

Parallelizzazione dell'algoritmo di intersezione di MöllerTrumbore

Gallina Roberto

06/11/2023

Indice

1	Algoritmo di MöllerTrumbore	3
2	Prima semplice implementazione	4
2.1	Struttura Point3D	4
2.2	Struttura Triangle	4
2.3	Differenza tra vettori	4
2.4	Dot product	4
2.5	Cross product	4
2.6	Codice	5
2.6.1	Raggio interseca triangolo	6
2.6.2	Raggio non interseca triangolo	6
3	Esempio complesso	7
3.1	Lettura dei file	7
3.2	Implementazione	9
3.3	Output su file	9
4	Parallelizzazione	10
4.1	Introduzione a CUDA	10
4.1.1	Passaggio dati	11
4.2	Implementazione algoritmo nel Kernel	13
4.3	Uso della shared memory	15
5	Metriche	16
5.1	Specifiche	16
5.2	Metriche	16
5.3	Profiling	17
5.3.1	Parallelo senza SH	17
5.3.2	Parallelo con SH (Size 256)	18
5.4	Conclusioni	18

1 Algoritmo di MöllerTrumbore

L'algoritmo di intersezione di MöllerTrumbore è un metodo usato per capire se un raggio (linea con un punto d'inizio e una direzione) intersecano un triangolo nello spazio 3D. È usato nella Computer Graphics per determinare quali parti di modelli 3D sono visibili da un certo punto di vista.

```
1  bool RayIntersectsTriangle(  
2      Vector3D rayOrigin,  
3      Vector3D rayVector,  
4      Triangle* inTriangle,  
5      Vector3D& outIntersectionPoint  
6  ){  
7      const float EPSILON = 0.0000001;  
8      Vector3D vertex0 = inTriangle->vertex0;  
9      Vector3D vertex1 = inTriangle->vertex1;  
10     Vector3D vertex2 = inTriangle->vertex2;  
11  
12     Vector3D edge1, edge2, h, s, q;  
13     edge1 = vertex1 - vertex0;  
14     edge2 = vertex2 - vertex0;  
15  
16     float a, f, u, v;  
17     h = rayVector.crossProduct(edge2);  
18     a = edge1.dotProduct(h);  
19  
20     if (a > -EPSILON && a < EPSILON)  
21         return false;  
22  
23     f = 1.0 / a;  
24     s = rayOrigin - vertex0;  
25     u = f * s.dotProduct(h);  
26  
27     if (u < 0.0 || u > 1.0)  
28         return false;  
29  
30     q = s.crossProduct(edge1);  
31     v = f * rayVector.dotProduct(q);  
32  
33     if (v < 0.0 || u + v > 1.0)  
34         return false;  
35  
36     float t = f * edge2.dotProduct(q);  
37  
38     if (t > EPSILON){  
39         // ray intersection  
40         outIntersectionPoint = rayOrigin + rayVector * t;  
41         return true;  
42     }else  
43         return false;  
44 }
```

Figura 1: Algoritmo di MöllerTrumbore da Wikipedia

2 Prima semplice implementazione

Prima di procedere con la prima implementazione, va da creare le strutture e le funzioni basilari che serviranno nei vari calcoli.

2.1 Struttura Point3D

```
1 struct Point3D{
2     double x;
3     double y;
4     double z;
5 };
```

2.2 Struttura Triangle

```
1 struct Triangle {
2     Point3D p1;
3     Point3D p2;
4     Point3D p3;
5 };
```

2.3 Differenza tra vettori

```
1 Point3D difference(Point3D a, Point3D b){
2     return {a.x - b.x, a.y - b.y, a.z - b.z};
3 }
```

2.4 Dot product

```
1 double dotProduct(Point3D &v1, Point3D &v2) {
2     return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
3 }
```

