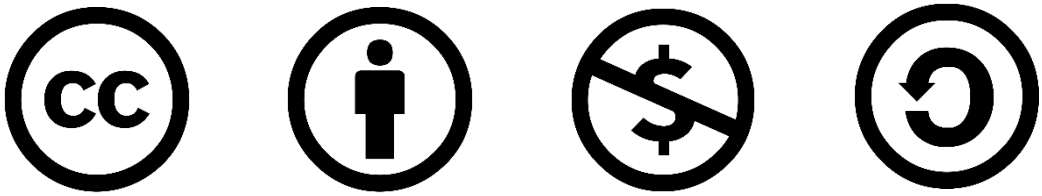


Report Common Assignment 1

Graph traversal:
Breath First Search Algorithm

Lecturer: Francesco Moscato - fmoscato@unisa.it

Student: Canzolino Gianluca - 0622701806 - g.canzolino3@studenti.unisa.it

Sommario

Breath First Search

In questo documento viene trattata la parallelizzazione e la valutazione delle performance dell'algoritmo "Breath First Search" su GPU Nvidia.

Setup sperimentale

Hardware

CPU

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model               : 79
model name      : Intel(R) Xeon(R) CPU @ 2.20GHz
stepping        : 0
microcode       : 0x1
cpu MHz         : 2199.998
cache size      : 56320 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid : 0
fpu                 : yes
fpu_exception   : yes
cpuid level     : 13
wp                  : yes
flags               : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall
nx  pdpe1gb  rdtscp  lm  constant_tsc  rep_good  nopl  xtopology
nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid
sse4_1  sse4_2  x2apic  movbe  popcnt  aes  xsave  avx  f16c  rdrand
hypervisor lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb
stibp  fsgsbase  tsc_adjust  bmi1  hle  avx2  smep  bmi2  erms  invpcid
rtm rdseed adx smap xsaveopt arat md_clear arch_capabilities
bugs                :    cpu_meltdown    spectre_v1    spectre_v2
spec_store_bypass l1tf mds swapgs taa
bogomips        : 4399.99
clflush size    : 64
cache_alignment     : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

GPU

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Mon_Oct_12_20:09:46_PDT_2020
Cuda compilation tools, release 11.1, V11.1.105
Build cuda_11.1.TC455_06.29190527_0
Sun Jan 23 15:06:21 2022

Device name: Tesla K80
Compute capability: 3.7

Clock Rate: 823500 kHz
Total SMs: 13
Shared Memory Per SM: 114688 bytes
Registers Per SM: 131072 32-bit
Max threads per SM: 2048
L2 Cache Size: 1572864 bytes
Total Global Memory: 11996954624 bytes
Memory Clock Rate: 2505000 kHz

Max threads per block: 1024
Max threads in X-dimension of block: 1024
Max threads in Y-dimension of block: 1024
Max threads in Z-dimension of block: 64

Max blocks in X-dimension of grid: 2147483647
Max blocks in Y-dimension of grid: 65535
Max blocks in Z-dimension of grid: 65535

Shared Memory Per Block: 49152 bytes
Registers Per Block: 65536 32-bit
Warp size: 32
```

GPU Bandwidth

```
Device 0: Tesla K80
 Range Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)    Bandwidth(MB/s)
   1000                  202.8
   101000               5941.6
   201000               6660.5
   301000               6858.3
   401000               6896.0
   501000               7181.2
   601000               7223.8
   701000               7261.4
   801000               7433.8
   901000               7398.1
```

```
Device to Host Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)    Bandwidth(MB/s)
  1000                      409.0
  101000                   6495.7
  201000                   7137.9
  301000                   7317.1
  401000                   7426.9
  501000                   7486.8
  601000                   7552.9
  701000                   7571.9
  801000                   7613.0
  901000                   7622.8

