

UPSCALING DI IMMAGINI CON KERNEL GAUSSIANO

OTTIMIZZAZIONI AVANZATE IN CUDA:
DALLA BASELINE AL THROUGHPUT VETTORIALE

SISTEMI DI ELABORAZIONE ACCELERATA
A.A. 2025/2026

Francesco Galli
Desirée Pellegrini

L'Upscaling delle immagini

L'obiettivo principale del progetto è l'implementazione di un sistema di **Upscaling 2x** di un'immagine digitale. In un processo di ingrandimento standard, il rischio principale è la perdita di definizione o la creazione di artefatti visivi.

Per superare i limiti della semplice interpolazione lineare, abbiamo adottato il **Resampling Gaussiano**. Questo metodo non si limita a duplicare i dati esistenti, ma ricostruisce ogni pixel dell'immagine di output calcolando una media pesata dei pixel circostanti nell'immagine originale. L'approccio permette di ottenere un risultato finale più naturale e nitido, rappresentando però una sfida computazionale superiore a causa della complessità del calcolo dei pesi per ogni singolo punto.

Il Cuore del Calcolo: La Distribuzione Gaussiana

Matematicamente, il valore di ogni pixel finale P è determinato dal rapporto tra la somma dei pixel vicini moltiplicati per i rispettivi pesi e la somma totale dei pesi stessi.

Il peso W_i segue la distribuzione normale (Gaussiana). La scelta di questo modello è dettata dalla necessità di dare massima importanza al pixel più vicino al centro del kernel, diminuendo gradualmente l'influenza dei vicini all'aumentare della distanza. Il parametro Sigma (σ) gioca un ruolo fondamentale: controlla la nitidezza del filtro, determinando quanto 'ampia' debba essere la campana di influenza del kernel.

$$P = \frac{\sum_{i \in Kernel} Pixel_i \times W_i}{\sum_{i \in Kernel} W_i}$$

$$W(dx, dy) = e^{-\frac{dx^2 + dy^2}{2\sigma^2}}$$

I Colli di Bottiglia dell'Accelerazione

Prima di procedere con l'implementazione, è stata condotta un'analisi teorica per identificare le criticità che limitano le prestazioni su GPU. Sono stati individuati tre fattori principali:

1

Memory Bound: per ogni pixel di output è necessario leggere un'intera finestra di pixel vicini, e migliaia di thread finirebbero per richiedere contemporaneamente gli stessi dati alla memoria globale, saturando la banda passante.

2

Compute Bound: l'utilizzo della funzione `exp()` all'interno del kernel per il calcolo dei pesi rappresenta un'operazione estremamente onerosa. Questo rischia di saturare le unità di calcolo prima ancora di aver sfruttato la memoria.

3

Latenza e Disallineamento: l'accesso a dati non allineati (come il formato RGB a 3 byte) impedisce alla GPU di raggruppare le richieste di memoria, causando sprechi di cicli di clock.

Hardware e Strumenti di Analisi

Il processo di ottimizzazione è stato validato su un'architettura **NVIDIA Turing**, utilizzando una scheda video **GeForce GTX 1660 Ti**.

Per guidare lo sviluppo in modo oggettivo, abbiamo adottato una metodologia basata sul profiling con **NVIDIA Nsight Compute**. Lo strumento principale di analisi è stato il [Roofline Model](#), che permette di visualizzare il rapporto tra l'intensità aritmetica del kernel e i limiti fisici della GPU, distinguendo tra problemi di calcolo e problemi di larghezza di banda.

