Generazione della catena di mipmap per un'immagine

Kevin Della Schiava

Luglio 2021

1 Introduzione

In questo documento verranno presentate possibili soluzioni e miglioramenti al problema di calcolare le mipmap di una singola immagine. Saranno presentati e analizzati diversi modi di parallelizzare la computazione attraverso strumenti e costrutti messi a disposizione da CUDA e OpenMP.

2 Descrizione del problema

Il problema che vogliamo risolvere è quello di generare una catena di mipmap per un'immagine. Una mipmap è una versione a risoluzione ridotta dell'immagine originale, utilizzata per attenuare fenomeni di aliasing in applicazioni di grafica tridimensionale oppure per data augmentation nel machine learning.
Partendo dall'immagine in input creiamo una mipmap che ha metà dell'altezza e della larghezza dell'originale, ripetendo questo procedimento sull'immagine così generata creiamo una "catena" di immagini progressivamente più piccole[1].

[1] è possibile anche arrivare a un immagine con un singolo pixel, più comunemente si sceglie una dimensione minima

3 Soluzione e Miglioramenti

3.1 Soluzione seriale

Il problema può essere risolto applicando un filtro box all'immagine originale per crearne una versione a più bassa risoluzione. Ripetendo il procedimento su questa nuova immagine a più bassa risoluzione possiamo generare tutta la catena di mipmap.
Un filtro box è implementato attraverso una convoluzione in cui ogni elemento del kernel ha peso $\frac{1}{k_s}$, dove k_s è la dimensione in pixel del filtro. Questa convoluzione viene applicata con stride k_w (la dimensione del filtro su un asse) per ottenere un'immagine che ha dimensioni $(\frac{larghezza}{k_w}, \frac{altezza}{k_w})$.
L'algoritmo seriale per ottenere l'intera catena di mipmap è quindi (per semplicità omettiamo la lettura e scrittura delle immagini su disco):

Algorithm 1 Generazione seriale della catena di mipmap

```
1:  procedure GenerateMipMap(in_image, out_image)
2:      k_w è la dimensione del kernel su un asse
3:      k_s è la dimensione totale del kernel (k_w^2)
4:      for pixel in out_image do
5:          result ← (0, 0, 0, 0)
6:          x, y ← pixel.coords
7:          for row from 0 to k_w do
8:              for column from 0 to k_w do
9:                  in_image_coords ← (k_w * x + row, k_w * y + column)
10:                 //Evitiamo coordinate fuori dall'immagine in input
11:                 in_image_coords ← max(in_image_coords, (0, 0))
12:                 max_image_coords ← in_image.size − (1, 1)
13:                 in_image_coords ← min(in_image_coords, max_image_coords)
14:                 result ← result + 1/k_s * in_image[in_image_coords]
15:         out_image[pixel.coords] ← result
16: procedure MipChain(in_image)
17:     mip_levels ← [n_of_mips + 1]      ▷ contenitori per i livelli delle mipmap
18:     mip_levels[0] ← in_image          ▷ la prima immagine è l'originale
19:     for i from 0 to mip_levels.size − 1 do
20:         GenerateMipMap(mip_levels[i], mip_levels[i + 1])
```

Partendo dall'immagine originale, che consideriamo la prima mipmap nella catena, applichiamo il filtro box per calcolare l'immagine successiva. Quando la seconda immagine nella catena è stata calcolata possiamo procedere all'applicazione del filtro box su di essa per generare il livello successivo e così via fino a che arriviamo ad aver generato ogni della catena. In questo modo riduciamo progressivamente la risoluzione dell'immagine in input ottenendo così tutta la catena di mipmap.

3.2 Implementazione in CUDA

Parallelizziamo ora lo schema di risoluzione illustrato precedentemente sfruttando i costrutti messi a disposizione da CUDA.