Device to Device Bandwidth, 1 Device(s)
PINNED Memory Transfers
  Transfer Size (Bytes)    Bandwidth(MB/s)
  1000                      253.4
  101000                  25891.7
  201000                  41917.7
  301000                  60030.0
  401000                  73698.0
  501000                  80673.8
  601000                  91201.4
  701000                  99054.6
  801000                  90735.4
  901000                  77983.9
```

Software

Google Colab

Riguardo le misure

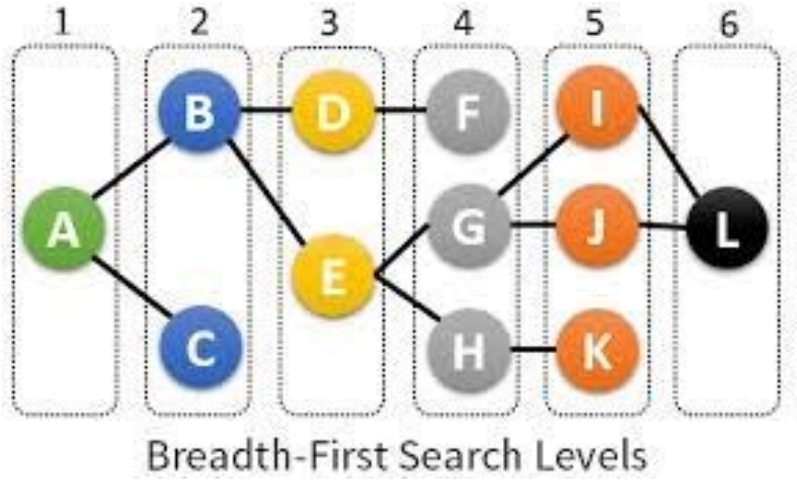
Ogni misurazione è stata eseguita 20 volte in modo tale da avere un'accuratezza migliore.

Tutti i test sono stati eseguiti utilizzando la global memory e la texture memory su 2 grafi differenti, con il rispettivo numero di vertici: 500'000, 1'000'000.

Report Breath-First Search

Breve descrizione

Nella teoria dei grafi, la Breadth-First Search (in italiano “ricerca in ampiezza”), è un algoritmo di ricerca per grafi che partendo da un vertice (o nodo) detto sorgente permette di cercare tutti gli altri nodi. La BFS si basa su livelli, ovvero per quanti nodi separano il nodo sorgente da un generico nodo v.



Algoritmo Sequenziale

Input: un grafo *G* e un nodo radice *v* appartenente a *G*

```
1   function BFS(G,v):
2       crea una coda Q
3       inserisci v in Q
4       marca v
5       while Q non è vuota:
6           t ← Q.dequeue()
7           for all archi e in G.incident_edge(t) do
8              u ← G.opposite_node(t,e)
9              if u non è marcato:
10                   marca u
11                   inserisci u in Q
12      return none
```

L’algoritmo può essere suddiviso in due semplici passi:

1) Prendo il vertice v dalla coda Q
2) Aggiungo in coda tutti i vicini di v che non sono già stati marcati

Algoritmi di parallelizzazione

La BFS ha una complessità O(|V| + |E|), ovvero, visitiamo ogni vertice esattamente una volta e ogni arco al massimo una volta. L'obiettivo durante la stesura dell'algoritmo parallelo è quello di non modificare tale complessità asintotica in modo tale da avere un'alta efficienza.

Una semplice strategia di parallelizzazione con complessità O(|V| + |E|) è la seguente: per ogni livello la BFS attraversa semplicemente tutti i vertici della coda in parallelo e crea una nuova coda. I nuovi vertici vengono aggiunti utilizzando l'operazione atomicAdd sulla posizione nella coda di output in ogni iterazione. Molto simile a quello sequenziale.

Esistono vari altri algoritmi più sofisticati. Una strategia potrebbe essere quella di espandere i vicini adiacenti in parallelo, implementare le frontiere dei vertici e dei bordi, utilizzare la somma dei prefissi locali al posto delle operazioni atomiche locali per determinare gli offset della coda e, infine, utilizzare una maschera di bit best-effort per un filtraggio efficiente dei vicini.

Algoritmo di parallelizzazione semplice

È stato scelto di implementare un semplice algoritmo di parallelizzazione molto simile a quello sequenziale in modo tale da avere un paragone in termini di tempo con gli altri sviluppati su CPU, sia puramente sequenziali, sia i paralleli (OpenMP e MPI).

L'algoritmo si basa su due code, F e N (frontiera e vicinato). Ogni thread ha un proprio vertice v da esplorare, esso esplora tutti i vicini di v e li inserisce all'interno di N.

Il processore si occupa di controllare la coda N se è vuota, di gestire queste code e di incrementare il livello. La GPU, invece, si occupa di esplorare i vertici e di aggiornare il vettore delle distanza (in questo caso contiene l'appartenenza al livello per ogni vertice).

Punti di forza

Il punto di forza più evidente è la semplicità nella realizzazione dato che i passi da seguire sono molto simili all'algoritmo sequenziale con l'aggiunta di una chiamata al kernel della GPU per il calcolo massiccio.

Punti di debolezza

Il punto di debolezza principale sta nel fatto che non ci sono tecniche aggiuntive per ottimizzare al meglio l'algoritmo, inoltre, utilizzando la matrice di adiacenza (la quale ha proporzionalità quadratica), non è possibile utilizzare grafi aventi molti vertici. Per evitare in parte questo problema, è stata utilizzata una lista di adiacenza con dimensione fissa in modo tale da evitare di avere molti elementi inutili e di conoscere a priori la dimensione.

Implementazione e dettagli specifici

Premesse

Dato che è stato implementato un algoritmo molto simile al sequenziale, esso non fa riferimento a scambio di informazioni tra i thread sia di diversi blocchi, sia dello stesso, per tale motivo sono state proposte solo due implementazioni: memoria globale e memoria texture.

Implementazione Global memory

CPU

Inizialmente sono state dichiarate tutte le variabili per la GPU ed è stato effettuato il trasferimento in memoria globale.

