

GPU COMPUTING

# Seam Carving GPU accelerated

Amadori Tommaso, n° 963792

Papparotto Luca, n° 960853

# Seam carving

L'algoritmo SeamCarving è un algoritmo di ridimensionamento delle immagini content-aware: è in grado di cambiarne dimensione, ma andandone a preservare gli elementi importanti. È un algoritmo che non scala in dimensione l'immagine, ma va a rimuovere al suo interno i pixel meno importanti.

Esso si basa "sull'energia" dei pixel, ovvero quanta importanza ha un pixel nell'immagine, ed è utilizzata per calcolarne il path da rimuovere (chiamati seams).

Questo procedimento viene iterato in base al ridimensionamento desiderato (se si vuole ridurre la dimensione dell'immagine di 300 pixel, allora l'algoritmo verrà ripetuto per 300 volte).

Se si ridimensionasse un'immagine tramite crop o stretch, il risultato non permetterebbe di ottenere la stessa immagine ridimensionata senza alterare eccessivamente il contenuto di essa : proporzioni e/o soggetti importanti verrebbero alterati.

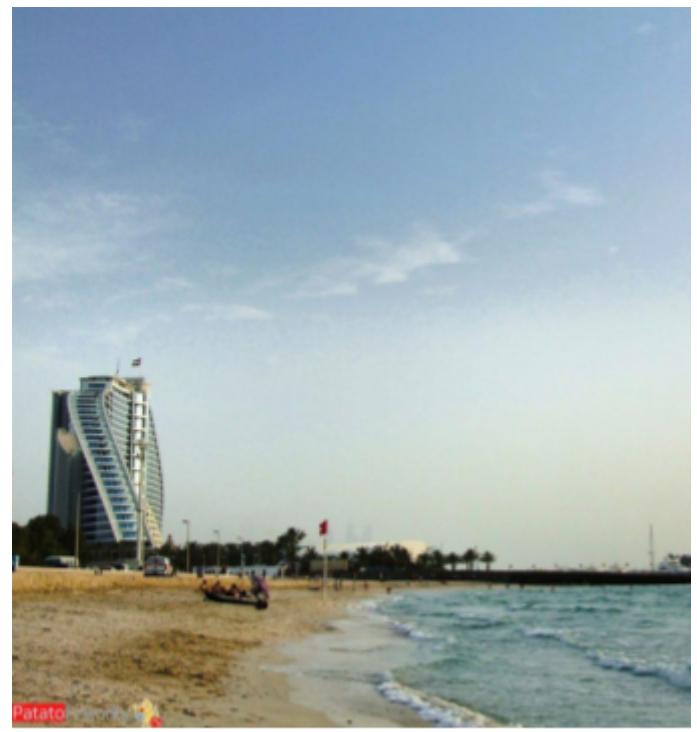
Qui di seguito sono mostrate le differenze tra un crop, uno stretch e un ridimensionamento con seam carving, tutte con una riduzione di 350 pixel(da 1140 a 855).



Original



Stretched



Cropped



Seam carved

Ovviamente queste riduzioni sono estreme, ma con le adeguate accortezze è impossibile capire se una foto sia originale o meno, come il caso che segue:



dove l'immagine è stata ridotta di ben 800 pixel.

## L'algoritmo

L'algoritmo si divide in 5 fasi principali:

- 1) **Lettura immagine e pre-elaborazione:** in questa fase viene letta l'immagine da file, viene allocata tutta la memoria necessaria all'algoritmo e, l'immagine, viene convertita in scala di grigi.
- 2) **Calcolo energia dei pixel:** ad ogni pixel viene assegnato un valore corrispondente alla sua importanza all'interno dell'immagine.
- 3) **Computazione dei seams:** calcolo di tutti i seam dell'immagine. Un seam è un cammino da un margine all'altro dell'immagine scelto seguendo il minimo tra le energie dei pixel.
- 4) **Computazione del minimum seam:** seam con peso minimo.
- 5) **Aggiornamento dell'immagine:** rimozione dei pixel che costituiscono il seam minimo.

## Lettura immagine e pre-elaborazione

L'algoritmo prende in input un'immagine BMP da ridimensionare e il numero di pixel orizzontali da rimuovere:

```
./SeamCarving.exe "/src/assets/images/big.bmp" 300
```

L'immagine di input ha una risoluzione di 7680 x 4320 pixel.

L'immagine di output "reduced.bmp" avrà una risoluzione finale di 7380 x 4320.

