Advanced Computer Architectures Report

Enrico Sgarbanti

VR446095

Contents

0.1 Introduzione

Lo scopo del progetto è realizzare il codice parallelo in CUDA per l'esecuzione dell'operazione di trasposizione di un tensore 3-dimensionale. L'obiettivo è di riuscire a migliorare il codice del file body.py del progetto pytorch-openpose. Nello specifico la trasposizione nella configurazione (1,2,0) eseguita con numpy, libreria python altamente ottimizzata a basso livello.
I test sono stati effettuati con scheda video NVIDIA GeForce GTX 1070 e CPU Intel i5-4690K (4) @3.900GHz e i risultati sono stati ottenuti dalla media di più esecuzioni con l'esclusione della prima.

0.2 Trasposizione

La trasposizione di un tensore 3-dimensionale è data scambiando gli indici nel seguente modo:

Configurazione	Operazione
012	$Output[z][y][x] = Input[z][y][x]$
021	$Output[z][x][y] = Input[z][y][x]$
102	$Output[y][z][x] = Input[z][y][x]$
120	$Output[y][x][z] = Input[z][y][x]$
201	$Output[x][z][y] = Input[z][y][x]$
210	$Output[x][y][z] = Input[z][y][x]$

0.3 Python

In questa prima fase ho esplorato tre differenti approcci all'esecuzione di codice CUDA in python. Per semplicità ho eseguito i test, confrontando la trasposizione di matrice **1024x1024** usando come riferimento i valori ottenuti dall'esecuzione del codice parallelo in CUDA/C++:

dim=1024x1024	**time (ms)**
host	13.6392
kernel	0.0440525
speedup	309.613x

0.3.1 pybind11

Questo approccio consiste nello scrivere un wrapper per poter usare codice c++ in python e viceversa.

dim=16384x16384	**time (ms)**
numpy	0.023603439331054688
c++	2988.4612560272217
cuda	896.3353633880615

dim=1024x1024	**time (ms)**
numpy	0.009298324584960938
c++	7.504463195800781
cuda	117.80309677124023

I risultati evidenziano un qualche tipo di problema in quest'approccio. L'overhead probabilmente è dovuto al passaggio da dati della struttura numpy a quelle in C++. Essendo questo metodo problematico e quello più difficile da gestire è stato subito abbandonato.

0.3.2 pycuda

Questa libreria mette a disposizione delle API per eseguire le tipiche operazioni CUDA (cudaMalloc, cudaMemcpy, ...) e il kernel direttamente in python. In alternativa è possibile scrivere il kernel in CUDA/C++, compilarlo in un file .cubin e poi importarlo in Python.

dim=1024x1024	**time (ms)**
numpy	0.00072479248046875
kernel	0.04473472010344267
reshaping	0.44356346130371094

Dai risultati si nota che il tempo di esecuzione del kernel è molto simile ai valori ottenuti nel test in CUDA/C++. C'è però da sommare a questo il tempo necessario a trasformare i dati 2-dimensionali in 1-dimensionali per il kernel CUDA. Il tutto risulta essere meno performante di numpy.

0.3.3 cupy

Questa libreria, come pycuda, permette di scrivere codice CUDA direttamente in Python, ma ha il grosso vantaggio di usare le stesse interfaccie di Numpy.

dim=1024x1024	**time (ms)**
numpy	0.0007653236389160156
cupy1	0.00034332275390625
cupy2	0.0011385600041830912
custom	0.05046464022248984

Nella tabella dei risultati, **custom** rappresentata il tempo di esecuzione del kernel scritto da me, che risulta molto simile ai valori ottenuti inizialmente in CUDA/C++. Il valori di **cupy1** e **cupy2** sono i tempi di esecuzione del kernel dalla funzione di trasposizione messa a disposizione dalla librearia cupy. Da notare che cupy necessita come input dei dati di cupy, ciò potrebbe portare all'eliminazione dei tempi di reshaping. Quello che cambia da **cupy1** e **cupy2** è semplicemente il metodo utilizzato per la misurazione del tempo. Nel primo caso è stata usata la librearia Python time(), mentre nel secondo i cudaEvents.

