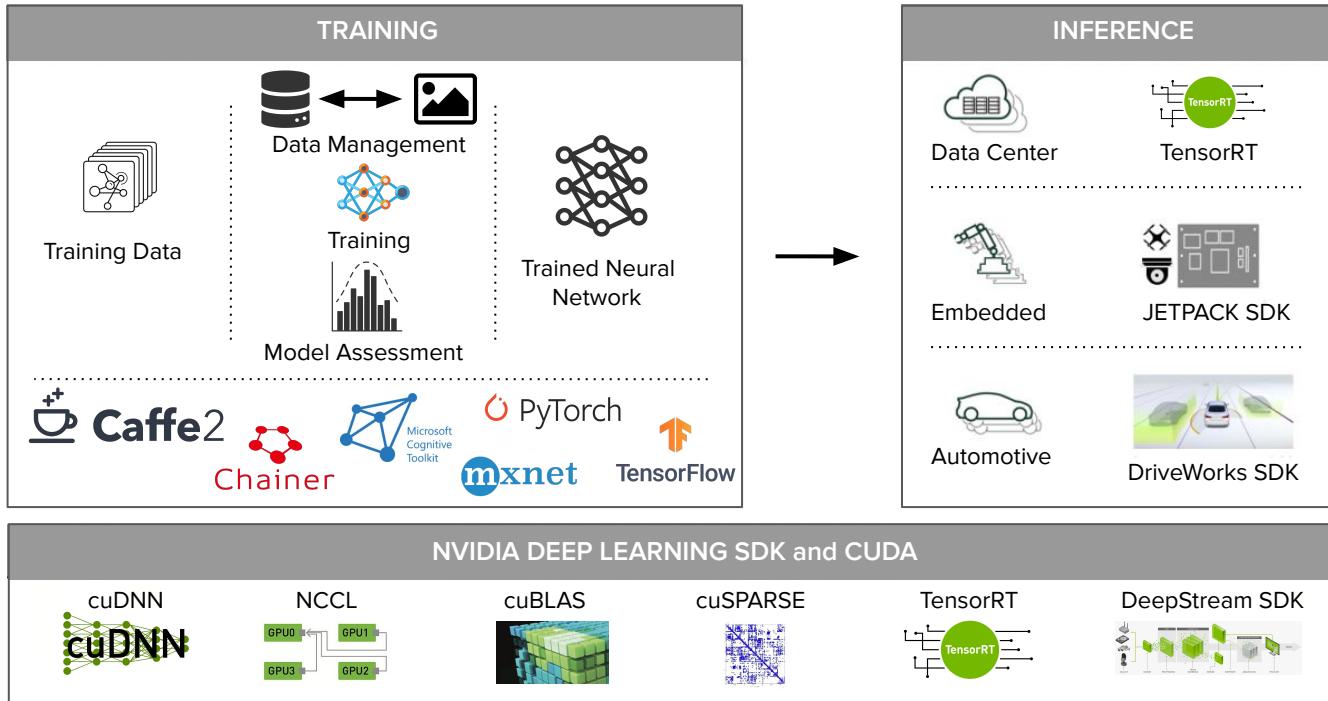
cuDNN

Libreria accelerata da GPU di primitive per deep neural network.

[Documentazione](#)

Librerie di Deep Learning in CUDA

- Librerie accelerate da GPU per applicazioni di deep learning che utilizzano CUDA e componenti hardware specializzati delle GPU.



Librerie CUDA e Applicazioni

➤ Introduzione alle Librerie CUDA

- Cos'è una Libreria
- Principali Librerie CUDA

➤ Utilizzo delle Librerie CUDA

- Workflow di Importazione e Utilizzo
- Esempi Pratici con cuBLAS e cuRAND

➤ CUDA in Python

- PyCUDA
- CuPy
- Accenni ad altri Framework

➤ Applicazioni

- AI e Computer Vision
- Opportunità di Tesi

Workflow delle Librerie CUDA

1. Inizializzazione

- Creare l'**handle della libreria** per gestire le informazioni contestuali necessarie al suo funzionamento.

2. Preparazione dei Dati

- **Allocare memoria GPU** per input/output.
- **Convertire dati** nel formato della libreria (es. convertire array 2D in *column-major order*).
- **Trasferire i dati convertiti** nella memoria GPU.

3. Configurazione ed Esecuzione

- **Impostare parametri di computazione** (es. dimensioni dei dati o opzioni specifiche).
- **Chiamare la funzione** di libreria per delegare la computazione alla GPU.

4. Gestione dei Risultati

- **Recuperare risultati** dalla memoria GPU.
- **Riconvertire i dati** nel formato dell'applicazione (se necessario).

5. Pulizia

- **Rilasciare tutte le risorse CUDA** allocate durante il processo.

***Nota:** Questo workflow può variare leggermente tra le diverse librerie CUDA.

Workflow delle Librerie CUDA: Esempio con cuBLAS

Obiettivo

- **cuBLAS (CUDA Basic Linear Algebra Subprogram)**: Libreria CUDA per operazioni di algebra lineare, come prodotti scalari, moltiplicazione di matrici e vettori, e prodotti di matrici, e altre operazioni di base.
- Queste operazioni sono utilizzate come blocchi costitutivi per problemi più complessi, come la risoluzione di sistemi lineari o il calcolo di autovalori e autovettori.
- Per **compatibilità con gli standard BLAS e Fortran**, cuBLAS sceglie di usare la memorizzazione **column-major**.
- **Esempio di Utilizzo**: Calcolare il prodotto di due matrici ($C = A * B$) in modo efficiente sfruttando la potenza di calcolo parallelo delle GPU tramite la libreria cuBLAS.

1. Includere l'Header e Inizializzare cuBLAS

- Includere l'header `<cublas_v2.h>` per accedere alle funzioni di cuBLAS e creare un **handle** per gestire le operazioni della libreria.

```
#include <cuda_runtime.h> // Include necessario per le funzioni CUDA
#include <cublas_v2.h>    // Include necessario per le funzioni cuBLAS

cublasHandle_t handle;
cublasCreate(&handle); // Inizializza il contesto cuBLAS
```

Workflow delle Librerie CUDA: Esempio con cuBLAS

2. Preparazione dei Dati

- Allocare la memoria necessaria sulla GPU per le matrici di input (A, B) e di output (C) utilizzando `cudaMalloc`.
- Copiare i dati delle matrici dalla memoria dell'host alla memoria del device con `cudaMemcpy`.

```
float *h_A, *h_B, *h_C; // Matrici su CPU (host)
float *d_A, *d_B, *d_C; // Matrici su GPU (device)
int m, n, k; // Dimensioni matrici (es. A: m x k, B: k x n, C: m x n)
// (Memoria GPU allocata con cudaMalloc)
cudaMemcpy(d_A, h_A, m * k * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, k * n * sizeof(float), cudaMemcpyHostToDevice);
```

3. Configurare ed Eseguire la Moltiplicazione

- Imposta i parametri per la moltiplicazione, come le **dimensioni delle matrici**, il **tipo di dato** (`float` in questo caso) ed eventuali **coefficienti di scaling** (α , β). **Nota:** cuBLAS usa *column-major order*.
- Eseguire la moltiplicazione delle matrici sulla GPU chiamando la funzione `cublasSgemm` (Single-precision).

```
float alpha = 1.0f, beta = 0.0f; // C =  $\alpha$  * A * B +  $\beta$  * C
cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, m, k, &alpha, d_B, n, d_A, k, &beta, d_C, n);
```

