Parallel Floyd-Warshall

Alessandro Sebastiano Catinello

a.a. 2022/23

Contents

1 Introduzione

L'algoritmo di Floyd-Warshall è un algoritmo semplice e ampiamente usato, capace di calcolare i percorsi minimi tra tutte le coppie di un grafo pesato e orientato. Ci si pone l'obiettivo di parallelizzare la sua esecuzione sfruttando la GPU riducendo il più possibile i tempi di esecuzione. L'unica limitazione dell'implementazione proposta dovrebbe essere solo la RAM disponibile.

1.1 Floyd-Warshall

Algorithm 1 Floyd-Warshall(M, n)

```
1: M[i, j] = ∞   ∀i ≠ j
2: M[i, i] = 0   ∀i
3: M[i, j] = c(i, j)   ∀(i, j) ∈ E(G)
4:
5: for k = 1 to n do
6:     for i = 1 to n do
7:         for j = 1 to n  do
8:             M[i, j] = min(M[i, j], (M[i, k] + M[k, j]))
9:         end for
10:    end for
11: end for
12:
13: for i = 1 to n do
14:    if(M[i, i] < 0) then return Error("ciclo negativo")
15: end for
```

L'algoritmo di Floyd-Warshall [Algorithm 1] ha una complessità nel caso peggiore di $O(V^3)$ con V pari al numero di vertici nel grafo. È capace di

calcolare il risultato corretto finché non ci sono cicli negativi nel grafo. Nel caso in cui dovesse essere presente un ciclo negativo, il problema dei percorsi minimi diventa NP-Hard e l'algoritmo non è quindi capace di calcolare un risultato valido. È però capace di rilevare la presenza di almeno un ciclo negativo a fine esecuzione: basterà semplicemente controllare la presenza di pesi negativi nella diagonale della matrice M risultante.

L'idea generale dell'algoritmo è quella di scegliere tutti i vertici come intermedi nei percorsi più brevi attraverso N iterazioni, dove N è la larghezza e l'altezza della matrice di adiacenza del grafo in input. Quindi, si testa se il vertice scelto è nel percorso più breve. Se è un intermediario nel percorso, aggiorniamo la distanza, altrimenti manteniamo la distanza originale.

Dato $G(V, E)$ per defininire il grafo stesso, con $V(G) = \{1, ..., n\}$ per definire i vertici e la funzione $c : E(G) \to R$ per definire i pesi degli archi, possiamo rappresentare il grafo come una matrice quadrata M detta "di adiacenza" tale che $M[i, j]$ contiene il peso dell'arco tra i vertici i e j: $M[i, j] = c(i, j) \quad \forall i \neq j$

1.2 Generazione dei grafi

I grafi utilizzati sono generati secondo il modello di generazione di grafi random **Erdős-Rényi**. Usando solo 3 parametri, n numero di vertici, s seed per la generazione di numeri casuali e p percentuale di connessione, l'algoritmo

Algorithm 2 Erdős-Rényi(n, s, p)

```
1: G = Matrix(n × n)
2: srand(s) [Set seed for random number's generation]
3: for i = 1 to n do
4:     for j = 1 to n do
5:         if i == j then
6:             G[i,j] = 0
7:             continue
8:         end if
9:         perc = rand() [random number between 1 - 100]
10:        if perc ≥ p then
11:            G[i, j] = rand() [random number for weight]
12:        else
13:            G[i, j] = ∞
14:        end if
15:    end for
16: end for
17: return G
```

è capace di generare un grafo. Questo sarà formato da n vertici e per ogni possibile coppia si genera un arco con probabilità p, così facendo generiamo casualmente tutti gli archi secondo la densità attesa data dalla probabilità p. Utilizzando questo metodo è stato possibile creare facilmente nuovi grafi relativi ai due fattori più importanti, ovvero il numero di vertici e il numero di archi. Infine, il peso di ogni arco varia tra 1 e 16 per semplificare l'output grafico (quando richiesto), gli archi mancanti invece vengono rappresentati come "infinito", ovvero lo **short** più grande rappresentabile tale da poterne sommare due senza andare in overflow.

1.3 Performance Test

L'algoritmo è implementato in C++ e CUDA e le prestazione sono state misurate in base ai risultati di varie esecuzioni, tutte su una GPU **"Nvidia RTX 3070"** dotata di 8GB di VRAM e CPU **"Intel i9-9900k"**, entrambi impostati con tutti i valori di fabbrica.

In particolare, data la sua natura "dinamica" (programmazione), l'algoritmo è estremamente **memory bound**. L'obiettivo principale sarà quindi quello di velocizzare il più possibile gli accessi in memoria, massimizzando dove possibile la **località** dell'algoritmo, mantenendo comunque però al minimo gli accessi in memoria. In base a quanto detto quindi si misureranno le prestazioni in base al tempo di esecuzione dell'algoritmo e alla banda effettiva prodotta, tenendo conto del limite teorico di banda della memoria della GPU utilizzata pari a **448.0 GB/s**.

2 Implementazione

Si mostrano diversi kernel per la parallelizzazione dell'algoritmo. In particolare:

- Simple parallelize
- Simple parallelize Vectorized
- Simple parallelize Pseudo-Blocked
- Blocked
- Blocked Vectorized

Ognuno di questi kernel ha differenze prestazionali in base ai parametri a esso passato, a tal fine è stata inserita la possibilità di specificare da riga di comando gli argomenti da utilizzare in fase di esecuzione:

Comando	Descrizione
-p <percentage>	Percentuale di connessione per la generazione del grafo
-s <seed>	Seed per la generazione del grafo
-b <blocksize>	BlockSize da utilizzare nel kernel
-a <algorithm>	Kernel da utilizzare per l'esecuzione
-v	Abilita (se disponibile) la vettorizzazione dell'algoritmo selezionato
-c	Verifica il risultato dell'esecuzione GPU con il risultato della CPU
-V	(Verbose) Stampa l'output dell'esecuzione

Inoltre, per tutte le implementazioni, si utilizza l'allocazione su GPU con pitch tramite la chiamata CUDA "**cudaMallocPitch**". Dato che l'algoritmo lavora con una matrice e dato che CUDA alloca le matrici **linearmente**, ovvero come un array dove le righe sono contigue tra loro, in base al quantitavo di colonne della matrice è possibile che l'allocazione si "disallinei" rispetto ai banchi di memoria. Per minimizzare le transazioni necessarie per ogni warp per leggere quanto richiesto a una singola lettura per tutta una riga,

minimizzando così gli accessi in global memory, è necessario che ogni riga inizi a un indirizzo **allineato**. Per fare ciò, usando esplicitamente l'allocazione con pitch, la GPU allocherà una riga della matrice e controllerà poi se i byte di questa danno la possibilità alla prossima riga di essere allineata, ad esempio controllando se i byte finali all'allocazione sono multipli a quanto desiderato dall'hardware: se così non dovesse essere allora alloca i byte aggiuntivi necessari prima della prossima riga.