0.3.4 Problematiche riscontrare

Come evidenziato dall'ultimo approccio, ho riscontrato un problema nella misurazione del tempo che potrebbe rendere inaffidabili i risultati ottenuti.
Ho realizzato un test (Codice 0.3.4) i cui risultati mostrano come i tempi di esecuzione non crescano con la dimensione dell'input.

```
import time
import numpy as np
from math import ceil

NUM_REPS  = 100
NUM_CROP = ceil(NUM_REPS * 0.1)

def benchmark(dim, input, n_reps, n_crop):
    timeElapsed = list()
    for _ in range(n_reps+n_crop):
        start = time.time()
        res = np.transpose(input)
        end = time.time()
        timeElapsed.append((end-start)*1e3)
    return np.mean(timeElapsed[n_crop:])

# TEST 1
dim = (100, 100)
input = np.arange(np.product(dim), dtype=np.float32).reshape(dim)
t = benchmark(dim, input, NUM_REPS, NUM_CROP)
print(f'{dim}\n->{t}ms\n')

# TEST 2
dim = (1000, 1000)
input = np.arange(np.product(dim), dtype=np.float32).reshape(dim)
t = benchmark(dim, input, NUM_REPS, NUM_CROP)
print(f'{dim}\n->{t}ms\n')

# TEST 3
dim = (10000, 10000)
input = np.arange(np.product(dim), dtype=np.float32).reshape(dim)
t = benchmark(dim, input, NUM_REPS, NUM_CROP)
print(f'{dim}\n->{t}ms\n')

# TEST 4
dim = (100000, 100000)
```

```
input = np.arange(np.product(dim), dtype=np.float32).reshape(dim)
t = benchmark(dim, input, NUM_REPS, NUM_CROP)
print(f'{dim}\n->{t}ms\n')
```

test	dimension	time (ms)
1	100x100	0.0008058547973632812
2	1000x1000	0.008201599121093750
3	10000x10000	0.0007677078247070312
4	100000x100000	MemoryError

0.4 C++

Per la realizzazione del codice CUDA per la trasposizione di un tensore 3-dimensionale ho quindi deciso di usare strettamente C++ in modo da poter valutare con maggiore affidabilità i vari miglioramenti.

0.4.1 Sequential matrix transpose

```
void transpose_cpu(const float* input, float* output, const int* iDim,
                   const int* perm) {
    int oDim[] = {iDim[perm[0]], iDim[perm[1]], iDim[perm[2]]};
    for (int z = 0; z < iDim[0]; ++z) {
        for (int y = 0; y < iDim[1]; ++y) {
            for (int x = 0; x < iDim[2]; ++x) {
                int idx[]  = {z, y, x};
                int odx[]  = {idx[perm[0]], idx[perm[1]], idx[perm[2]]};
                int iIndex = (idx[0] * iDim[1] * iDim[2]) + (idx[1] * iDim[2]) +
                             (idx[2]);
                int oIndex = (odx[0] * oDim[1] * oDim[2]) + (odx[1] * oDim[2]) +
                             (odx[2]);
                output[oIndex] = input[iIndex];
            }
        }
    }
}
```

0.4.2 Parallel matrix transpose

La funzione transpose_simple_kernel (codice 0.4.2) permette di eseguire la trasposizione in qualsiasi configurazione e dimensione.

```
__global__ void transpose_simple_kernel(const float* d_input, float* d_output,
                                        int dimz, int dimy, int dimx, int pz,
                                        int py, int px) {
    int idx[3]  = {blockIdx.z * blockDim.z + threadIdx.z,
                   blockIdx.y * blockDim.y + threadIdx.y,
                   blockIdx.x * blockDim.x + threadIdx.x};
    int iDim[3] = {dimz, dimy, dimx};
    int oDim[3] = {iDim[pz], iDim[py], iDim[px]};
    int odx[3]  = {idx[pz], idx[py], idx[px]};
    int iIndex  = (idx[0] * iDim[1] * iDim[2]) + (idx[1] * iDim[2]) + idx[2];
    int oIndex  = (odx[0] * oDim[1] * oDim[2]) + (odx[1] * oDim[2]) + odx[2];

    if (idx[0] < dimz && idx[1] < dimy && idx[2] < dimx) {
```

```
        d_output[oIndex] = d_input[iIndex];
    }
}
```