Implementazione Naïve e Profiling

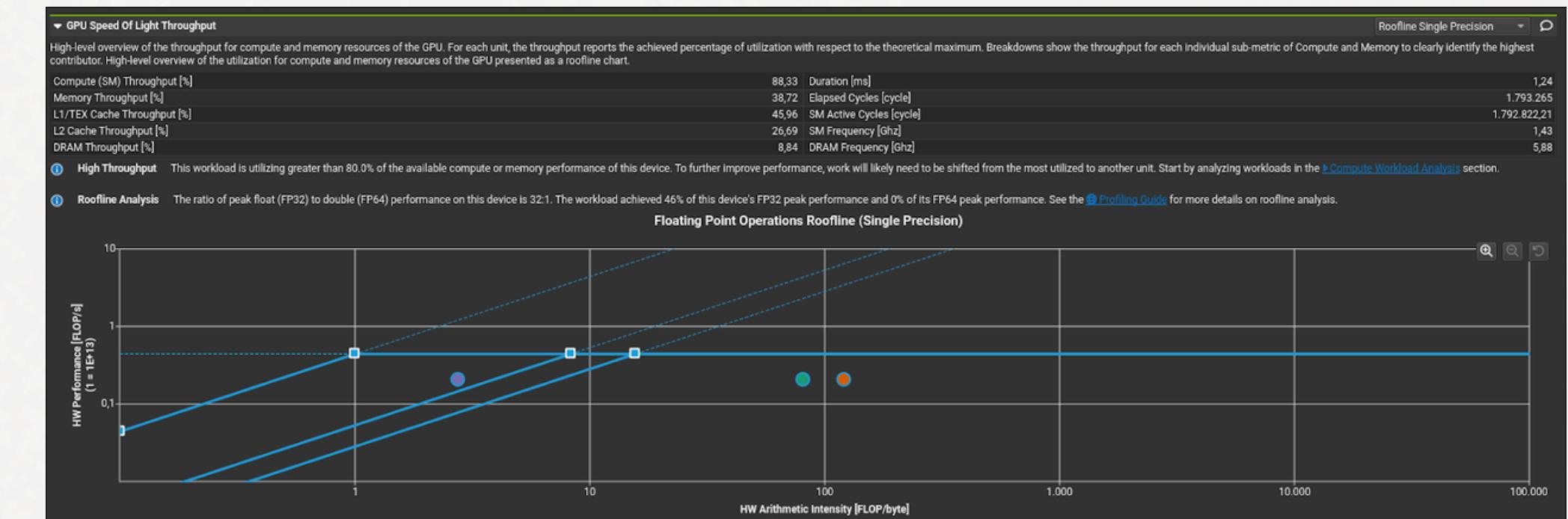
Ogni thread è responsabile di un pixel di output (griglia 16x16).

```
// Ogni thread calcola il peso individualmente
float weight = expf(-(dx*dx + dy*dy) / (2.0f*sigma*sigma));

// Accesso RGB non ottimizzato alla Global Memory
r_sum += d_input[idx + 0] * weight;
g_sum += d_input[idx + 1] * weight;
b_sum += d_input[idx + 2] * weight;
```

Il modello conferma che il kernel è Compute-Bound (88% del throughput FP32), saturato da calcoli matematici ridondanti.

- Tempo di esecuzione (FHD): **1.284 ms.**
- Elevata pressione sulle unità di calcolo (SFU) per la funzione `expf()`.
- Accessi a memoria non coalescenti (formato RGB a 3-byte).
- Basso riutilizzo dei dati (data reuse) tra thread vicini.