```
//Inizializzaione variabili GPU
int *d_adjacencyMatrix;
int *d_firstQueue;
int *d_secondQueue;
int *d_nextQueueSize;
int *d_distance;

const int size = n_vertices * sizeof(int);
const int adjacencySize = n_vertices * graph->max_neighbours * sizeof(int);

//Allocazione su GPU
cudaMalloc((void **)&d_adjacencyMatrix, adjacencySize);
cudaMalloc((void **)&d_firstQueue, size);
cudaMalloc((void **)&d_secondQueue, size);
cudaMalloc((void **)&d_distance, size);
cudaMalloc((void **)&d_nextQueueSize, sizeof(int));

cudaMemcpy(d_adjacencyMatrix, graph->adjMatrix, adjacencySize, cudaMemcpyHostToDevice);
cudaMemcpy(d_nextQueueSize, &NEXT_QUEUE_SIZE, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_firstQueue, &start_vertex, sizeof(int), cudaMemcpyHostToDevice);

graph->visited[start_vertex] = 1;
cudaMemcpy(d_distance, graph->visited, n_vertices * sizeof(int), cudaMemcpyHostToDevice);
```

Successivamente passo nella parte principale del codice. Il processore si occupa di gestire le code e di richiamare il kernel in modo tale da poter ottenere la prossima coda.

```
//Avvio il timer
STARTTIME
while (currentQueueSize > 0) {
  int *d_currentQueue;
  int *d_nextQueue;
  if (level % 2 != 0) {
    d_currentQueue = d_firstQueue;
    d_nextQueue = d_secondQueue;
  }
  else {
    d_currentQueue = d_secondQueue;
    d_nextQueue = d_firstQueue;
  }
  computeNextQueue<<<blocks, th_p_block>>> (d_adjacencyMatrix, graph->max_neighbours, d_distance, currentQueueSize,
                                            d_currentQueue, d_nextQueueSize, d_nextQueue, level);
  cudaDeviceSynchronize();
  level++;
  cudaMemcpy(&currentQueueSize, d_nextQueueSize, sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(d_nextQueueSize, &NEXT_QUEUE_SIZE, sizeof(int), cudaMemcpyHostToDevice);
}
//Salvo il tempo trascorso
STOPTIME
```

GPU

Il kernel si occupa di controllare i vari vertici della coda calcolata nella precedente iterazione ed esplorare i vicini. Per poter aggiungere gli elementi nella coda N, è stato usato l'atomicAdd in modo tale da garantire la mutua esclusione della risorsa. Il valore di ritorno dell'atomicAdd è il valore precedente alla somma, il quale corrisponde all'indirizzo di memoria dedicato per il vertice v, ovvero un vicino di u.