Come si deduce dalla risoluzione finale, l'algoritmo non fa un semplice resizing (che manterebbe il rapporto d'aspetto originario), bensì rimuove dei pixel in maniera "intelligente" andando a modificare il contenuto dell'immagine.

Sì è scelto di utilizzare il formato BMP poichè è quello più semplice da trattare e rielaborare.

In questo passo dell'algoritmo si ha una fase di allocazione di memoria. Si è deciso di utilizzare le primitive **cudaMallocManaged()** e di conseguenza la memoria unificata poichè è il metodo più nuovo che CUDA mette a disposizione.

Dopo la lettura dell'immagine da file, si ha la conversione della stessa in scala di grigi. Si è dedicato, quindi, un kernel per tale conversione: ad ogni pixel dell'immagine viene dedicato un thread che provverà a fare una semplice somma e divisione  $(r+g+b)/3$ . L'immagine, inoltre, viene salvata come un array lineare, si hanno, quindi, accessi contigui in ogni warp e di conseguenza ottimo utilizzo della memoria.

Il numero di thread per block verrà discusso in seguito nella sezione "Performance e speedUp"

L'output di questa fase sarà l'immagine seguente:



## Calcolo energia dei pixel

L'approccio alla base di questa fase è pressoché identico a quella precedente: ad ogni thread viene assegnato un pixel per il quale calcolerà la sua energia.

Con "energia" (*energy*) si intende l'importanza che ha quel pixel rispetto ai suoi vicini..

$$energy = \sqrt{dx^2 + dy^2}$$

dove

$$dx = \text{pixelSuccessivoRiga} - \text{pixelPrecedenteRiga}$$

$$dy = \text{pixelSuperiore} - \text{pixelInferiore}$$

Come si può notare, è fondamentale conoscere la posizione del pixel nell'immagine, ovvero se angolo, se bordo o se al suo interno. Di conseguenza sarà necessario, oltre ad ottenere la posizione, fare calcoli differenti per ogni caso.

Nonostante ciò la divergenza è minima poichè, utilizzando un array lineare, solo in 4 casi un thread di un warp farà operazioni differenti rispetto agli altri, ma ciò è inevitabile.

Ovviamente questo ragionamento va applicato ad immagini più grandi di 32 x 32, ma in questo caso non avrebbe senso utilizzare una GPU.

Il risultato di questa fase è il seguente:



Come si può osservare, l'immagine presenta contorni/bordi più chiari in corrispondenza di soggetti/contorni di figure, a sottolineare la loro importanza nel contesto in cui sono inserite. Tali pixel, quindi, difficilmente verranno eliminati (come giusto che sia).

## Computazione dei seams

Questa è certamente la fase cruciale e più importante dell'algoritmo e che ha avuto diverse implementazioni e test durante tutto lo svolgimento del progetto.

Un seam è un path di pixel che inizia dal bordo inferiore dell'immagine e arriva a quello superiore attraversandola nella sua interezza con spostamenti solo verticali o diagonali, ma mai orizzontali.

La costruzione del path, quindi, è una operazione sequenziale: preso un determinato pixel posso andare al successivo, ma devo conoscere anche il precedente.

Quello che è altamente parallelizzabile è la costruzione di questi path per ogni pixel di origine (tutti quelli del bordo inferiore): si hanno così  $N$  thread che in parallelo costruiscono tutti i seam, dove  $N$  è il numero di pixel di larghezza dell'immagine.

Tali seam devono essere, però, costruiti seguendo la logica del minimo locale; ci si muove andando sul pixel vicino (vedi sopra) con energia minore; durante la costruzione eseguo anche la somma delle energie calcolando, quindi, quella totale del seam.

E' dunque necessario la conoscenza, da parte di ogni pixel, di quale sia il prossimo minimo da raggiungere.

Una prima versione prevedeva che, il calcolo del minimo, avvenisse durante la costruzione del path.

Successivamente è stata adottata una strategia di pre-computazione che permettesse di ridurre i tempi di esecuzione totale, in particolare ci siamo valsi di un kernel che, per ogni pixel dell'immagine, calcoli il "pixel minimo vicino" e aggiorni l'apposita struct. Ovviamente tale operazione può essere svolta in parallelo, 1 thread = 1 pixel dell'immagine.