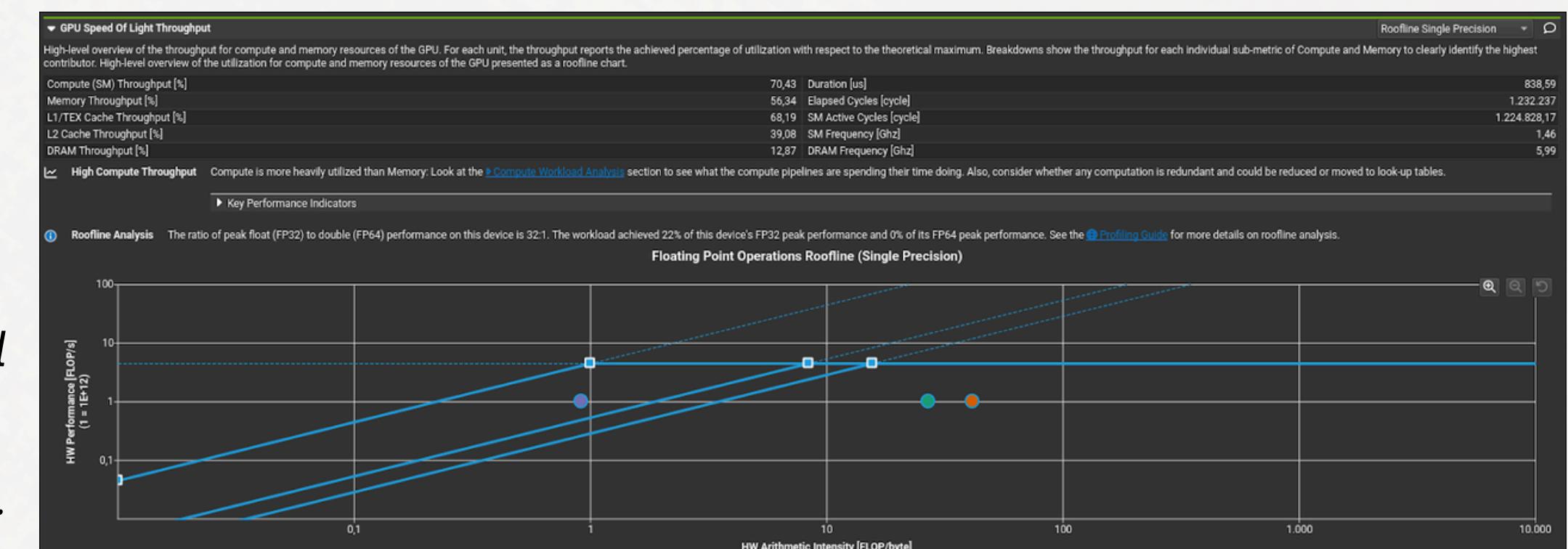
Abbattimento del carico computazionale: Constant Memory

- Sostituzione di `expf()` con un lookup table pre-calcolato dalla CPU.
- La Constant Memory è ottimizzata per il broadcast: tutti i thread di un warp leggono lo stesso peso simultaneamente.
- Il tempo scende a **0.842 ms** (Speedup: 1.52x).

I punti operativi si abbassano drasticamente: il kernel non è più limitato dal calcolo puro, ma inizia a spostarsi verso il limite della memoria.

```
// I pesi non vengono più calcolati, ma letti
float weight = c_weights[ky * radius + kx];

r_sum += d_input[idx] * weight;
```



Gestione Manuale della Cache: Implementazione del Tiling

```
// Caricamento cooperativo nella memoria condivisa
__shared__ unsigned char s_tile[TILE_H][TILE_W];

s_tile[ty][tx] = d_input[global_idx];
__syncthreads(); // Sincronizzazione necessaria

// Calcolo effettuato sui dati in Shared Memory (veloce)
r_sum += s_tile[local_y + ky][local_x + kx] * weight;
```