Algorithm 2 Generazione attraverso CUDA della catena di mipmap

1: **procedure** GENERATEMIPMAP(in_image, out_image)
2: k_w è la dimensione del kernel su un asse
3: k_s è la dimensione totale del kernel (k_w^2)
4: $block\_coords \leftarrow (blockDim.x * blockIdx.x, blockDim.y * blockIdx.y)$
5: $x, y \leftarrow block\_coords + threadIdx$
6: **if** $x >= out\_image.size.x$ or $y >= out\_image.size.y$ **then**
7: *Return*
8: $result \leftarrow (0, 0, 0, 0)$
9: **for** row from 0 to k_w **do**
10: **for** $column$ from 0 to k_w **do**
11: $in\_image\_coords \leftarrow (k_w * x + row, k_w * y + column)$
12: //Evitiamo coordinate fuori dall'immagine in input
13: $in\_image\_coords \leftarrow max(in\_image\_coords, (0, 0))$
14: $max\_image\_coords \leftarrow in\_image.size - (1, 1)$
15: $in\_image\_coords \leftarrow min(in\_image\_coords, max\_image\_coords)$
16: $result \leftarrow result + \frac{1}{k_s} * in\_image[in\_image\_coords]$
17: $out\_image[(x, y)] \leftarrow result$
18: **procedure** MIPCHAIN(in_image)
19: $mip\_levels \leftarrow [n\_of\_mips + 1]$ ▷ contenitori per i livelli delle mipmap
20: $mip\_levels[0] \leftarrow in\_image$ ▷ la prima immagine è l'originale
21: ...Codice per allocare le immagini sul device
22: ...e copiare l'immagine originale
23: **for** i from 0 to $mip\_levels.size - 1$ **do**
24: //Calcolo dei blocchi necessari all'elaborazione
25: //di tutta l'immagine
26: $block\_dims \leftarrow (32, 32, 1)$
27: $block\_num \leftarrow ceiling(mip\_levels[i + 1].size/block\_dims)$
28: $GenerateMipMap <<< block\_num, block\_dims >>>$
29: $(mip\_levels\_dev[i], mip\_levels\_dev[i + 1])$
30: ..Codice per trasferire le immagini calcolate
31: ..dal device all'host

A ogni thread viene associato un pixel dell'immagine in output, questo thread effettua la convoluzione dei pixel dell'immagine in input con il filtro e scrive il risultato nelle coordinate associate al thread.
Per iniziare ad applicare il filtro a un immagine abbiamo bisogno che tutte le scritture sull'immagine che la precede nella catena siano state completate per

evitare di leggere dei pixel in cui le scritture non sono ancora terminate. Dobbiamo quindi aspettare che abbiano completato le scritture non solo i thread nel singolo blocco ma anche tutti i thread nei vari blocchi, purtroppo CUDA non offre dei meccanismi di sincronizzazione tra i blocchi quindi dobbiamo utilizzare i lanci dei kernel come punto di sincronizzazione globale.
In molte applicazioni che manipolano immagini ed eseguono convoluzioni può risultare più efficiente utilizzare la shared memory per evitare di rileggere più volte il valore dello stesso pixel dalla memoria principale. La latenza della memoria principale è infatti molto più alta di quella della shared memory, arrivando addirittura a un fattore di 100 per l'accesso a memoria globale non presente in nessuna cache[2]. Questo si verifica poiché le convoluzioni, se applicate con uno stride di 1 (il centro della convoluzione avanza di un pixel alla volta), leggono più volte molti pixel. Ad esempio in un filtro gaussiano 3x3 che avanza con stride 1 abbiamo che 6 pixel sono letti due volte su un totale di 18 pixel letti. In questo problema però non ci troviamo mai nella situazione di leggere due volte lo stesso pixel, ogni thread infatti legge solamente una volta un'area univoca dell'immagine di input. Pertanto risulta molto più efficiente accumulare i valori in un registro che ha una latenza inferiore anche a quella della shared memory.