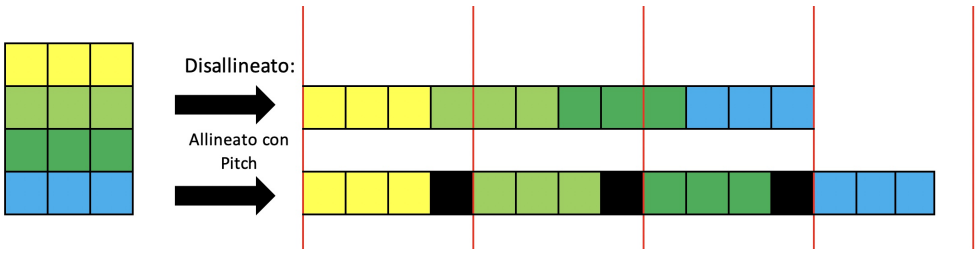

Figure 1: Allocazione con Pitch

2.1 Simple Parallelization

Si mostrano in principio le prestazioni di un kernel basilare "naive" - Simple Parallelize - per avere un'idea delle prestazioni della GPU e osservare fin da subito il guadagno prestazionale della GPU rispetto alla CPU, se e quando presente.

```
// Kernel Call
for(int k = 0; k < numVertices; k++)
    FW_simple_kernel<<<numBlock, dimBlock>>>
        (d_matrix, pitch, numVertices, k);
```

```
void FW_simple_kernel(short *graph, long long pitch, long long k){
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    short *graph_Pitch_i = (short*)((char*)graph + i * pitch);
    short *graph_Pitch_k = (short*)((char*)graph + k * pitch);

    if (i < n && j < n) {
        short ij = graph_Pitch_i[j];
        short ik = graph_Pitch_i[k];
        short kj = graph_Pitch_k[j];

```

```
13          if (ik + kj < ij)
14              graph_Pitch_i[j] = ik + kj;
15      }
16  }
```

Il kernel lavora con 3 parametri:

- **short** *graph: il puntatore device al grafo
- **long long** pitch: il pitch della matrice
- **long long** k: il k indicativo dell'iterazione corrispondente

Il funzionamento è semplice: nell'algoritmo Floyd-Warshall esistono dipendenze solo tra le iterazioni del ciclo più esterno "k", i due "for" interni sono parallelizzabili. Per ogni chiamata del kernel, inizialmente, si calcolano gli indici ($j = x$ e $i = y$ così da lavorare in coalescenza sulle righe) e si calcolano poi i puntatori all'inizio riga di i e k (righe 5 - 6), così da seguire l'allocazione con pitch (in bytes). Infine si esegue Floyd-Warshall normalmente: si selezionano le 3 componenti ij, ik, kj e si valuta se il percorso minimo è migliorato usando il vertice k.

2.2 Simple Parallelization Vectorized

Si mostra un approccio alla vettorizzazione dell'algoritmo, utilizzando il **tipo vettorizzato** di CUDA **short4**. Così facendo si sfruttano le operazioni vettoriali della GPU, velocizzando le prestazioni dell'algoritmo. In realtà si

presenta tale kernel così da avere un'implementazione che sfrutta la vettorizzazione che verrà poi riutilizzata nella vettorizzazione della variante **"Blocked"** dell'algoritmo.

```
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < (n << 2) && j < n) {
        short tempIk;
        short4 ij, ik, kj;
        short4 *graph_Pitch_i = (short4*)((char*)graph + i * pitch);
        short4 *graph_Pitch_k = (short4*)((char*)graph + k * pitch);

        ij = graph_Pitch_i[j];
        kj = graph_Pitch_k[j];

        int mask = ~((~0) << 2);
        int lsb_2 = (k & mask);
        tempIk = *(((short*)(graph_Pitch_i + (k >> 2))) + lsb_2);

        ik = make_short4(tempIk, tempIk, tempIk, tempIk);
        short4 res = ik + kj;
        graph_Pitch_i[j] = make_short4( res.x < ij.x ? res.x : ij.x,
                                        res.y < ij.y ? res.y : ij.y,
```

```
21                                      res.z < ij.z ? res.z : ij.z,
22                                      res.w < ij.w ? res.w : ij.w );
23      }
```

Il kernel lavora esattamente come nella sua versione non vettorizzata, l'unica differenza consiste nella lettura di ik: leggere infatti ik come **short4** avrebbe poco senso in quanto è in realtà uno solo l'elemento valido durante l'iterazione k. Bisogna quindi indicizzare il singolo short valido tra i 4: questo segue una ciclicità modulo 4 in base al valore di iterazione k. Basterà quindi selezionare gli ultimi 2 bit di k (riga 13 - 15) e indicizzare poi all'interno della struct short4. Questo viene fatto dal $+ \quad lsb_2$ a riga 15, che sostituisce il codice:

```
1       if(lsb_2 == 0)
2           tempIk = graph_Pitch_i[(k >> 2)].x;
3       if(lsb_2 == 1)
4           tempIk = graph_Pitch_i[(k >> 2)].y;
5       if(lsb_2 == 2)
6           tempIk = graph_Pitch_i[(k >> 2)].z;
7       if(lsb_2 == 3)
8           tempIk = graph_Pitch_i[(k >> 2)].w;
```

