Parallelizzazione algoritmo All Pairs Shortest Path (Floyd-Warshall)

Simone Girardi - VR421147
Dipartimento di Informatica
Universitá Degli Studi Di Verona
Email: simone.girardi@studenti.univr.it

***Sommario*—In questo report, vengono introdotte tecniche di computazione su un insieme di grafi utilizzando delle unitá di elaborazione visiva, comunemente chiamate schede grafiche o GPUs (Graphic Processing Units). L'obbiettivo di questo report é quindi mostrare l'efficienza di un'implementazione dell'algoritmo Floyd-Warshall in grado di sfruttare il piú possibile le risorse messe a disposizione dalla GPU comparando i risultati ottenuti con tecniche parallele alternative come OpenMP.**

I. Introduzione

Molti problemi computazionali su grafi presentano una complessitá temporale/spaziale del tutto impraticabile anche per le moderne CPU. Al fine di abbattere quanto piú possibile tali complessitá, conoscenze teoriche e pratiche per la programmazione e l'analisi di architetture di calcolo avanzate come ad esempio piattaforme multiprocessore e GPU, sono ad oggi di fondamentale importanza. In primo luogo é necessario essere in grado di effettuare un'analisi delle performance e profiling del codice, con individuazione delle zone critiche per riuscire poi ad applicare alcune tecniche di ottimizzazione considerando le caratteristiche architetturali della piattaforma su cui si sta sviluppando. Inoltre é importante prestare particolare attenzione nella scelta dei pattern di parallelismo da applicare, in modo da prendere una decisione adeguata a seconda del contesto d'uso (e.g. consumo energetico, potenza di calcolo, area ridotta etc...).

II. Background

A. Floyd-Warshall

Dato un grafo $G = (V, E)$ dove i pesi di tutti i suoi archi sono esplicitamente specificati, l'algoritmo *Floyd-Warshall* [1] risolve il problema di trovare il percorso a costo minimo tra tutte le coppie di vertici del grafo. L'algoritmo, nella sua versione standard, opera con una complessitá computazionale pari a $O(|V^3|)$ ed una complessitá spaziale pari a $O(|V^2|)$. Lo pseudocodice che rappresenta l'algoritmo in esame é mostrato in Algorithm 1, in cui: per un grafo di n nodi la funzione $W(i, j)$ rappresenta il peso dell'arco tra il nodo i e il nodo j.

Algorithm 1 Floyd-Warshall

$D^0 = W$
for $k = 0$ `to` V **do**
  for $i = 0$ `to` V **do**
    for $j = 0$ `to` V **do**
      $D^k[i,j] = min(D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j])$
    end for
  end for
end for

B. Computed Unified Architecture (CUDA)

La piattaforma di elaborazione in parallelo CUDA [2] fornisce alcune semplici estensioni di C e C++ che consentono di esprimere sia i dati "fine-grained" che quelli "coarse-grained", oltre al parallelismo dei task. Il programmatore puó scegliere di esprimere tale parallelismo in linguaggi ad alto livello come C, C++, Fortran o in standard aperti come le direttive OpenACC. In figura 1 vengono messe a confronto un'implementazione sequenziale *C* ed un'implementazione parallela *CUDA* per effettuare la medesima procedura *SAXPY*.

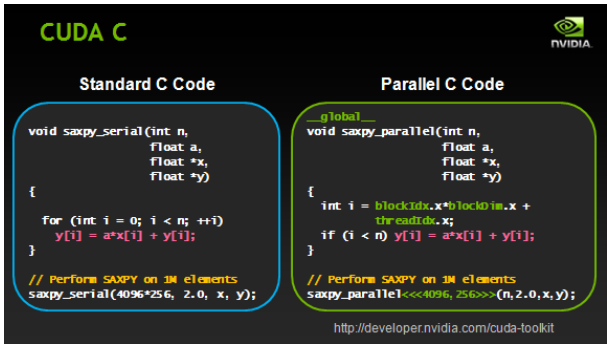

Figura 1: Confronto fianco a fianco di codice C standard e C in parallelo

C. OpenMP

OpenMP (Open Multiprocessing) [3] è un API multipiattaforma per la creazione di applicazioni parallele su sistemi a memoria condivisa. Questa piattaforma é composta da un insieme di direttive di compilazione, routine di librerie e variabili d'ambiente che ne definiscono il funzionamento a run-time. OpenMP é un'implementazione del concetto di multithreading ovvero di un metodo di parallelizzazione per cui un master thread (una serie di istruzioni eseguite consecutivamente) crea un certo numero di slave thread e un task (processo) é diviso tra i vari slave thread. I thread vengono eseguiti in modo concorrente mentre il run-time system alloca i thread su processori disponibili.