3.3 Miglioramenti

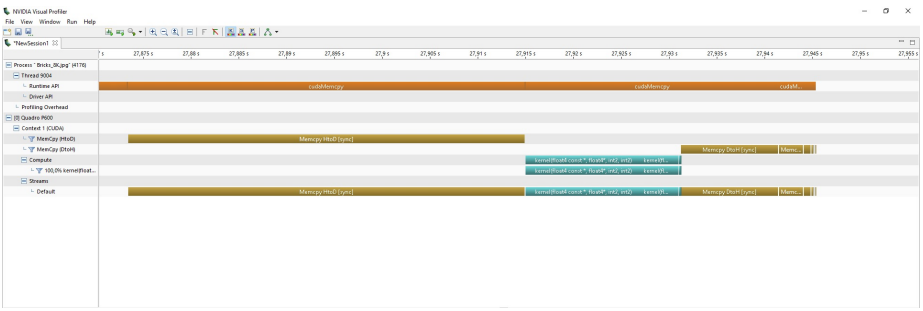

Figure 1: GPU Timeline utilizzando le funzioni sincrone

Analizzando la soluzione precedente attraverso nvvp vediamo che una sostanziale fetta del tempo è spesa nella fase di trasferimento dei dati (i box colorati di marrone denominati HtoD e DtoH), mentre i kernel (i box in azzurro) occupano meno di metà del tempo.
Possiamo migliorare la soluzione precedente facendo in modo che trasferimenti ed elaborazioni abbiano la possibilità di sovrapporsi. Per rendere questa sovrapposizione possibile dividiamo le immagini in bande orizzontali[3] e le trasferiamo contemporaneamente, elaborandole poi tutte in parallelo.
Questo schema a colpo d'occhio può sembrare identico al precedente, l'unica

[2] `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`
[3] Le bande orizzontali ci permettono di copiare un blocco di memoria contigua

differenza è che il trasferimento e l'elaborazione sono suddivisi su più canali. Dobbiamo però tenere conto delle risorse fisiche del dispositivo su cui effettuiamo l'elaborazione, le moderne GPU infatti possono eseguire due trasferimenti parallelamente a patto che questi siano in direzioni opposte (Ad esempio due copie HtoD non vengono parallelizzate). Inoltre se lanciamo diversi kernel non è detto che questi vengano eseguiti parallelamente dato che un solo kernel potrebbe saturare la GPU occupandola totalmente e costringendola ad eseguire successivamente il secondo kernel.
Questo modello quindi, considerando le risorse del device, ci permette di ottenere un pipelining dei trasferimenti e delle elaborazioni, preservando l'abilità di sfruttare GPU più avanzate con molte unità di elaborazione.
Per mettere in pratica questo schema utilizziamo OpenMP e diversi CUDA streams per registrare contemporaneamente i comandi di trasferimento e di elaborazione di ciascuna banda. Per ogni stream registriamo i comandi di trasferimento da host a device (HtoD), di elaborazione (kernel) e di trasferimento da device a host (DtoH), che, essendo sullo stesso stream, non avranno bisogno di ulteriore sincronizzazione. Utilizziamo poi gli eventi (costrutti messi a disposizione da cuda per la sincronizzazione tra stream) per segnalare al device che vogliamo aspettare che tutte le bande di un immagine abbiano terminato l'elaborazione prima di iniziare il livello successivo della catena di mipmap. Inoltre dato che effettuiamo trasferimenti asincroni tutta la memoria allocata dalla CPU dovrà essere allocata come pinned.

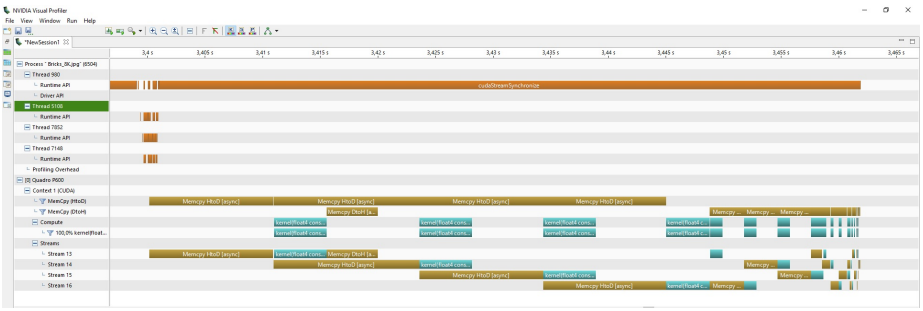

Figure 2: GPU Timeline utilizzando le funzioni asincrone e OpenMP

Per evitare che l'overhead della sincronizzazione e della divisione in bande superi i benefici ottenuti questo schema viene applicato fino a che le dimensioni dell'immagine sono sopra un certo limite. Quando l'immagine diventa molto piccola utilizziamo rispettivamente il primo e secondo stream per le elaborazioni e per i trasferimenti per godere comunque dei vantaggi della sovrapposizione delle due operazioni.
La figura 2 conferma le nostre supposizioni sui kernel, anche usando diversi stream non otteniamo un' esecuzione parallela poiché un solo kernel satura tutta la GPU che deve quindi far attendere gli altri. Vediamo anche che i trasferimenti nella stessa direzione vengono eseguiti uno alla volta, mentre quelli in direzioni

opposte vengono invece eseguiti in parallelo.
Inoltre attraverso l'uso di OpenMP è stato possibile parallelizzare la scrittura delle immagini su disco. Questo è stato fatto perché prima della scrittura è necessario convertire le immagini in un formato appropriato, operazione che trae beneficio dalla parallelizzazione offerta da OpenMP.