```
__global__ void computeNextQueue(int *adjacencyMatrix, int n_vertices, int *distance, int queueSize,
                                 int *currentQueue, int *nextQueueSize, int *nextQueue, int level) {
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;  // thread id
  //Creo un index per l'iterazione
  int idx_iter = tid;

  while (idx_iter < queueSize) {
    int current = currentQueue[idx_iter];
    for (int i = 0; i < n_vertices; i++) {
      int v = adjacencyMatrix[current * n_vertices + i];
      if(v==-1) break;
      if(distance[v] == 0){
        distance[v] = level + 1;
        int position = atomicAdd(nextQueueSize, 1);
        nextQueue[position] = v;
      }
    }
    idx_iter += blockDim.x * gridDim.x;
  }
}
```

CPU

L'ultima fase riguarda il trasferimento dell'array delle distanze dalla GPU alla CPU.

```
cudaMemcpy(graph->visited, d_distance, size, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

//Free delle variabili su GPU
cudaFree(d_adjacencyMatrix);
cudaFree(d_firstQueue);
cudaFree(d_secondQueue);
cudaFree(d_distance);
```

Implementazione Texture memory

CPU

Inizialmente sono state dichiarate tutte le variabili per la GPU ed è stato effettuato il trasferimento in memoria texture.

```
//Inizializzaione variabili GPU
int *d_adjacencyMatrix;
int *d_firstQueue;
int *d_secondQueue;
int *d_nextQueueSize;
int *d_distance;

const int size = n_vertices * sizeof(int);
const int adjacencySize = n_vertices * graph->max_neighbours * sizeof(int);

//Allocazione su GPU
cudaMalloc((void **)&d_adjacencyMatrix, adjacencySize);
cudaMalloc((void **)&d_firstQueue, size);
cudaMalloc((void **)&d_secondQueue, size);
cudaMalloc((void **)&d_distance, size);
cudaMalloc((void **)&d_nextQueueSize, sizeof(int));

cudaMemcpy(d_adjacencyMatrix, graph->adjMatrix, adjacencySize, cudaMemcpyHostToDevice);
cudaMemcpy(d_nextQueueSize, &NEXT_QUEUE_SIZE, sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_firstQueue, &start_vertex, sizeof(int), cudaMemcpyHostToDevice);

graph->visited[start_vertex] = 1;
cudaMemcpy(d_distance, graph->visited, n_vertices * sizeof(int), cudaMemcpyHostToDevice);

cudaChannelFormatDesc channel = cudaCreateChannelDesc<int>();
cudaBindTexture(0,text_mem, d_adjacencyMatrix, channel);
```

Successivamente passo nella parte principale del codice. Il processore si occupa di gestire le code e di richiamare il kernel in modo tale da poter ottenere la prossima coda.

```
//Avvio il timer
STARTTIME
while (currentQueueSize > 0) {
  int *d_currentQueue;
  int *d_nextQueue;
  if (level % 2 != 0) {
    d_currentQueue = d_firstQueue;
    d_nextQueue = d_secondQueue;
  }
  else {
    d_currentQueue = d_secondQueue;
    d_nextQueue = d_firstQueue;
  }
  computeNextQueue<<<blocks, th_p_block>>> (graph->max_neighbours, d_distance, currentQueueSize,
                                            d_currentQueue, d_nextQueueSize, d_nextQueue, level);
  cudaDeviceSynchronize();
  level++;
  cudaMemcpy(&currentQueueSize, d_nextQueueSize, sizeof(int), cudaMemcpyDeviceToHost);
  cudaMemcpy(d_nextQueueSize, &NEXT_QUEUE_SIZE, sizeof(int), cudaMemcpyHostToDevice);
}
//Salvo il tempo trascorso
STOPTIME
```

GPU

Il kernel si occupa di controllare i vari vertici della coda calcolata nella precedente iterazione ed esplorare i vicini. Per poter aggiungere gli elementi nella coda N, è stato usato l'atomicAdd in modo tale da garantire la mutua esclusione della risorsa. Il valore di ritorno dell'atomicAdd è il valore precedente alla somma, il quale corrisponde all'indirizzo di memoria dedicato per il vertice v, ovvero un vicino di u. La differenza rispetto alla Global memory sta nel fatto che la lettura del vertice avviene tramite la texture memory.