2.3 Simple Parallelization Pseudo-Blocked

Prima di presentare l'ottimizzazione principale "blocked" dell'algoritmo si presenta un'ultima implementazione che lavora comunque in global memory. Questa simula la creazione di sottomatrici dette "blocchi", in particolare lavora in modo tale che ogni thread si occupi di un blocco 4×4 (modificabile da variabile d'ambiente) ottimizzando così le operazioni in global memory e simulando il comportamento delle operazioni vettoriali.

```
1  #define PSEUDO_BLOCKSIZE 4
2  __global__
3  void FW_simple_kernel_vectorized_4x4(short *graph,
4          ll pitch,
5          ll n,
6          int k){
7      int j = blockIdx.x * blockDim.x + threadIdx.x;
8      int i = blockIdx.y * blockDim.y + threadIdx.y;
9
10     if(i < n && j < n){
11         #pragma unroll
12         for(int h = 0; h < PSEUDO_BLOCKSIZE; h++){
13             short *graph_i =
14                 (short*)((char*)graph + ((i * PSEUDO_BLOCKSIZE) + h)
                       * pitch);
```

```
15              short *graph_k = (short*)((char*)graph + k * pitch);
16
17              #pragma unroll
18              for(int w = 0; w < PSEUDO_BLOCKSIZE; w++){
19                  short ij, ik, kj;
20
21                  ij = graph_i[(j * PSEUDO_BLOCKSIZE) + w];
22                  kj = graph_k[(j * PSEUDO_BLOCKSIZE) + w];
23                  ik = graph_i[k];
24
25                  if (ik + kj < ij)
26                      graph_i[(j * PSEUDO_BLOCKSIZE) + w] = ik + kj;
27              }
28          }
29      }
30  }
```

2.4 Blocked Floyd-Warshall

Esistono varie ottimizzazioni applicabili all'algoritmo, molte simili a quelle per il prodotto tra matrici (lavorando con una matrice di adiacenza) [1]. L'ottimizzazione più significativa che si può fare però è quella di aumentarne la **località**, ovvero ottimizzare gli accessi in memoria sfruttando l'utilizzo di cache o, nel caso delle GPU, di shared memory. Per fare ciò si divide la matrice in sottomatrici dette **blocchi**, di uguale misura (idealmente la dimensione dei blocchi divide la dimensione della matrice ma ciò non è indispensabile). È necessario però modificare l'algoritmo originale per lavorare a blocchi: nella versione **blocked** [2] di Floyd-Warshall infatti si sfrutta una funzione per l'esecuzione dell'algoritmo originale all'interno di un singolo blocco, la correttezza della variante blocked è dettata poi da varie dipendenze sull'ordine con il quale si chiama tale funzione su blocchi specifici.

```
1 __forceinline__
2 __device__ void blockedUpdateFW(short *graph_ij, short *graph_ik,
      short *graph_kj, int i, int j, const int blockSize){
3     for(int k = 0; k < blockSize; k++){
4         short sum = graph_ik[i * blockSize + k] + graph_kj[k *
              blockSize + j];
5         if(graph_ij[i * blockSize + j] > sum)
6             graph_ij[i * blockSize + j] = sum;
7         __syncthreads();
```

```
8       }
9   }
```

La variante blocked è condizionata da 2 principali dipendenze: il ciclo for più esterno non è parallelizzabile (come nella parallelizzazione semplice) e inoltre, a causa della divisione in blocchi, è necessario dividere ogni iterazione in 3 fasi da eseguire in ordine per aggiornare le dipendenze e assicurare la correttezza dell'algoritmo. In particolare ogni iterazione corrisponderà a un **blocco principale** ovvero un blocco di dimensione $D \times D$ che costruisce la diagonale della matrice divisa per blocchi, per ognuna di queste iterazioni l'algoritmo si divide in:

1. **Fase Dipendente**
2. **Fase Parzialmente-Dipendente**
3. **Fase Indipendente**

Le tre fasi prendono il nome dalla dipendenza che il blocco da aggiornare ha rispetto al blocco principale da cui devono leggere. Il guadagno prestazionale che questa variante offre è quello che, lavorando progressivamente su blocchi di dimensione minore, si copiano i singoli blocchi in shared memory che gode di una velocità di trasferimento di molto superiore alla global memory. Così facendo, un algoritmo memory bound come Floyd-Wharsall ne beneficia notevolmente.