4 Raccolta dei dati e Risultati Sperimentali

4.1 Raccolta dei dati

Gli algoritmi sviluppati e utilizzati per la comparazione sono:

1. **Seriale** (serial.cu): Tutta la computazione è completamente seriale.
2. **OpenMP** (openmp.cu): l'iterazione su ciascun pixel dell'immagine è parallelizzato attraverso il costrutto di OpenMP parallel for.
3. **CUDA** (cudasync.cu): la generazione di ogni livello della catena di mipmap è parallelizzata utilizzando un kernel CUDA. Le funzioni di CUDA utilizzate sono quelle bloccanti.
4. **OpenMP + CUDA** (openmpcuda.cu): la generazione di ogni livello della catena di mipmap è effettuata attraverso il metodo descritto nella sezione 3.3.

Misuriamo i tempi di esecuzione su tre versioni di grandezza differente della stessa texture[4].
Le categorie di grandezza sono:

Categoria	Larghezza	Altezza
(2K) M	2048	1024
(4K) L	4096	2048
(8K) XL	8916	4096

Generiamo le mipmap fino a che una dimensione dell'immagine è maggiore di un certo limite (32 pixel nel nostro caso).
Oltre a registrare il tempo totale di esecuzione misuriamo anche il tempo che l'algoritmo impiega per:

1. Leggere l'immagine dal disco.
2. Generare tutte le mipmap della catena (inclusi eventuali trasferimenti tra host e device).
3. Scrivere tutte le immagini sul disco.

[4]Dal sito `https://kaimoisch.com/free-textures/brick-wall-red-15x3-seamless/`

Poiché queste tre fasi non sempre sono necessarie, pensiamo ad esempio alla generazione di mipmap per un'immagine che è già nella memoria RAM del dispositivo oppure a un processo in cui alla fine le mipmap vengono semplicemente scartate.
Le misure dei tempi vengono effettuate su 10 esecuzioni dell'algoritmo per attenuare eventuali fattori esterni che possono influenzare le misurazioni, i dati presentati nei grafici sono la media di questi 10 valori.
Tutti i test sono stati effettuati sullo stesso computer con le specifiche sottostanti.

CPU	Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz (4 CPUs), 3.0GHz
GPU	NVIDIA Quadro P600
RAM	8192MB RAM

4.2 Risultati Sperimentali

Analizziamo ora i tempi totali di esecuzione dei vari algoritmi.