```
__global__ void computeNextQueue(int n_vertices, int *distance, int queueSize, int *currentQueue,
                                 int *nextQueueSize, int *nextQueue, int level) {
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;  // thread id
  //Creo un index per l'iterazione
  int idx_iter = tid;

  while (idx_iter < queueSize) {
    int current = currentQueue[idx_iter];
    for (int i = 0; i < n_vertices; i++) {
      int v = tex1Dfetch(text_mem,current * n_vertices + i);
      if(v==-1) break;
      if(distance[v] == 0){
        distance[v] = level + 1;
        int position = atomicAdd(nextQueueSize, 1);
        nextQueue[position] = v;
      }
    }
    idx_iter += blockDim.x * gridDim.x;
  }
}
```

CPU

L'ultima fase riguarda il trasferimento dell'array delle distanze dalla GPU alla CPU.

```
cudaUnbindTexture(text_mem);

cudaMemcpy(graph->visited, d_distance, size, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

//Free delle variabili su GPU
cudaFree(d_adjacencyMatrix);
cudaFree(d_firstQueue);
cudaFree(d_secondQueue);
cudaFree(d_distance);
```

Analisi delle misure

Misure 500'000 vertici – Global Memory vs Texture Memory

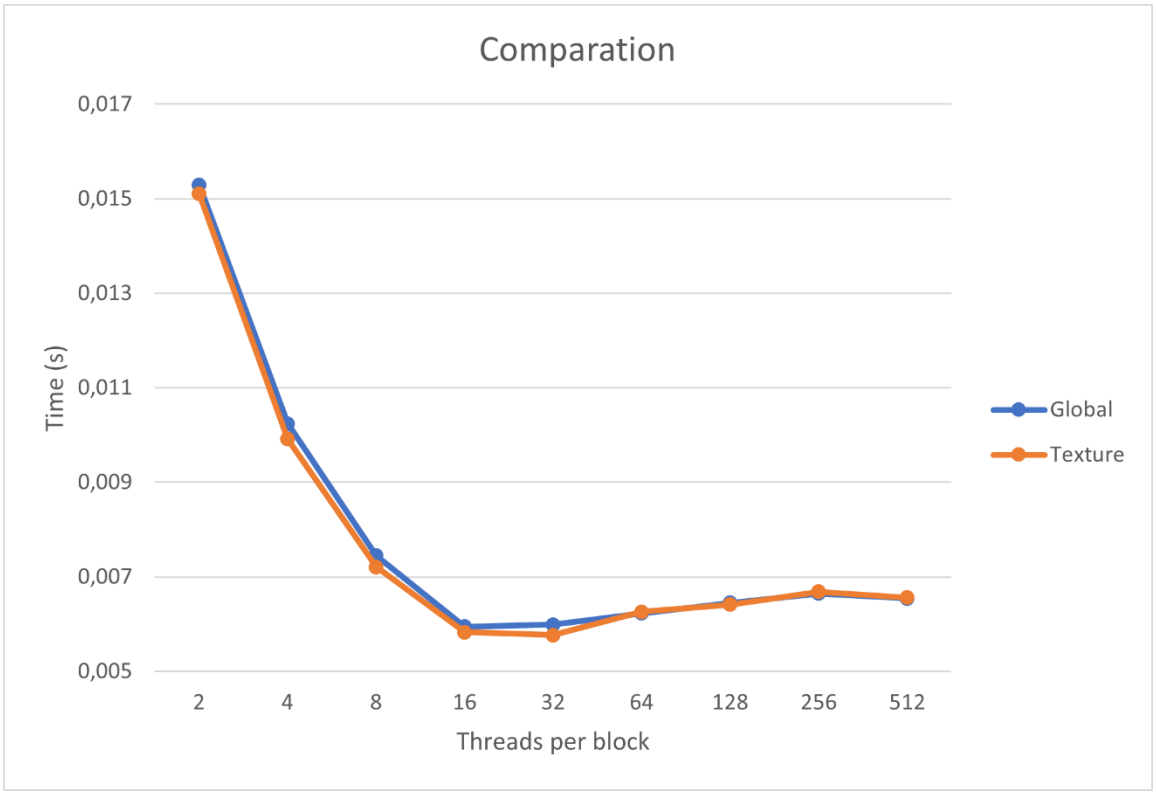
Global memory

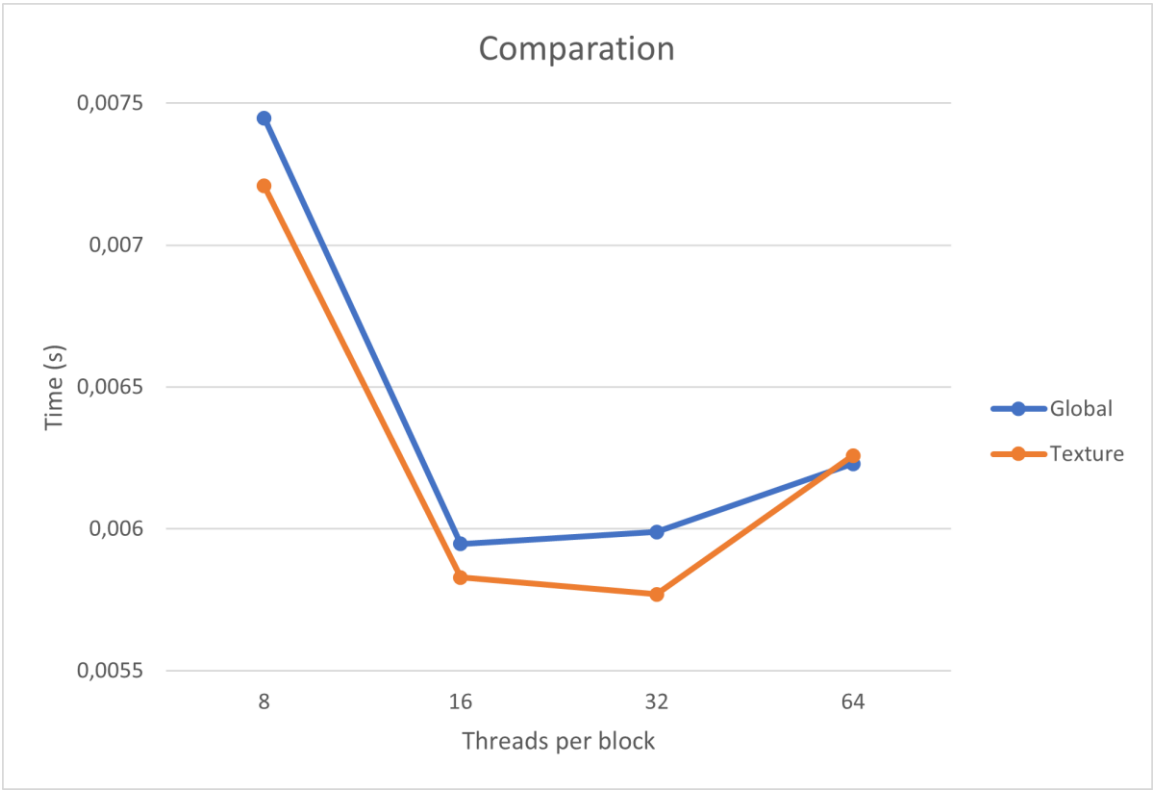
Size	*BlockSize*	*GridSize*	*Time*
500000	*2*	*13312*	*0,015292*
500000	*4*	*6656*	*0,010241*
500000	*8*	*3328*	*0,007448*
500000	*16*	*1664*	*0,005948*
500000	*32*	*832*	*0,005989*
500000	*64*	*416*	*0,006229*
500000	*128*	*208*	*0,006453*
500000	*256*	*104*	*0,00665*
500000	*512*	*52*	*0,006544*

Texture Memory

Size	*BlockSize*	*GridSize*	*Time*
500000	*2*	*13312*	*0,015107*
500000	*4*	*6656*	*0,009913*
500000	*8*	*3328*	*0,007208*
500000	*16*	*1664*	*0,005829*
500000	*32*	*832*	*0,00577*
500000	*64*	*416*	*0,006258*
500000	*128*	*208*	*0,006412*
500000	*256*	*104*	*0,006685*
500000	*512*	*52*	*0,006559*

Confronto


Comparation
0,017
0,015
0,013
0,011
0,009
0,007
0,005
Time (s)
2
4
8
16
32
64
128
256
512
Threads per block
Global
Texture


Comparation
0,0075
0,007
0,0065
0,006
0,0055
Time (s)
8
16
32
64
Threads per block
Global
Texture

Misure 1’000'000 vertici – Global Memory vs Texture Memory

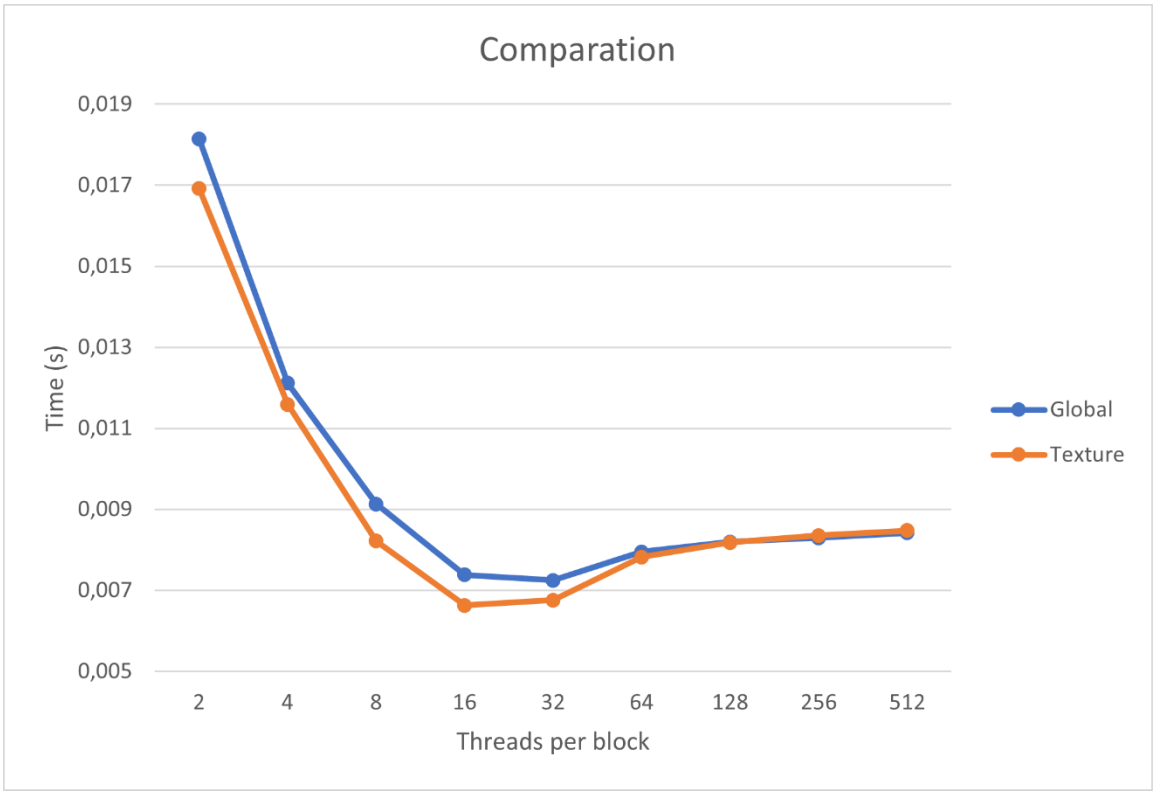
Global memory