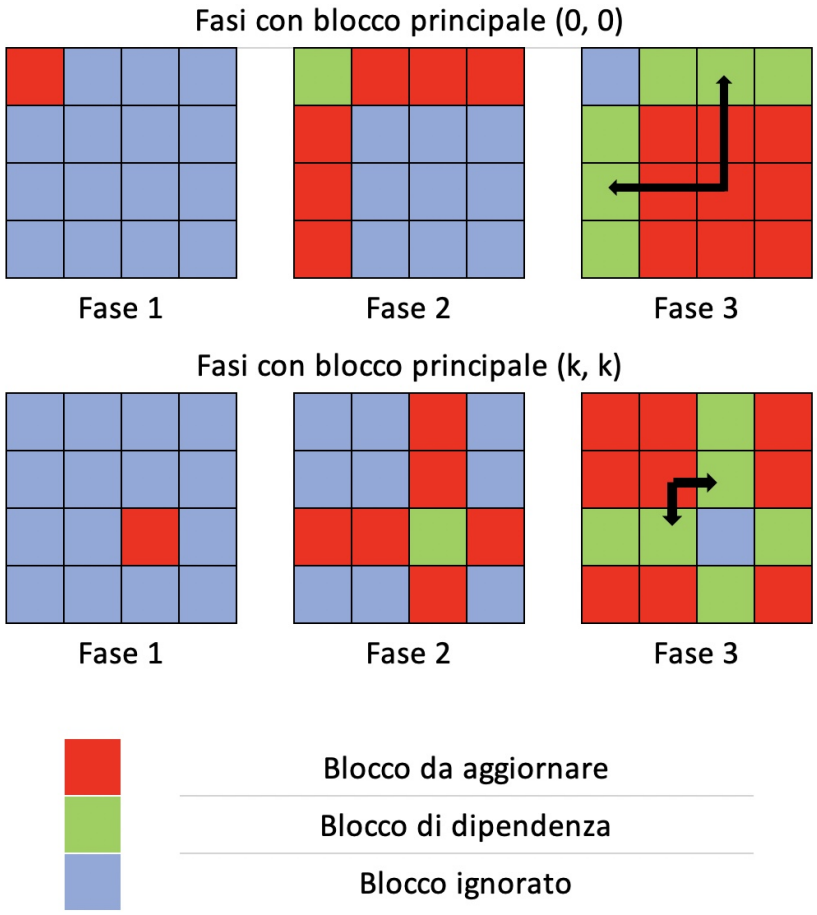

Figure 2: Fasi in variante Blocked

2.4.1 Fase Dipendente (1)

La prima fase consiste semplicemente nell'aggiornare i valori all'interno del blocco principale corrispondente all'iterazione k.

```
// Kernel Call
blocked_FW_phase1<<<1, dimBlock, sharedMemSize>>>
    (d_matrix, pitch, pitch / sizeof(short), k, blockSize);
```

```
__global__ void blocked_FW_phase1(short *graph, ll pitch, ll n,
    int k, const int blockSize){
int i = threadIdx.y;
int j = threadIdx.x;

extern __shared__ short lmem[];

short *graph_Pitch_main =
    (short*)((char*)graph + (k * blockSize * pitch));

lmem[i * blockSize + j] =
    graph_Pitch_main[(k * blockSize) + (i * n + j)];
__syncthreads();

blockedUpdateFW(lmem, lmem, lmem, i, j, blockSize);
__syncthreads();
```

```
16
17 graph_Pitch_main[(k * blockSize) + (i * n + j)] =
18     lmem[i * blockSize + j];
19 }
```

2.4.2 Fase Parzialmente-Dipendente (2)

La seconda fase si occupa di aggiornare i blocchi nella stessa riga e colonna del blocco principale. Per fare ciò, si lancia una griglia **monodimensionale** grande quanti sono i blocchi nella matrice lungo una dimensione. Il kernel si divide poi in due fasi:

1. **Fase 2.1**: Si aggiorna il blocco nella stessa colonna del blocco principale
2. **Fase 2.2**: Si aggiorna il blocco nella stessa riga del blocco principale

```
1 // Kernel Call
2 blocked_FW_phase2<<<numBlocks, dimBlock, 2 * sharedMemSize>>>
3     (d_matrix, pitch, pitch / sizeof(short), k, blockSize);
```

```
1 __global__ void blocked_FW_phase2(short *graph, ll pitch, ll n,
      int k, const int blockSize){
2     int x = blockIdx.x;
3
4     int i = threadIdx.y;
5     int j = threadIdx.x;
```

```
6
7       if (x == k)
8           return;
9
10      extern __shared__ short lmem[];
11      short *lmem_Block = (short*)lmem;
12      short *lmem_Main = (short*)
13      ((char*)lmem_Block + (blockSize * blockSize) * sizeof(short));
14
15      short *graph_Pitch_col =
16          (short*)((char*)graph + (x * blockSize * pitch));
17      short *graph_Pitch_main =
18          (short*)((char*)graph + (k * blockSize * pitch));
19      short *graph_Pitch_row =
20          (short*)((char*)graph + (k * blockSize * pitch));
21
22      // Fase 2.1
23      lmem_Block[i * blockSize + j] =
24          graph_Pitch_col[(k * blockSize) + (i * n + j)];
25      lmem_Main[i * blockSize + j] =
26          graph_Pitch_main[(k * blockSize) + (i * n + j)];
27      __syncthreads();
28
29      blockedUpdateFW(lmem_Block, lmem_Block, lmem_Main, i, j,
```

```
        blockSize);
30      __syncthreads();
31
32      graph_Pitch_col[(k * blockSize) + (i * n + j)] = lmem_Block[i *
            blockSize + j];
33
34      // Fase 2.2
35      lmem_Block[i * blockSize + j] =
36          graph_Pitch_row[(x * blockSize) + (i * n + j)];
37      lmem_Main[i * blockSize + j] =
38          graph_Pitch_main[(k * blockSize) + (i * n + j)];
39      __syncthreads();
40
41      blockedUpdateFW(lmem_Block, lmem_Main, lmem_Block, i, j,
            blockSize);
42      __syncthreads();
43
44      graph_Pitch_row[(x * blockSize) + (i * n + j)] =
45          lmem_Block[i * blockSize + j];
46  }
```

2.4.3 Fase Indipendente (3)

Nella terza fase invece si lancia una griglia bidimensionale sempre sfruttando il **blocco** come unità di misura. Si aggiorna ogni blocco che non è il principale e non è nella stessa colonna o riga del principale (Fase 2). Per ognuno di questi si usano come dipendenze i blocchi aggiornati nella fase 2 tali che:

- Il blocco è nella stessa riga del blocco principale e nella stessa colonna del blocco da aggiornare
- Il blocco è nella stessa colonna del blocco principale e nella stessa riga del blocco da aggiornare

```
// Kernel Call
blocked_FW_phase3<<<dimBlock_phase3, dimBlock, 3 * sharedMemSize>>>
    (d_matrix, pitch, pitch / sizeof(short), k, blockSize);
```

```
__global__ void blocked_FW_phase3(short *graph, ll pitch, ll n,
    int k, const int blockSize){
    int x = blockIdx.y;
    int y = blockIdx.x;

    int i = threadIdx.y;
    int j = threadIdx.x;

    if(x == k || y == k)
```

```
9           return;
10
11      extern __shared__ short lmem[];
12      short *lmem_Block = (short*)lmem;
13      short *lmem_Col =
14          (short*)((char*)lmem_Block + (blockSize * blockSize) *
                sizeof(short));
15      short *lmem_Row =
16          (short*)((char*)lmem_Col + (blockSize * blockSize) *
                sizeof(short));
17
18      short *graph_Pitch_block =
19          (short*)((char*)graph + (x * blockSize * pitch));
20      short *graph_Pitch_col =
21          (short*)((char*)graph + (x * blockSize * pitch));
22      short *graph_Pitch_row =
23          (short*)((char*)graph + (k * blockSize * pitch));
24
25      lmem_Block[i * blockSize + j] =
26          graph_Pitch_block[(y * blockSize) + (i * n + j)];
27      lmem_Col[i * blockSize + j] =
28          graph_Pitch_col[(k * blockSize) + (i * n + j)];
29      lmem_Row[i * blockSize + j] =
30          graph_Pitch_row[(y * blockSize) + (i * n + j)];
```

```
31      __syncthreads();
32
33      blockedUpdateFW(lmem_Block, lmem_Col, lmem_Row, i, j,
            blockSize);
34      __syncthreads();
35
36      graph_Pitch_block[(y * blockSize) + (i * n + j)] =
37          lmem_Block[i * blockSize + j];
38  }
```