Questa pre-computazione ha fatto sì che il tempo totale di calcolo del minimo locale sommata al tempo di costruzione del seams, sia migliore di un kernel che faccia entrambe le cose assieme (1° versione), dovuto principalmente al miglior sfruttamento della GPU causato dal coinvolgimento di un maggior numero di thread.

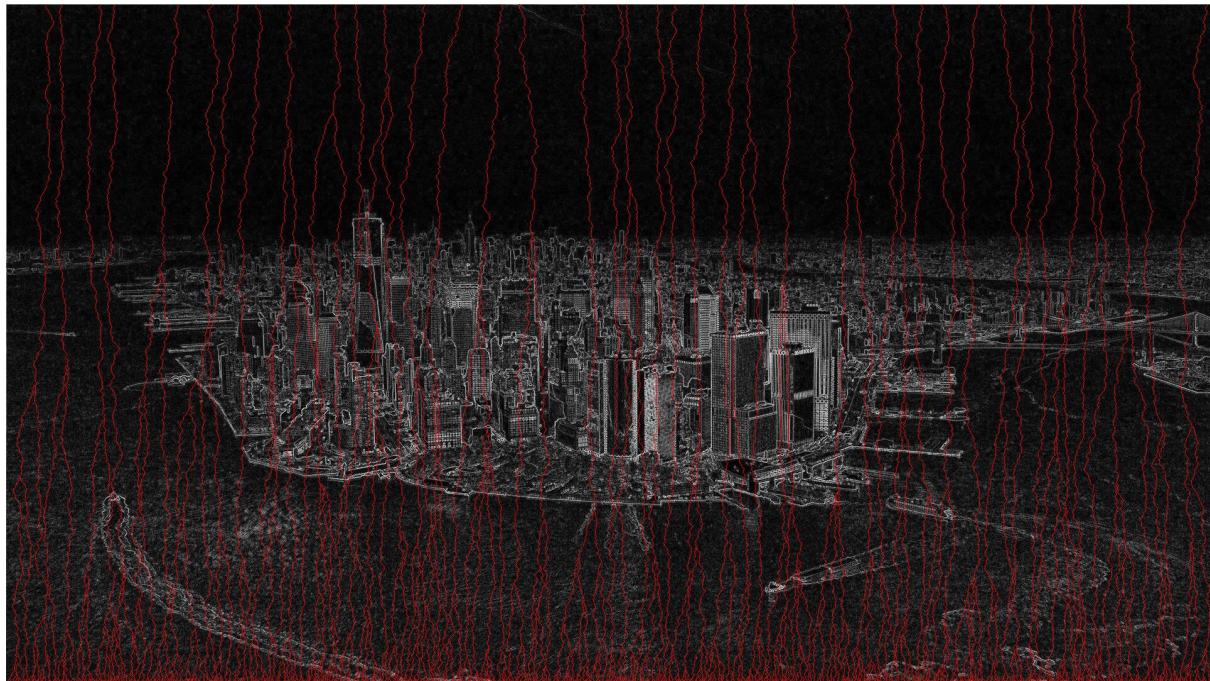
Da notare che è possibile che due seams condividano parte del path, ciò comporta che alcuni thread per determinati pixel (quelli in comune), ri-computino lo stesso percorso. Questo non è troppo un problema: essendo un'operazione parallela, è impossibile determinare quali dei due thread arrivi prima e di conseguenza quale non necessiti di ricalcolare il path rimanente. Nel caso comunque si riuscisse, il thread che non deve ricalcolare il percorso dovrà necessariamente rimanere in attesa degli altri (i vari step dell'algoritmo sono tutti concatenati l'uno con l'altro).

Sì è provato, inoltre, ad estremizzare la parallelizzazione in questa fase dell'algoritmo cercando, per prima cosa, di calcolare il path di un singolo seam in parallelo e successivamente eseguire le somme delle energie sfruttando la parallel reduction. Queste due ottimizzazioni non sono state attuate per due differenti motivi:

- 1) Trovare il seam è una operazione sequenziale ed è impossibile, dati due sotto-path distinti, capire se appartengono allo stesso seam o a due differenti;
- 2) Calcolare la somma usando la parallel reduction è estremamente inefficiente: utilizzando il parallelismo dinamico, si lanciano N thread (uno per ogni pixel di larghezza) che a loro volta eseguano un kernel con M thread, al fine di computare il minimo. Si ha, quindi, un'esplosione nel numero di kernel, dipendente dalla grandezza dell'immagine, e, con la possibilità di eseguirne solo 32 in parallelo, le performance vanno a peggiorare. Inoltre, considerando il fatto che il path viene

costruito in maniera sequenziale, sommare passo per passo, non risulta una operazione così dispendiosa.