Un buon miglioramento di questo codice è ottenibile specializzando il kernel per ogni permutazione possibile (transpose_simple_kernel_template codice 0.4.2), potendo quindi ridurre il numero di operazioni da fare a runtime. Per assicurare che il compilatore generi il codice per ogni configurazione si utilizza la funzione transpose_selector (codice 0.4.2).

```
template<int pz, int py, int px>
__global__ void transpose_simple_kerneltmplt(const float* d_input,
                                             float* d_output, int dimz,
                                             int dimy, int dimx) {
    int idx[3]  = {blockIdx.z * blockDim.z + threadIdx.z,
                   blockIdx.y * blockDim.y + threadIdx.y,
                   blockIdx.x * blockDim.x + threadIdx.x};
    int iDim[3] = {dimz, dimy, dimx};
    int oDim[3] = {iDim[pz], iDim[py], iDim[px]};
    int odx[3]  = {idx[pz], idx[py], idx[px]};
    int iIndex  = (idx[0] * iDim[1] * iDim[2]) + (idx[1] * iDim[2]) + idx[2];
    int oIndex  = (odx[0] * oDim[1] * oDim[2]) + (odx[1] * oDim[2]) + odx[2];

    if (idx[0] < dimz && idx[1] < dimy && idx[2] < dimx) {
        d_output[oIndex] = d_input[iIndex];
    }
}
```

```
void transpose_simple_selector(const dim3& DimGrid, const dim3& DimBlock,
                               const float* d_input, float* d_output,
                               const int* dim, const int* perm) {
    if (perm[0] == 0 && perm[1] == 1 && perm[2] == 2) {
        transpose_simple_kerneltmplt<0, 1, 2>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 0 && perm[1] == 2 && perm[2] == 1) {
        transpose_simple_kerneltmplt<0, 2, 1>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 1 && perm[1] == 0 && perm[2] == 2) {
        transpose_simple_kerneltmplt<1, 0, 2>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 1 && perm[1] == 2 && perm[2] == 0) {
        transpose_simple_kerneltmplt<1, 2, 0>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 2 && perm[1] == 0 && perm[2] == 1) {
        transpose_simple_kerneltmplt<2, 0, 1>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 2 && perm[1] == 1 && perm[2] == 0) {
        transpose_simple_kerneltmplt<2, 1, 0>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    }
}
```

0.4.3 Parallel matrix transpose with shared-memory

Nei metodi precedenti (codice 0.4.2) la lettura dell'input avviene in modo tale da sfruttare la coalescenza della memoria, ma non è lo stesso per la scrittura dell'output.
Nella funzione transpose_shm_template (codice 0.4.3) l'operazione di trasposizione avviene nella shared memory, al fine di sfruttare la coalescenza della memoria anche per l'output.

```
template<int pz, int py, int px>
__global__ void transpose_shm_kerneltmplt(const float* d_input, float* d_output,
                                          int dimz, int dimy, int dimx) {
    __shared__ float buffer[TILE][TILE][TILE];

    int iDim[3] = {dimz, dimy, dimx};
    int x       = blockIdx.x * TILE + threadIdx.x;
    int y       = blockIdx.y * TILE + threadIdx.y;
    int z       = blockIdx.z * TILE + threadIdx.z;
    if (z < iDim[0] && y < iDim[1] && x < iDim[2]) {
        int iIndex     = (z * iDim[1] * iDim[2]) + (y * iDim[2]) + x;
        int threads[3] = {threadIdx.z, threadIdx.y, threadIdx.x};
        buffer[threads[pz]][threads[py]][threads[px]] = d_input[iIndex];
    }
    __syncthreads();

    int oDim[3]    = {iDim[pz], iDim[py], iDim[px]};
    int blocks[3] = {blockIdx.z, blockIdx.y, blockIdx.x};
    x              = blocks[px] * TILE + threadIdx.x;
    y              = blocks[py] * TILE + threadIdx.y;
    z              = blocks[pz] * TILE + threadIdx.z;
    if (z < oDim[0] && y < oDim[1] && x < oDim[2]) {
        int oIndex        = (z * oDim[1] * oDim[2]) + (y * oDim[2]) + x;
        d_output[oIndex] = buffer[threadIdx.z][threadIdx.y][threadIdx.x];
    }
}
```