Workflow delle Librerie CUDA: Esempio con cuBLAS

4. Recuperare i Dati

- Copia la matrice risultato (C) dalla memoria della GPU alla memoria dell'host utilizzando `cudaMemcpy`.

```
cudaMemcpy(h_C, d_C, m * n * sizeof(float), cudaMemcpyDeviceToHost);
```

5. Liberare le Risorse

- Libera la memoria allocata sulla GPU con `cudaFree` e distruggi l'handle cuBLAS con `cublasDestroy`.

```
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
cublasDestroy(handle);
```

Compilazione

- Per compilare un programma CUDA che utilizza cuBLAS, è necessario utilizzare `nvcc` con specifici flag:

```
nvcc program.cu -lcublas -o program
```

Workflow delle Librerie CUDA: Esempio con cuRAND

Obiettivo

- Generare un insieme di numeri casuali su GPU utilizzando la libreria [cuRAND](#) (CUDA Random Number Generation), che è ottimizzata per la generazione parallela di numeri casuali in grandi quantità.

1. Includere l'Header e Inizializzare cuRAND

- Includere l'header `<curand.h>` per accedere alle funzioni della libreria cuRAND.
- Creare e inizializzare uno stato del generatore di numeri casuali.

```
#include <cuda_runtime.h> // Include necessario per le funzioni CUDA
#include <curand.h>      // Include necessario per le funzioni cuRAND

curandGenerator_t gen;
// Crea il generatore di numeri casuali
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
// Inizializza il generatore con un seed
curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);
```

Workflow delle Librerie CUDA: Esempio con cuRAND

2. Preparazione dei Dati

- Riservare spazio sulla GPU per contenere i numeri casuali generati.
- Specificare il numero di numeri casuali da generare.

```
float *d_data;  
// Numero di numeri casuali da generare  
size_t n = 100000;  
// Alloca memoria sulla GPU per i numeri casuali  
cudaMalloc( (void**) &d_data, n * sizeof(float));
```

3. Generare i Numeri Casuali

- Usare cuRAND per generare numeri casuali uniformemente distribuiti in virgola mobile.

```
// Genera n numeri casuali uniformemente distribuiti in [0,1) su GPU  
curandGenerateUniform(gen, d_data, n);
```

Workflow delle Librerie CUDA: Esempio con cuRAND

4. Recuperare i Dati

- Copiare i numeri casuali generati dalla GPU alla CPU per ulteriori elaborazioni o analisi.

```
// Alloca memoria sulla CPU
float *h_data = (float*)malloc(n * sizeof(float));
// Copia i dati dalla GPU alla CPU
cudaMemcpy(h_data, d_data, n * sizeof(float), cudaMemcpyDeviceToHost);
```

5. Liberare le Risorse

- Liberare la memoria GPU e distruggere il generatore cuRAND per rilasciare le risorse allocate.

```
cudaFree(d_data); // Libera la memoria allocata sulla GPU
curandDestroyGenerator(gen); // Distrugge il generatore di numeri casuali
```

Compilazione

- Per compilare un programma CUDA che utilizza cuRAND, è necessario utilizzare **nvcc** con specifici flag:

```
nvcc program.cu -lcurand -o program
```

Librerie CUDA e Applicazioni

➤ Introduzione alle Librerie CUDA

- Cos'è una Libreria
- Principali Librerie CUDA

➤ Utilizzo delle Librerie CUDA

- Workflow di Importazione e Utilizzo
- Esempi Pratici con cuBLAS e cuRAND

➤ CUDA in Python

- PyCUDA
- CuPy
- Accenni ad altri Framework

➤ Applicazioni

- AI e Computer Vision
- Opportunità di Tesi

Introduzione all'uso delle GPU in Python



L'Ecosistema Python per il Calcolo Scientifico

- Python è tra i linguaggi più usati in ambito scientifico e ingegneristico, grazie alla sua semplicità e alle numerose librerie per il calcolo numerico e il machine learning.
- L'**ecosistema Python** include strumenti fondamentali come [NumPy](#) per il calcolo numerico, [SciPy](#) per il calcolo scientifico, [Matplotlib](#) e [Jupyter](#) per grafici e per l'esecuzione interattiva del codice.

Le Limitazioni del Python Tradizionale

- Python, essendo un **linguaggio interpretato con tipizzazione dinamica** e progettato per eseguire codice principalmente su CPU, risulta **significativamente più lento** rispetto a linguaggi compilati come C++ o Fortran.

Librerie/Framework Python per il Calcolo su GPU

- Per chi vuole scrivere **kernel GPU personalizzati**:
 - [Numba](#), [PyOpenCL](#), [PyCUDA](#), [CUDA Python](#)
- **Sostituti diretti** per NumPy, SciPy, scikit-learn, ecc.:
 - [CuPy](#), [RAPIDS](#)
- **Framework di Machine/Deep Learning**:
 - [PyTorch](#), [TensorFlow](#), [JAX](#)
- **Calcolo Distribuito / Multi-nodo**:
 - [mpi4py+X](#), [dask](#), [cuNumeric](#)



CuPy



RAPIDS

PyTorch



TensorFlow

Numba

Introduzione a PyCUDA

Cos'è PyCUDA? ([Documentazione Online](#))

- PyCUDA è una **libreria Python** che consente di eseguire codice CUDA su GPU NVIDIA direttamente da Python.
- Fornisce un'interfaccia per **scrivere kernel CUDA in C/C++** e lanciare la loro esecuzione, integrandosi con librerie Python come NumPy.

Perchè usare PyCUDA?

- **Facilità d'uso:** Sintassi Python semplice e intuitiva, ideale per chi ha familiarità con Python.
- **Integrazione Python:** Si integra perfettamente con librerie come NumPy, semplificando la gestione dei dati.
- **Compilazione dinamica:** PyCUDA permette di scrivere e compilare **kernel CUDA** (scritti in C/C++) al volo durante l'esecuzione del programma, senza bisogno di compilazione esterna.
- **Gestione semplificata della memoria:** Semplifica la gestione della memoria rispetto a CUDA C grazie al *garbage collector* di Python, ma l'allocazione e il trasferimento dei dati tra CPU e GPU restano manuali.
- **Debugging semplificato:** Errori CUDA gestiti come eccezioni Python, facilitando il debug.

Installazione

- PyCUDA può essere installato facilmente usando **pip**:
`pip install pycuda`
- **Prerequisiti:** È necessario avere driver CUDA e CUDA Toolkit installati e configurati correttamente.

Architettura e Principi di Funzionamento di PyCUDA

Compilazione Just-in-Time (JIT):

- Il codice CUDA, definito come stringhe all'interno del codice Python, viene **compilato dinamicamente** durante l'esecuzione del programma (caching dei kernel compilati per evitare ricompilazioni ridondanti).

Gestione della Memoria:

- PyCUDA permette di **allocare e gestire la memoria** del device GPU **direttamente da Python**, semplificando il trasferimento dati tra host e device.

Esecuzione di Kernel:

- PyCUDA offre un'interfaccia intuitiva per lanciare kernel CUDA, definendo la configurazione di griglia, blocchi, dimensione della shared memory ed altre proprietà del kernel tramite una **sintassi Pythonica**.