III. Ambiente di sviluppo

Un primo fattore a cui é necessario prestare attenzione, dopo aver compreso la tipologia di problema da affrontare, riguarda le risorse Hardware e Software che verranno utilizzate per implementare una soluzione in grado di soddisfare le specifiche.

In questa sezione vengono riportate in tabella I e II, le caratteristiche tecniche che compongono l'ambiente di sviluppo utilizzato per affrontare il caso di studio in esame.

Software	Descrizione
Compilatore	The GNU Compiler Collection's C++, g++ 7.3.0 con supporto a C++14
CPU profiler	Valgrind + Callgrind, KCachegrind 3.13.0
GPU profiler	Nvidia Visual Profiler 10.0
Piattaforma parallela CPU	OpenMP 4.5
Piattaforma parallela GPU	CUDA 10.0

Tabella I: Specifiche tecniche - Software

Hardware	Descrizione
CPU	IntelCore i5 6400 2,7GHz QuadCore
Operating System	Ubuntu 18.04 LTS 64 bit
RAM	Crucial 8GB DDR4 2400MHz
GPU	MSI Nvidia GEFORCE GTX 1060 3GT OC

Tabella II: Specifiche tecniche - Hardware

IV. Metodologia applicata

Il primo obbiettivo é rivolto all'analisi del codice sequenziale con l'uso di profiler. In secondo luogo vengono sviluppati dei kernel CUDA per esecuzione su GPU con ottimizzazione su:

- Kernel configuration
- Brench divergence
- Memory coalescence
- Shared memory usage
- Load Balancing
- Tecniche avanzate (ILP, etc.)

Il secondo obbiettivo riguarda la parallelizzazione del codice sequenziale con direttive OpenMP, in particolare:

- utilizzo di tutte le possibili direttive presentate durate il corso di architetture avanzate.

A. ANALISI CODICE SEQUENZIALE

Il codice sequenziale fornito inizialmente con le specifiche di progetto é mostrato in Listing 1.

Dopo una prima analisi del codice sequenziale, con l'aiuto di KCachegrind [4] é stato possibile notare come il carico di lavoro maggiore fosse concentrato principalmente nel ciclo piú interno dell'algoritmo (vedi figura 2).

In particolare, eseguendo il codice su tutti i benchmark forniti con le specifiche del progetto, si evince come il collo di bottiglia persista sulle istruzioni condizionali del ciclo piú interno. Di conseguenza, poiché la condizione *matrix[i][k] != INF* dipende solo da *i* e da *k*, il codice sequenziale é stato leggermente modificato spostando tale condizione nel costrutto *for* appena piú esterno. Dopo questa fine accortezza, l'algoritmo ha presentato un buon miglioramento delle performance

Listing 1 Codice sequenziale Floyd-Warshall

```
for (int k = 0; k < num_v; k++) {
  for (int i = 0; i < num_v; i++) {
      for (int j = 0; j < num_v; j++) {
          if (N[i][k] != INF && N[k][j] != INF &&
            ↪  N[i][k] + N[k][j] < N[i][j])
              N[i][j] = N[i][k] + N[k][j];
      }
  }
}
```

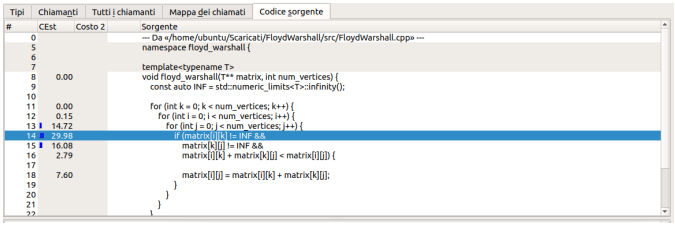

Figura 2: Profiling codice sequenziale

risparmiando cosí, nel caso migliore, n iterazioni. Allo stato attuale, peró, il codice che implementa l'algoritmo in maniera sequenziale non puó vantare di ulteriori ottimizzazioni significative. Di conseguenza una possibile soluzione é quella di implementare una versione parallela adottando una architettura hardware CUDA che verrá descritta nella sezione successiva.