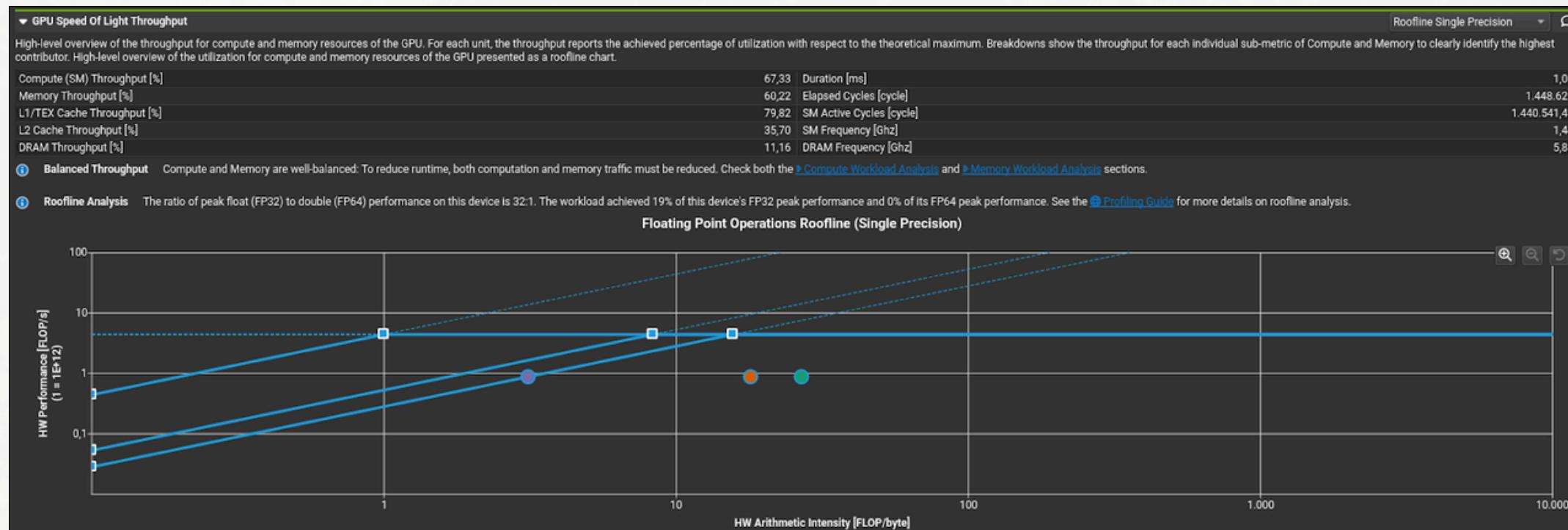
- Utilizzo di `_shared_` per minimizzare gli accessi alla Global Memory (DRAM).
- Obiettivo: ridurre la ridondanza delle letture caricando una “mattonella” (Tile) di pixel che viene riutilizzata da tutti i thread del blocco.
- Inserimento di un bordo di pixel extra per permettere la convoluzione Gaussiana anche ai bordi della Tile.
- Ci aspettiamo un abbattimento del traffico dati verso la memoria globale e accelerazione del kernel.

Risultati Inattesi e Analisi dei Colli di Bottiglia

L'architettura Turing dispone di una cache L1 estremamente efficiente che gestisce già automaticamente il riutilizzo dei dati. L'uso di `_syncthreads()` e la gestione complessa degli indici hanno introdotto una latenza superiore al risparmio ottenuto.

In conclusione, la gestione manuale della memoria ha “ostacolato” gli automatismi della GPU invece di aiutarli.

Versione	Tempo (ms)	Esito
Constant Memory	0.842	Migliore
Shared Memory	1.002	Rgresso (-19%)



Il grafico mostra uno spostamento verso un'intensità aritmetica maggiore, ma un'efficienza globale inferiore a causa degli stalli introdotti dalla sincronizzazione.