2.5 Blocked Floyd-Warshall Vectorized

Un'ulteriore ottimizzazione applicabile è quella di utilizzare i tipi vettorizzati. La logica di esecuzione è uguale a quella del kernel "Simple Parallelize Vectorized" e viene implementata nella funzione di aggiornamento che viene richiamata all'interno dei singoli blocchi.

```
__forceinline__
__device__ void blockedUpdateFW_vectorized(short4 *graph_ij,
    short4 *graph_ik, short4 *graph_kj, int j, const int blockSize){

    for(int k = 0; k < blockSize; k++){
        short4 *graph_KJ = (short4*)
            ((char*)graph_kj + (k * blockSize) * sizeof(short));

        short4 ij = graph_ij[j];
        short4 kj = graph_KJ[j];

        int mask = ~((~0) << 2);
        int lsb_2 = (k & mask);
        short tempIk = *(((short*)(graph_ik + (k >> 2))) + lsb_2);

        short4 ik = make_short4(tempIk, tempIk, tempIk, tempIk);
        graph_ij[j] = checkWeight(ik + kj, ij);
        __syncthreads();
```

```
18      }
19  }
```

Nel corpo del kernel cambia ovviamente l'indicizzazione degli elementi in quanto è necessario prima arrivare alla riga e successivamente indicizzare le colonne in short4. Cambia anche l'indicizzazione della shared memory: ogni thread anzichè avere l'indirizzo di inizio per ogni blocco, calcola ora l'indirizzo della riga corrispondente a esso per scriverci direttamente. Alla funzione di aggiornamento si passa poi l'indirizzo di inizio blocco kj per garantire la correttezza durante la lettura delle righe kj, dipendenti dal k del for interno alla funzione stessa.

3 Confronto di Prestazioni

Di seguito si riporta il grafico generale delle prestazioni in base al numero di vertici del grafo. I valori arrivano fino a un massimo di 59392 che, allocati come short, corrispondono circa a 7GB, limite spaziale della scheda utilizzata (RTX 3070).

3.1 Tempo di esecuzione per numero di vertici

Già da subito si nota come il kernel blocked vettorizzato è ovviamente il più prestazionale. È giusto però attenzionare che, finchè il grafo ha pochi vertici (circa 70), non conviene utilizzare totalmente la GPU. Per grafi invece con numero di vertici superiori a 16384, eseguire sulla CPU porta a tempi di esecuzione quasi intrattabili.

