Università degli studi di
Milano-Bicocca

Sistemi di calcolo parallelo

Project

Calcolo del coefficiente di correlazione di Pearson per time series in Python CUDA e OPENMP

Author:
Radice Carlo - 807159

September 21, 2020

1 Introduzione

In questa relazione viene mostrato un approccio basato su tecniche di calcolo parallelo per la computazione del coefficiente di correlazione di Pearson[1] per time series.
In ambito medico, *functional magnetic resonance imaging* (fMRI)[2] è una tecnica di scansione del cervello non invasiva che viene utilizzata per lo studio delle attività funzionali. Una ben nota misura per analizzare queste attività è il coefficiente di correlazione di Pearson. Il suo utilizzo permette di sviluppare algoritmi in grado di studiare a fondo la dinamica delle connessioni presenti all'interno del cervello umano.
Poichè questo tipo di scansioni generano una grande quantità di dati, l'utilizzo di tecniche basate su CPU ha portato ad avere tempi di attesa molto elevati per l'ottenimento di risultati.
Di conseguenza, sono state sviluppate tecniche basate su GPU per l'ottimizzazione del tempo di esecuzione.
In particolare, in questo progetto, mi sono concentrato sullo studio della tecnica *Fast-GPU-PCC*[1], inizialmente implementata in CUDA C++, e successivamente implementata da me in CUDA Python e in OPENMP.

[1] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
[2] https://en.wikipedia.org/wiki/Functional_magnetic_resonance_imaging

2 Descrizione generale della tecnica utilizzata

In questa sezione viene descritta la tecnica utilizzata per poter effettuare in modo efficiente il calcolo del coefficiente di correlazione di Pearson per time series con grandi quantità di dati.

Quando viene effettuata una fMRI, vengono acquisite da uno scanner una serie di immagini del soggetto nel tempo. I dati di queste scansioni consistono in centinaia di milioni di componenti chiamati *voxels*. Un voxel è il più piccolo elemento indicizzabile del cervello. Esso rappresenta una regione tridimensionale del cervello stesso, comprendendo milioni di neuroni.
Alterazioni emodinamiche all'interno del cervello corrispondono a variazioni dell'intensità dei voxel.
Quindi, tenendo traccia nel tempo della variazione di intensità, è possibile estrarre una time serie da ogni voxel.
Una tecnica molto utilizzata per lo studio di queste time series è il coefficiente di correlazione di Pearson (PCC). La PCC computa l'associazione lineare tra due variabili x e y T dimensionali.
La formula è la seguente:

$$\rho_{x_y} = \frac{\sum_{i=1}^{T}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{T}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{T}(y_i - \bar{y})^2}} \tag{1}$$

Il valore che si ottiene dalla formula è nel range compreso tra -1 e +1. Il valore -1 indica una perfetta relazione lineare negativa, 0 indica che non è prensente una relazione lineare tra x e y, mentre il valore +1 indica una perfetta relazione lineare positiva tra le due varibili.
Nel caso in studio dell'fMRI, x e y rappresentano due voxel, ognuno con T data points nella rispettiva time serie.
Nel corso degli anni sono stati studiati diverse tecniche per poter utilizzare questo tipo di dato per ricavare informazioni per diversi studi medici, tra i quali la diagnosi di malattie quali l'Alzheimer o in generale disordini della cognizione.
Tuttavia, a causa del tempo di computazione necessario per calcolare la PCC a coppie di voxel, sono stati applicati modelli più semplici che implicano l'uso di gruppi di voxel o regioni del cervello invece dei singoli voxel, e si è passati dalla computazione su singolo processo a quella parallela.
L'approccio utilizzato si basa sul fatto che il coefficiente di correlazione di

Pearson abbia la proprietà di *simmetria* ovvero: *corr(x,y) = corr(y,x)*. Da questa proprietà si può ricavare che tutte le correlazioni tra N elementi possono essere rappresentate tramite un array di dimensione pari a N(N - 1)/2. Questo fà si che non si debbano salvare in memoria N^2 elementi, risparmiando notevolmente spazio di memoria nel caso in cui le time series sono molto lunghe.

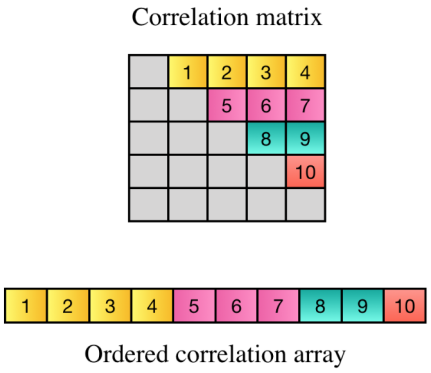

Figure 1: matrice di correlazione e rispettiva rappresentazione monodimensionale

Come si può vedere in figura 1 viene consiederata solo la matrice triangolare superiore in quanto gli elementi della diagonale sono scartati perchè rappresentano la correlazione con sé stessi mentre gli elementi della matrice triangolare inferiore hanno gli stessi valori in quanto la matrice è simmetrica. Inoltre viene mostrata la rappresentazione con array monodimensionale della matrice.
La seconda considerazione che è stata adottata, mostrata nell'articolo studiato, è quella di poter ottenere il coefficiente di correlazione di Pearson come prodotto di una matrice U con la sua trasposta U^{T} se tutte le time series dei voxel vengono normalizzate tramite la seguente equazione:

$$u_i = \frac{v_i - \bar{v}_i}{||v_i - \bar{v}_i||_2} \tag{2}$$

Nell'equazione v_{i} rappresenta la time serie del voxel i-esimo, mentre u_{i} rappresenta la time serie normalizzata.
Tutti i voxel sono quindi aggregati nella matrice $U = [u_1, u_2, u_3, ..]$. Infine, la matrice U viene moltiplicata per la sua trasposta U^{T} così da ottenere la

matrice di correlazione.
Al variare della dimensione della matrice e della disponibilità di memoria potrebbe essere necessario dividere la matrice in blocchi più piccoli per poter eseguire il prodotto. La tecnica basata su GPU di divisione e salvataggio dei risultati utilizzata verrà spiegata meglio nella sezione successiva.

3 Descrizione del metodo su GPU

Fino ad adesso si è parlato in modo molto generale di come viene calcolata la matrice di correlazione. In questa sezione ci si concentrerà nel spiegare in dettaglio i vari passi che portano ad ottenere il vettore rappresentante la matrice triangolare superiore.

All'inizio, su CPU, vengono normalizzate le time series utilizzando l'equazione (2).
Una volta ottenuta la matrice U di dimensione $N\ x\ M$ dove N è il numero di voxels e M è la lunghezza delle time series, si trasferisce il suo contenuto sulla memoria globale della GPU. A questo punto si presentano due casi:

1. la matrice di correlazione è abbastanza piccola da poter essere salvata completamente sulla memoria globale della GPU;

2. la matrice di correlazione è troppo grande per poter essere salvata completamente sulla memoria globale della GPU.