2.5 Cross product

```
1 Point3D crossProduct(Point3D &v1, Point3D &v2) {
2     return {
3         v1.y * v2.z - v1.z * v2.y,
4         v1.z * v2.x - v1.x * v2.z,
5         v1.x * v2.y - v1.y * v2.x
6     };
7 }
```

2.6 Codice

Vado quindi ad implementare la versione del codice vista precedentemente

```
1  bool rayIntersectsTriangle(  
2      Point3D rayOrigin,  
3      Point3D rayVector,  
4      Triangle inTriangle  
5  ){  
6      const float EPSILON = 0.0000001;  
7      Point3D vertex0 = inTriangle.p1;  
8      Point3D vertex1 = inTriangle.p2;  
9      Point3D vertex2 = inTriangle.p3;  
10  
11     Point3D edge1, edge2, h, s, q;  
12     double a, f, u, v;  
13  
14     edge1 = difference(vertex1, vertex0);  
15     edge2 = difference(vertex2, vertex0);  
16  
17     h = crossProduct(rayVector, edge2);  
18     a = dotProduct(edge1, h);  
19  
20     if (a > -EPSILON && a < EPSILON)  
21         return false;  
22  
23     f = 1.0 / a;  
24     s = difference(rayOrigin, vertex0);  
25     u = f * dotProduct(s, h);  
26  
27     if (u < 0.0 || u > 1.0)  
28         return false;  
29  
30     q = crossProduct(s, edge1);  
31     v = f * dotProduct(rayVector, q);  
32     if (v < 0.0 || u + v > 1.0)  
33         return false;  
34  
35     double t = f * dotProduct(edge2, q);  
36  
37     if (t > EPSILON)  
38         return true;  
39  
40     return false;  
41 }
```

Implemento ora il 'main' verificando che l'algoritmo funzioni correttamente sui casi base

2.6.1 Raggio interseca triangolo

```

1  int main() {
2      Point3D rayOrigin = {0.0, 0.0, 0.0};
3      Point3D rayDirection = {0.0, 0.0, 2.0};
4
5      Triangle triangle = {
6          {-0.5, 0.5, 0.5},
7          {0.5, 0.0, 0.5},
8          {0.5, -1.0, 0.5}
9      };
10
11     cout << rayIntersectsTriangle(rayOrigin, rayDirection,
12                                     triangle);
13
14     return 0;
15 }
```

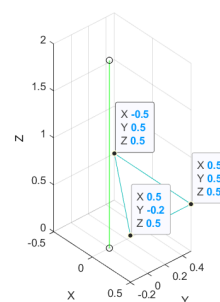
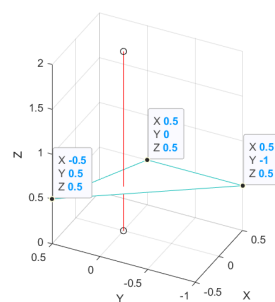
1 1

2.6.2 Raggio non interseca triangolo

```

1  ..
2  Triangle triangle = {
3      {-0.5, 0.5, 0.5},
4      {0.5, 0.5, 0.5},
5      {0.5, -0.2, 0.5}
6  };
7  ..
```

1 0



3 Esempio complesso

3.1 Lettura dei file

Vado ora a implementare delle funzioni per poter leggere i file dei punti e delle mesh, sappiamo infatti che il file *verts.csv* è composto da tre numeri con virgola scritti in notazione scientifica.

Es. $-6.195182353258132935e-02, -5.577753782272338867e-01, 5.792263150215148926e-01$

Vado quindi a creare una funzione che dato il nome del file genera un vettore di punti.

```
1  vector<Point3D> readPoints(string filename) {
2      vector<Point3D> punti;
3      string linea;
4      ifstream file(filename);
5      if (file.is_open()) {
6          while (getline(file, linea)) {
7              Point3D punto;
8              stringstream ss(linea);
9              string valore;
10             if (getline(ss, valore, ',')) {
11                 punto.x = stod(valore);
12             }
13             if (getline(ss, valore, ',')) {
14                 punto.y = stod(valore);
15             }
16             if (getline(ss, valore)) {
17                 punto.z = stod(valore);
18             }
19             punti.push_back(punto);
20         }
21         file.close();
22     }
23     return punti;
24 }
```

Allo stesso modo sappiamo che il file *mesh.csv* è composto da tre numeri scritti in notazione scientifica, corrispondenti al numero (indice) del punto.