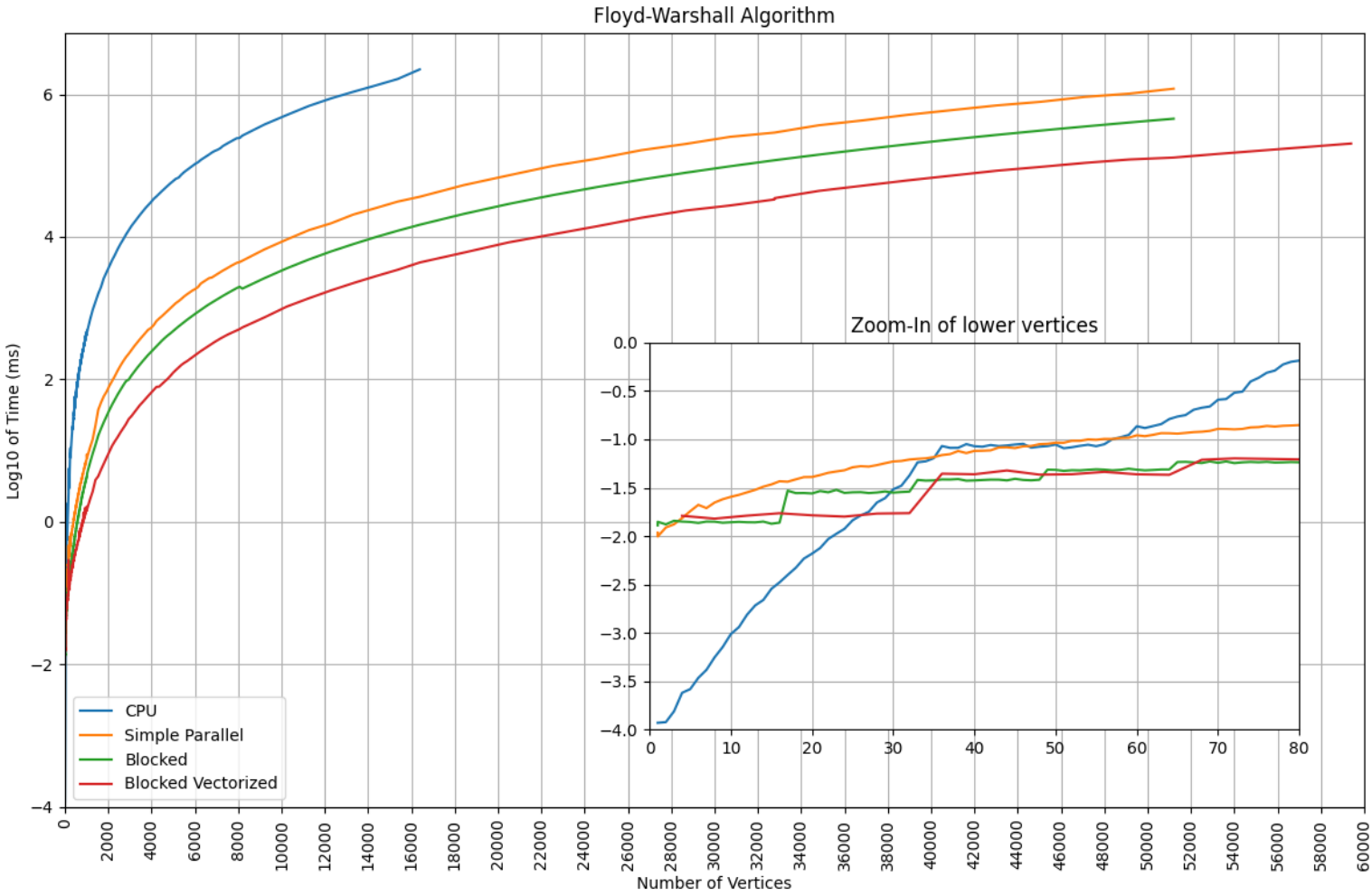

Figure 3: Prestazioni

3.2 Bandwidth e confronto di kernel

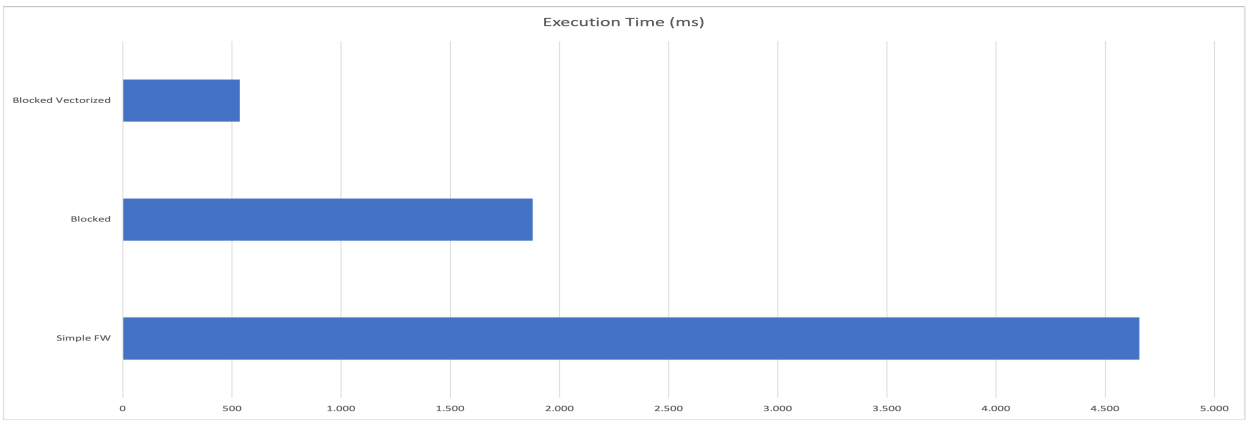

Figure 4: Tempo di esecuzione

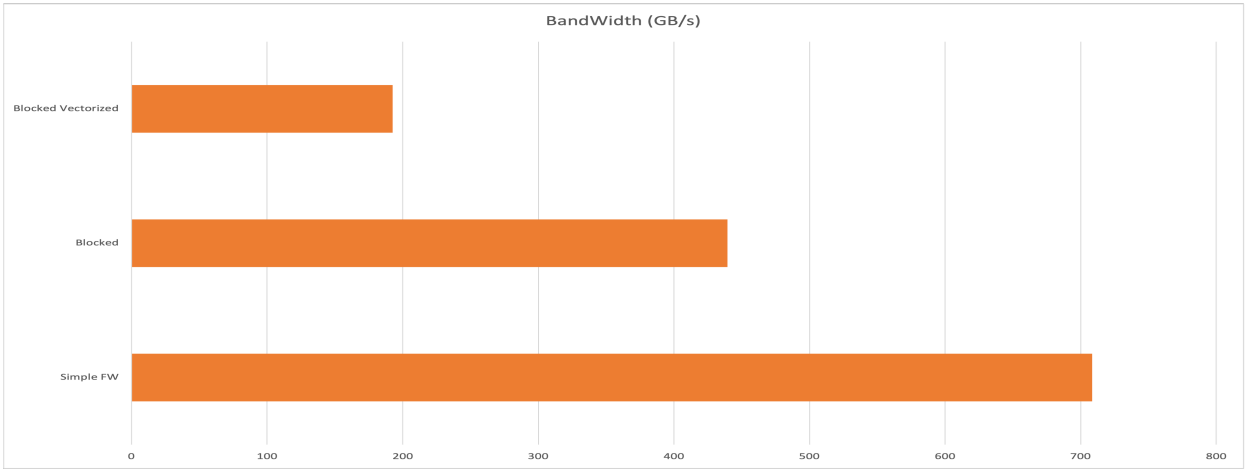

Figure 5: Bandwidth

I test sono stati effettuati tutti con 16384 vertici e con blocksize pari a 256 thread (16×16) effettivi.

Caso particolare è il "Simple FW" ovvero il kernel naïve utilizzato come baseline per le prestazioni. Si nota come la bandwidth sia nettamente su-

periore al limite teorico dell'hardware: ciò è probabilmente dovuto a un alto rateo di cache hit, vista soprattutto la natura del kernel di lavorare in place anzichè su una seconda matrice. Tale scelta dipende soprattutto dalle prestazioni ottenute: nel caso dell'algoritmo Floyd Warshall, si osserva che non sempre è necessario scrivere il valore risultante. A seconda del kernel utilizzato, potrebbe non essere necessario effettuare tutte le scritture o, nel peggiore dei casi, potrebbero essere riscritti gli stessi valori precedentemente letti. Questa situazione può portare a un miglioramento delle prestazioni grazie a cache hit o grazie a ottimizzazioni che evitano completamente la scrittura.