Nel primo caso non ci sono problemi in quanto si può ottenere il prodotto tra le matrici complete. L'algoritmo finisce dopo solo 1 iterazione.
Nel secondo caso, invece, è necessario effettuare il prodotto in più iterazioni così che ad ogni passo viene effettuato il prodotto su GPU di una parte della matrice, viene salvato e viene liberata la memoria per l'iterazione successiva.
In entrambi i casi bisogna calcolare uno spazio su GPU tale da poter permettere oltre a caricare la matrice anche di salvare momentariamente i coefficienti riordinati della matrice di correlazione, rappresentata in forma monodimensionale, prima di passare su CPU.
A livello di memoria è necessario tenere in considerazione che bisogna salvare:

1. la matrice delle time series normalizzate che ha dimensione NxM;

2. il prodotto che ha dimensione N^2 (poichè si moltiplica una matrice NxM con una MxN);

3. la matrice triangolare che ha dimensione N(N-1)/2.

Di conseguenza, la memoria necessaria per salvare i dati, effettuare la computazione della matrice di correlazione e riordinare l'array in GPU, è pari a:

$$N^2 + \frac{N(N-1)}{2} + NM \tag{3}$$

Se il valore è più piccolo della memoria globale della GPU la computazione viene eseguita in una iterazione, altrimenti sono necessarie più iterazioni.
La figura 2 mostra il processo in questo secondo caso.

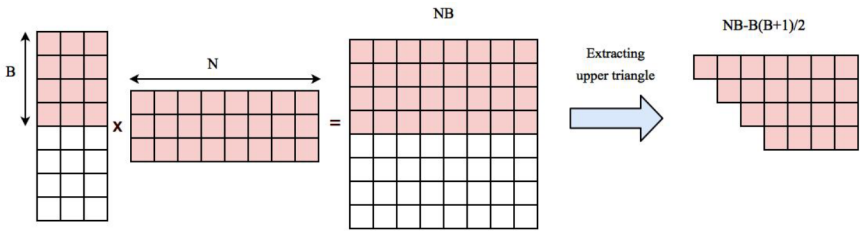

Figure 2: divisione della matrice ad ogni iterazione se troppo grande

Viene effettuata la correlazione della matrice di dimensione B voxel con tutta la matrice. Viene calcolata la matrice triangolare superiore, salvata nell'array monodimensionale e trasferito su CPU. Lo spazio necessario ad ogni iterazione è:

$$NM + NB + NB - \frac{B(B+1)}{2} \tag{4}$$

Al variare di B varia quanto spazio in memoria GPU rimane libero.
Di conseguenza B si può calcolare come X ovvero lo spazio in memoria libero diviso 2N.

$$B = \frac{X}{2N} \tag{5}$$

Se questo valore è maggiore di N allora non si è in grado di poter eseguire l'algoritmo in una sola iterazione.
Ora distinguiamo più dettagliamente i due casi.

3.1 Caso 1: la correlazione viene eseguita in una iterazione

Una volta calcolata la matrice di correlazione si può ottenere da essa la matrice triangolare superiore assegnando ad ogni cella presente nella parte

triangolare superiore un thread GPU che quindi copierà il valore in una cella specifica dell'array monodimensionale. L'id del tread viene ottenuto in questo modo: *idx = blockDim.x * blockIdx.x + threadIdx.x*. La figura 3 mostra come avviene il calcolo dell'indice dell'array e il successivo salvataggio.
Infine, l'array viene copiato su CPU.

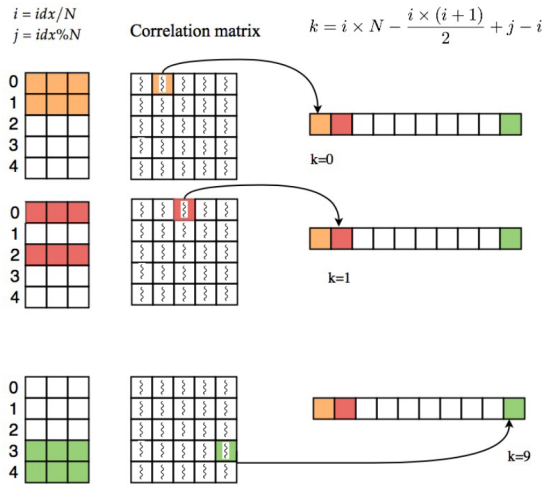

Figure 3: rappresentazione visuale del salvataggio nell'array monodimensionale

3.2 Caso 2: la correlazione viene eseguita in più iterazioni

Come abbiamo visto nella sezione precedente, in questo caso non si può effettuare il calcolo di tutta la matrice in una sola iterazione in quanto non si dispone della memoria necessaria. Quindi per ogni sottomatrice di correlazione si ottiene la matrice triangolare superiore e poi i valori vengono scritti nell'array monodimensionale che in questo caso viene riempito, man mano, ad ogni iterazione. Vengono, quindi, ricalcolati i vari parametri per poter eseguire la successiva iterazione oppure viene terminata l'esecuzione se si è arrivati a calcolare l'ultima sottomatrice. Infine, anche in questo caso l'array viene copiato su CPU.

4 Python CUDA

La prima implementazione effettuata consiste nella computazione del coefficiente di correlazione di Pearson, per time series, in python.

4.1 Librerie

Si è scelto, dopo aver effettuato diverse sperimentazioni, di utilizzare *PyCuda*[2] come principale libreria di python per cuda. Infatti, questa libreria permette di poter utilizzare funzionalità in cuda scritte nativamente in C++ e di conseguenza avere sia velocità di esecuzione, completezza grazie alle API ma anche convenienza grazie alle astrazioni Python. Inoltre permette di utilizzare le stesse funzioni del codice sviluppato in C++ e di conseguenza si ha un miglior confronto a livello di prestazioni nei due casi.
Altre librerie utilizzate sono *numpy*[3] per la rappresentazione dei vettori, *skcuda*[4] per poter usare nativamente codice scritto in C++ in Python, *math*[5] per funzioni matematiche e infine *time*[6] per tener traccia del tempo di esecuzione del codice.

4.2 Scelte tecniche

Di seguito è presente un elenco delle varie scelte tecniche usate:

- tutte le variabili sono state inizializzate come np.float32 ovvero float a 32 bit così che si conosca esattamente la loro occupazione in memoria;
- si è scelto di importare codice scritto in C++ per poter ottimizzare il tempo di calcolo della funzione che copia il contenuto della matrice triangolare superiore;
- in presenza di cicli innestati si è cercato il più possibile di ottimizzare il tempo di esecuzione utilizzando strutture Python più efficienti.

[3]https://numpy.org
[4]https://scikit-cuda.readthedocs.io/en/latest/
[5]https://docs.python.org/3/library/math.html
[6]https://docs.python.org/3/library/time.html

4.3 Funzioni

In questa sezione verranno trattate le funzioni che sono state definite nei due file per il calcolo della matrice di correlazione.