B. SVILUPPO KERNEL CUDA

Una prima implementazione parallela del FLoyd-Warshall vuole focalizzarsi sulla funzionalitá dell'algoritmo, senza dare troppa enfasi alle performance. Questa implementazione nasce sotto il nome di *Naive Floyd-Warshall* 2 e consiste nel parallelizzare il codice sequenziale senza applicare alcuna tecnica di ottimizzazione. Ossia una volta configurata una griglia 2D, viene eseguito lo stesso kernel un numero di volte pari alla dimensione (riga o colonna) della matrice.

Un approccio di questo tipo, testato sui vari benchmark, ci porta ad avere un miglioramento marginale, rispetto alle potenzialitá offerte dalla GPU, dovuto soprattutto al fatto che non é stata applicata alcuna tecnica di ottimizzazione. Un analisi piú accurata del kernel é resa possibile grazie al profiler messo a disposizione da NVIDIA (NVIDIA Visual Profiler).

In figura 3 é possibile notare come il tool, dopo un'analisi di latenza, mette in evidenza un'abbondante carico del kernel focalizzato principalmente sulla dipendenza dalla memoria. Questo risultato é in linea con le nostre aspettative in quanto l'algoritmo sappiamo giá essere di per se *memory bounded*.

Ripetendo l'esecuzione su dimensioni di blocchi di thread differenti si nota un calo di prestazioni quando si supera la dimensione 16x16. Dall'analisi del profiler si dimostra che questo fenomeno puó essere causato da un accesso a memoria inefficiente. Quindi proprio sotto suggerimento del profiler si é passati ad un'implementazione volta a risolvere il problema della coalescenza e ottimizzando quindi gli allineamenti di

Listing 2 Naive Floyd-Warshall Kernel

```
__global__ void naive_floyd_warshall_kernel(float
 ↪  *N, int n, int k) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;

  if (i >= n || j >= n) return;

  float i_k_value = N[i * n + k];
  float k_j_value = N[k * n + j];
  float i_j_value = N[i * n + j];

  if (i_k_value != INF && k_j_value != INF) {
      float sum = i_k_value + k_j_value;
      if (sum < i_j_value)
          N[i * n + j] = sum;
  }
}
```

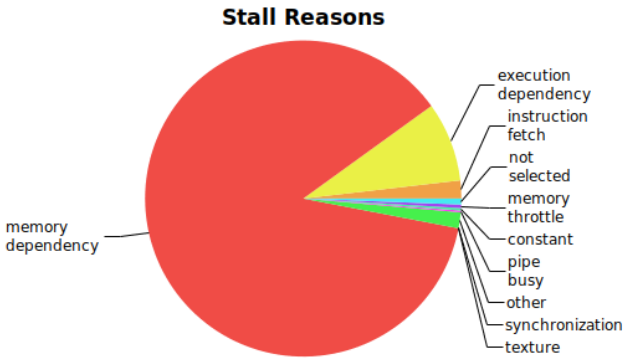

Figura 3: Profiling Naive Floyd-Warshall

accessi a memoria globale. Il nuovo kernel viene quindi configurato come mostrato in Listing 3.

É da notare che come ulteriore tecnica di ottimizzazione é stato implementato un meccanismo che permette ad ogni thread di computare piú di un elemento di output, con lo scopo di ottenere un ulteriore incremento delle performance sfruttando il principio di localitá temporale.

Con questa implementazione si nota un guadagno di prestazioni giá piú significativo passando da uno speedup di circa 14x ad uno speedup di circa 25x. Questo dimostra che eseguendo l'accesso alla memoria globale in maniera coalizzata la larghezza di banda della memoria viene utilizzata in modo più efficace, il che a sua volta aumenta il throughput.