Verso il Picco di Performance: La Vectorization

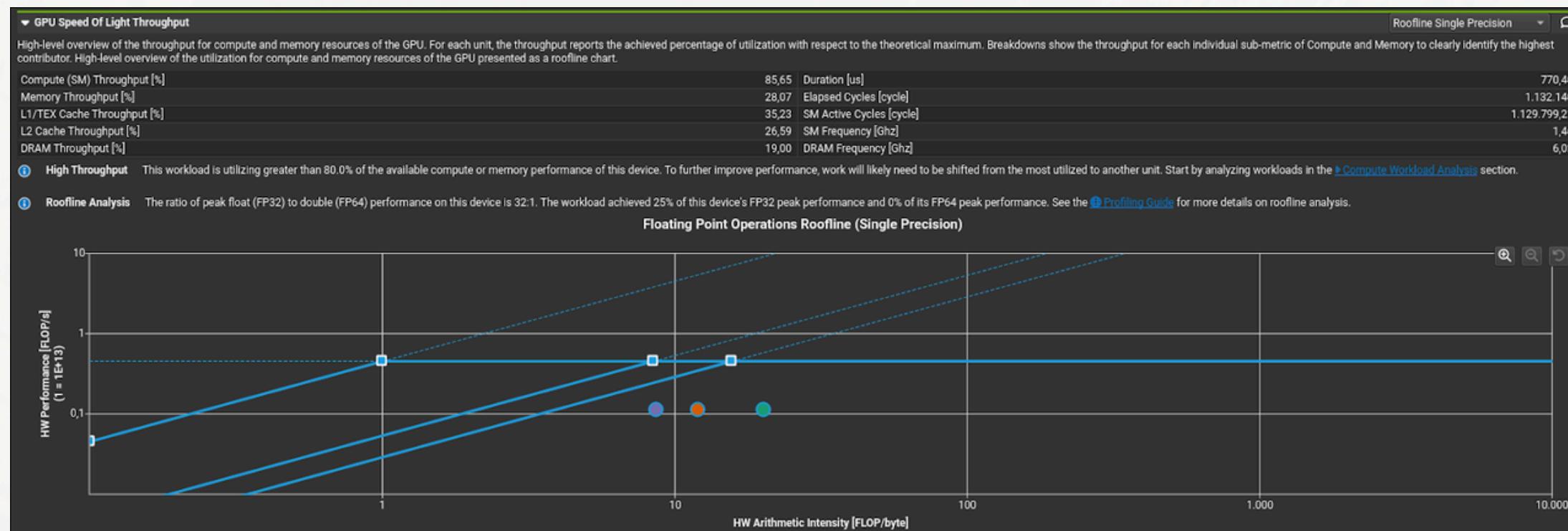
Utilizziamo uchar4 per garantire accessi alla memoria perfettamente coalescenti.

- Gli accessi a 3 byte (RGB) costringono la GPU a letture multiple e disallineate.
L'aggiunta di un canale “padding” (Alpha) allinea i dati a 4 byte.
- Ogni richiesta di memoria dei thread viene ora raggruppata in singole transazioni da 32 o 128 byte, saturando la banda passante.
- Raggiungimento del tempo minimo record: **0.777 ms.**

```
// Passaggio dal formato RGB (3 byte) al formato RGBA (4 byte)
// Accesso vettoriale a 32 bit (allineato)
uchar4 pixel = ((uchar4*)d_input)[idx];

r_sum += pixel.x * weight;
g_sum += pixel.y * weight;
b_sum += pixel.z * weight;
```

Validazione delle Performance: Il Massimo Throughput



Il modello mostra il raggiungimento del picco di throughput della memoria (DRAM), confermando l'efficacia dell'accesso a 32-bit (uchar4).

Passando da accessi a 3 byte (RGB) a 4 byte (RGBA), abbiamo eliminato le letture ridondanti della DRAM.

Il kernel non è più bloccato dalla latenza di calcolo, ma lavora alla velocità massima consentita dalla larghezza di banda della GTX 1660 Ti.

Incremento prestazionale del **65% (1.65x)** rispetto alla versione iniziale.

Analisi Comparativa delle Versioni

	Naïve	Constant	Shared	Vectorized
Memory [%]	38,72	56,34	60,22	28,07
Compute [%]	88,33	70,43	67,33	85,65
DRAM Thr. [GB/s]	24,94	36,99	31,53	54,88
L1 Hit Rate [%]	88,80	88,84	71,78	11,16
Sectors/Req	1,66	1,66	5,13	7,11
Achieved Occ. [%]	87,60	80,25	92,96	92,42
Regs/Thread	41	45	38	35
Stall L. Sc.	0,58	1,65	1,82	2,53
Duration [ms]	1,24	0,84	1,00	0,77
Speedup [x]	1.00	1.48	1.24	1.61