API Pythonica:

- Offre sia **API di alto livello** per operazioni comuni, che **API di basso livello** per controllo granulare.
- Mantiene l'accesso alle funzionalità native CUDA pur fornendo un'interfaccia Pythonica.

Error Handling e Debugging:

- Converte automaticamente gli **errori CUDA** in **eccezioni Python native**, fornendo informazioni dettagliate sul contesto degli errori, inclusi problemi di compilazione, esecuzione e gestione della memoria.

Interoperabilità con Ecosistema Python e Librerie CUDA:

- Integrazione con librerie scientifiche Python (NumPy, SciPy) e accesso alle librerie CUDA ottimizzate (cuBLAS, cuFFT, cuRAND, ecc.) tramite scikit-cuda.

Sintassi di un Kernel in PyCUDA: Esempio

Sintassi Kernel

- **CUDA C:** Il kernel viene scritto in C e compilato prima dell'esecuzione.
- **PyCUDA:** Il kernel CUDA viene definito come stringa all'interno del programma Python, compilato dinamicamente a runtime tramite la classe **SourceModule** e recuperato usando il metodo **get_function**.

CUDA C

```
__global__ void add_arrays(float *a, float *b, float *c, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) c[idx] = a[idx] + b[idx];
}
```

PyCUDA

```
mod = SourceModule("""
__global__ void add_arrays(float *a, float *b, float *c, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) c[idx] = a[idx] + b[idx];
}
""")

func = mod.get_function("add_arrays")
```

Sintassi di un Kernel in PyCUDA: Esempio

Lancio di un Kernel

- **CUDA C:** Il kernel è lanciato con `add_arrays<<<grid_size, block_size>>>(args)` dove `grid_size` definisce il numero di blocchi e `block_size` il numero di thread per blocco.
- **PyCUDA:** Il kernel è lanciato con `func(d_a, d_b, d_c, block=(block_size, 1, 1), grid=(grid_size, 1))`.

CUDA C

```
int main() {  
    // Allocazione memoria, copia dati, lancio kernel  
    int block_size = 256;  
    int grid_size = (N + block_size - 1) / block_size; // Calcolo dimensione griglia  
    add_arrays<<<grid_size, block_size>>>(d_a, d_b, d_c);  
}
```

PyCUDA

```
# Allocazione memoria, copia dati, lancio kernel  
block_size = 256  
grid_size = (N + block_size - 1) // block_size # Calcolo dimensione griglia  
func(d_a, d_b, d_c, block=(block_size, 1, 1), grid=(grid_size, 1))
```

Trasferimento Dati in PyCUDA: Approccio Low-Level

Compatibilità diretta con NumPy

- PyCUDA consente di **utilizzare array NumPy** per trasferire dati tra host e device.

Esempio di Trasferimento Dati

- Creazione di un array NumPy su host:

```
import numpy as np
a = np.random.randn(100).astype(np.float32) # Array NumPy su CPU
```

- Allocazione memoria sul device + Copia dati:

```
import pycuda.driver as cuda
import pycuda.autoinit

# Allocazione memoria sul device e copia dati
d_a = cuda.mem_alloc(a.nbytes) # Alloca memoria su GPU
cuda.memcpy_htod(d_a, a)      # Copia array da host a device
```

- Copia dati dal device all'host + Deallocazione della memoria sul device (opzionale):

```
a_host = np.empty_like(a) # Crea array vuoto per ricevere i dati
cuda.memcpy_dtoh(a_host, d_a) # Copia dati dal device all'host
d_a.free() # Deallocazione esplicita (se necessario)
```

Trasferimento Dati in PyCUDA: Approccio High-Level

Memoria in PyCUDA

- PyCUDA semplifica notevolmente la **gestione dei dati tra host e device** tramite la classe **gpuarray**, che rappresenta l'equivalente GPU degli array NumPy.
- Questa classe **gestisce automaticamente allocazione e deallocazione** della memoria GPU e fornisce funzionalità familiari come operazioni su array multidimensionali e slicing.

Esempio di Trasferimento Host-Device con **gpuarray**

```
import numpy as np
import pycuda.gpuarray as gpuarray
import pycuda.autoinit

# Array su host
host_data = np.array([1, 2, 3], dtype=np.float32)

# Trasferimento dati dalla CPU alla GPU
gpu_data = gpuarray.to_gpu(host_data)

# Recupero dei risultati dalla GPU alla CPU
result = gpu_data.get()
```

Esempio di Codice PyCUDA - Somma di Vettori

1. Inizializzazione e Contesto GPU

- **Cos'è il Contesto?**: Il "contesto" è come una sessione di lavoro che gestisce le risorse della GPU, come la memoria e i kernel. Serve a collegare il programma Python alla GPU, permettendo di eseguire codice CUDA.

```
import numpy as np
from pycuda import driver, compiler, gpuarray

driver.init()    # Inizializza PyCUDA
device = driver.Device(0) # Seleziona la prima GPU disponibile
context = device.make_context() # Crea un contesto CUDA per il device
```

2. Definizione e Compilazione del Kernel CUDA

- **Compilazione**: In PyCUDA, il kernel è scritto come stringa di testo in Python e viene compilato durante l'esecuzione del programma.

```
kernel_code = """
__global__ void add_arrays(float *a, float *b, float *c, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) c[idx] = a[idx] + b[idx];
}
"""

module = compiler.SourceModule(kernel_code)    # Compila il kernel CUDA
```

Esempio di Codice PyCUDA - Somma di Vettori

3. Allocazione Memoria su GPU e Lancio del Kernel

- Allocazione della Memoria:** Utilizzo della classe `gpuarray` per allocare memoria sulla GPU e trasferire facilmente dati tra la GPU e l'host.

```
a_gpu = gpuarray.to_gpu(np.random.randn(N).astype(np.float32))
b_gpu = gpuarray.to_gpu(np.random.randn(N).astype(np.float32))
c_gpu = gpuarray.empty_like(a_gpu)
add_arrays = module.get_function("add_arrays")
add_arrays(a_gpu, b_gpu, c_gpu, np.int32(N), block=(block_size,1,1), grid=(grid_size, 1))
```

4. Recupero dei Risultati e Pulizia

- Recupero dei Dati:** Il risultato dell'operazione sulla GPU viene trasferito di nuovo alla CPU con il metodo `.get()`.
- Rilascio del Contesto:** Il contesto GPU viene rilasciato, liberando le risorse. Questo passaggio è importante per evitare memory leaks e mantenere la GPU libera per altri usi.

```
c_cpu = c_gpu.get() # Copia il risultato dalla GPU alla CPU
context.pop() # Rilascia il contesto GPU
```

Esempio di Codice PyCUDA - Somma di Vettori

Esecuzione del Profiling con Nsight Compute

- Per eseguire il profiling del programma, è possibile utilizzare il comando **ncu** direttamente dal terminale.
Esempio di comando:

```
ncu --set full python vector_add.py
```

- Opzioni principali di **ncu**:
 - set full**: Esegue un profiling completo, includendo metriche dettagliate sulle prestazioni.
 - python vector_add.py**: Specifica il nome dello script Python che avvia l'esecuzione dei kernel CUDA su cui fare il profiling.