L'immagine risultante da questa fase è la seguente:



Come si osserva, viene creato un percorso per ogni pixel di larghezza e, con un po' di accortezza, si può notare come i seams costeggino i bordi (verticali) dei vari elementi delle immagini.

## Computazione del seam minimo

Una volta trovati tutti i seams dell'immagine, si procede con la rimozione di quello con peso minimo, ovvero del seam che nell'immagine contiene i pixel meno importanti.

Alla fine della fase precedente, nel concreto, si ha un array di seam con le relative energie.

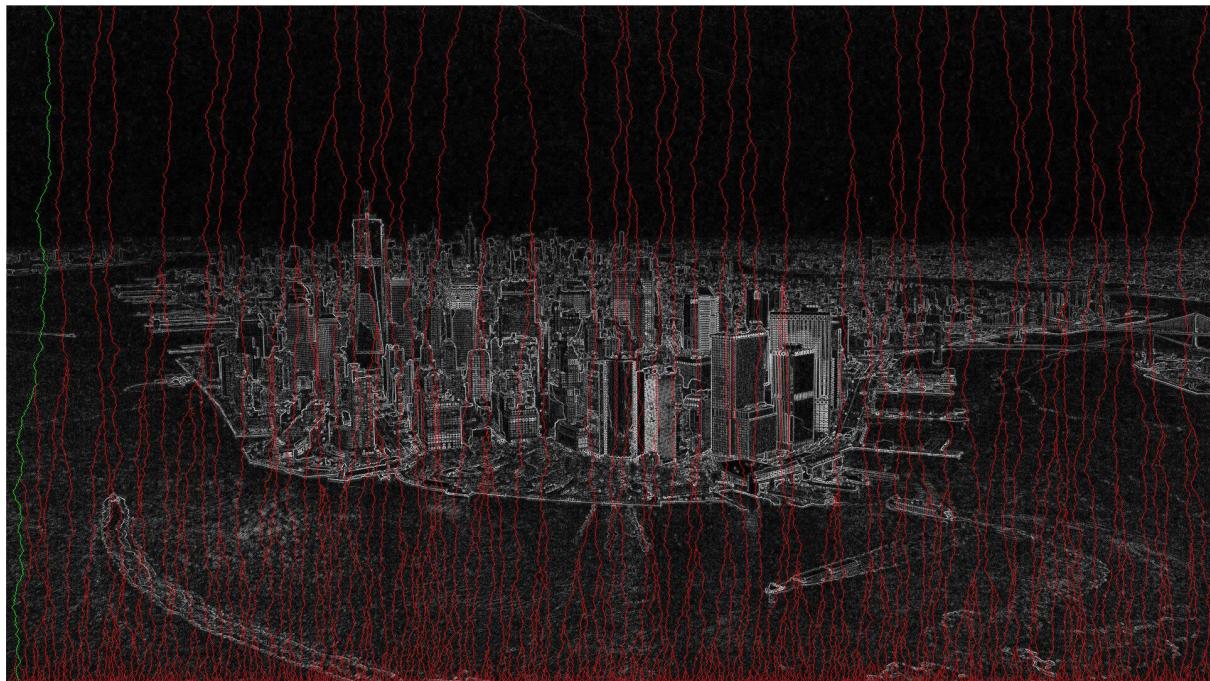
Questo rende possibile sfruttare l'algoritmo di parallel reduction applicato, invece che alla somma, all'operazione di comparazione, permettendoci di ottenere l'indice del seam minimo.

Con questa idea si è realizzato un kernel che, sfruttando la shared memory e l'algoritmo di parallel reduction, vada a comparare le energie dei vari seam e di conseguenza il minimo.

Si lanciano, quindi, tanti threads quanti sono i seams generati al passo precedente (ovvero pari alla larghezza dell'immagine).

Ovviamente, come risultato, si avranno tanti minimi quanti i blocchi di thread. Per trovare il reale minimo, si sfrutta la comparazione in CPU: il numero di blocchi è generalmente esiguo e di conseguenza inutile parallelizzare anche questo calcolo (a meno che si parli di immagini prossime al giga, ma a quel punto il problema si sposta sulla quantità di memoria disponibile).

L'output di questa fase è:



Si nota che il seam evidenziato è nella porzione di immagine più “scura” ovvero dove ogni pixel ha poca importanza rispetto ai vicini.