4.3.1 CPU_side.py

- **main**: viene caricato il file contenente le time series, istanziate le matrici *BOLD* ed *upper_tri* in CPU, calcolata la memoria rimanente e in base al suo valore viene chiamata la funzione *cor_mat_2* se basta una iterazione per ottenere l'array con la matrice di correlazione oppure *cor_mat_3* se sono necessarie più iterazioni. Infine viene scritto su file l'array;

- **remaining_mem**: permette di calcolare la memoria rimanente se viene istanziata la matrice nella GPU. Viene calcolata dividendo per 32 (dimensione delle variabili tipo float) e sottrando al valore la dimensione della matrice NxL dove N è il numero di voxel ed L è la lunghezza della time serie.

4.3.2 GPU_side.py

- **cor_ma_2**: effettua il calcolo della matrice di correlazione nel caso in cui c'è abbastanza spazio nella GPU per effettuare il prodotto in una sola iterazione. Per fare ciò viene usata la funzione *cublasSgemm* che permette di svolgere prodotti di matrici in modo molto efficiente. Successivamente come descritto nella sezione precedente si ottiene dal risultato la matrice triangolare superiore che viene scritta in un array monodimensionale. L'array ora presente in memoria globale della GPU viene poi copiato in memoria RAM;

- **cor_ma_3**: effettua il calcolo della matrice di correlazione nel caso in cui non ci sia abbastanza spazio nella GPU per effettuare il prodotto in una sola iterazione. Di conseguenza, è presente un ciclo while al suo interno che effettua le varie iterazioni allocando ad ogni passo una parte della matrice per effettuare il prodotto ed ottenere la sottomatrice triangolare superiore. Viene riempito in parte l'array *upper_tri* fino a che all'ultima iterazione viene riempito tutto. L'array ora presente in memoria globale della GPU viene poi copiato in memoria RAM;

- **preprocessing**: svolto su CPU, effettua la normalizzazione delle time series come descritto nella sezione 3;
- **remaining_N2**: permette di calcolare la memoria rimanente ad ogni iterazione.

5 C OPENMP

La seconda implementazione effettuata consiste nella computazione del coefficiente di correlazione di Pearson, per time series, in C OPENMP.

5.1 Librerie

Le librerie incluse comprendono librerie standard come *stdio* e *stdlib*, *math* per funzioni matematiche e infine *omp* per le funzioni specifiche dell'implementazione ma anche per l'uso di *omp_get_wtime* per ottenere il calcolo del tempo di esecuzione delle varie sezioni.

5.2 Scelte tecniche

Si considera la matrice come array unidimensionale dove l'elemento in posizione i nella matrice bidimensionale NxL (N=numero di voxel, L=lunghezza della time serie) è rappresentato dall'elemento in posizione i*L+j nell'array unidimensionale.
Inoltre si è deciso di utilizzare al più 10 GB di memoria RAM per l'esecuzione, sui 16 GB disponibili, in quanto si è voluto lasciare un buffer di RAM per non causare problemi come swap che possono ridurre notevolmente l'efficienza, in termini di tempo, del codice.

5.3 Funzioni

In questa sezione verranno trattate le funzioni che sono state definite nel singolo file per il calcolo della matrice di correlazione.

5.3.1 CPU_side.c

- **CorMat_2**: effettua il calcolo della matrice di correlazione nel caso in cui c'è abbastanza spazioo nella memoria RAM per effettuare il prodotto in una sola iterazione. Per fare ciò viene usata la struttura di OPENMP che parallelizza il ciclo for. Si ottiene il risultato del prodotto e tramite un'altra sezione parallelizzata viene estapolata la matrice triangolare superiore. Infine viene liberata la memoria precedentemente allocata;

- **CorMat_3**: effettua il calcolo della matrice di correlazione nel caso in cui non ci sia abbastanza spazio nella memoria RAM per effettuare il prodotto in una sola iterazione. Anche in questa implementazione è presente un ciclo while che effettua le varie iterazioni allocando ad ogni passo una struttura che contiene una parte della matrice di correlazione. Viene poi ottenuta la parte di matrice triangolare superiore e copiata nell'array *upper_tri* fino a che nell'ultima iterazione viene riempito tutto.Infine viene liberata la memoria precedentemente allocata;

- **main**: viene caricato il file contenete le time series, istanziate le matrici *BOLD*, e *upper_tri*, calcolata la memoria RAM rimanente e in base al suo valore viene chiamata la funzione *CorMat_2* se basta una iterazione per ottenere l'array con la matrice di correlazione oppure *CorMat_3* se sono necessarie più iterazioni. Infine viene scritto su file l'array;

- **preprocessing**: effettua la normalizzazione delle time series come descritto nella sezione 3;

- **remaininig_mem**: permette di calcolare la memoria RAM rimanente, a partire da un valore predefinito dall'utente (in questo caso 10 GB), per capire se si deve effettuare il calcolo della matrice di correlazione in una o più iterazioni;

- **remaining_N2**: permette di calcolare la memoria rimanente ad ogni iterazione.

6 Risultati

Per l'hardware e software utilizzati fare riferimento all'appendice A, mentre per il codice fare riferimento all'appendice B.
Per effettuare i diversi confronti tra linguaggi di programmazione e implementazioni viene utilizzato uno script Python che crea un file contenente il numero di voxel e la loro lunghezza in base ai parametri che prende in input. Si è scelto di eseguire diversi esperimenti variando la quantità di voxel considerati ma mantenendo la lunghezza delle time series pari a 100. Inoltre l'intervallo di valore delle singole misurazioni dei voxel varia tra -6 e +6.

Vengono mostrati ora i vari esperimenti.

6.1 Python CUDA

Per avere un quadro più specifico sulle performance del codice si è pensato di calcolare diversi tempi:

- **Preprocessing**: rappresenta il tempo necessario ad effettuare la fase di preprocessing dei dati su CPU;
- **HostToDevice**: rappresenta il tempo necessario per copiare la matrice dalla memoria RAM nella memoria globale della GPU unito all'inizializzazione delle strutture in memoria GPU;
- **CorrMat**: rappresenta il tempo necessario per effettuare il calcolo della matrice di correlazione;
- **MatrixToVector**: rappresenta il tempo necessario per ottenere e copiare la matrice triangolare superiore nel vettore monodimensionale;
- **Total**: rappresenta il tempo totale dell'esecuzione escludendo il tempo necessario ad effettuare la stampa su file, se richista.

Da 30 000 voxels in poi, l'esecuzione avviene utilizzando la funzione *mat_cor_3*. Di consenguenza non basta più una sola iterazione per ottenere la matrice di correlazione.

In tabella 1 sono presenti i tempi delle esecuzioni in Python CUDA mentre in tabella 2 sono presenti i tempi delle esecuzioni in C++ CUDA.