Inclusione File CUDA Esterno in PyCUDA - Somma di Vettori

File CUDA Esterno (vector_add.cu)

- Invece di scrivere i kernel come stringhe in Python, è possibile **caricare e compilare un file CUDA esterno**, migliorando la leggibilità e il riutilizzo del codice.

File CUDA C - vector_add.cu

```
__global__ void add_arrays(float *a, float *b, float *c, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) c[idx] = a[idx] + b[idx];
}
```

Integrazione in Python (vector_add.py)

```
with open('vector_add.cu', 'r') as f:
    kernel_code = f.read()
module = compiler.SourceModule(kernel_code)
add_arrays = module.get_function("add_arrays")

# Configurazione griglia per N elementi
block_size = 256
grid_size = (N + block_size - 1) // block_size
add_arrays(a_gpu, b_gpu, c_gpu, np.int32(N), block=(block_size,1,1), grid=(grid_size,1,1))
```

Confronto fra CUDA C e PyCUDA

	CUDA C	PyCUDA
Sintassi	Codice più complesso , ma offre controllo preciso .	Sintassi semplificata , più leggibile, ma con meno controllo .
Compilazione	Compilazione statica del codice in fasi distinte (build separata).	Dinamica , più flessibile ma potenzialmente meno efficiente.
Gestione della Memoria	Manuale , ma garantisce un controllo totale sulle risorse.	Semplificata , riduce il rischio di errori, ma meno flessibile.
Interfaccia host-device	Completo controllo su trasferimenti, allocazioni e sincronizzazioni.	Semplificata , ma con meno possibilità di ottimizzazioni avanzate.
Debugging	Più laborioso, richiede gestione manuale degli errori .	Gestito tramite eccezioni Python , semplificando il debug.
Flessibilità	Richiede molte linee di codice per compiti semplici.	Ideale per prototipazione veloce e per interfacce con altre librerie Python.
Astrazione	Basso livello , controllo completo su hardware e risorse.	Livello di astrazione più alto , automatizza molte operazioni.

Introduzione a CuPy



Cos'è CuPy? ([Documentazione Online](#))

- CuPy è una libreria **open-source**, compatibile con NumPy, progettata per il calcolo scientifico su GPU.
- Fornisce un'interfaccia quasi identica a NumPy ma esegue i calcoli su **GPU NVIDIA tramite CUDA**.
- CuPy **sfrutta librerie del CUDA Toolkit** (come cuBLAS, cuRAND, cuSOLVER) per garantire elevate prestazioni.
- È disponibile anche il **supporto per GPU AMD tramite ROCm**, sebbene con compatibilità più limitata.

Perché usare CuPy?

- **Familiarità:** API simile a NumPy, curva di apprendimento minima.
- **Performance:** Accelerazione GPU senza scrivere codice CUDA.
- **Compatibilità:** Supporta la maggior parte delle funzioni NumPy e SciPy.
- **Facilità d'Uso:** Non richiede conoscenza diretta di CUDA.
- **Ottimizzazione Automatica:** Gestisce automaticamente memoria e trasferimenti CPU-GPU.

Funzionalità Chiave

- **Array N-dimensionali:** Supporto completo per array multi-dimensionali eseguiti su GPU.
- **Operazioni Matematiche:** Funzioni matematiche avanzate (equivalenti a Numpy e SciPy) ottimizzate per GPU.
- **Kernel Personalizzati:** Possibilità di scrivere kernel CUDA personalizzati.

Installazione

- PyCUDA può essere installato facilmente usando **pip**:

```
pip install cupy-cudatoolkit # sostituire con versione CUDA appropriata
```

Introduzione agli Array CuPy

Core della Libreria

- La classe `cupy.ndarray` è il componente fondamentale di CuPy, offrendo un'**alternativa GPU completamente compatibile** con `numpy.ndarray`.

Inizializzazione

- Importazione delle librerie simile a NumPy:

```
import numpy as np  
import cupy as cp
```

Creazione Array GPU

- Sintassi identica a NumPy ma con allocazione su GPU:

```
x_gpu = cp.array([1, 2, 3])      # Array su GPU  
x_cpu = np.array([1, 2, 3])      # Array su CPU
```

Computazione

- Stesse funzioni di NumPy ma eseguite su GPU:

```
# Calcolo norma L2 (cp utilizza cubLAS internamente)  
l2_gpu = cp.linalg.norm(x_gpu)    # Esecuzione su GPU  
l2_cpu = np.linalg.norm(x_cpu)    # Esecuzione su CPU
```

Gestione dei Dispositivi GPU in CuPy

Il Device Corrente

- Ogni operazione CuPy avviene su un "device corrente" (GPU predefinita).
- Esempio con **GPU default** (ID 0):

```
x = cp.array([1, 2, 3]) # Allocato su GPU 0
```

Selezione del Device (Con una sola GPU, non serve gestire manualmente i device)

- Cambio permanente:

```
cp.cuda.Device(1).use() # Passa a GPU 1
```

- Cambio temporaneo tramite Context Manager:

```
with cp.cuda.Device(1): # Usa GPU 1 solo in questo blocco
    x = cp.array([1, 2, 3])
```

Regole e Limitazioni

- Gli array devono essere elaborati sulla stessa GPU dove sono allocati. Errori comuni:

```
with cp.cuda.Device(0):
    x = cp.array([1, 2, 3])
with cp.cuda.Device(1):
    x * 2 # Errore: array su GPU 0, operazione su GPU 1
```

Trasferimento Dati tra CPU e GPU in CuPy

Da CPU a GPU

- `cp.asarray()` sposta dati su GPU corrente array numpy, liste e tuple python, qualsiasi oggetto compatibile con `numpy.array()`.

```
x_cpu = np.array([1, 2, 3])
x_gpu = cp.asarray(x_cpu) # trasferimento su GPU
```

Tra GPU diverse

- Stesso metodo per trasferire tra device:

```
with cp.cuda.Device(0):
    x_gpu_0 = cp.array([1, 2, 3]) # crea un array su GPU 0
with cp.cuda.Device(1):
    x_gpu_1 = cp.asarray(x_gpu_0) # muove l'array su GPU 1
```

Ottimizzazione Copie

- `cp.asarray()` evita copie non necessarie se possibile, restituendolo direttamente se già sul device corrente.
- Per forzare la copia: `cp.array(arr, dtype, copy=True)`
- `cp.asarray()` equivale a `cp.array(arr, dtype, copy=False)`

Trasferimento Dati tra CPU e GPU in CuPy

Da GPU a CPU

- Con **asnumpy()**:

```
x_gpu = cp.array([1, 2, 3])          # crea array su GPU  
x_cpu = cp.asnumpy(x_gpu)           # sposta su CPU
```

- Con **get()**:

```
x_cpu = x_gpu.get()                 # sposta su CPU
```

Note

- Entrambi i metodi **creano un array NumPy sulla CPU**.
- L'array originale rimane sulla GPU.