## Rimozione del seam minimo

In questa fase, viene creata un'immagine di supporto con dimensione pari al numero di pixel dell'immagine che ha subito tutti i passaggi precedenti, ma sottratta di un numero di pixel pari a quelli della sua altezza.

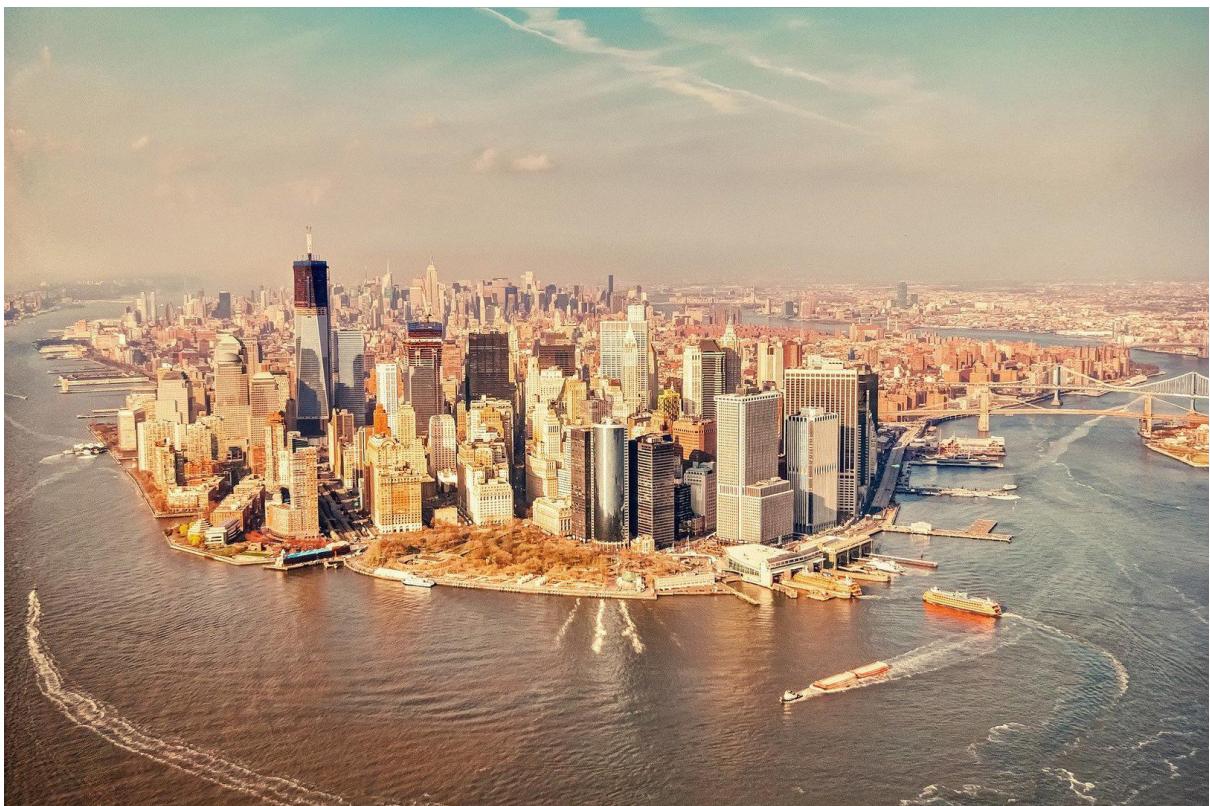
Viene utilizzato, quindi, un kernel di  $N$  thread (pari al numero di pixel dell'immagine finale) dove ogni thread verifica se si trovi a sinistra o a destra (nella riga) rispetto al pixel da rimuovere. Nel primo caso copia il valore del pixel nella stessa posizione, altrimenti dovrà effettuare uno shift a sinistra di 1.

Questo procedimento sfrutta il fatto che il seam è stato costruito sequenzialmente, di conseguenza `seam[0]` conterrà l'indice del pixel della riga 0, `seam[1]` quello della riga 1 ecc..

Da sottolineare che tale rimozione di pixel viene eseguita solamente per l'immagine in scala di grigi e, quindi, aggiornata ad ogni iterazione. Ciò è richiesto poichè il calcolo dell'energia viene fatto mediante il coinvolgimento dei pixel vicini che, se sono stati modificati a causa della rimozione, sono differenti rispetto a quelli dell'iterazione precedente.