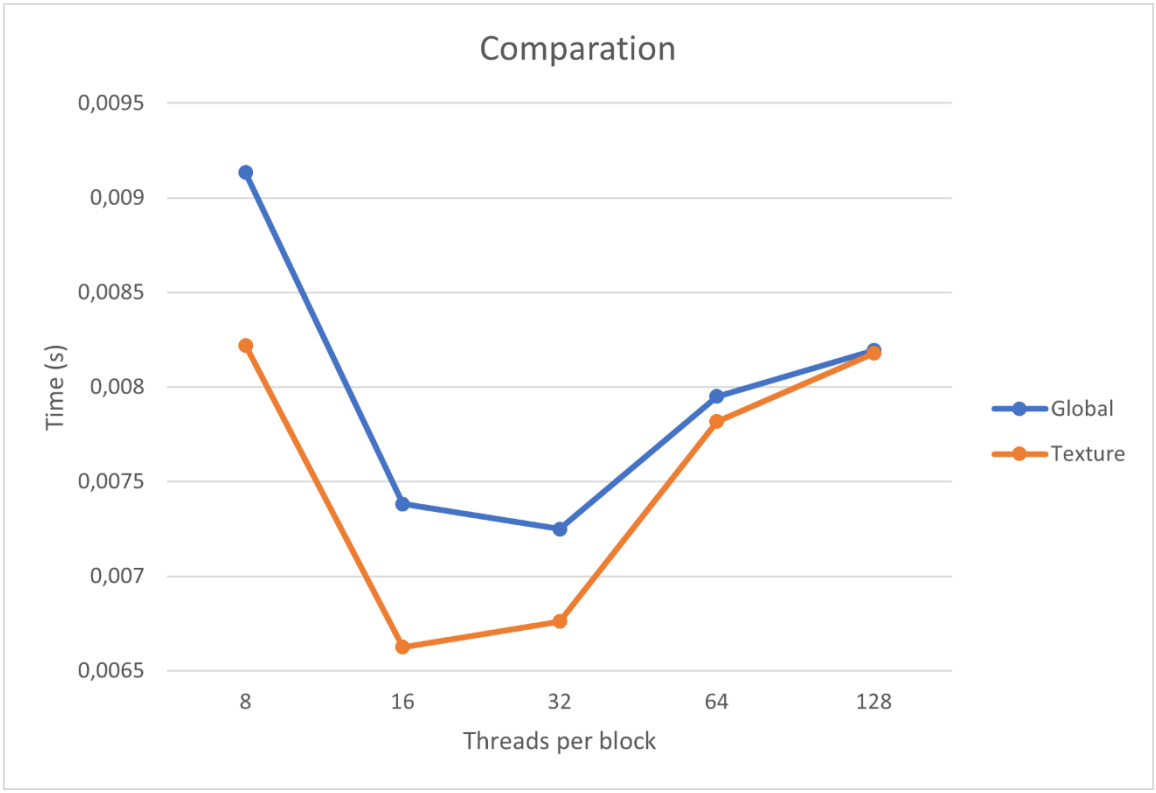
Size	BlockSize	GridSize	Time
1000000	2	13312	0,018138
1000000	4	6656	0,012114
1000000	8	3328	0,009134
1000000	16	1664	0,007382
1000000	32	832	0,007249
1000000	64	416	0,00795
1000000	128	208	0,008194
1000000	256	104	0,008294
1000000	512	52	0,008418

Texture Memory

Size	BlockSize	GridSize	Time
1000000	2	13312	0,016919
1000000	4	6656	0,011587
1000000	8	3328	0,008219
1000000	16	1664	0,006626
1000000	32	832	0,006761
1000000	64	416	0,007818
1000000	128	208	0,008178
1000000	256	104	0,00835
1000000	512	52	0,008476

Confronto


Comparation
0,019
0,017
0,015
0,013
0,011
0,009
0,007
0,005
Time (s)
2
4
8
16
32
64
128
256
512
Threads per block
Global
Texture


Comparation
0,0095
0,009
0,0085
0,008
0,0075
0,007
0,0065
Time (s)
8
16
32
64
128
Threads per block
Global
Texture

Conclusioni e considerazioni

L'algoritmo implementato risulta mediamente efficiente su grafi con molti vertici. Come è stato già accennato in precedenza, tale algoritmo è atto a spiegare come l'utilizzo di G-GPU è molto utile in applicazioni reali.