Codice Agnóstico CPU/GPU in CuPy

get_array_module()

- `cupy.get_array_module()` restituisce il modulo corretto (Numpy o CuPy) in base agli argomenti. Esempio:

```
def softplus(x):  
    xp = cp.get_array_module(x) # alias per numpy o cupy, a seconda del tipo di x  
    return xp.maximum(0, x) + xp.log1p(xp.exp(-abs(x)))
```

Conversioni tra CPU e GPU

- Uso dei metodi `asarray()` e `asnumpy()` per gestire array di tipo CPU e GPU:

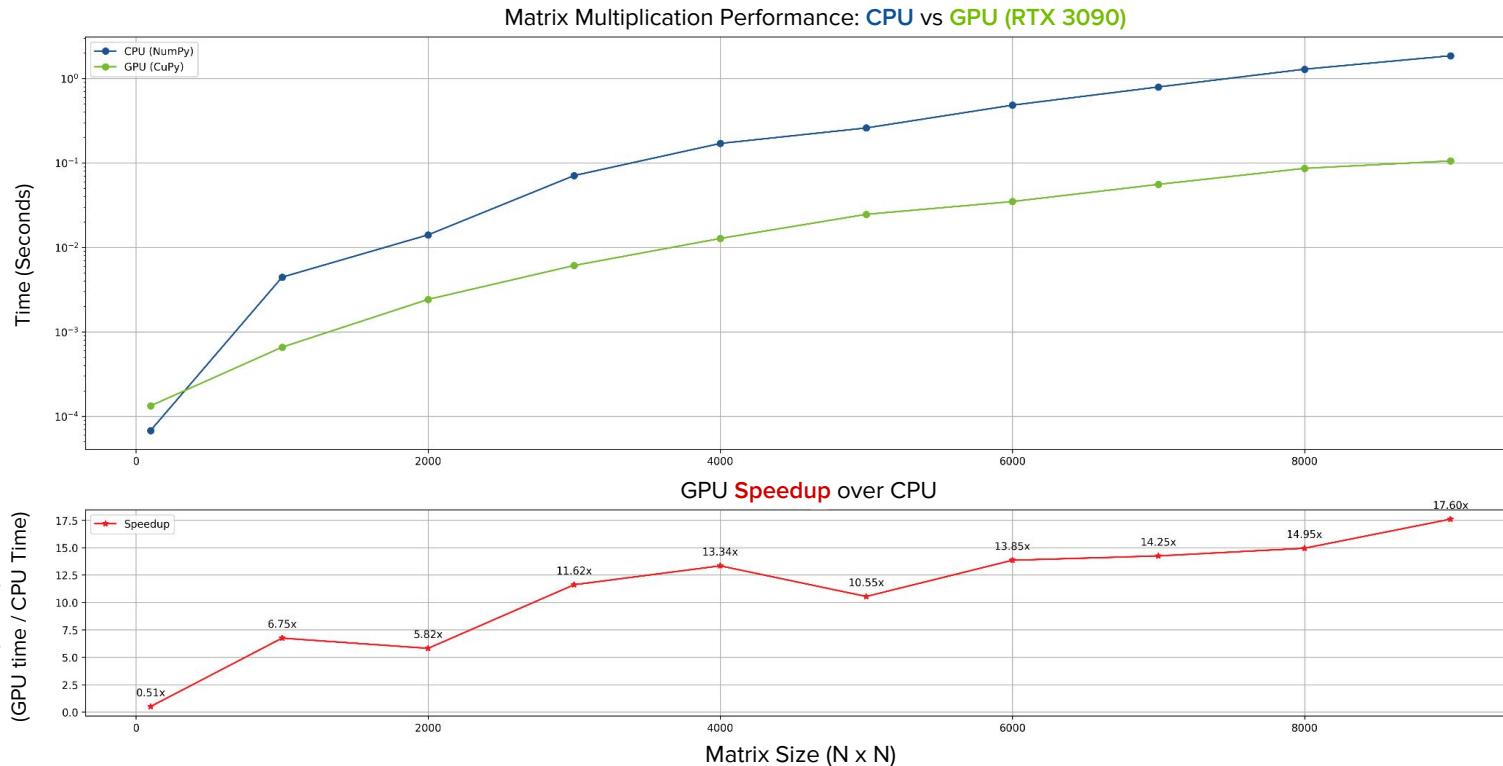
```
y_cpu = np.array([4, 5, 6])  
x_gpu = cp.array([1, 2, 3])  
  
x_gpu + y_cpu          # Errore: tipi incompatibili  
cp.asnumpy(x_gpu) + y_cpu # OK: entrambi su CPU  
x_gpu + cp.asarray(y_cpu) # OK: entrambi su GPU
```

Regola Chiave

- Le operazioni richiedono array dello **stesso tipo** (entrambi CPU o GPU).

Moltiplicazione di Matrici con GPU (CuPy) e CPU (NumPy)

- Confronto prestazionale tra le operazioni `cp.matmul()` e `np.matmul()` per la moltiplicazione di matrici quadrate di dimensione crescente.



Definizione di Kernel in CuPy

User-Defined Kernels

- CuPy permette di definire tre tipi di kernel CUDA (*elementwise*, *reduction* e *raw*).
- Ci focalizziamo sui **raw kernel** che offrono il controllo completo sull'implementazione.

```
# Definizione kernel (compilazione JIT alla prima invocazione)
add_kernel = cp.RawKernel(r'''
extern "C"
__global__ void add_arrays(float *a, float *b, float *c, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) c[idx] = a[idx] + b[idx];
}
'', 'my_add')

# Utilizzo
x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
y = cp.zeros((5, 5), dtype=cp.float32)
add_kernel((grid_size,), (block_size,), (x1, x2, y, N)) # griglia, blocchi, argomenti
```

Definizione di Kernel in CuPy

Raw Module in CuPy

- **RawModule** gestisce codice sorgente CUDA o binari CUDA esistenti. Si inizializza con codice sorgente CUDA o percorso del binario, offrendo accesso ai kernel tramite **get_function()**.

```
# Definizione modulo con più kernel
module = cp.RawModule(code=r'''
extern "C"{
    __global__ void test_sum(const float* x1, const float* x2, float* y, unsigned int N) {
        unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
        if (tid < N) {
            y[tid] = x1[tid] + x2[tid];
        }
    }

    __global__ void test_multiply(const float* x1, const float* x2, float* y,
                                 unsigned int N) {
        unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
        if (tid < N) {
            y[tid] = x1[tid] * x2[tid];
        }
    }
}
'''')
```

Caricamento da file

```
module = cp.RawModule(path='path/to/kernels.cubin')
```

Definizione di Kernel in CuPy

Raw Module in CuPy

- **RawModule** gestisce codice CUDA esteso o binari esistenti. Si inizializza con codice sorgente CUDA o percorso del binario, offrendo accesso ai kernel tramite **get_function()**.

```
# Recupero e utilizzo dei kernel
ker_sum = module.get_function('test_sum')
ker_times = module.get_function('test_multiply')

# Test dei kernel
N = 10
x1 = cp.arange(N**2, dtype=cp.float32).reshape(N, N)
x2 = cp.ones((N, N), dtype=cp.float32)
y = cp.zeros((N, N), dtype=cp.float32)

ker_sum((N,), (N,), (x1, x2, y, N**2))      # y = x1 + x2
ker_times((N,), (N,), (x1, x2, y, N**2))     # y = x1 * x2
```

Confronto fra PyCUDA e CuPy

	PyCUDA	CuPy
Approccio Base	Wrapper Python per CUDA che mantiene controllo di basso livello.	Replica NumPy su GPU con automazione delle operazioni.
Livello di Astrazione	Basso livello, richiede scrittura diretta di kernel CUDA.	Alto livello, operazioni simili a NumPy e SciPy.
Gestione della Memoria	Gestione perlopiù manuale dei trasferimenti host-device.	Gestione automatica con memory pool.
Compilazione	JIT dei kernel definiti come stringhe CUDA C.	Ibrido: librerie pre-compilate (cuBLAS, cuFFT, ecc.) + JIT per kernel custom.
Integrazione	Richiede codice esplicito per interagire con NumPy.	Compatibilità diretta con NumPy/SciPy.
Curva Apprendimento	Richiede conoscenza CUDA C/C++.	Richiede solo conoscenza NumPy e SciPy.
Supporto Hardware	GPU NVIDIA	GPU NVIDIA, con supporto ROCm (AMD).
Manutenibilità	Codice più lungo, richiede più manutenzione.	Codice conciso, più facile da mantenere.