Ripetendo di nuovo l'esecuzione su dimensioni di blocchi di thread differenti, come mostrato in figura 4 e 5, si nota un calo di prestazioni quando i blocchi di thread vengono configurati in modo da avere una dimensione maggiore o minore di 64. Avvalendosi di nuovo del supporto del profiler é possibile individuare ulteriori colli di bottiglia, in particolare notiamo ancora una volta che gli accessi a memoria globale, benché vengano coalizzati, rendono l'intera computazione notevolmente piú lenta delle aspettative rispetto alle risorse

Listing 3 Coalesced Floyd-Warshall Kernel

```
dimGrid=dim3(ceil(1.0*n*n/(BLK_COA*SEGMENT_SIZE)));
dimBlock=dim3(BLOCK_COA, 1.0,1.0);

__global__ void coa_floyd_warshall_kernel(float *N,
 ↪  int n, int k) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  const int size = n*n;

  if (idx >= size) return;

  int i,j;
  float i_k_value, k_j_value, sum;
  const int seg_id = SEGMENT_SIZE*idx;

  #pragma unroll
  for (int offset = 0; offset < SEGMENT_SIZE &&
   ↪  offset + seg_id < size; offset++) {
    i = (seg_id + offset) / n;
    j = seg_id + offset - i * n;

    i_k_value = N[i * n + k];
    k_j_value = N[k * n + j];

    if (i_k_value != INF && k_j_value != INF) {
      sum = i_k_value + k_j_value;
      if (sum < N[i * n + j])
        N[i * n + j] = sum;
    }
  }
}
```

hardware a disposizione (vedi figura 6).

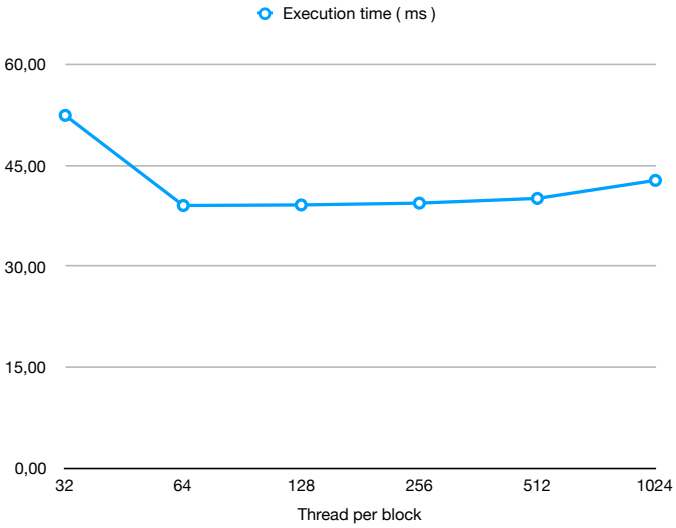

Figura 4: Coalesced Floyd-Warshall

La fase successiva é dunque quella di ricorrere all'uso della memoria condivisa, mantenendo peró la coalescenza tra le thread dello stesso blocco. L'uso della memoria condivisa ha lo scopo di ridurre la latenza dovuta agli accessi che altrimenti verrebbero effettuati in memoria globale. Di conseguenza, se