Figure 1: Occupancy testata con nvvp

0.4.4 Parallel matrix transpose with shared-memory bank-conflict free

La Shared Memory è organizzata in 32 banchi i quali possono essere letti simultaneamente dalle 32 thread del warp. Durante la scrittura in shared memory, word successive vengono mappate in bank differenti in modo da sfruttare l'accesso simultaneo. Se durante l'esecuzione, due thread dello stesso warp devono accedere allo stesso bank si ha un bank conflict. L'idea di questa ottimizzazione è quindi cambiare la grandezza della shared memory, in accordo al vincolo di shared memory massima per blocco, al fine di minimizzare i bank conflicts. Per fare ciò ho realizzato uno script python per trovare il numero di bank conflict per una data dimensione e permutazione e ho preso le dimensioni minime per avere 0 bank conflicts. A questo punto nella funzione transpose_shm_bank_selector (codice 0.4.4) ho passato al template anche le dimensioni della shared memory.

```
template<int pz, int py, int px, int tilez, int tiley, int tilex>
__global__ void transpose_shm_bank_kerneltmplt(const float* d_input,
                                               float* d_output, int dimz,
                                               int dimy, int dimx) {
    __shared__ float buffer[tilez][tiley][tilex];

    int iDim[3] = {dimz, dimy, dimx};
    int x       = blockIdx.x * TILE + threadIdx.x;
    int y       = blockIdx.y * TILE + threadIdx.y;
    int z       = blockIdx.z * TILE + threadIdx.z;
    if (z < iDim[0] && y < iDim[1] && x < iDim[2]) {
        int iIndex     = (z * iDim[1] * iDim[2]) + (y * iDim[2]) + x;
        int threads[3] = {threadIdx.z, threadIdx.y, threadIdx.x};
        buffer[threads[pz]][threads[py]][threads[px]] = d_input[iIndex];
    }
    __syncthreads();

    int oDim[3]    = {iDim[pz], iDim[py], iDim[px]};
    int blocks[3] = {blockIdx.z, blockIdx.y, blockIdx.x};
    x             = blocks[px] * TILE + threadIdx.x;
    y             = blocks[py] * TILE + threadIdx.y;
    z             = blocks[pz] * TILE + threadIdx.z;
    if (z < oDim[0] && y < oDim[1] && x < oDim[2]) {
        int oIndex       = (z * oDim[1] * oDim[2]) + (y * oDim[2]) + x;
        d_output[oIndex] = buffer[threadIdx.z][threadIdx.y][threadIdx.x];
    }
}
```

```
void transpose_shm_bank_selector(const dim3& DimGrid, const dim3& DimBlock,
                                 const float* d_input, float* d_output,
                                 const int* dim, const int* perm) {
    if (perm[0] == 0 && perm[1] == 1 && perm[2] == 2) {
        transpose_shm_bank_kerneltmplt<0, 1, 2, 8, 8, 8>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 0 && perm[1] == 2 && perm[2] == 1) {
        transpose_shm_bank_kerneltmplt<0, 2, 1, 8, 8, 12>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 1 && perm[1] == 0 && perm[2] == 2) {
        transpose_shm_bank_kerneltmplt<1, 0, 2, 8, 9, 8>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 1 && perm[1] == 2 && perm[2] == 0) {
        transpose_shm_bank_kerneltmplt<1, 2, 0, 8, 10, 10>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 2 && perm[1] == 0 && perm[2] == 1) {
        transpose_shm_bank_kerneltmplt<2, 0, 1, 8, 8, 9>
            <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
    } else if (perm[0] == 2 && perm[1] == 1 && perm[2] == 0) {
```

```
            transpose_shm_bank_kerneltmplt<2, 1, 0, 8, 12, 9>
                <<<DimGrid, DimBlock>>>(d_input, d_output, dim[0], dim[1], dim[2]);
        }
}
```