Es. $1.0000000000000000e+00, 2.0000000000000000e+00, 0.0000000000000000e+00$

Vado quindi a creare una funzione che dato il nome del file e un vettore di punti genera un vettore di triangoli (mesh).

```
1  vector<Triangle> readTriangles(string filename, vector<Point3D>
   punti) {
2      vector<Triangle> triangoli;
3      string linea;
4      ifstream file(filename);
5      if (file.is_open()) {
6          while (getline(file, linea)) {
7              Triangle t;
8              stringstream ss(linea);
9              string valore;
10             if (getline(ss, valore, ',')) {
11                 index = (int) stod(valore);
12                 t.p1 = punti[index];
13             }
14             if (getline(ss, valore, ',')) {
15                 index = (int) stod(valore);
16                 t.p2 = punti[index];
17             }
18             if (getline(ss, valore)) {
19                 index = (int) stod(valore);
20                 t.p3 = punti[index];
21             }
22             triangoli.push_back(t);
23         }
24         file.close();
25     }
26     return triangoli;
27 }
```

Nel main uso queste funzioni per caricare i dati.

```
1  int main() {
2      vector<Point3D> punti = readPoints("verts.csv");
3      vector<Triangle> triangoli = readTriangles("meshes.csv",
   punti);
4      .
5  }
```


3.2 Implementazione

Vado ora a implementare una funzione che data l'origine, la direzione e un vettore di triangoli verifica se quel raggio interseca almeno un triangolo

```
1 bool rayIntersectsAnyTriangle(Point3D origin, Point3D dir,
   vector<Triangle> triangles) {
2     for (const Triangle &triangle : triangles) {
3         bool r = rayIntersectsTriangle(origin, dir, triangle);
4         if (r) {
5             return true;
6         }
7     }
8     return false;
9 }
```

Nel main vado quindi a verificare se ogni punto interseca almeno un triangolo

```
1 int main() {
2     vector<Point3D> punti = readPoints("verts.csv");
3     vector<Triangle> triangoli = readTriangles("meshes.csv",
4         punti);
5
6     Point3D rayOrigin = {0.0, 0.0, 0.0};
7
8     for (const Point3D &punto : punti) {
9         if (punto.x > 0 || punto.x <= 0)
10             cout << rayIntersectsAnyTriangle(rayOrigin, punto,
11                 triangoli) << endl;
12     }
13     return 0;
14 }
```

3.3 Output su file

Non rimane che aggiungere la stampa direttamente su un file.

```
1 int main(){
2     ofstream outFile("out.txt");
3     vector<Point3D> punti = readPoints("verts.csv");
4     vector<Triangle> triangoli = readTriangles("meshes.csv",
5         punti);
6
7     Point3D rayOrigin = {0.0, 0.0, 0.0};
8
9     for (const Point3D &punto : punti)
10     {
11         if (punto.x > 0 || punto.x <= 0)
12             outFile << rayIntersectsAnyTriangle(rayOrigin, punto,
13                 triangoli) << endl;
14     }
15     outFile.close();
16     return 0;
17 }
```

4 Parallelizzazione

4.1 Introduzione a CUDA

La prima cosa da parallelizzare è il ciclo che viene fatto su punti, ricordiamo infatti che per ogni punto viene verificato se il segmento che unisce l'origine con esso interseca qualche triangolo.

Vado quindi a creare un Kernel CUDA che presa l'origine, un punto e il vettore di triangoli (e una indirizzo dove salvare il risultato) verifichi la condizione.