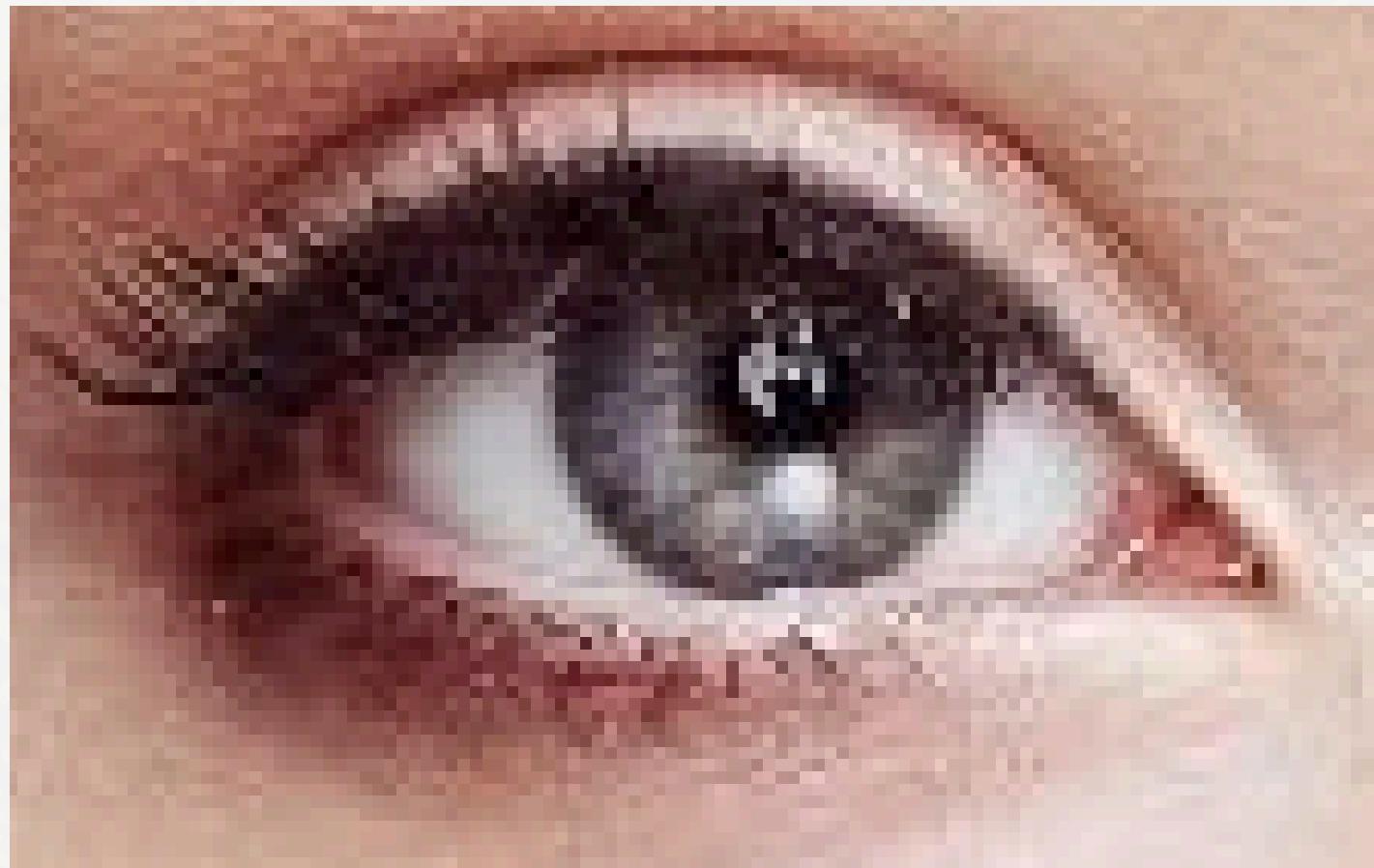
Analisi della Scalabilità su Diversi Dataset

Per verificare la robustezza delle ottimizzazioni, il kernel è stato testato su diverse risoluzioni, dai piccoli dataset (Bassa Risoluzione) fino a immagini 8K. I risultati confermano che lo speedup rimane consistente all'aumentare del carico di lavoro.