Librerie CUDA e Applicazioni

➤ Introduzione alle Librerie CUDA

- Cos'è una Libreria
- Principali Librerie CUDA

➤ Utilizzo delle Librerie CUDA

- Workflow di Importazione e Utilizzo
- Esempi Pratici con cuBLAS e cuRAND

➤ CUDA in Python

- PyCUDA
- CuPy
- Accenni ad altri Framework

➤ Applicazioni

- AI e Computer Vision
- Opportunità di Tesi

Applicazioni CUDA

Perché CUDA è Importante?

- CUDA è diventata una tecnologia fondamentale in innumerevoli campi applicativi:
 - **Intelligenza Artificiale & Deep Learning:** Training e inferenza di reti neurali.
 - **Calcolo Scientifico:** Simulazioni fisiche, analisi dei dati, chimica computazionale.
 - **Crypto & Blockchain:** Mining e validazione.
 - **Media & Intrattenimento:** Rendering 3D, elaborazione video, gaming.

Opportunità nel CVLab

- Il **Computer Vision Laboratory (CVLab)** della nostra università si occupa di Computer Vision utilizzando tecniche di deep learning, con focus su:
 - **Comprensione della Scena 3D:** Monocular Depth, Multi-View/Binocular Stereo, SLAM.
 - **Analisi della Scena:** Segmentazione semantica, Object Detection, Tracking.
 - **Novel View Synthesis:** NeRF, 3D Gaussian Splatting.
 - **AI Generativa:** Generazione di immagini e video tramite Diffusion Model.

Tesi e Progetti

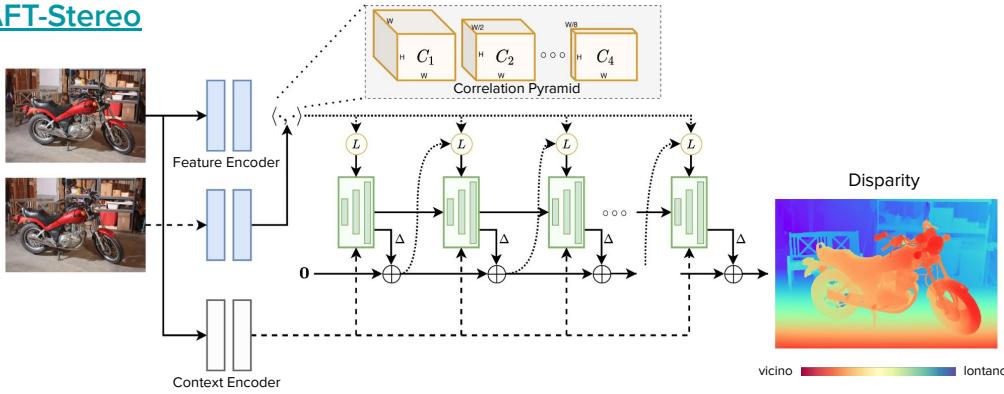
- Siamo aperti a **studenti (fortemente) interessati a svolgere la tesi su questi temi**. Le competenze CUDA acquisite nel corso sono direttamente applicabili nei nostri progetti di ricerca, dove:
 - **Ideiamo e implementiamo** nuove soluzioni per problemi di visione artificiale.
 - Utilizziamo **GPU NVIDIA** e **framework** basati su CUDA (PyTorch).
 - Sviluppiamo **moduli CUDA personalizzati** per ottimizzare algoritmi specifici.

Deep Stereo Matching

- Date due immagini rettificate della stessa scena, la profondità viene calcolata tramite triangolazione, determinando la **disparità** come la differenza tra le coordinate orizzontali dei punti corrispondenti.