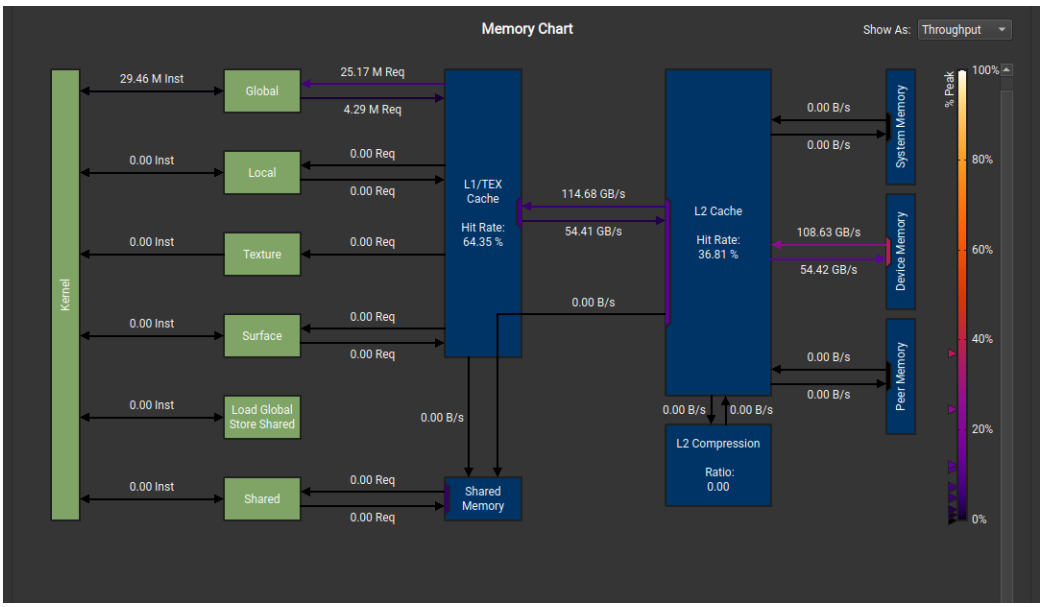

Figure 6: Profiling Simple FW Kernel

3.3 Simple Parallelization Pseudo-Block

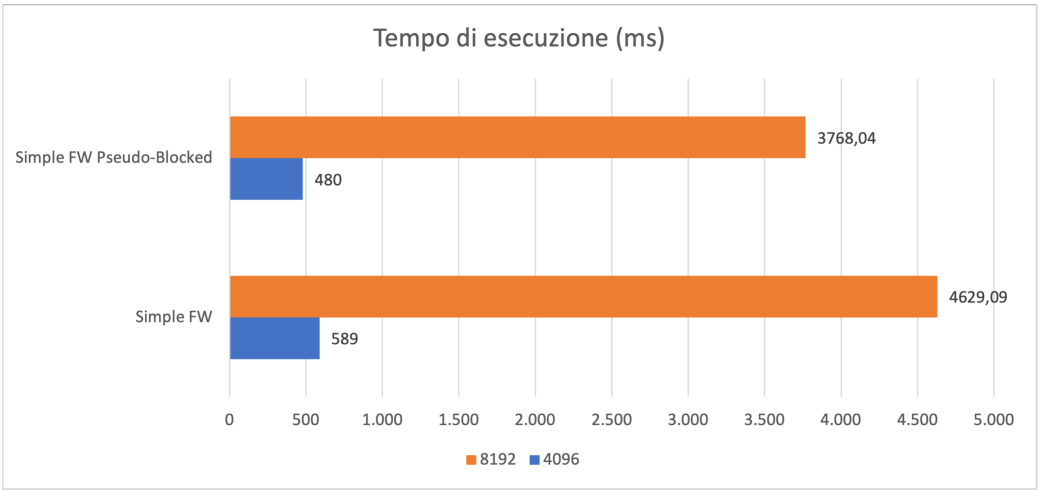

Figure 7: Execution Time of Pseudo-Blocked variant

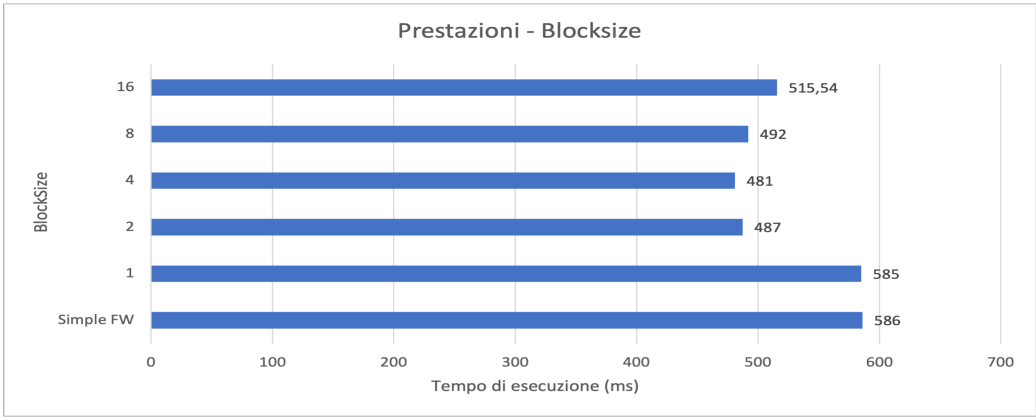

Figure 8: Performance based on different "blocksize" (4096 Vertices)

[Tutti i test sono stati effettuati con BlockSize CUDA pari a 256 thread per blocco]

Osservando invece la variante che implementa gli "pseudo-blocchi" possiamo notare un leggero miglioramento di prestazioni rispetto alla semplice implementazione "Simple FW", questo perchè ogni singolo thread si sta occupando di più elementi migliorando sia l'efficienza generale del kernel ma migliorando anche le richieste effettuate alla memoria in quanto vengono aggiornate più righe alla volta.

È interessante invece osservare il cambio di prestazioni che si genera utilizzando "blocchi" più grandi. Idealmente le prestazioni dovrebbero sempre migliorare ma ciò si equilibra con la mole di accessi in global memory che aumenta con l'aumento di grandezza della blocksize, riducendo le prestazioni del kernel.

Si nota infatti [Figura 8] come per blocksize pari a 1 il kernel si riduce a comportarsi come il "Simple FW". Abbiamo però miglioramenti di prestazioni solo fino a blocksize pari a 4, per misure maggiori le prestazioni ne risentono in modo graduale fino a rendere inutile l'ottimizzazione (con blocksize pari a 32 si riscontrano risultati sull'ordine dei 2 secondi).

3.4 Blocksize

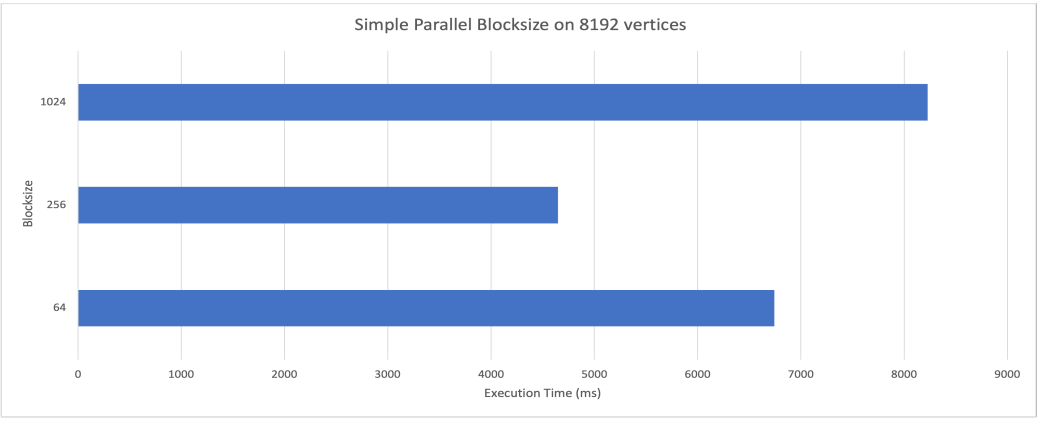

Figure 9: "Simple Kernel" performance's differencies based on different blocksizes