Ricordiamo che le parti principali per usare un Kernel Cuda sono:

- Allocazione memoria del device
- Passaggio dati dall'host al device
- Passaggio dati dal device all'host
- Deallocazione memoria del device

4.1.1 Passaggio dati

La prima versione del kernel servirà per verificare il corretto passaggio dei dati tra l'*host* e il *device* pertanto restituirà sempre *true*.

Vado quindi a preparare i parametri da passare: il punto origine, un array di punti, un array di triangoli, il numero di triangoli e un array di booleani che serve per salvare i valori di ritorno. Per fare ciò definisco alcuni puntatori per l'array di punti e di triangoli e l'array di ritorno.

```
1 Point3D *d_points;  
2 Triangle *d_triangles;  
3 bool *d_result;
```

Vado poi ad allocarne lo spazio.

```
1 cudaMalloc(&d_points, Np * sizeof(Point3D) );  
2 cudaMalloc(&d_triangles, Nt * sizeof(Triangle) );  
3 cudaMalloc(&d_result, Np * sizeof(bool) );
```

E copio i dati.

```
1 cudaMemcpy(  
2     d_points, h_punti,  
3     Np * sizeof(Point3D),  
4     cudaMemcpyHostToDevice  
5 );  
6 cudaMemcpy(  
7     d_triangles,  
8     h_triangles,  
9     Nt * sizeof(Triangle),  
10    cudaMemcpyHostToDevice  
11 );
```

Andrò poi ad eseguire il kernel (spiegato successivamente), prima però calcolo la dimensione dei blocchi

```
1 dim3 DimGrid(Np/256, 1, 1);  
2 if (Np % 256)  
3     DimGrid.x++;  
4  
5 dim3 DimBlock(256, 1, 1);  
6  
7 rayIntersectsAnyTrianglesKernel<<<DimGrid, DimBlock>>>(  
8     origin,  
9     d_points,  
10    d_triangles,  
11    Nt,  
12    d_result  
13 );
```

E copiare i risultati.

```
1 cudaMemcpy(  
2     result,  
3     d_result,  
4     Np * sizeof(bool),  
5     cudaMemcpyDeviceToHost  
6 );
```

E infine libero lo spazio allocato.

```
1 cudaFree( d_points );  
2 cudaFree( d_triangles );  
3 cudaFree( d_result );
```

Il kernel sarà una funzione void e dovrà ricevere i parametri precedentemente preparato, ogni thread analizzerà un unico punto, per fare ciò calcola l'indice del punto che deve analizzare.

```
1  __global__
2  void rayIntersectsAnyTrianglesKernel(
3      Point3D rayOrigin,
4      Point3D *ps,
5      Triangle *ts,
6      int Nt,
7      bool *result
8  ){
9      int idx = blockIdx.x * blockDim.x + threadIdx.x;
10
11      result[idx] = true;
12  }
```

4.2 Implementazione algoritmo nel Kernel

Per definizione devo restituire *true* se almeno un triangolo è intersecato dal raggio che unisce l'origine al punto, pertanto il kernel sarà un ciclo su tutti i triangoli e se interseca restituirà *true* altrimenti a fine ciclo restituirà *false*.

```
1  __global__
2  void rayIntersectsAnyTrianglesKernel(
3      Point3D rayOrigin,
4      Point3D *ps,
5      Triangle *ts,
6      int Nt,
7      bool *result
8  ){
9      int idx = blockIdx.x * blockDim.x + threadIdx.x;
10
11     Point3D point = ps[idx];
12     Point3D dir = point;
13
14     result[idx] = false;
15
16     for(int i = 0; i < Nt; i++){
17         Triangle t = ts[i];
18
19         bool r = rayIntersectsTriangle(rayOrigin, dir, t);
20         if(r){
21             result[idx] = true;
22         }
23     }
24 }
```