#Voxel	Preprocessing	HostToDevice	CorrMat	MatrixToVector	Total
100	0.005	0.0003	0.134	0.012	0.15
1000	0.050	0.0014	0.139	0.014	0.20
10000	0.500	0.0820	0.130	0.110	0.83
20000	0.998	0.3230	0.133	0.465	1.93
30000	1.462	0.6642	0.134	1.755	4.08
40000	1.910	1.1271	0.133	3.143	6.45
50000	2.473	1.5852	0.138	4.919	9.45

Table 1: Tabella dei tempi di esecuzione di Python CUDA

#Voxel	Preprocessing	HostToDevice	CorrMat	MatrixToVector	Total
100	0.00007	0.00002	0.00005	0.00005	0.10
1000	0.00055	0.00014	0.00012	0.00064	0.10
10000	0.00531	0.00122	0.00794	0.03849	0.16
20000	0.01071	0.00318	0.03074	0.22736	0.38
30000	0.01650	0.46270	0.05475	0.38778	0.87
40000	0.02244	0.48825	0.09582	1.01749	1.76
50000	0.02683	0.65842	0.10985	1.13414	2.26

Table 2: Tabella dei tempi di esecuzione di C++ CUDA

Come si può vedere, in generale, il tempo di esecuzione totale per quanto riguarda Python è sempre maggiore rispetto a C++.
Nello specifico si può notare come gran parte del deficit che accumula Python è concentrato principalmente nella fase di *Preprocessing* ma anche nelle fasi *HostToDevice* e *MatrixToVector*. Uno dei motivi per cui fase di preprocessing è così onerosa in termini di tempo è dovuta al fatto che sono presenti diversi cicli innestati e di conseguenza l'ottimizzazione in Python ne risente rispetto a C++. La fase *CorrMat* ha all'incirca lo stesso tempo di esecuzione in entrambi i casi.
Le figure in seguito mostrano il confronto tra i vari parametri di tempo utilizzati.

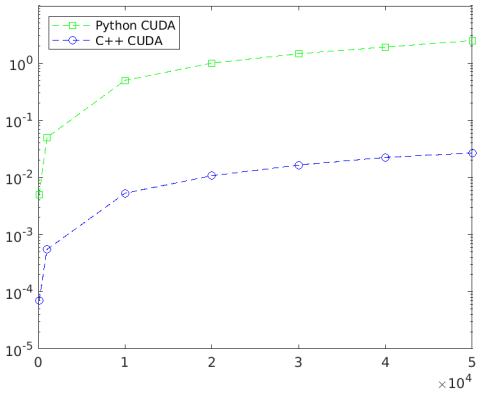

Figure 4: Preprocessing

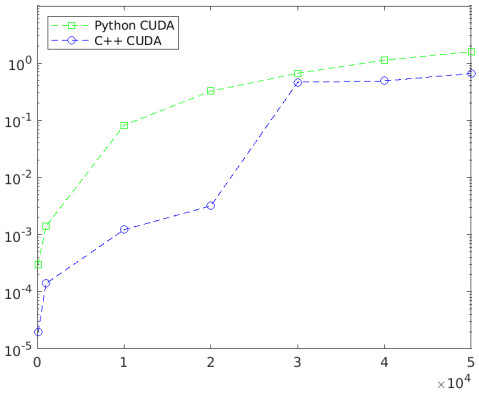

Figure 5: HostToDevice

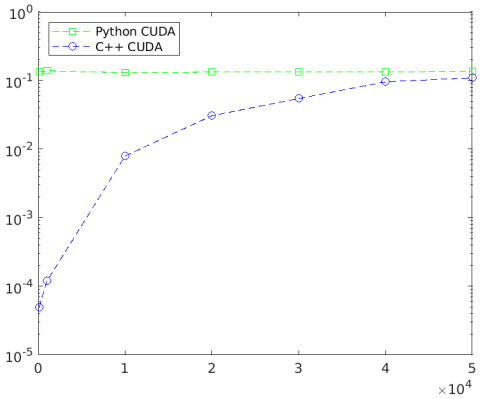

Figure 6: CorrMat

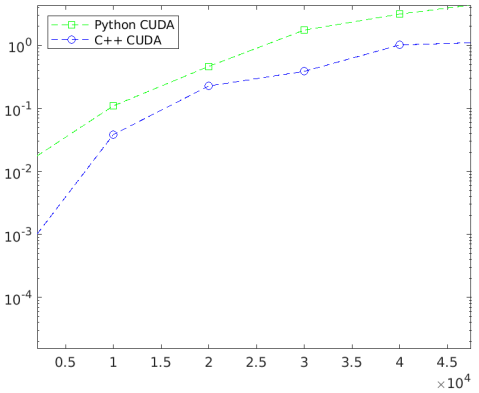

Figure 7: MatrixToVector

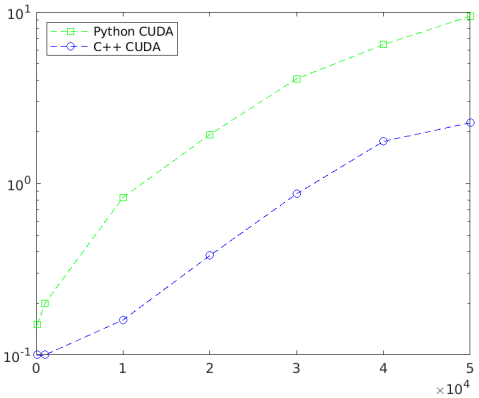

Figure 8: Tempo totale

6.2 C OPENMP

La seconda implementazione che è stata effettuata riguarda OPENMP in linguaggio C. Non potendo utilizzare GPUs il codice è stato implementato mantenendo la fase di preprocessing del metodo ma modificando la sua struttura per il calcolo della matrice di correlazione.

#Voxel	Preprocessing	CorrMat	UpperTri	Total
100	0.00011	0.00240	0.00003	0.002
1000	0.00071	0.11684	0.00637	0.125
10000	0.00750	11.8260	0.36946	12.22
20000	0.01509	48.5726	1.38228	50.02
30000	0.02232	112.175	3.38338	115.6
40000	0.02899	209.093	5.53110	214.8
50000	0.03657	268.174	7.60408	281.7

Table 3: Tabella dei tempi di esecuzione di C OPENMP

Si può notare da subito come in figura 3 i tempi di esecuzione sono nell'ordine delle centinaia di secondi per quanto riguarda le matrici più grandi. Infatti, benchè, il tempo di *Preprocessing* ed ottenimento della matrice triangolare superiore *UpperTri* siano dello stesso ordine di grandezza di Python CUDA, il tempo impiegato per effettuare il prodotto matriciale è ordini di grandezza superiore e di conseguenza ha un peso decisamente elevato nel calcolo del tempo totale. L'utilizzo della CPU per effettuare questo tipo di operazioni ha portato ad avere performance molto inferiori alle GPU che grazie al loro maggior numero di core (CUDA core, in questo caso) permettono una miglior parallelizzazione.
Le figure in seguito mostrano il confronto tra l'implementazione in Python CUDA e C OPENMP. Poichè non tutti i parametri di tempo coincidono, per motivi intrinseci all'implementazione, vengono considerati quelli comuni o che si possono accomunare.