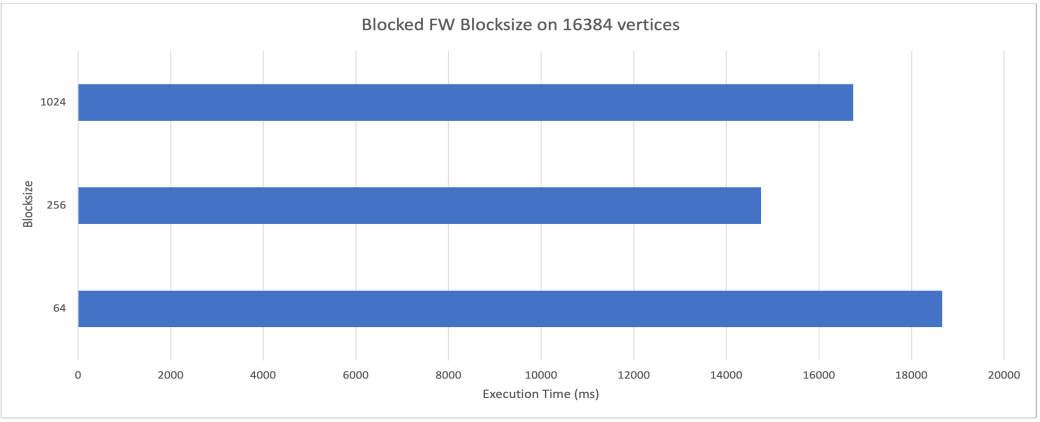

Figure 10: "Blocked Kernel" performance's differencies based on different blocksizes

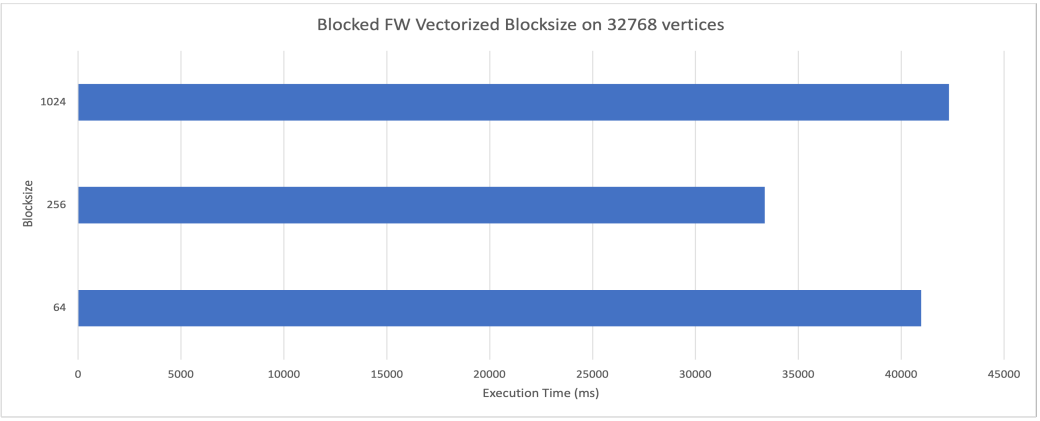

Figure 11: "Blocked Kernel Vectorized" performance's differencies based on different blocksizes

Si mostra ora come diverse blocksize, ovvero la quantità di thread lanciati per ogni blocco, condizionano le prestazioni dei vari kernel. I test sono stati effettuati con numero di vertici variabile in modo tale da mostrare le differenze in modo sostanziale senza aumentare di troppo il range dei valori, equilibrando però il tutto con le prestazioni differenti in base all'implementazione utilizzata.

Tutti i kernel hanno prestazioni migliori lavorando con blocksize pari a 256, per blocksize differenti invece si perdono prestazioni per motivazioni differenti.

Il kernel naïve "Simple FW" beneficia fortemente di blocksize "piccole" in quanto riesce a comprimere tutte le letture in una singola transazione. Aumentando invece di blocksize (1024), probabilmente è necessaria più di una singola transazione e quindi più accessi alla global memory.

Nei kernel blocked invece nasce un compromesso tra la mole di dati che

un blocco di grandi dimensioni riuscirebbe a elaborare contro le stesse limitazione alle transazioni del kernel Simple. Si nota infatti che nel kernel "Blocked" abbiamo la stessa perdita di prestazioni tra 256 e 1024 ma, al contrario di quanto visto prima, utilizzare blocksize pari a 64 causa una perdita drastica di prestazioni. Questo perchè l'algoritmo divide la matrice in molti **blocchi principali** e di conseguenza sono necessarie molte più iterazioni, causando anche più chiamate di kernel. Osservando invece la sua versione vettorizzata, il comportamento è simile ma con perdite meno drastiche per blocksize pari a 64.

4 Conclusioni

Dai risultati dei test effettuati si nota che, come da iniziali aspettative, l'utilizzo di GPU per la risoluzione di Floyd-Warshall porta grandi guadagni prestazionali: il kernel blocked vettorizzato è circa 570 volte più veloce della CPU con 16384 vertici.

È interessante notare che nonostante la banda passante dichiarata per la GPU utilizzata (RTX 3070) sia di 448 GB/s, i test riportano una banda passante teorica superiore o molto vicina al limite.

Concentrandosi invece sull'implementazione blocked dell'algoritmo, anche qui l'utilizzo di due matrici distinte per input/output ha portato a perdite di prestazioni. Allo stesso modo anche l'unificazione delle tre fasi in un unico kernel non ha portato a nessuna ottimizzazione: ciò è causato probabilmente dalla necessità di lanciare una griglia di dimensioni maggiori per la fase 3, limitando l'efficienza dei thread nelle precedenti due fasi.

L'utilizzo della shared memory ha portato invece sempre benefici, anche se l'esecuzione è limitata dalle frequenti barriere necessarie per la sincronizzazione dei thread dopo le scritture.

References

[1] Manish Pandey and Sanjay Sharma. Opencl parallel blocked approach for solving all pairs shortest path problem on gpu. *International Journal of Computer Applications*, 111(15):8–17, 2015.

[2] J-S Park, Michael Penner, and Viktor K Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on parallel and distributed systems*, 15(9):769–782, 2004.