Implemento quindi la funzione che verifica l'intersezione di un raggio in un triangolo, devo quindi definire le funzioni dotProduct, crossProduct e difference come funzioni `__device__` in modo che i thread possono usarle.

```
1  __device__
2  Point3D difference(Point3D a, Point3D b) {
3      return {a.x - b.x, a.y - b.y, a.z - b.z};
4  }
5
6  __device__
7  Point3D crossProduct(Point3D &v1, Point3D &v2){
8      return {    v1.y * v2.z - v1.z * v2.y,
9                 v1.z * v2.x - v1.x * v2.z,
10                v1.x * v2.y - v1.y * v2.x};
11 }
12
13 __device__
14 double dotProduct(Point3D &v1, Point3D &v2){
15     return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
16 }
```

Vado ora a creare una funzione partendo da quanto visto sulla Wiki, usando le funzioni precedentemente create.

```
1  __device__
2  bool rayIntersectsTriangle(
3      Point3D rayOrigin,
4      Point3D rayVector,
5      Triangle inTriangle
6  ){
7      const float EPSILON = 0.0000001;
8      Point3D vertex0 = inTriangle.p1;
9      Point3D vertex1 = inTriangle.p2;
10     Point3D vertex2 = inTriangle.p3;
11
12     Point3D edge1, edge2, h, s, q;
13     double a, f, u, v;
14
15     edge1 = difference(vertex1, vertex0);
16     edge2 = difference(vertex2, vertex0);
17
18     h = crossProduct(rayVector, edge2);
19     a = dotProduct(edge1, h);
20
21     if (a > -EPSILON && a < EPSILON)
22         return false;
23
24     f = 1.0 / a;
25     s = difference(rayOrigin, vertex0);
26     u = f * dotProduct(s, h);
27
28     if (u < 0.0 || u > 1.0)
29         return false;
30
31     q = crossProduct(s, edge1);
32     v = f * dotProduct(rayVector, q);
33     if (v < 0.0 || u + v > 1.0)
34         return false;
35
36     double t = f * dotProduct(edge2, q);
37
38     if (t > EPSILON)
39         return true;
40     return false;
41 }
```

4.3 Uso della shared memory

Attualmente ogni thread, per eseguire il calcolo, deve ciclare su tutti i triangoli, che risiedono in memoria globale; questo comporta un notevole tempo d'accesso. Vado quindi ad introdurre una memoria condivisa (shared) tra i vari thread, essa memorizzerà alcuni triangoli in modo che ci siano meno accessi in memoria globale.

Aggiungo la dimensione della shared memory, e dentro la funzione eseguita dai vari thread aggiungo un array di triangoli condiviso.

```
1 #define SH_SIZE 512

1 __global__
2 void rayIntersectsAnyTrianglesKernel(
3     ..
4 ){
5     ..
6     __shared__ Triangle sh_triangle[SH_SIZE];
7     ..
8 }
```

Essendo la dimensione della shared minore del numero totale di triangoli, sarà necessario caricare una parte di triangoli, attendere che tutti i thread finiscano di analizzarli, per poi caricare i successivi

```
1 __global__
2 void rayIntersectsAnyTrianglesKernel(
3     Point3D rayOrigin,
4     Point3D *ps,
5     Triangle *ts,
6     int Nt,
7     bool *result
8 ){
9     int idx = blockIdx.x * blockDim.x + threadIdx.x;
10
11     Point3D point = ps[idx];
12     Point3D dir = point;
13     result[idx] = false;
14
15     __shared__ Triangle sh_triangle[SH_SIZE];
16
17     bool res = false;
18
19     for(int m = 0; m < Nt/SH_SIZE; m++){
20         if(idx < SH_SIZE){
21             sh_triangle[idx] = ts[m * SH_SIZE + idx];
22         }
23         __syncthreads();
24
25         for(int i = 0; i < SH_SIZE; i++){
26             Triangle t = sh_triangle[i];
27
28             bool r = rayIntersectsTriangle(rayOrigin, dir, t);
29             if(r){
30                 res = true;
31             }
32         }
33         __syncthreads();
34     }
35     result[idx] = res;
36 }
```