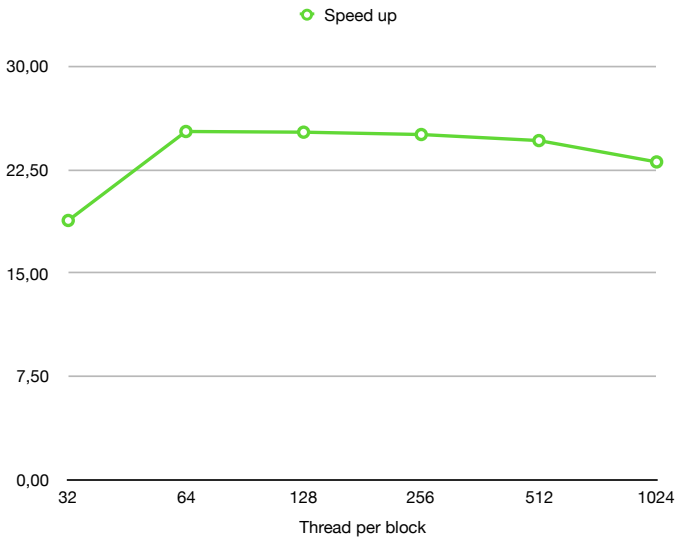

Figura 5: Coalesced Floyd-Warshall

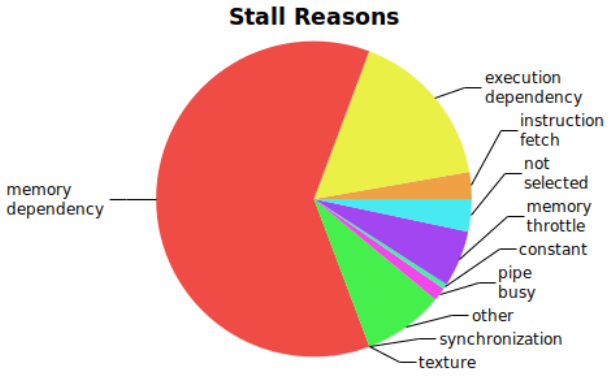

Figura 6: Coalesced Floyd-Warshall

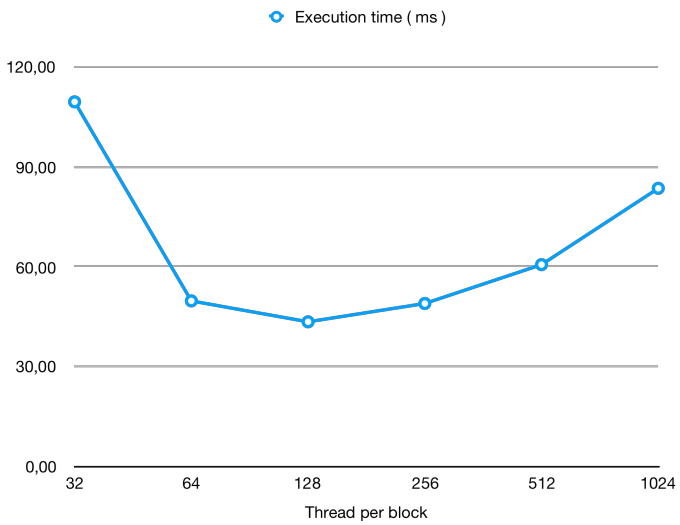

Figura 7: Shared Floyd-Warshall

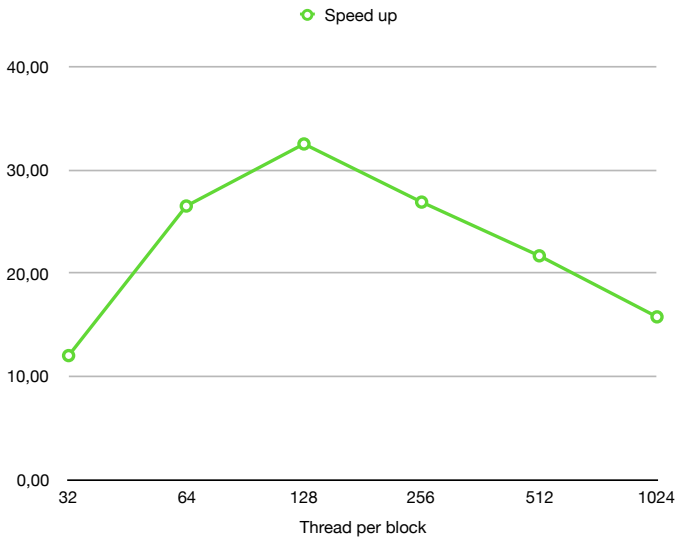

Figura 8: Shared Floyd-Warshall

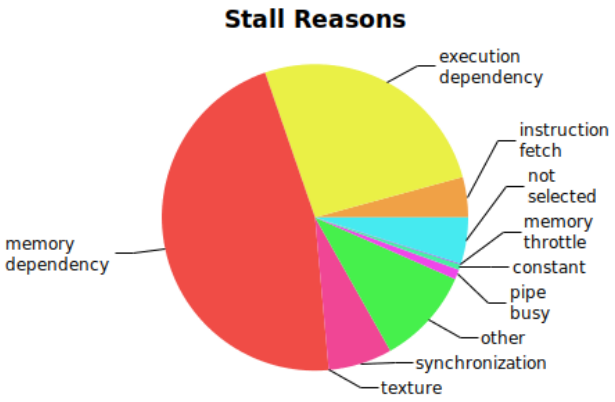

Figura 9: Shared Floyd-Warshall

l'algoritmo prevede molti accessi agli stessi elementi quanti piú ne vengono caricati in memoria condivisa, tanto piú elevato sará il guadagno in termini di performance. In Listing 4 vengono riportati il codice e alcuni risultati da confrontare con le implementazioni precedentemente analizzate.