L'aggiornamento dell'immagine a colori, invece, viene fatta una singola volta. Ogni pixel dell'immagine in GrayScale, memorizza la sua posizione originaria: questo ci permette di eseguire un kernel che copi dall'immagine originale a colori solo i pixel rimanenti dall'ultima iterazione, parallelizzando, di fatto, anche questa fase.

Il processo si conclude con la scrittura su file di questa nuova immagine ridotta di 300px:



## Performance e speedUp

Riportiamo in questa sezione tutti i vari accorgimenti adottati in fase di sviluppo utili a migliorare le performance e lo speedUp della GPU.

I test sono stati eseguiti su una macchina dotata di CPU i7-7700k e di GTX 1080.

Si riportano in seguito i tempi di esecuzione:

	fox.bmp (10 iterazioni)	castle.bmp (10 iterazioni)	big.bmp (10 iterazioni)
CPU	372 ms	2027 ms	58557 ms
GTX 1080	26 ms	57 ms	742 ms
GTX 1050ti	59 ms	174 ms	3343 ms

Con speedUp di:

	fox.bmp (10 iterazioni)	castle.bmp (10 iterazioni)	big.bmp (10 iterazioni)
GTX 1080	1430%	3556%	7891%
GTX 1050ti	630%	1164%	1751%

Come si può notare, lo speedUp ottenuto è notevole. Osservando i risultati si vede come il divario di performance aumenti sempre di più all'aumentare della grandezza delle immagini, segno che l'implementazione vada a sfruttare in maniera intelligente la parallelizzazione e che sia in grado di scalare in base all'immagine di input.

Sì osserva anche un divario importante tra le performance delle due GPU, la 1080 possiede il triplo dei cuda core e SM rispetto alla 1050Ti. Tali differenze mostrano come l'algoritmo, all'aumentare delle risorse disponibili, sia in grado di sfruttarle al massimo.

Per quanto riguarda l'ottimizzazione e lo sfruttamento della GPU per kernel, si è utilizzato nvprof e nsight monitor.

Le metriche prese in considerazione sono state le seguenti:

- *achieved\_occupancy*
- *branch\_divergence*
- *warp\_execution\_efficiency*

Da sottolineare che tali valori possono variare anche di molto (soprattutto l'occupancy) in base all'immagine sottoposta a riduzione. Riportiamo ora i valori finali ottenuti durante il processo di ottimizzazione (per tutti i dati raccolti vedere il repository, directory reports/profiling):

```

==5700== NVPROF is profiling process 5700, command: ./SeamCarving.exe C:/aa/fox.bmp 2
#bytes: 768002
Input BMP dimension: (640 x 400)
IHeader[2] 768056

operation took 352 milliseconds
FINAL HEIGHT 400
FINAL WIDTH 638
Imagin C:\aa\reduced.bmp generate
==5700== Profiling application: ./SeamCarving.exe C:/aa/fox.bmp 2
==5700== Profiling result:
==5700== Metric result:
Invocations                               Metric Name           Metric Description      Min      Max      Avg
Device "NVIDIA GeForce GTX 1080 (0)"
  Kernel: removeSeam_(EnergyPixelStruct*, int*, ImgPropStruct*, EnergyPixelStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.874577  0.874673  0.874625
    2          branch_efficiency          Branch Efficiency   98.42%  98.91%  98.67%
    2          warp_execution_efficiency Warp Execution Efficiency 99.75%  99.82%  99.79%
  Kernel: computeSeams2_(EnergyPixelStruct*, PixelStruct*, seamStruct*, ImgPropStruct*, bool)
    2          achieved_occupancy           Achieved Occupancy   0.831238  0.831240  0.831239
    2          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    2          warp_execution_efficiency Warp Execution Efficiency 99.84%  100.00% 99.92%
  Kernel: energyMap_(EnergyPixelStruct*, ImgPropStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.888135  0.888140  0.8874769
    2          branch_efficiency          Branch Efficiency   98.85%  98.86%  98.85%
    2          warp_execution_efficiency Warp Execution Efficiency 96.54%  96.62%  96.58%
  Kernel: updateImageGray_(EnergyPixelStruct*, EnergyPixelStruct*, ImgPropStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.844029  0.844815  0.846222
    2          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    2          warp_execution_efficiency Warp Execution Efficiency 100.00% 100.00% 100.00%
  Kernel: removePixelsFromSrc_(PixelStruct*, PixelStruct*, EnergyPixelStruct*, ImgPropStruct*)
    1          achieved_occupancy           Achieved Occupancy   0.888980  0.888980  0.888980
    1          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    1          warp_execution_efficiency Warp Execution Efficiency 100.00% 100.00% 100.00%
  Kernel: toGrayScale_(PixelStruct*, EnergyPixelStruct*, int)
    1          achieved_occupancy           Achieved Occupancy   0.911564  0.911564  0.911564
    1          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    1          warp_execution_efficiency Warp Execution Efficiency 100.00% 100.00% 100.00%
  Kernel: min_(seamStruct const *, seamStruct*, ImgPropStruct*, int)
    2          achieved_occupancy           Achieved Occupancy   0.030381  0.030382  0.030381
    2          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    2          warp_execution_efficiency Warp Execution Efficiency 79.34%  81.19%  80.27%
  Kernel: computeMinsPerPixel_(EnergyPixelStruct*, ImgPropStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.868203  0.875066  0.871634
    2          branch_efficiency          Branch Efficiency   95.33%  95.33%  95.33%
    2          warp_execution_efficiency Warp Execution Efficiency 95.63%  95.64%  95.64%