5 Metriche

5.1 Specifiche

Tutte le esecuzioni sono state fatte su un pc con:

- CPU: AMD Ryzen 7 5800x 8-Core (3.80GHz)
- GPU: NVIDIA GeForce RTX 3060 (circa 3500 CUDA Core da 1.70 GHz)

5.2 Metriche

Tipo di codice	Tempo
Sequenziale	4391245824ns = 4391245µs 4391ms = 4.3s
Parallelo senza SH	536918784ns = 536918µs 536ms = 0.5s
Parallelo con SH (Size 256)	299524300ns = 299524µs 299ms = 0.3s

5.3 Profiling

Usando un programma di profiling (NVIDIA Nsight Compute) ho cercato di analizzare se i programmi scritti fossero compute-bound (spendevano più tempo eseguendo i calcoli) o memory-bound (spendevano più tempo eseguendo spostando i dati dalla memoria centrale a quella del dispositivo o viceversa).

5.3.1 Parallelo senza SH

Analizzando l'eseguibile che non usa la shared memory, noto che il tempo speso per la parte di computazione è quasi 15 volte superiore a quello per lo scambio dati in memoria (81.66% contro il 4.96%), l'eseguibile è sicuramente compute-bound.

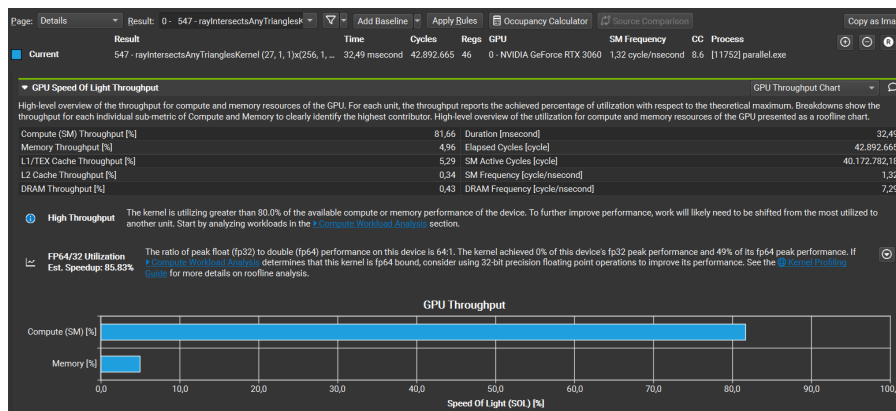


Figura 2: Profiling per l'eseguibile parallelo senza SH

Analizzo poi il diagramma degli accessi in memoria, dai cui si nota come tutte le richieste passino dal Kernel, alla memoria globale (passando poi per le varie memorie cache) fino ad arrivare alla memoria del device.