Notiamo che ora lo speedup supera abbondantemente il 30x, ma un ulteriore sguardo al profiler sottolinea come un collo di bottiglia rimanga l'accesso alla memoria globale. Un implementazione di questo tipo, pur offrendo un ulteriore incremento delle performance, sembra non fare un uso molto efficiente della memoria condivisa. É quindi necessario rivedere l'algoritmo per riuscire ad identificare le fasi di computazione dipendenti e indipendenti tra di loro, in modo da riuscire a separare queste fasi ed aumentare quindi l'efficienza di un parallelismo piú capiente ed efficiente.

Da questa esigenza, nasce quindi una nuova implementazione chiamata *Blocked Floyd-Warshall*; questa versione é stata proposta per la prima volta da Venkataraman et al [5]. L'algoritmo inizialmente partiziona il grafo in blocchi la cui dimensione é dettata dalla capacitá della cache della specifica GPU, succesivamente le iterazioni che prima venivano effettuate sui singoli elementi (riga/colonna), avvengono ora per

Listing 4 Shared Memory Floyd-Warshall Kernel

```
dimGrid  = dim3(ceil(n/(float)BLOCK_SM),n,1.0);
dimBlock = dim3(BLOCK_SM, 1.0, 1.0);

__global__ void sm_floyd_warshall_kernel(float *N,
 ↪  int n, int k) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;

  if (i >= n || j >= n) return;

  float i_j_value = N[i * n + j];
  float k_j_value = N[k * n + j];

  __shared__ float i_k_value;

  if (threadIdx.x == 0) {
    i_k_value = N[i * n + k];
  }
  __syncthreads();

  if(i_k_value != INF && k_j_value != INF) {
    float sum = i_k_value + k_j_value;
    if (sum < i_j_value)
      N[i * n + j] = sum;
  }
}
```

mezzo dei blocchi appena creati, in cui ad ogni iterazione seguono 3 fasi principali:

1) La computazione in ogni iterazione inizia da un blocco (pivot) sulla diagonale della matrice, dall'alto verso il basso. Ogni blocco sulla diagonale é indipendente dal resto della matrice quindi puó essere processato in loco.
2) Nella seconda fase, tutti i blocchi che si trovano sulla stessa riga e colonna del blocco pivot, processato alla fase precedente, vengono a loro volta processati in parallelo tra di loro. Tutti i blocchi coinvolti in questa fase sono quindi dipendenti solo da se stessi e dal rispettivo blocco pivot.
3) Nella terza fase, tutti i blocchi rimanenti sono dipendenti tra di loro e dai blocchi riga/colonna calcolati alla fase 2.

A causa delle dipendenze tra i blocchi, é molto importante che le 3 fasi vengano eseguite in ordine. Esaminando meglio l'algoritmo é possibile intuire come esso riesca a fare un uso intenso della memoria condivisa come cache locale, e con il supporto del profiler possiamo notare come questa soluzione impatti pesantemente sulle performance ottenute.

Quest'ultima soluzione ha dato prova di riuscire a spingere le performance ben oltre le precedenti implementazioni. In figura 10 e 11 possiamo notare come i tempi siano abbondamente abbattuti con uno speedup di circa 194x.

Osservando con piú attenzione i risultati ottenuti con l'analisi del profiler mostrati in figura 13 si nota che questa soluzione presenta ulteriori margini di miglioramento, dovuti soprattutto al fatto che nonostante un uso molto efficiente della memoria