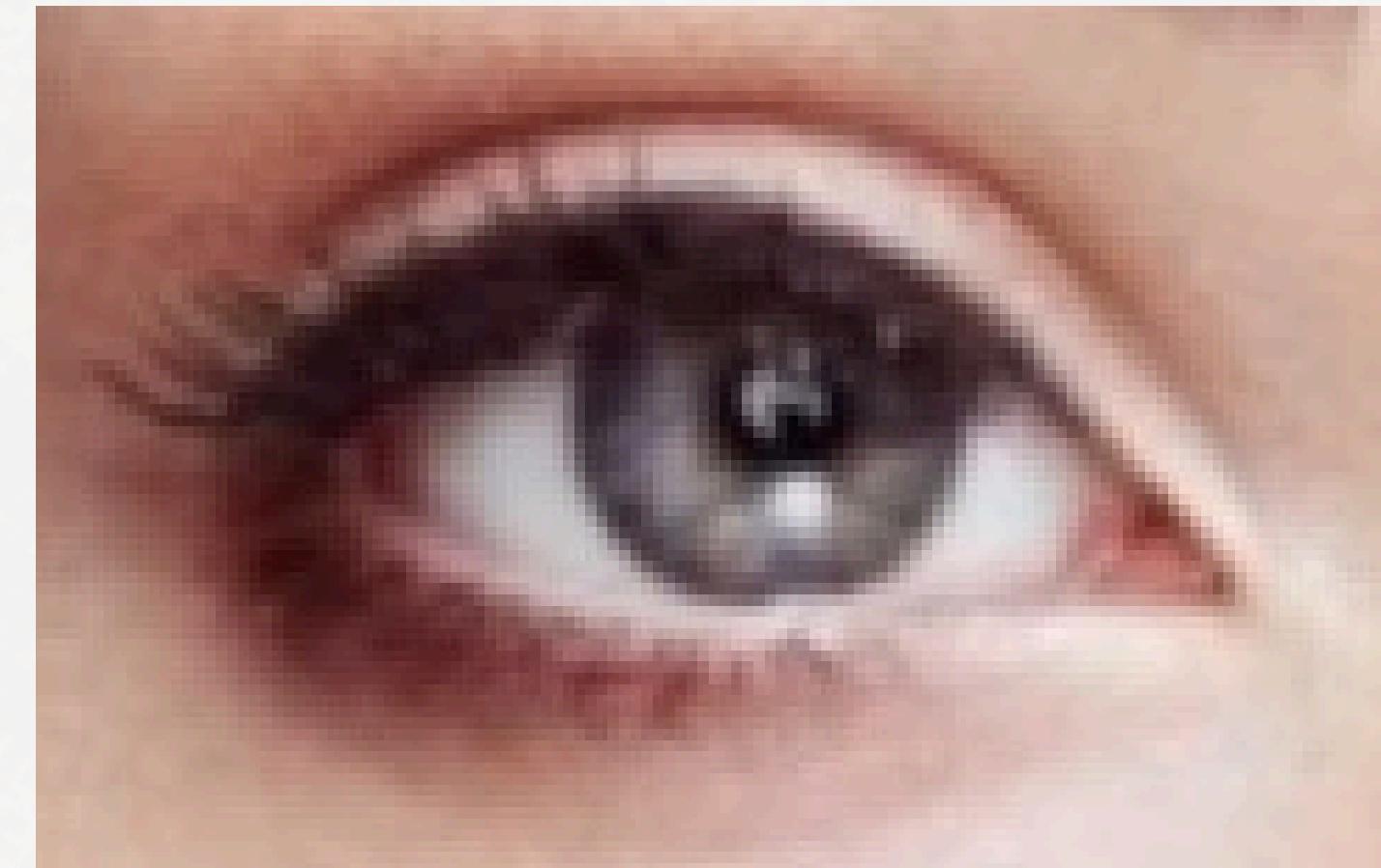
Dataset	Risoluzione	Pixel Totali (Mpx)	Tempo Naïve (ms)	Tempo Vectorized (ms)	Speedup (x)
Bassa	800 x 500	0.4	0.117	0.024	4.86
Viso	1000 x 700	0.7	0.456	0.263	1.73
FHD	1920 x 1080	2.1	1.284	0.777	1.65
4K	3992 x 2242	8.9	5.552	3.342	1.66
6K	5472 x 3648	20.0	12.015	7.534	1.59
8K	7680 x 4320	33.2	17.551	11.003	1.59