0.5 Risultati

0.5.1 Kernel analysis

permutation	(2,1,0)	(2,0,1)	(1,2,0)	(1,0,2)	(0,2,1)	(0,1,2)
A	216.72	217.641	215.352	217.179	216.262	215.579
B	169.116	172.827	171.237	169.396	169.959	168.838
C	78.7692	146.286	77.9596	192.3	141.534	190.689
D	79.1957	147.55	78.3773	221.405	142.025	216.49
E	122.342	122.635	122.123	122.929	123.299	122.635
F	155.387	156.935	154.566	164.63	167.184	166.234
G	169.818	169.116	164.895	167.047	165.161	166.099

Table 1: Bandwidth (GB/s) di (A) Copy simple, (B) Copy shared-memory, (C) Transpose simple, (D) Transpose simple template, (E) Transpose shared-memory, (F) Transpose shared-memory template, (G) Transpose shared-memory bank-conflict-free template

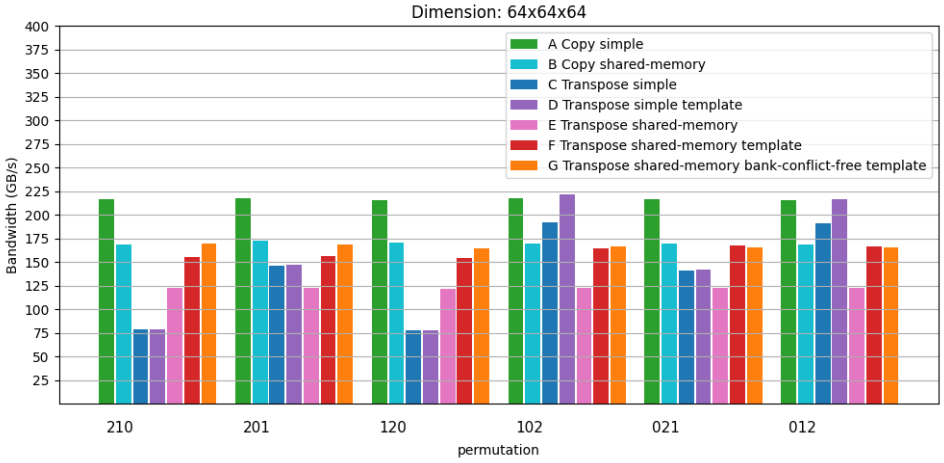

Figure 2: Bandwidth effettiva dei vari kernel provati nelle possibili configurazioni.

L'obiettivo è quello di raggiungere la bandwidth effettiva della copia, cioè del numero di gigabyte spostati per input e output in un secondo.
La bandwidth della copia (A) è uguale in tutte le permutazioni e sfrutta al massimo la coalescenza della memoria quindi risulta essere un buon punto di riferimento per i risultati da raggiungere. L'impatto della coalescenza delle memoria è ben visibile osservando come la bandwidth della trasposa semplice (C) (codice 0.4.2) varia nelle varie permutazioni. Si nota che la badwitdh della permutazione identità 012 con la trasposa semplice (C) è minore di quella della copia semplice (A). Il problema

viene risolto con la versione con template (D) (codice 0.4.2).
A questo punto ho realizzato la copia con shared memory (B) per capire quanta bandwidth potessi raggiungere. La versione con shared memory (E) mostra che c'è ancora un buon margine di miglioramento, raggiungibile grazie all'uso dei template (F) (codice 0.4.3) e all'eliminazione dei bank conflicts (G) (codice 0.4.4).

permutation	**dimension**	**bank conflict**	**dimension**	**bank conflict**
(0,1,2)	(8,8,8)	0.0	(8,8,8)	0.0
(0,2,1)	(8,8,8)	5.3	(8,8,12)	0.0
(1,0,2)	(8,8,8)	5.1	(8,9,8)	0.0
(1,2,0)	(8,8,8)	16.0	(8,10,10)	0.0
(2,0,1)	(8,8,8)	16.0	(8,8,9)	0.0
(2,1,0)	(8,8,8)	16.0	(8,12,9)	0.0