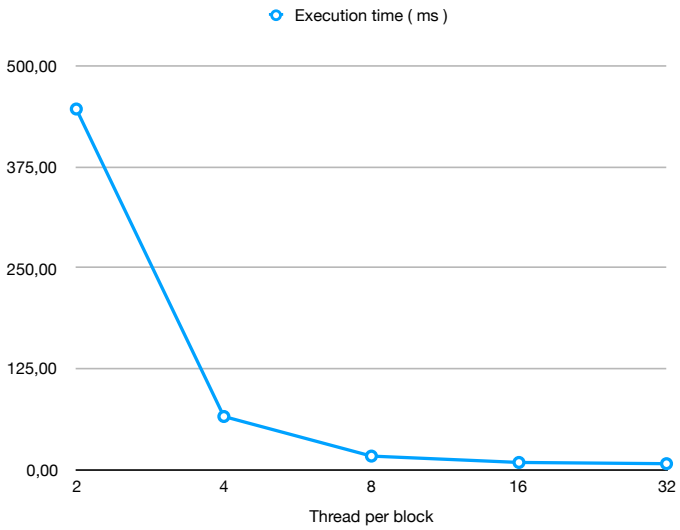

Figura 10: Blocked Floyd-Warshall

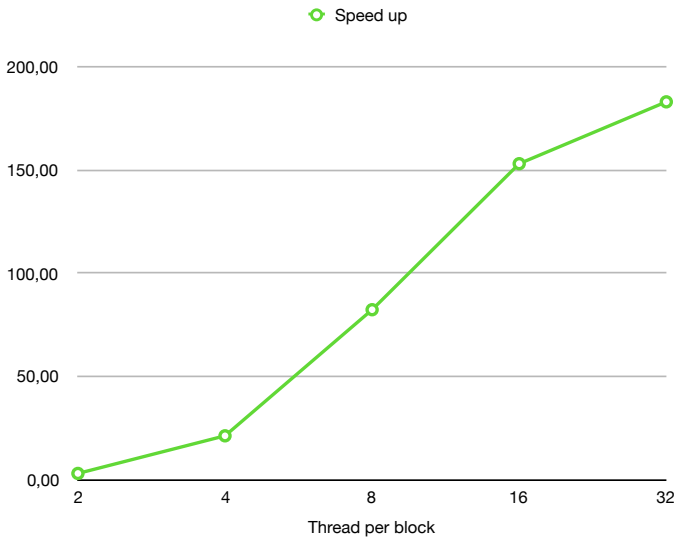

Figura 11: Blocked Floyd-Warshall

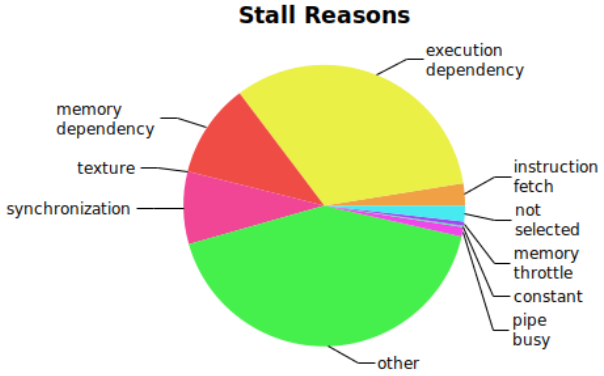

Figura 12: Blocked Floyd-Warshall

condivisa, vi sono poche operazioni che vengono effettuate su di essa. Un'ulteriore ottimizzazione (non implementata in questo report) puó essere applicata sfruttando ancora una volta il principio di localitá temporale, facendo in modo che ogni thread computi piú di un risultato e aumentando cosí il numero di operazioni svolte sulla memoria condivisa.

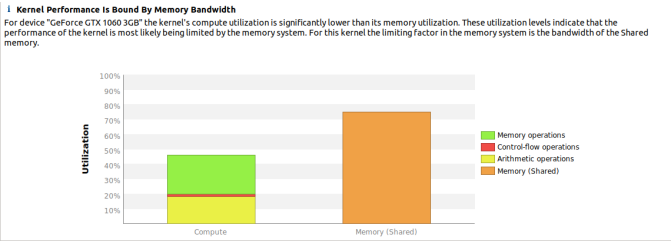

Figura 13: Blocked Floyd-Warshall

C. UTILIZZO DELLE DIRETTIVE OPENMP

Un altro modo per sperimentare i vantaggi nell'adozione di una soluzione parallela puó essere messo in pratica sfruttando la versatilitá di OpenMP. Infatti, con un effort davvero ridotto, é possibile istruire il codice sequenziale in modo da eseguire porzioni di esso in parallelo tramite la chiamata a delle direttive ben definite dallo standard. In particolare come mostrato dal codice in Listing 5

Listing 5 OpenMP Floyd-Warshall

```
void floyd_warshall_omp(T** matrix, int num_v) {
  int i,j,k;
  #pragma omp parallel shared(num_v, matrix)
   ↪  private(i,j, k); default(none)
  for (k = 0; k < num_v; k++) {
      #pragma omp for schedule(dynamic)
      for (i = 0; i < num_v; i++) {
          if (matrix[i][k] != INF) {
            for (j = 0; j < num_v; j++) {
                if (matrix[k][j] != INF &&
                 ↪  matrix[i][k] + matrix[k][j] <
                 ↪  matrix[i][j])
                    matrix[i][j] = matrix[i][k] +
                     ↪  matrix[k][j];
            }
          }
      }
  }
}
```

Aggiungere semplicemente due direttive con le opportune configurazioni, é stato piú che sufficiente per ottenere un notevole incremento delle performance.