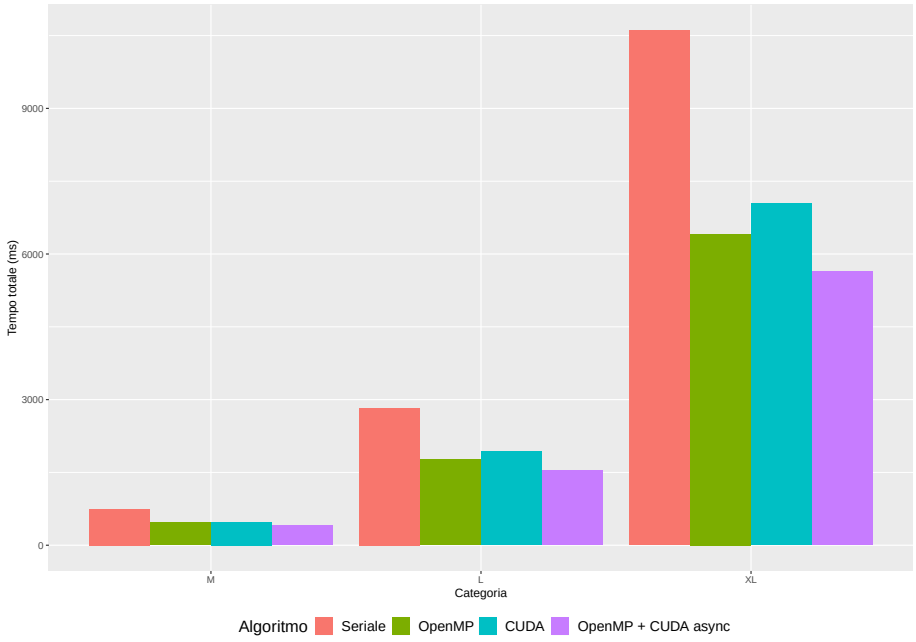

Figure 3: Tempi totali per ogni algoritmo per ogni categoria

Sorprendentemente la versione CUDA sembra addirittura avere un impatto negativo sul tempo di esecuzione dell'algoritmo risultando più lenta della versione OpenMP, per capire le cause di questo fenomeno dobbiamo isolare e osservare esclusivamente i tempi di esecuzione e scrittura (i tempi di lettura sono uguali per tutte le versioni).

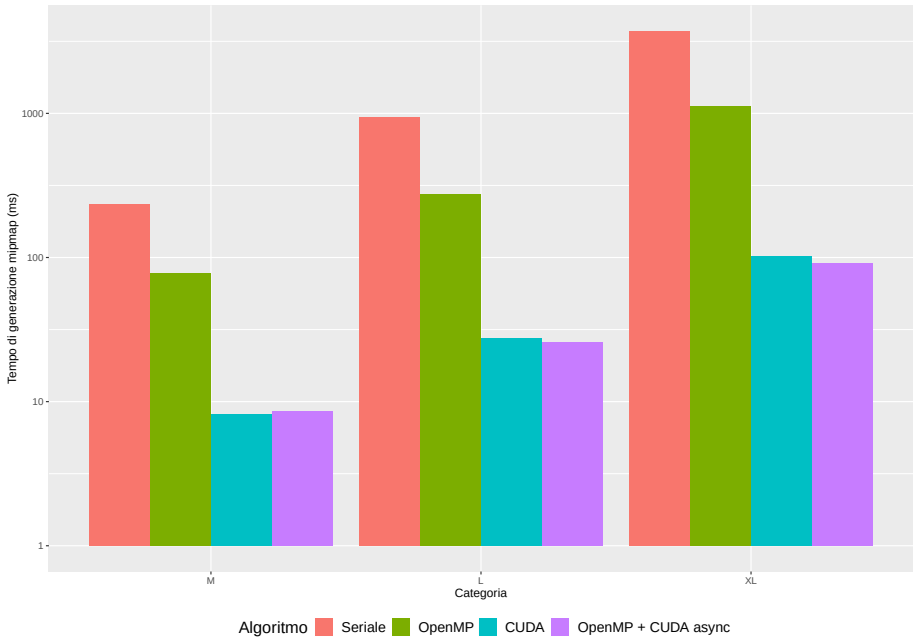

Figure 4: Tempi di generazione delle mipmap

Analizzando i tempi di esecuzione possiamo vedere come nella fase di elaborazione lo speed-up ottenuto attraverso l'introduzione di CUDA sia sostanziale (la scala della figura 4 è logaritmica).
Vediamo anche che tra CUDA e OpenMP+CUDA nei tempi di elaborazione non c'è una differenza sostanziale dato che la registrazione parallela dei comandi non è molto più veloce di quella seriale a meno di immagini molto molto grandi. Anche se la versione OpenMP+CUDA permette di sovrapporre trasferimenti ed le elaborazioni questi devono comunque essere effettuati.
Il contributo di OpenMP si rivela invece notevole nella fase di scrittura delle immagini su disco. Generalmente parallelizzare le scritture non dà alcun vantaggio poiché la banda dell'hard-disk è limitata e suddividerla su diverse scritture contemporaneamente non cambia il tempo totale di scrittura. Inoltre parallelizzando le scritture il disco deve saltare da un settore all'altro degradando ulteriormente le performance. Il miglioramento nei tempi di scrittura evidenziato dal grafico in figura 5 è dovuto sopratutto a un operazione di conversione dell'immagine (che può trarre vantaggio dalla parallelizzazione offerta da OpenMP) necessaria per la successiva scrittura su disco.