Table 2: bank conflict con la dimensione iniziale di tile 8x8x8 e con gli aggiustamenti

permutation	(2,1,0)	(2,0,1)	(1,2,0)	(1,0,2)	(0,2,1)	(0,1,2)
host	4.12189	3.4565	3.21223	1.5661	1.59269	1.59624
C	0.026624	0.014336	0.0269005	0.0109056	0.0148173	0.0109978
D	0.0264806	0.0142131	0.0267571	0.009472	0.0147661	0.00968704
E	0.0171418	0.0171008	0.0171725	0.0170598	0.0170086	0.0171008
F	0.0134963	0.0133632	0.013568	0.0127386	0.012544	0.0126157
G	0.0123494	0.0124006	0.0127181	0.0125542	0.0126976	0.0126259

Table 3: Time (ms) di Host, (C) Transpose simple, (D) Transpose simple template, (E) Transpose shared-memory, (F) Transpose shared-memory template, (G) Transpose shared-memory bank-conflict-free template

permutation	(2,1,0)	(2,0,1)	(1,2,0)	(1,0,2)	(0,2,1)	(0,1,2)
C	154.818	241.106	119.412	143.605	107.489	145.142
D	155.657	243.191	120.051	165.34	107.862	164.781
E	240.459	202.125	187.057	91.8002	93.6403	93.3431
F	305.408	258.658	236.75	122.941	126.969	126.528
G	333.771	278.736	252.572	124.746	125.433	126.426

Table 4: Speedup di (C) Transpose simple, (D) Transpose simple template, (E) Transpose shared-memory, (F) Transpose shared-memory template, (G) Transpose shared-memory bank-conflict-free template

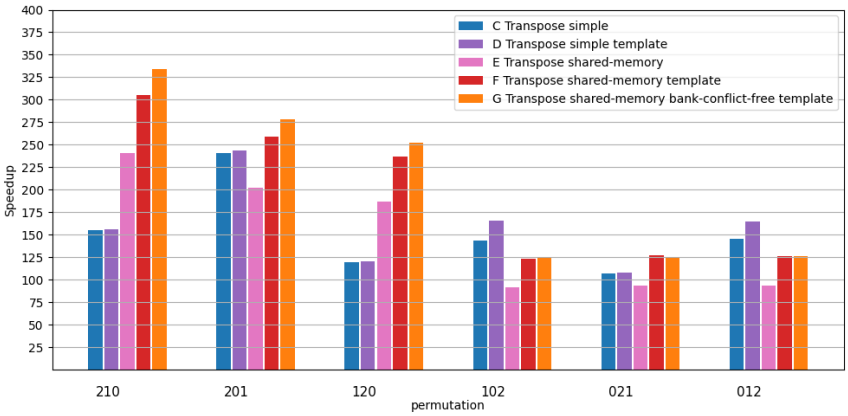

Figure 3: Speedup dei vari kernel provati nelle possibili configurazioni rispetto all'host.

Come evidenziato dalla tabella 3, il tempo di esecuzione del codice sequenziale impiega tempi differenti in base alla configurazione, enfatizzando la differenza di speedup nella tabella 4 e figura 3

0.5.2 Device analysis

permutation	(2,1,0)	(2,0,1)	(1,2,0)	(1,0,2)	(0,2,1)	(0,1,2)
D	11.7406	9.90654	9.05044	4.63124	4.73946	4.52751
F	12.1282	10.0843	9.51553	4.63303	4.67633	4.66133
G	12.2494	9.86121	9.50978	4.64616	4.7512	4.68133

Table 5: Speedup di (D) Transpose simple template, (F) Transpose shared-memory template, (G) Transpose shared-memory bank-conflict-free template

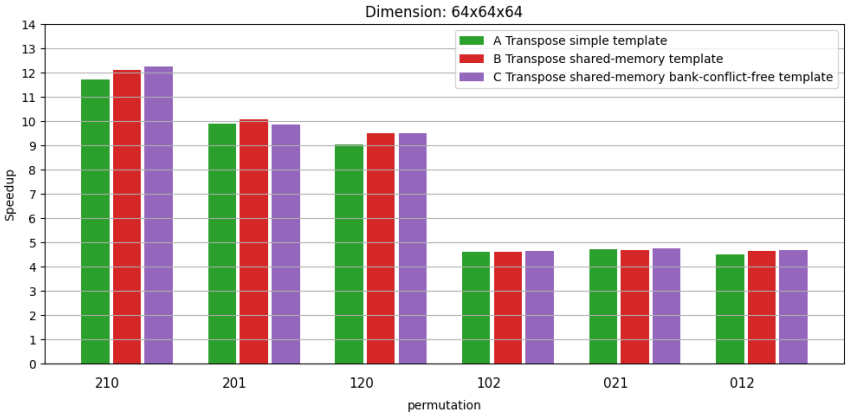

Figure 4: Speedup dei vari kernel provati nelle possibili configurazioni rispetto all'host.