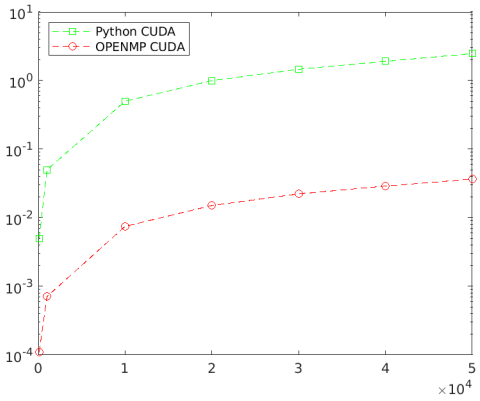

Figure 9: Preprocessing

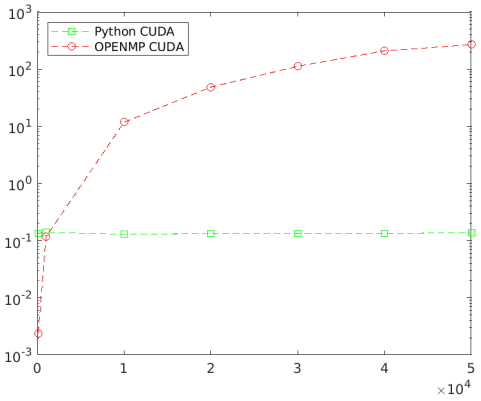

Figure 10: CorrMat

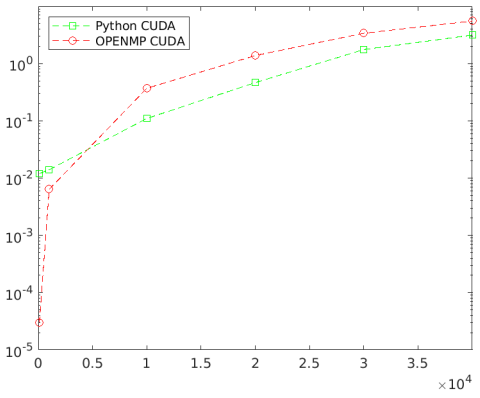

Figure 11: MatToVec Vs UpperTri

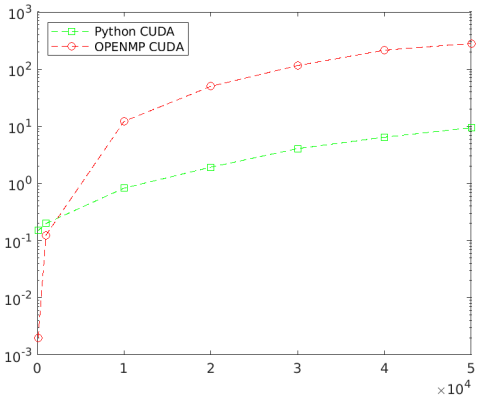

Figure 12: Total

7 Conclusioni

In questo progetto è stata approfondita la tecnica fMRI (functional magnetic resonance imaging) per lo studio delle attività funzionali del cervello. Per poter elaborare i dati sottoforma di serie temporali di due o più voxel, viene applicato il coefficiente di correlazione di Pearson che è dimostrato in letteratura essere un buon indicatore della correlazione lineare tra variabili.
Il metodo consiste nell'uso di GPUs per la computazione di prodotti di matrici molto grandi per ottenere come risultato la matrice di correlazione. Il codice sviluppato in Python CUDA effettua proprio questa operazione ritornando in output il vettore che rappresenta la sottomatrice triangolare superiore della matrice di correlazione.
Il codice Python ottiene dei buoni risultati in termini di prestazioni ed utilizzo ottimale della memoria. Inoltre, è stato eseguito il confronto con il codice originario scritto in C++. Si è notato come l'ottimizzazione migliore di un linguaggio di programmazione come C++ permette di ottenere tempi di esecuzione inferiori usando strutture simili tra i due linguaggi.
Successivamente è stata effettuata l'implementazione usando OPENMP in linguaggio C. Si è visto come questa implementazione permetta di poter avere tempi di esecuzione anche superiori C++ per quanto riguarda il *preprocessing*, tuttavia poichè viene usata solo la CPU, ottenendo una minore parallelizzazione del processo, il tempo per effettuare il calcolo della matrice di correlazione è ordini di grandezza superiori rispetto sia al più efficiente C++ ma anche a Python CUDA.

Appendices

A Hardware & Software

Hardware:

- **processore**: Quad-Core Intel® Core™ i5-4690K CPU @ 3.50GHz;
- **memoria RAM**: 16 GB;
- **scheda grafica**: NVIDIA Corporation GM204 [GeForce GTX 970].

Software:

- **sistema operativo**: elementary OS 5.1.7 Hera (Built on Ubuntu 18.04.4 LTS);
- **versione di Python**: Python 3.6.9
- **versione di GCC**: 7.5.0
- **versione di pycuda**: 2019.1.2
- **versione di scikit-cuda**: 0.5.3
- **versione di numpy**: 1.19.1

B Codice

In questa sezione dell'appendice viene mostrato il codice sviluppato in Python CUDA e C OPENMP.

B.1 Python

B.1.1 CPU_side.py

```
import numpy as np
import pycuda.driver as cuda
from GPU_side import cor_mat_2, cor_mat_3
import time

# questa funzione controlla che ci sia abbastanza memoria per
    calcolare la matrice
def remaining_mem(N, L, flag):
  meminfo = cuda.mem_get_info()
  print("free: %s bytes, total, %s bytes" % (meminfo[0], meminfo
      [1]))
  available_mem = float(meminfo[0])
  available_mem /= np.dtype(np.float32).itemsize
  NL = N * L
  if flag is 0:
    available_mem -= NL
  x = available_mem
  temp = N * 2
  x /= temp
  return int(x)


def main():
  # salvo la matrice come numpy array
  BOLD = np.array(np.loadtxt("/home/carlo/Documents/progetto-
      calcolo-parallelo/random_matrix.txt"), np.float32)

  # ottengo il numero di voxel e la lunghezza della time serie
      dalle dimensioni della matrice
  N = BOLD.shape[0]
  L = BOLD.shape[1]
  print("Number of voxels: ", N)
  print("Length of time series: ", L)
```

```
    # creazione matrice triangolare superiore e inizializzazione a
        zero
    size = int(((N-1) * N) / 2)
    upper_tri = np.zeros(size, np.float32)
    # print("upper_tri shape:", upper_tri.shape)

    # calcolo memoria necessaria per la matrice
    rem_mem = remaining_mem(N, L, 0)
    print("Remaining mem:", rem_mem)

    print("Computing correlations ...")
    # se la matrice occupa meno memoria del totale
    if N <= rem_mem:

      print("cor_mat_2")
      start_time = time.time()
      upper_tri = cor_mat_2(BOLD, upper_tri, N, L)
      stop_time = time.time()
      delta = stop_time - start_time
      print("Running time for computing correlations: ", delta, "\
          n")

    # se la matrice occupa piu memoria del totale
    if N > rem_mem:

      print("cor_mat_3")
      start_time = time.time()
      upper_tri = cor_mat_3(BOLD, upper_tri, N, L, rem_mem)
      stop_time = time.time()
      delta = stop_time - start_time
      print("Running time for computing correlations: ", delta, "\
          n")

    # print to file
    nome_file = "/home/carlo/Documents/progetto-calcolo-parallelo/
        python_gpu_pcc_corr.txt"
    upper_tri.tofile(nome_file, sep="\n", format="%.7f")


if __name__ == '__main__':
    main()
```