Infatti, come possiamo notare dalla tabella seguente, il tempo necessario ad effettuare una BFS su CPU con 12 thread in memoria condivisa (OpenMP) su un grafo di 10'000 vertici è molto simile al tempo necessario ad effettuare una BFS su GPU con 26.624 threads (32 x 832) su un grafo di 1'000'000 di vertici.

Type	Threads	Vertices	Time elapsed
Sequential O3	*1*	*10'000*	*0,016932*
OpenMP O3	*12*	*10'000*	*0,005203*
MPI	*12*	*10'000*	*0,02291*
CUDA – Global Memory	*32 x 832*	*1'000'000*	*0,007249*
Cuda – Texture Memory	*32 x 832*	*1'000'000*	*0,006761*

Inoltre, l'utilizzo della Texture Memory per poter leggere dalla Global Memory risulta più efficiente, dato che ogni thread esplorerà elementi adiacenti, i quali corrispondono ai vicini di un vertice v.

Come eseguire i test

1) Utilizza il notebook colab nella cartella condivisa di google drive.

2) Per poter visualizzare i dati relativi alle misurazioni, aprire il file excel “BFS_measure_CUDA”, dirigersi in “Dati” e cliccare su “Aggiorna tutti”. In questo modo automaticamente verrano presi i dati e inseriti in excel per poterli vedere sia in formato testuale, sia sottoforma di grafici.