La tabella 4 e figura 3 mostrano i dati sullo speedup, calcolati confrontando il tempo di esecuzione del codice sequenziale e il tempo del codice parallelo comprendendo anche il tempo di spostamento dei dati (cudamemcpy).

0.5.3 Matrix analysis

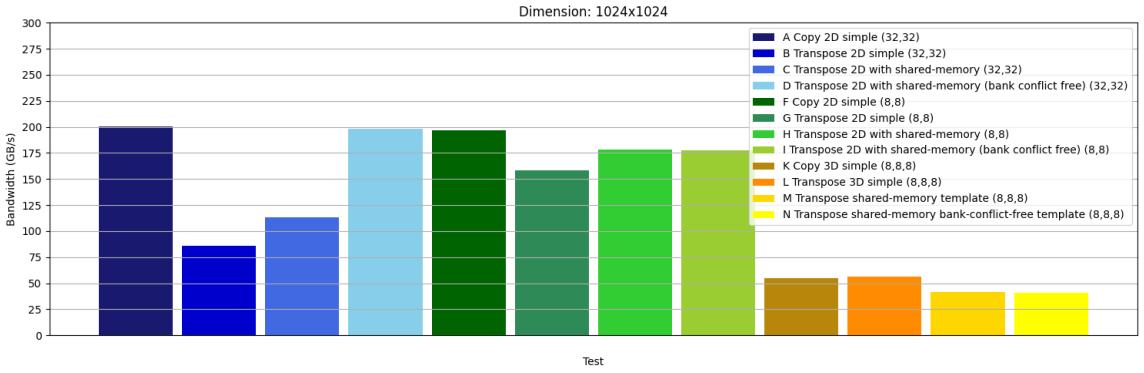

Figure 5: Bandwidth effettiva transpose2D e transpose3D

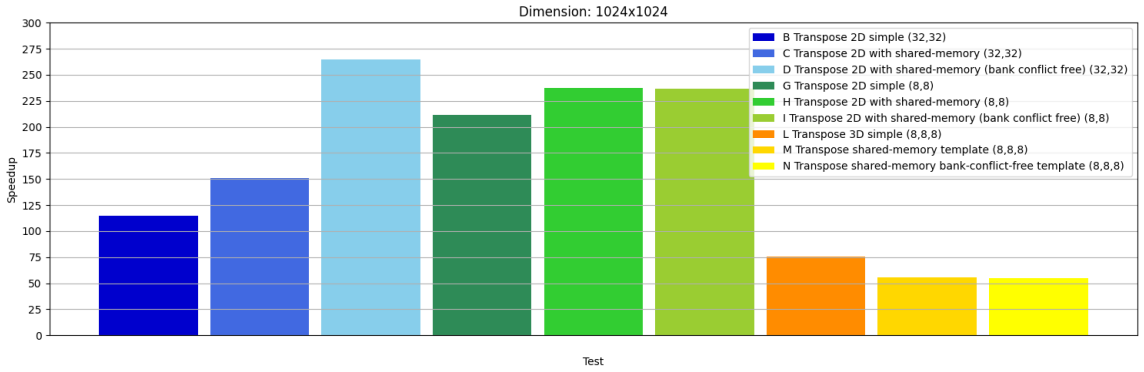

Figure 6: Speedup transpose2D e transpose3D

Come ultimo esperimento ho provato a confrontare la bandwidth effettiva della trasposa di una matrice con TILE 32x32 e 8x8 e della trasposa di tensore 3-dimensionale nella configurazione (0,2,1) con TILE 8x8x8. Quello che si nota è che la versione 3D risulta essere meno efficiente di quella 2D.