B.2 GPU_side.py

```
#from numba import vectorize, cuda as cd, float32
```

```
import numpy as np
import math
import time
#import cupy as cp
import skcuda.cublas as cublas
import  pycuda.gpuarray  as  gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import pycuda.compiler


def remaining_N2(N, L, available_mem):
  x = available_mem
  temp = N
  temp *= 2
  x /= temp
  return int(x)


# preprocessing della matrice in CPU
def preprocessing(BOLD, N, L):
  for i in range(0, N):
    B = BOLD[i,:]
    sum1 = sum(B) / L
    sum2 = np.sqrt(sum(np.power((B - sum1), 2)))
    if sum2 is not 0:
      B = (B - sum1) / sum2
    else:
      B = 0
    BOLD[i, :] = B
  return BOLD


def cor_mat_2(BOLD, upper_tri, N, L):
  # preprocessing fMRI data in CPU
  start_time = time.time()
  BOLD = preprocessing(BOLD, N, L)
  stop_time = time.time()
  delta = stop_time - start_time
  print("Running time for preprocessing: ", delta, "\n")

  alpha = np.float32(1.0)
  beta = np.float32(0.0)

  # passaggio su device
```

```
start_time = time.time()
BOLD_device = gpuarray.to_gpu(BOLD)
result = np.zeros((BOLD.shape[0], BOLD.shape[0]), np.float32)
result_device = gpuarray.to_gpu(result)
# print("BOLD_device shape:", BOLD_device.shape)
# print("result_device shape:", result_device.shape)
stop_time = time.time()
delta = stop_time - start_time
print("Running time matrices to device: ", delta, "\n")

start_time = time.time()
h = cublas.cublasCreate()
cublas.cublasSgemm(h,
                   'T',
                   'n',
                   N,
                   N,
                   L,
                   alpha,
                   BOLD_device.gpudata,
                   L,
                   BOLD_device.gpudata,
                   L,
                   beta,
                   result_device.gpudata,
                   N)
stop_time = time.time()
delta = stop_time - start_time
print("Running time core function: ", delta, "\n")

start_time = time.time()
threads_per_block = 1024
blocks_per_grid = int(math.ceil(1 + ((N*N - 1) /
    threads_per_block)))
mod = pycuda.compiler.SourceModule("""
  __global__ void ker(float * cormat, float * upper,int n1,int
      n)
  {
    long idx = blockDim.x*blockIdx.x+threadIdx.x;
    long i = idx%n1;
    long j = idx/n1;
    if(i<j && i<n1 && j<n)
    {
          long tmp=i;
          tmp*=(i+1);
```

```
            tmp/=2;
            long tmp_2=i;
            tmp_2*=n;
            tmp_2=tmp_2-tmp;
            tmp_2+=j;
            tmp_2-=i;
            upper[tmp_2-1]=cormat[j*n+i];
        }
      }
      """)
    result_device = result_device.reshape(-1)
    # print("result device shape:", result_device.shape)
    upper_tri_device = gpuarray.to_gpu(upper_tri)
    funct = mod.get_function("ker")
    funct(result_device,
        upper_tri_device,
        np.int32(N),
        np.int32(N),
        block=(threads_per_block, 1, 1),
            grid=(blocks_per_grid, 1)
            )
    upper_tri = upper_tri_device.get()
    stop_time = time.time()
    delta = stop_time - start_time
    print("Running time to get upper tri: ", delta, "\n")

    cublas.cublasDestroy(h)

    return upper_tri


def cor_mat_3(BOLD, upper_tri, N, L, OOO):
    # calcolo memoria disponibile
    meminfo = cuda.mem_get_info()
    print("free: %s bytes, total, %s bytes" % (meminfo[0], meminfo
        [1]))
    available_mem = float(meminfo[0])
    available_mem /= np.dtype(np.float32).itemsize
    available_mem -= N * L
    # print("Available memory: ", available_mem)

    # preprocessing fMRI data in CPU
    start_time = time.time()
    BOLD = preprocessing(BOLD, N, L)
    stop_time = time.time()
```

```
delta = stop_time - start_time
print("Running time for preprocessing: ", delta, "\n")


# passaggio di BOLD in device
start_time = time.time()
BOLD_device = gpuarray.to_gpu(BOLD)
stop_time = time.time()
delta = stop_time - start_time
print("Running time matrices to device: ", delta, "\n")

# calcolo memoria disponibile
# meminfo = cuda.mem_get_info()
# print("After BOLD_device free: %s bytes, total, %s bytes" %
    (meminfo[0], meminfo[1]))

# inizializzazione variabili
flag = 1
ii=0
upper_size = (N-1) * N / 2
block = 000
N_prime = N
temp = 0
temp2 = 0
temp3 = 0
pak = 0
so_far = 0
count = 1
temp4 = 0

alpha = np.float32(1.0)
beta = np.float32(0.0)

while flag is 1:
  print("###### ITERAZIONE ", count, " #####")
  # calcolo memoria disponibile
  # meminfo = cuda.mem_get_info()
  # print("After BOLD_device free: %s bytes, total, %s bytes"
     % (meminfo[0], meminfo[1]))

  # print("block: ", block)
  # print("N_prime: ", N_prime)
  # checking for the last chunk
  if block == N_prime:
    flag = 0
```

```
if pak is not 0:
  del dev_upper
  del result_device

temp = block
temp *= (block + 1)
temp /= 2
# M1 is the size of upper triangle part of chunk
M1 = N_prime
M1 *= block
M1 -= temp

M1 = int(M1)

# print("M1: ", M1)

pak += 1

# print("so_far*L: ", so_far*L)
start_time = time.time()
result = np.zeros((block, N_prime), np.float32)
# print("result shape: ", result.shape)

BOLD_device = BOLD_device.reshape(-1)

# allocate memory on the device for the result
result_device = gpuarray.to_gpu(result)
# print("result_device shape: ", result_device.shape)

# # calcolo memoria disponibile
# meminfo = cuda.mem_get_info()
# print("Before cublasSgemm free: %s bytes, total, %s bytes"
    % (meminfo[0], meminfo[1]))
stop_time = time.time()
delta = stop_time - start_time
print("Running time matrices to device: ", delta, "\n")

start_time = time.time()
h = cublas.cublasCreate()
cublas.cublasSgemm(h,
        'T',
        'n',
        block,
        N_prime,
```

```
        L,
        alpha,
        BOLD_device[so_far*L:].gpudata,
        L,
        BOLD_device[so_far*L:].gpudata,
        L,
        beta,
        result_device.gpudata,
        block)
stop_time = time.time()
delta = stop_time - start_time
print("Running time core function: ", delta, "\n")

temp2 = block
temp2 *= N_prime

# calcolo memoria disponibile
# meminfo = cuda.mem_get_info()
# print("free: %s bytes, total, %s bytes" % (meminfo[0],
    meminfo[1]))

# result_device = gpuarray.to_gpu(result1)

start_time = time.time()
threads_per_block = 1024
blocks_per_grid = 1 + math.ceil(((temp2-1) /
    threads_per_block))
grid = (blocks_per_grid, 1)

# print("temp2:", temp2)
# print("threads_per_block: ", threads_per_block)
# print("blocks_per_grid: ", blocks_per_grid)

upper = np.zeros(M1, np.float32)
# print("upper shape:", upper.shape)
dev_upper = gpuarray.to_gpu(upper)

# print("dev_upper shape: ", dev_upper.shape)
# print("result_device shape: ", result_device.shape)

mod = pycuda.compiler.SourceModule("""
  __global__ void ker2(float * cormat, float * upper,int n1,
      int n,long long upper_size,int N,int i_so_far,long long
       M1)
  {
```

```
    long long idx = blockDim.x;
    idx*=blockIdx.x;
    idx+=threadIdx.x;
    long i = idx/n;
    long j = idx%n;

    if(i<j && i<n1 && j<n)// &&i<N &&j<N && idx<(n1*n))
    {
            long long tmp=i;
            tmp*=(i+1);
            tmp/=2;
            long long tmp_2=i;
            tmp_2*=n;
            tmp_2=tmp_2-tmp;
            tmp_2+=j;
            tmp_2-=i;
            long long indexi=n1;
            indexi*=j;
            indexi=indexi+i;
            upper[tmp_2-1]=cormat[indexi];
    }
  }
    """)
# calcolo memoria disponibile
# meminfo = cuda.mem_get_info()
# print("free: %s bytes, total, %s bytes" % (meminfo[0],
    meminfo[1]))

funct = mod.get_function("ker2")
funct(result_device,
    dev_upper,
    np.int32(block),
    np.int32(N_prime),
    np.int64(upper_size),
    np.int32(N),
    np.int32(ii),
    np.int64(M1),
    block=(threads_per_block, 1, 1),
          grid=grid
          )

temp3+=M1
# print("upper_tri shape:", upper_tri.shape)
upper_tri[temp4:temp3] = dev_upper.get()
stop_time = time.time()
```

```
    delta = stop_time - start_time
    print("Running time to get upper tri: ", delta, "\n")

    temp4 += M1
    ii += block

    cublas.cublasDestroy(h)

    so_far += block

    if N_prime > block:
      N_prime = N_prime - block
      block = remaining_N2(N_prime, L, available_mem)
      if N_prime < block:
        block = N_prime

    count += 1

  # liberare la memoria
  del BOLD_device
  del result_device
  del dev_upper

  # calcolo memoria disponibile
  # meminfo = cuda.mem_get_info()
  # print("free: %s bytes, total, %s bytes" % (meminfo[0],
      meminfo[1]))

  return upper_tri
```