```

**fox.bmp**

```

==3976== NVPROF is profiling process 3976, command: ./SeamCarving.exe C:/aa/castle.bmp.bmp 2
#bytes: 4146914
Input BMP dimension: (1428 x 968)
IHeader[2] 4146968

Operation took 355 milliseconds
FINAL HEIGHT 968
FINAL WIDTH 1426
Imagin C:\aa\reduced.bmp generate
==3976== Profiling application: ./SeamCarving.exe C:/aa/castle.bmp.bmp 2
==3976== Profiling result:
==3976== Metric result:
Invocations                               Metric Name           Metric Description      Min      Max      Avg
Device "NVIDIA GeForce GTX 1080 (0)"
  Kernel: removeSeam_(EnergyPixelStruct*, int*, ImgPropStruct*, EnergyPixelStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.890343  0.890978  0.890660
    2          branch_efficiency          Branch Efficiency   99.32%  99.34%  99.33%
    2          warp_execution_efficiency Warp Execution Efficiency 99.89%  99.89%  99.89%
  Kernel: computeSeams2_(EnergyPixelStruct*, PixelStruct*, seamStruct*, ImgPropStruct*, bool)
    2          achieved_occupancy           Achieved Occupancy   0.035175  0.035180  0.035178
    2          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    2          warp_execution_efficiency Warp Execution Efficiency 99.10%  99.17%  99.13%
  Kernel: energyMap_(EnergyPixelStruct*, ImgPropStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.901388  0.903460  0.902424
    2          branch_efficiency          Branch Efficiency   98.89%  98.89%  98.89%
    2          warp_execution_efficiency Warp Execution Efficiency 97.43%  97.44%  97.43%
  Kernel: updateImageGray_(EnergyPixelStruct*, EnergyPixelStruct*, ImgPropStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.858456  0.862478  0.860467
    2          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    2          warp_execution_efficiency Warp Execution Efficiency 100.00% 100.00% 100.00%
  Kernel: removePixelsFromSrc_(PixelStruct*, PixelStruct*, EnergyPixelStruct*, ImgPropStruct*)
    1          achieved_occupancy           Achieved Occupancy   0.913642  0.913642  0.913642
    1          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    1          warp_execution_efficiency Warp Execution Efficiency 100.00% 100.00% 100.00%
  Kernel: toGrayScale_(PixelStruct*, EnergyPixelStruct*, int)
    1          achieved_occupancy           Achieved Occupancy   0.932122  0.932122  0.932122
    1          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    1          warp_execution_efficiency Warp Execution Efficiency 100.00% 100.00% 100.00%
  Kernel: min_(seamStruct const *, seamStruct*, ImgPropStruct*, int)
    2          achieved_occupancy           Achieved Occupancy   0.030362  0.030421  0.030392
    2          branch_efficiency          Branch Efficiency   100.00% 100.00% 100.00%
    2          warp_execution_efficiency Warp Execution Efficiency 79.90%  79.97%  79.94%
  Kernel: computeMinsPerPixel_(EnergyPixelStruct*, ImgPropStruct*)
    2          achieved_occupancy           Achieved Occupancy   0.885205  0.887655  0.886445
    2          branch_efficiency          Branch Efficiency   95.40%  95.40%  95.40%
    2          warp_execution_efficiency Warp Execution Efficiency 97.17%  97.17%  97.17%