RAFT-Stereo

Left Image
Right Image

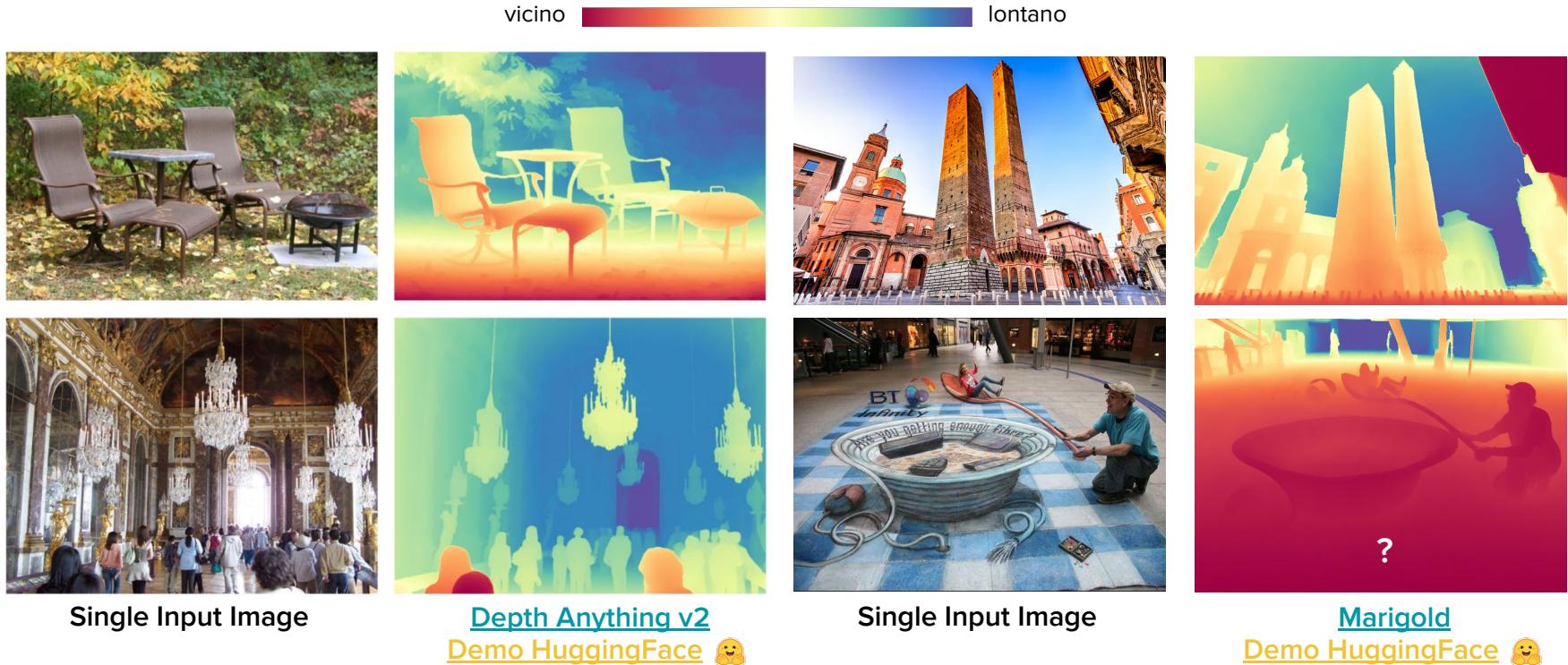


sampler_kernel.cu

```
13 #define BLOCK 16
14
15 __forceinline__ __device__ bool within_bounds(int h, int w, int H, int W) {
16     return h >= 0 && h < H && w >= 0 && w < W;
17 }
18
19 template <typename scalar_t>
20 __global__ void sampler_forward_kernel(
21     const torch::PackedTensorAccessor32<scalar_t, 4, torch::RestrictPtrTraits> volume,
22     const torch::PackedTensorAccessor32<float, 4, torch::RestrictPtrTraits> coords,
23     torch::PackedTensorAccessor32<scalar_t, 4, torch::RestrictPtrTraits> corr,
24     int r)
25 {
26     // batch index
27     const int x = blockIdx.x * blockDim.x + threadIdx.x;
28     const int y = blockIdx.y * blockDim.y + threadIdx.y;
29     const int n = blockIdx.z;
30
31     const int h1 = volume.size(1);
32     const int w1 = volume.size(2);
33     const int w2 = volume.size(3);
34
35     if (!within_bounds(y, x, h1, w1)) {
36         return;
37     }
38
39     float x0 = coords[n][0][y][x];
40     float y0 = coords[n][1][y][x];
41
42     float dx = x0 - floor(x0);
43     float dy = y0 - floor(y0);
44
45     int rd = 2*r + 1;
46     for (int i=0; i<rd; i++) { // i is X
47         int x1 = static_cast<int>(floor(x0)) - r + i;
48
49         if (within_bounds(0, x1, 1, w2)) {
50             scalar_t s = volume[n][y][x][x1];
51
52             if (i > 0)
53                 corr[n][i-1][y][x] += s * scalar_t(dx);
54
55             if (i < rd)
56                 corr[n][i][y][x] += s * scalar_t((1.0f-dx));
57         }
58     }
59 }
60 }
```

Single-view Depth Estimation

- A partire da una **singola immagine**, la profondità (depth) viene stimata utilizzando **reti neurali** addestrate su dataset di coppie immagine-profoundità, sfruttando indizi visivi prospettiva, dimensioni relative, ombre, texture etc.



Single-view Depth Estimation

- A partire da una **singola immagine**, la profondità (depth) viene stimata utilizzando **reti neurali** addestrate su dataset di coppie immagine-profoundità, sfruttando indizi visivi prospettiva, dimensioni relative, ombre, texture etc.

Python Code - Depth Anything v2

```
import cv2
import torch

from depth_anything_v2.dpt import DepthAnythingV2

DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

model_configs = {
    'vits': {'encoder': 'vits', 'features': 64, 'out_channels': [48, 96, 192, 384]},
    'vitb': {'encoder': 'vitb', 'features': 128, 'out_channels': [96, 192, 384, 768]},
    'vitl': {'encoder': 'vitl', 'features': 256, 'out_channels': [256, 512, 1024, 1024]},
    'vitg': {'encoder': 'vitg', 'features': 384, 'out_channels': [1536, 1536, 1536, 1536]}
}

encoder = 'vitl' # or 'vits', 'vitb', 'vitg'
model = DepthAnythingV2(**model_configs[encoder])
model.load_state_dict(torch.load(f'checkpoints/depth_anything_v2_{encoder}.pth', map_location='cpu'))
model = model.to(DEVICE).eval()

raw_img = cv2.imread('your/image/path')
depth = model.infer_image(raw_img) # HxW raw depth map in numpy
```

Temporally Consistent Monocular Depth Estimation

- Estende la stima monoculare della profondità a **sequenze video**, garantendo coerenza temporale tra i frame consecutivi per produrre una ricostruzione 3D fluida e stabile nel tempo.

vicino  lontano



Monocular Input Video

Output Depth



Monocular Input Video



Output Depth

[Rolling Depth](#)

[Demo HuggingFace](#) 😊

Novel View Synthesis

- Tecnica che permette di generare **nuove viste di una scena** a partire da una o più immagini esistenti, ricostruendo la geometria 3D e le texture per renderizzare viewpoint mai osservati.

3D Gaussian Splatting



rasterizer.cu

```
| 236 dim3 tile_grid((width + BLOCK_X - 1) / BLOCK_X, (height + BLOCK_Y - 1) / BLOCK_Y, 1);
| 237
| 238
| 239 // Dynamically resize image-based auxiliary buffers during training
| 240 size_t img_chunk_size = requiredImageState->width * height;
| 241 char* img_chunkptr = imageBuffer(img_chunk_size);
| 242 ImageState imgState = ImageState::fromChunk(img_chunkptr, width * height);
| 243
| 244
| 245 if (NUM_CHANNELS != 3 && colors_precomp == nullptr)
| 246 {
| 247     throw std::runtime_error("For non-RGB, provide precomputed Gaussian colors!");
| 248 }
| 249
| 250 // Run preprocessing per-Gaussian (transformation, bounding, conversion of SHs to RGB)
| 251 CHECK_CUDA(FORWARD)::preprocess(
| 252     P, D, M,
| 253     means3D,
| 254     (glm::vec3*)scales,
| 255     scale_modifier,
| 256     (glm::vec4*)rotations,
| 257     opacities,
| 258     shs,
| 259     geomState.clamped,
| 260     cov3D_precomp,
| 261     colors_precomp,
| 262     viewmatrix, projmatrix,
| 263     (glm::vec3*)cam_pos,
| 264     width, height,
| 265     focal_x, focal_y,
| 266     tan_fovx, tan_fovy,
| 267     radii,
| 268     geomState.means2D,
| 269     geomState.depths,
| 270     geomState.cov3D,
| 271     geomState.rgb,
| 272     geomState.conic_opacity,
| 273     tile_grid,
| 274     geomState.tiles_touched,
| 275     prefiltered,
| 276     antialiasing
| 277 ), debug)
```

Novel View Synthesis

- Tecnica che permette di generare **nuove viste di una scena** a partire da una o più immagini esistenti, ricostruendo la geometria 3D e le texture per renderizzare viewpoint mai osservati.