B.3 C

B.3.1 CPU_side.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>


long long remaining_mem(int N, int L) {
  long long available_mem = 10000000000;
  // division by float size = 4 byte
  available_mem /= 4;
  available_mem -= N * L;
```

```
  float temp = N * 2;
  available_mem /= temp;
  printf("available_mem: %lld \n", available_mem);
  return available_mem;
}


float remaining_N2(int N_prime, int L, long long rem_mem) {

  long long x = rem_mem;
    long long temp = N_prime;
    temp *= 2;
    x /= temp;
    printf("rem_mem: %lld \n", rem_mem);
    printf("N_prime: %d \n", N_prime);
    printf("x: %lld \n", x);
    return x;
}


void preprocessing(float * BOLD, int N, int L) {
  for(size_t i = 0; i < N; i++) {
    float * row = BOLD + i * L;
    float sum1 = 0, sum2 = 0;
    for(size_t l = 0; l < L; l++) {
      sum1 += row[l];
    }
    sum1 /= L;
    for(size_t l = 0; l < L; l++) {
      sum2 += (row[l] - sum1) * (row[l] - sum1);
    }
    sum2 = sqrt(sum2);
    for(size_t l = 0; l < L; l++) {
      if(sum2 != 0) {
        row[l] = (row[l] - sum1) / sum2;
      }
      else {
        if(sum2 == 0) {
          row[l] = 0;
        }
      }
    }
  }
}
```

```
void CorMat_2(float * BOLD, float * upper_tri, int N, int L) {

  size_t i, j, k;
  float * BOLD_transpose;
  float * result;

  double start_time, stop_time;

  // preprocessing fMRI data
  start_time = omp_get_wtime();
  preprocessing(BOLD, N, L);
  stop_time = omp_get_wtime();
  printf("Running time for preprocessing: %f \n", stop_time -
      start_time);

  BOLD_transpose = (float *) malloc((L * N) * sizeof(float));
  result = (float *) malloc((N * N) * sizeof(float));

  // get BOLD_transpose
  start_time = omp_get_wtime();
  float temp;
  for(i = 0; i < N; i++) {
    for(j = 0; j < L; j++) {
      temp = BOLD[i * L + j];
      BOLD_transpose[j * N + i] = temp;
    }
  }
  stop_time = omp_get_wtime();
  printf("Running time for transpose: %f \n", stop_time -
      start_time);

  // matrix product
  start_time = omp_get_wtime();
  # pragma omp parallel shared (N, L, BOLD, BOLD_transpose,
      result) private (i, j, k)
  {
    # pragma omp for
    for(i = 0; i < N; i++) {
      for(j = 0; j < N; j++) {
        result[i * N + j] = 0;
        for(k = 0; k < L; k++) {
          result[i * N + j] += BOLD[i * L + k] * BOLD_transpose[
              k * N + j];
        }
```

```
      }
    }
  }
  stop_time = omp_get_wtime();
  printf("Running time core function: %f \n", stop_time -
      start_time);

  // get upper triangular matrix
  start_time = omp_get_wtime();
  # pragma omp parallel shared (N, upper_tri, result) private (i
      , j)
  {
    int idx = 0;
    for(i = 0; i < N; i++) {
      for(j = 0; j < N; j++) {
        idx = (N * i) + j - ((i * (i+1)) / 2);
        idx -= (i+1);
        if(i < j) {
          upper_tri[idx] = result[i * N + j];
        }
      }
    }
  }
  stop_time = omp_get_wtime();
  printf("Running time to get upper tri: %f \n", stop_time -
      start_time);

  free(result);
  free(BOLD_transpose);
}


void CorMat_3(float * BOLD, float* upper_tri, int N, int L, long
    long OOO) {

  size_t i, j, k;

  float * BOLD_transpose;
  float * result;
  float * BOLD_section;
  float * upper_section;

  long long available_mem = 10000000000;
  available_mem /= sizeof(float);
  available_mem -= (N*L);
```

```
printf("Available memory: %lld \n", available_mem);

double start_time, stop_time;

// preprocessing fMRI data
start_time = omp_get_wtime();
preprocessing(BOLD, N, L);
stop_time = omp_get_wtime();
printf("Running time for preprocessing: %f \n", stop_time -
    start_time);

BOLD_transpose = (float *) malloc((L * N) * sizeof(float));