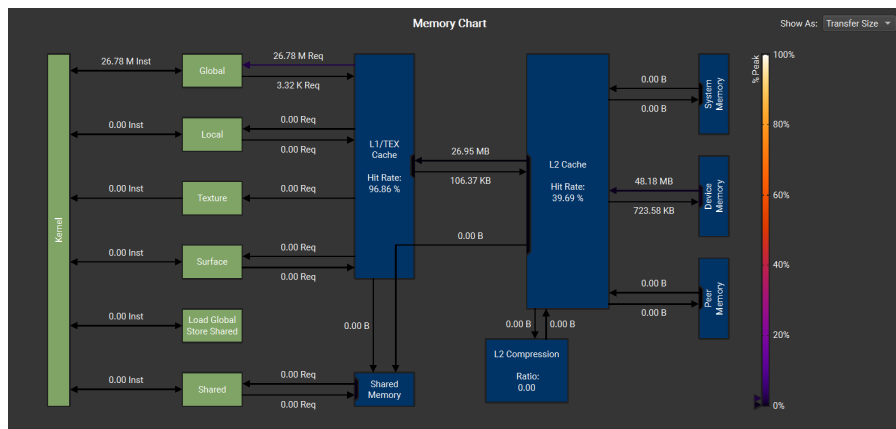


Figura 3: Memory access diagram per l'eseguibile parallelo senza SH

5.3.2 Parallelo con SH (Size 256)

Analizzando l'eseguibile che usa la shared memory, noto che anche in esso il tempo speso per la parte di computazione è quasi 15 volte superiore a quello per lo scambio dati in memoria (81.70% contro il 4.99%).

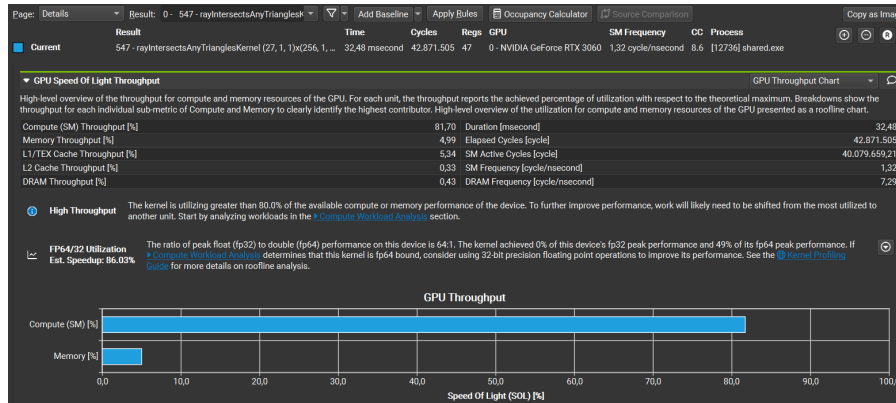


Figura 4: Profiling per l'eseguibile parallelo con SH

Invece, analizzando il diagramma degli accessi in memoria, si nota come quasi tutte le richieste passino dalla shared memory (passando poi per le varie memorie cache) fino ad arrivare alla memoria del device e questa è la motivazione delle prestazioni migliori.

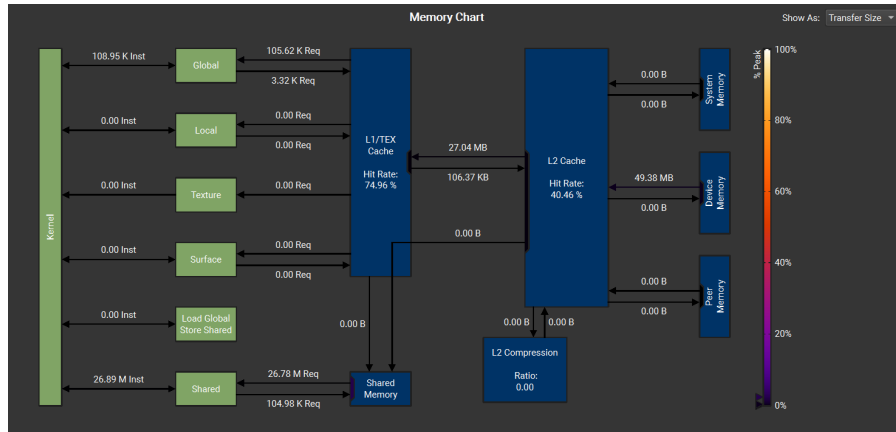


Figura 5: Memory access diagram per l'eseguibile parallelo con SH

5.4 Conclusioni

Anche se siamo su un caso di studio abbastanza piccolo, l'esecuzione tramite CPU risulta essere circa 10 volte più lenta rispetto all'uso della GPU; inoltre implementando la shared memory si vedono dei leggeri benefici, credo queste differenze verrebbero amplificate in un caso di studio più grande.