V. RISULTATI

In questa sezione vengono riportati i risultati delle esecuzioni effettuate su tutti i benchmark mettendo a confronto i tempi di esecuzione di: CPU, OpenMP e CUDA con le loro migliori configurazioni individuate in fase di sviluppo.

#Nodes	CPU time	OpenMP time	CUDA time
100	0,00107	0,00061	0,000062
1074	1,33	0,26	0,01
1272	1,86	0,27	0,011
10728	981,34	91,73	6,37
16320	2927,99	147,38	22,11

Tabella III: Tempi di esecuzione

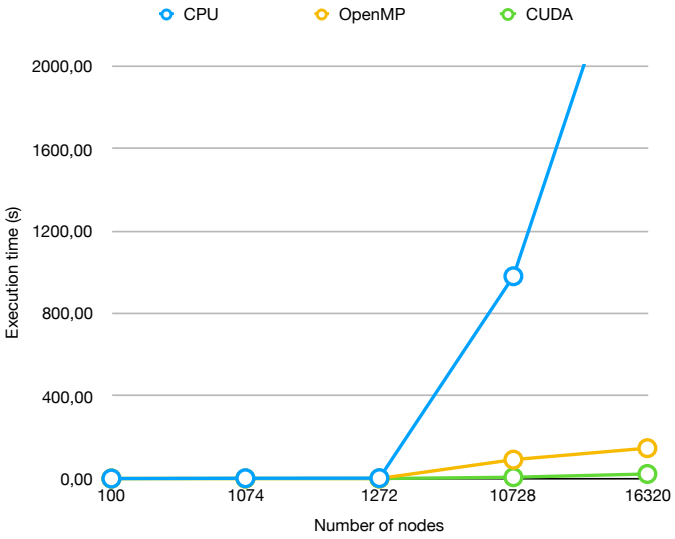

Figura 14: Comparazione Tempi di esecuzione

VI. CONCLUSIONI

In conclusione possiamo dire che le performance di un kernel sono sensibili ai parametri di configurazione, cosí come il sotto utilizzo della larghezza di banda e una subottima thread occupancy, possono degradare molto le performance di una GPU. Per la scelta corretta delle dimensioni di griglia, blocchi ed eventuale dimensione di segmenti (o pitch), é necessario svolgere le azioni di profiling e testare il codice su benchmark diversi. Inoltre, é da notare che, nonostante la cura nella scelta dei parametri appena citati, é possibile che quest'ultimi funzionino meglio su certe implementazioni e peggio su altre. In ogni caso i principi generali quali la coalescenza degli accessi a memoria e la riduzione degli accessi a quest'ultima per mezzo della memoria condivisa, hanno sempre un forte impatto sulle performance quali speedup e tempo di esecuzione, come dimostrato negli esperimenti di questo report.

RIFERIMENTI BIBLIOGRAFICI

[1] Wikipedia, "Algoritmo di Floyd-Warshall — Wikipedia, the free encyclopedia," http://it.wikipedia.org/w/index.php?title=Algoritmo\%20di\%20Floyd-Warshall&oldid=97075577, 2019, [Online; accessed 17-March-2019].

[2] N. Corporation. (2007) Nvidia cuda for developer. [Online]. Available: www.nvidia.it/object/cuda_home_temp.html

[3] Wikipedia, "OpenMP — Wikipedia, the free encyclopedia," http://it.wikipedia.org/w/index.php?title=OpenMP&oldid=95478054, 2019, [Online; accessed 17-March-2019].

[4] J. Weidendorfer, "Kcachegrind: A profile data visualization tool," https://kcachegrind.github.io/html/Home.html, 2013.

[5] S. M. G. Venkataraman, S. Sahni, "A blocked all-pairs shortest-paths algorithm," *Experimental Algorithmics (JEA), August*, 2003.