// get BOLD_transpose
start_time = omp_get_wtime();
float temporary;
for(i = 0; i < N; i++) {
  for(j = 0; j < L; j++) {
    temporary = BOLD[i * L + j];
    BOLD_transpose[j * N + i] = temporary;
  }
}
stop_time = omp_get_wtime();
printf("Running time for transpose: %f \n", stop_time -
    start_time);

int flag = 1;
int count = 0;
int block = 000;
int N_prime = N;
int so_far = 0;
int pak = 0;
long long M1, temp;
int limit = 0;
long long cormat_fullsize;

while(flag == 1) {
  printf("########## ITERAZIONE %d \n", count);

  if(block == N_prime) {
    flag = 0;
  }

  temp = block;
```

```
temp *= (block + 1);
temp /= 2;
M1 = N_prime;
M1 *= block;
M1 -= temp;

if(pak != 0) {
  free(result); //result = devCormat in C++ CUDA
  free(upper_section);
  free(BOLD_transpose);
}

cormat_fullsize = block;
cormat_fullsize *= N_prime;

printf("block: %d \n", block);
printf("N_prime: %d \n", N_prime);
printf("cormat_fullsize: %lld \n", cormat_fullsize);

result = (float *) malloc(cormat_fullsize * sizeof(float));
upper_section = (float *) malloc(M1 * sizeof(float));

BOLD_section = (float *) malloc(N_prime * L * sizeof(float))
    ;


if (count != 0) {
  printf("count -> 0: get BOLD_section\n");
  // get BOLD_section transpose
  float temporary;
  for(i = 0; i < N_prime; i++) {
    for(j = 0; j < L; j++) {
      temporary = BOLD[(so_far * L) + (i * L) + j];
      BOLD_section[j * N_prime + i] = temporary;
    }
  }
}

pak++;

// matrix product
start_time = omp_get_wtime();
if (count != 0) {
  // caso passo: da dopo la prima iterazione
  # pragma omp parallel shared (block, N_prime, L, so_far,
```

```
      BOLD, BOLD_section, result) private (i, j, k)
  {
    # pragma omp for
    for(i = 0; i < block; i++) {
      for(j = 0; j < N_prime; j++) {
        result[i * N_prime + j] = 0;
        for(k = 0; k < L; k++) {
          result[i * N_prime + j] += BOLD[(so_far * L) + (i
              * L) + k] * BOLD_section[(k * N_prime) + j];
        }
      }
    }

  }
}
else {
  // caso base: prima iterazione
  # pragma omp parallel shared (block, N_prime, L, so_far,
      BOLD, BOLD_transpose, result) private (i, j, k)
  {
    # pragma omp for
    for(i = 0; i < block; i++) {
      for(j = 0; j < N_prime; j++) {
        result[i * N_prime + j] = 0;
        for(k = 0; k < L; k++) {
          result[i * N_prime + j] += BOLD[L * i + k] *
              BOLD_transpose[k * N_prime + j];
        }
      }
    }
  }
}
stop_time = omp_get_wtime();
printf("Running time core function: %f \n", stop_time -
    start_time);

// get upper triangular matrix
start_time = omp_get_wtime();
# pragma omp parallel shared (block, N_prime, upper_section,
     result) private (i, j)
{
  int idx = 0;
  for(i = 0; i < block; i++) {
    for(j = 0; j < N_prime; j++) {
      idx = (N_prime * i) + j - ((i * (i+1)) / 2);
```

```
            idx -= (i+1);
            if(i < j) {
              upper_section[idx] = result[i * N_prime + j];
            }
          }
        }
      }
      stop_time = omp_get_wtime();
      printf("Running time to get upper tri: %f \n", stop_time -
          start_time);

      long long M11 = (N-1);
      M11 *= N;
      M11 /= 2;
      long long idx = 0;
      for(long long index = 0; index < M11; index++) {
        if(upper_tri[index] == 0.000000 && idx < M1) {
          upper_tri[index] = upper_section[idx];
          idx++;
        }
      }

      so_far += block;

      if(N_prime>block) {
              N_prime = N_prime-block;
              block=remaining_N2(N_prime, L, available_mem);

              if(N_prime < block) {//checking last chunk
                block = N_prime;
                printf("END \n");
            }

        }

        printf("block: %d \n", block);
        printf("N_prime: %d \n", N_prime);

      count++;
    }

    free(BOLD_section);
    free(result);
    free(upper_section);
```

```
}


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

  int N, L;
  char wr;

  N = (int) strtol(argv[1], (char **)NULL, 10);
  L = (int) strtol(argv[2], (char **)NULL, 10);
  wr = *argv[3];

  printf("Number of voxels: %d, Length of time series: %d \n", N
      , L);

  double start_time, stop_time;

  float * BOLD;
  BOLD = (float *) malloc((N* L) * sizeof(float));

  printf("Allocated matrix dimension: %ld \n", N*L * sizeof(
      float));

  // open file to read matrix
  FILE *fp;
  fp = fopen("/home/carlo/Documents/progetto-calcolo-parallelo/
      random_matrix.txt", "r");
  // copy matrix in BOLD
  for(size_t k = 0; k < N; k++) {
    for(size_t l = 0; l < L; l++) {
      fscanf(fp, "%f", &BOLD[k*L+l]);
    }
  }
  fclose(fp);

  printf("Copied matrix in BOLD \n");

  long long M11 = (N-1);
  M11 *= N;
  M11 /= 2;

  // creazione matrice triangolare superiore di dimensione (N-1)
      * N / 2
  float * upper_tri;
  upper_tri = (float *) malloc(M11 * sizeof(float));
```

```
  for(long long idx=0; idx<M11; idx++) {
    upper_tri[idx] = 0;
  }

  long long rem_mem = remaining_mem(N, L);

  if(N <= rem_mem) {
    printf("CorMat_2 \n");
    printf("Computing correlations ... \n");
    // inizio calcolo tempo di esecuzione
    start_time = omp_get_wtime();
    // calcolo della matrice triangolare superiore
    CorMat_2(BOLD, upper_tri, N, L);
    stop_time = omp_get_wtime();
    printf("Running time for computing correlations: %f \n",
        stop_time - start_time);
  }

  if(N > rem_mem) {
    printf("CorMat_3 \n");
    printf("Computing correlations ... \n");
    // inizio calcolo tempo di esecuzione
    start_time = omp_get_wtime();
    // calcolo della matrice triangolare superiore
    CorMat_3(BOLD, upper_tri, N, L, rem_mem);
    stop_time = omp_get_wtime();
    printf("Running time for computing correlations: %f \n",
        stop_time - start_time);
  }

  printf("Writing correlation values into the text file ... \n")
      ;
  fp = fopen("/home/carlo/Documents/progetto-calcolo-parallelo/
      openmp_pcc_corrs_TEST.txt", "w");
  for(long long idx=0; idx<M11; idx++) {
      fprintf(fp, "%f \n", upper_tri[idx]);
  }
  fclose(fp);

  free(upper_tri);
  free(BOLD);

}
```

References

[1] T. Eslami and F. Saeed, "Fast-GPU-PCC: A GPU-Based Technique to Compute Pairwise Pearson's Correlation Coefficients for Time Series Data - fMRI Study," *high-throughput*, 2018.

[2] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.