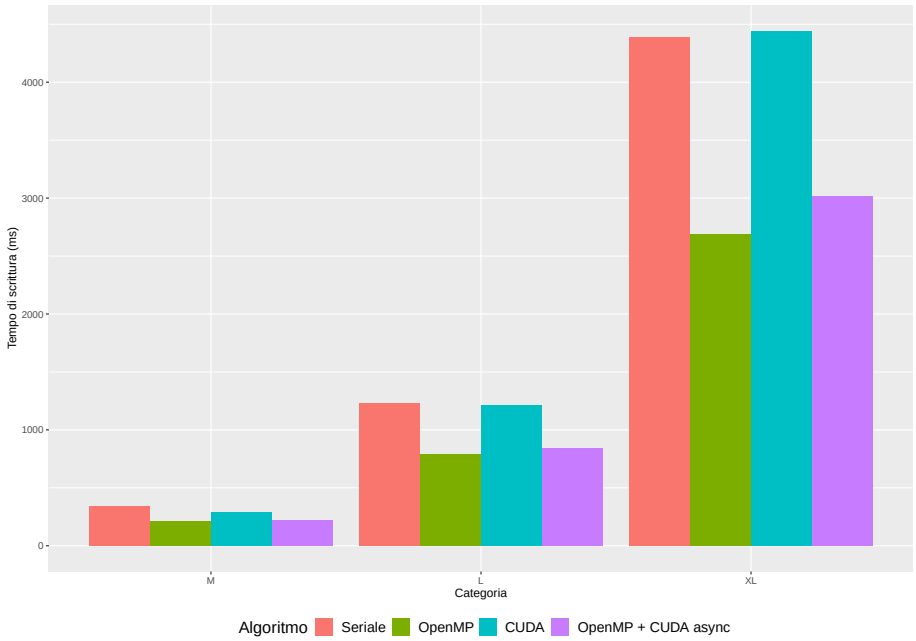

Figure 5: Tempi di scrittura delle immagini

5 Valutazioni e osservazioni

In conclusione lo speed-up più sostanziale è stato ottenuto attraverso l'utilizzo di CUDA per la parallelizzazione dell'algoritmo seriale.
Nel caso della versione OpenMP + CUDA la maggior parte del tempo guadagnato rispetto alla versione CUDA è dovuto all'uso degli streams. La piccola diminuzione nei tempi di esecuzione tra CUDA e OpenMP + CUDA non è quindi da attribuirsi a OpenMP, bensì all'utilizzo degli stream che ci permettono di sovrapporre trasferimenti ed elaborazioni. Tuttavia vediamo un miglioramento sostanziale nei tempi di scrittura delle immagini dovuto a OpenMP.

6 Guida alla compilazione e alla raccolta dei dati

Per compilare le diverse versioni dell'algoritmo è disponibile il file build.bat su windows, mentre su linux i comandi sono:

```
nvcc serial.cu -o build\serial
nvcc cudasync.cu -o build\cudasync
nvcc openmp.cu -o build\openmp -Xcompiler "-openmp"
nvcc openmpcuda.cu -o build\openmpcuda -Xcompiler "-openmp"
```

Per generare i dati in formato CSV nella sotto cartella Misure i comandi linux sono (la versione windows è nel file collect.bat):

```
build\serial Dataset\ Output\ Bricks_8K.jpg > Misure\serial_XL.csv
build\serial Dataset\ Output\ Bricks_4K.jpg > Misure\serial_L.csv
build\serial Dataset\ Output\ Bricks_2K.jpg > Misure\serial_M.csv

build\cudasync Dataset\ Output\ Bricks_8K.jpg > Misure\cudasync_XL.csv
build\cudasync Dataset\ Output\ Bricks_4K.jpg > Misure\cudasync_L.csv
build\cudasync Dataset\ Output\ Bricks_2K.jpg > Misure\cudasync_M.csv

build\openmp Dataset\ Output\ Bricks_8K.jpg > Misure\openmp_XL.csv
build\openmp Dataset\ Output\ Bricks_4K.jpg > Misure\openmp_L.csv
build\openmp Dataset\ Output\ Bricks_2K.jpg > Misure\openmp_M.csv

build\openmpcuda Dataset\ Output\ Bricks_8K.jpg > Misure\openmpcuda_XL.csv
build\openmpcuda Dataset\ Output\ Bricks_4K.jpg > Misure\openmpcuda_L.csv
build\openmpcuda Dataset\ Output\ Bricks_2K.jpg > Misure\openmpcuda_M.csv
```

La sintassi del comando è:

```
build\serial
    [cartella dov'è l'immagine]
    [cartella dove verranno scritte le mipmap]
    [nome dell'immagine]
```