```

**castle.bmp**

Invocations	Metric Name	Metric Description	Min	Max	Avg
Device "NVIDIA GeForce GTX 1080 (0)"					
Kernel: removeSeam_(EnergyPixelStruct*, int*, ImgPropStruct*, EnergyPixelStruct*)	achieved_occupancy	Achieved Occupancy	0.896526	0.897646	0.897086
2	branch_efficiency	Branch Efficiency	99.87%	99.91%	99.89%
2	warp_execution_efficiency	Warp Execution Efficiency	99.98%	99.99%	99.98%
Kernel: computeSeams2_(EnergyPixelStruct*, PixelStruct*, seamStruct*, ImgPropStruct*, bool)	achieved_occupancy	Achieved Occupancy	0.187332	0.187352	0.187342
2	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%
2	warp_execution_efficiency	Warp Execution Efficiency	99.99%	100.00%	99.99%
Kernel: energyMap_(EnergyPixelStruct*, ImgPropStruct*)	achieved_occupancy	Achieved Occupancy	0.917366	0.917687	0.917487
2	branch_efficiency	Branch Efficiency	99.17%	99.17%	99.17%
2	warp_execution_efficiency	Warp Execution Efficiency	98.59%	98.59%	98.59%
Kernel: updateImageGray_(EnergyPixelStruct*, EnergyPixelStruct*, ImgPropStruct*)	achieved_occupancy	Achieved Occupancy	0.864598	0.864926	0.864762
2	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%
2	warp_execution_efficiency	Warp Execution Efficiency	100.00%	100.00%	100.00%
Kernel: removePixelsFromSrc_(PixelStruct*, PixelStruct*, EnergyPixelStruct*, ImgPropStruct*)	achieved_occupancy	Achieved Occupancy	0.912506	0.912506	0.912506
1	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%
1	warp_execution_efficiency	Warp Execution Efficiency	100.00%	100.00%	100.00%
Kernel: toGrayScale_(PixelStruct*, EnergyPixelStruct*, int)	achieved_occupancy	Achieved Occupancy	0.814780	0.814780	0.814780
1	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%
1	warp_execution_efficiency	Warp Execution Efficiency	100.00%	100.00%	100.00%
Kernel: min_(seamStruct const *, seamStruct*, ImgPropStruct*, int)	achieved_occupancy	Achieved Occupancy	0.087682	0.089352	0.088517
2	branch_efficiency	Branch Efficiency	100.00%	100.00%	100.00%
2	warp_execution_efficiency	Warp Execution Efficiency	88.57%	88.63%	88.60%
Kernel: computeMinsPerPixel_(EnergyPixelStruct*, ImgPropStruct*)	achieved_occupancy	Achieved Occupancy	0.878984	0.900325	0.889655
2	branch_efficiency	Branch Efficiency	95.48%	95.48%	95.48%
2	warp_execution_efficiency	Warp Execution Efficiency	98.20%	98.20%	98.20%

big.bmp

Interessante è notare come la branch divergence è ridotta al minimo, riducendo quindi i tempi di attesa tra thread. Questo è dovuto dal fatto che thread dello stesso warp eseguono generalmente le stesse cose, nonostante, nei vari kernel, siano presenti differenti branch: due thread seguono generalmente branch differenti quando sono in posizione differenti dell'immagine, per esempio: th[0] = angolo in basso a sinistra, th[1] = bordo di destra.

L'occupancy, come si nota, è generalmente molto alta, ad eccezione dei kernel *min\_* e *computeSeam2\_*.

Questo è dovuto principalmente al numero di thread che vengono lanciati concorrentemente, in particolare *min\_* opera su una quantità esigua di dati(che ricordiamo essere pari alla larghezza dell'immagine), e di conseguenza non vengono schedulati molti thread. Caso analogo è quello di *computeSeam2\_* dove, essendo un'operazione sequenziale, non è possibile utilizzare più thread della larghezza dell'immagine.

Negli altri casi, è stato effettuato un lavoro empirico di ricerca del miglior compromesso tra tempo di esecuzione e occupancy: si sono variati i numeri di blocchi e thread per blocco al fine di minimizzare le metriche prima citate. Si rimanda al repository per visionare tutti i test effettuati in questo senso.

Oltre all'occupancy, è molto alta anche la thread utilization, segno che l'algoritmo è stato implementato in maniera intelligente.