*Stereo training data from image sequences collected
with a single handheld camera*



[NeRF-Stereo](#) / [Instant-NGP](#)

render_buffer.cu

```
332 __global__ void overlay_depth_kernel(
333     ivec2 resolution,
334     float alpha,
335     const float* __restrict__ depth,
336     float depth_scale,
337     ivec2 image_resolution,
338     int fov_axis,
339     float zoom,
340     vec2 screen_center,
341     cudaSurfaceObject_t surface
342 ) {
343     uint32_t x = threadIdx.x + blockDim.x * blockIdx.x;
344     uint32_t y = threadIdx.y + blockDim.y * blockIdx.y;
345
346     if (x >= resolution.x || y >= resolution.y) {
347         return;
348     }
349
350     float scale = image_resolution[fov_axis] / float(resolution[fov_axis]);
351
352     float fx = x + 0.5f;
353     float fy = y + 0.5f;
354
355     fx -= resolution.x * 0.5f; fx /= zoom; fx += screen_center.x * resolution.x;
356     fy -= resolution.y * 0.5f; fy /= zoom; fy += screen_center.y * resolution.y;
357
358     float u = (fx - resolution.x * 0.5f) * scale + image_resolution.x * 0.5f;
359     float v = (fy - resolution.y * 0.5f) * scale + image_resolution.y * 0.5f;
360
361     int srcx = floorf(u);
362     int srcy = floorf(v);
363     uint32_t srcidx = srcx + image_resolution.x * srcy;
364
365     vec4 color;
366     if (srcx >= image_resolution.x || srcy >= image_resolution.y || srcx < 0 || srcy < 0) {
367         color = {0.0f, 0.0f, 0.0f, 0.0f};
368     } else {
369         float depth_value = depth[srcidx] * depth_scale;
370         vec3 c = colormap_turbo(depth_value);
371         color = {c[0], c[1], c[2], 1.0f};
372     }
373
374     vec4 prev_color;
375     surf2DRead<float4>(&prev_color, surface, x * sizeof(float4), y);
376     color = color * alpha + prev_color * (1.f - alpha);
377     surf2DWrite<float4>(color, surface, x * sizeof(float4), y);
378 }
```

SLAM (Simultaneous Localization and Mapping)

- Tecnica che permette a un sistema di costruire una **mappa dell'ambiente** e contemporaneamente determinare la propria posizione all'interno di essa, utilizzando sensori come telecamere RGB-D per l'elaborazione in tempo reale.



GLORIE-SLAM

droid_kernels.cu

```
| 427 __global__ void projmap_kernel(
| 428     const torch::PackedTensorAccessor32<float,2,torch::RestrictPtrTraits> poses,
| 429     const torch::PackedTensorAccessor32<float,3,torch::RestrictPtrTraits> disps,
| 430     const torch::PackedTensorAccessor32<float,1,torch::RestrictPtrTraits> intrinsics,
| 431     const torch::PackedTensorAccessor32<long,1,torch::RestrictPtrTraits> ii,
| 432     const torch::PackedTensorAccessor32<long,1,torch::RestrictPtrTraits> jj,
| 433     torch::PackedTensorAccessor32<float,4,torch::RestrictPtrTraits> coords,
| 434     torch::PackedTensorAccessor32<float,4,torch::RestrictPtrTraits> valid)
| 435     {
| 436
| 437         const int block_id = blockIdx.x;
| 438         const int thread_id = threadIdx.x;
| 439
| 440         const int ht = disps.size(1);
| 441         const int wd = disps.size(2);
| 442
| 443         __shared__ int ix;
| 444         __shared__ int jx;
| 445
| 446         __shared__ float fx;
| 447         __shared__ float fy;
| 448         __shared__ float cx;
| 449         __shared__ float cy;
| 450
| 451         __shared__ float ti[3], tj[3], tij[3];
| 452         __shared__ float qij[4], qij[4];
| 453
| 454         // load intrinsics from global memory
| 455         if (thread_id == 0) {
| 456             ix = static_cast<int>(ii[block_id]);
| 457             jx = static_cast<int>(jj[block_id]);
| 458             fx = intrinsics[0];
| 459             fy = intrinsics[1];
| 460             cx = intrinsics[2];
| 461             cy = intrinsics[3];
| 462         }
| 463
| 464         __syncthreads();
```

Semantic Segmentation

- Tecnica di computer vision che mira a **suddividere un'immagine in regioni semanticamente significative**, assegnando a ogni pixel una specifica **classe o categoria** (es. persona, auto, strada, edificio, cielo).

Segmenting objects across images and videos



[Segment Anything v2](#)

Generative AI

- Modelli generativi come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche secondo specifiche richieste, mantenendo coerenza fotorealistica.**

Midjourney



V1 (Feb 2022)



V2 (Apr 2022)



V3 (Jul 2022)



V4 (Nov 2022)



V5 (Mar 2023)



V5.1 (May 2023)



V5.2 (Jun 2023)



V6 (Dec 2023)

Generative AI

- Modelli generativi come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche secondo specifiche richieste, mantenendo coerenza fotorealistica.**

Create Anything in 4D with Multi-View Video Diffusion Models



CAT4D

Generative AI

- Modelli generativi come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche secondo specifiche richieste, mantenendo coerenza fotorealistica.**

ControlNet

Adding Conditional Control to Text-to-Image Diffusion Models

Demo HuggingFace 😊



Control (Edges)



Original Image



Prompt: “*cinematic, luxury apartment, colourful, highly detailed*”



Prompt: “*cinematic, cyberpunk apartment out of steel and concrete, colourful, highly detailed*”

Generative AI

- Modelli generativi come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche secondo specifiche richieste, mantenendo coerenza fotorealistica.**

Genie

A large-scale foundation world model



Prompt: The video shows a first person perspective of someone navigating difficult terrain in the middle of a volcanic area. This is a real world video shot from the perspective of a wheeled robot that needs to traverse across a terrain. The vehicle has chunky offroad tires that crunch under the blackened rock. The camera is an egocentric camera mounted to the vehicle, and you can see the front tires just on the bottom of the camera along with the body of the robot. In the distance you can see smoke and lava flowing from the volcano. There are no other visible signs of life. There are lava pools that the agent is trying to avoid and random rock formations. The sky is a vivid blue.



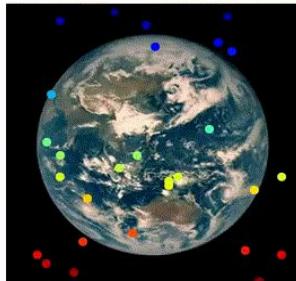
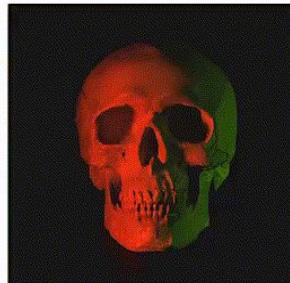
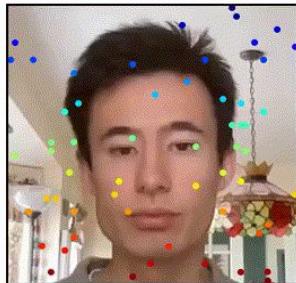
Prompt: POV action camera of a tan house being painted by a first person agent with a paint roller

Generative AI

- Modelli generativi come GAN e diffusion models vengono utilizzati per **sintetizzare e manipolare immagini realistiche secondo specifiche richieste, mantenendo coerenza fotorealistica.**

Motion Prompting

Controlling Video Generation with Motion Trajectories



Motion Transfer

Object Control

Riferimenti Bibliografici

Testi Generali

- Cheng, J., Grossman, M., McKercher, T. (2014). **Professional CUDA C Programming**. Wrox Pr Inc. (1^a edizione)
- Kirk, D. B., Hwu, W. W. (2013). **Programming Massively Parallel Processors**. Morgan Kaufmann (3^a edizione)
- Brian Tuomanen. (2018). **Hands-On GPU Programming with Python and CUDA**. Packt Publishing

NVIDIA Docs

- CUDA Programming:
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- CUDA C Best Practices Guide
<http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>

Risorse Online

- Corso GPU Computing (Prof. G. Grossi): Dipartimento di Informatica, Università degli Studi di Milano
 - <http://gpu.di.unimi.it/lezioni.html>