Risultati Visivi: Qualità della Ricostruzione (1000 x 700)

Prima

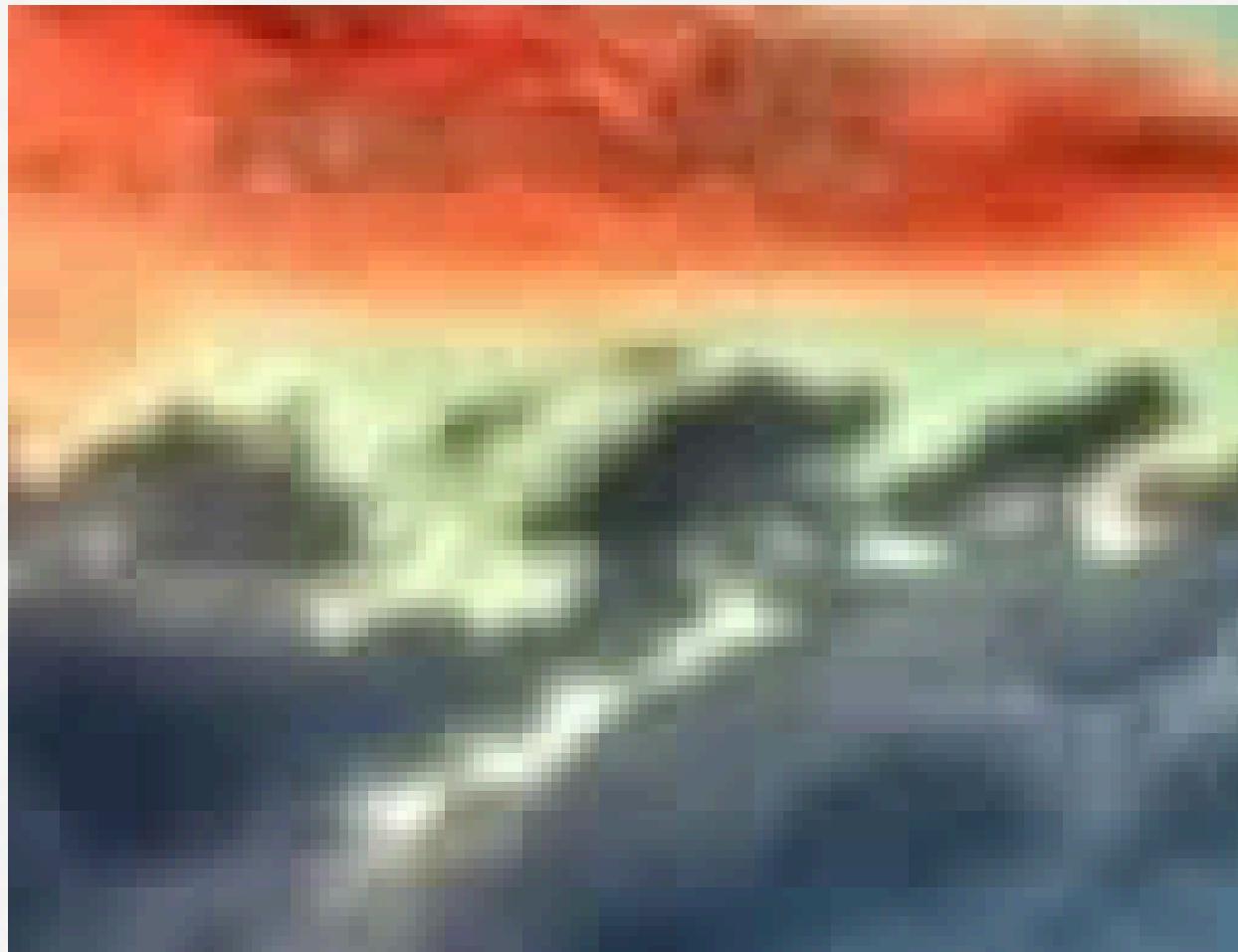


Dopo

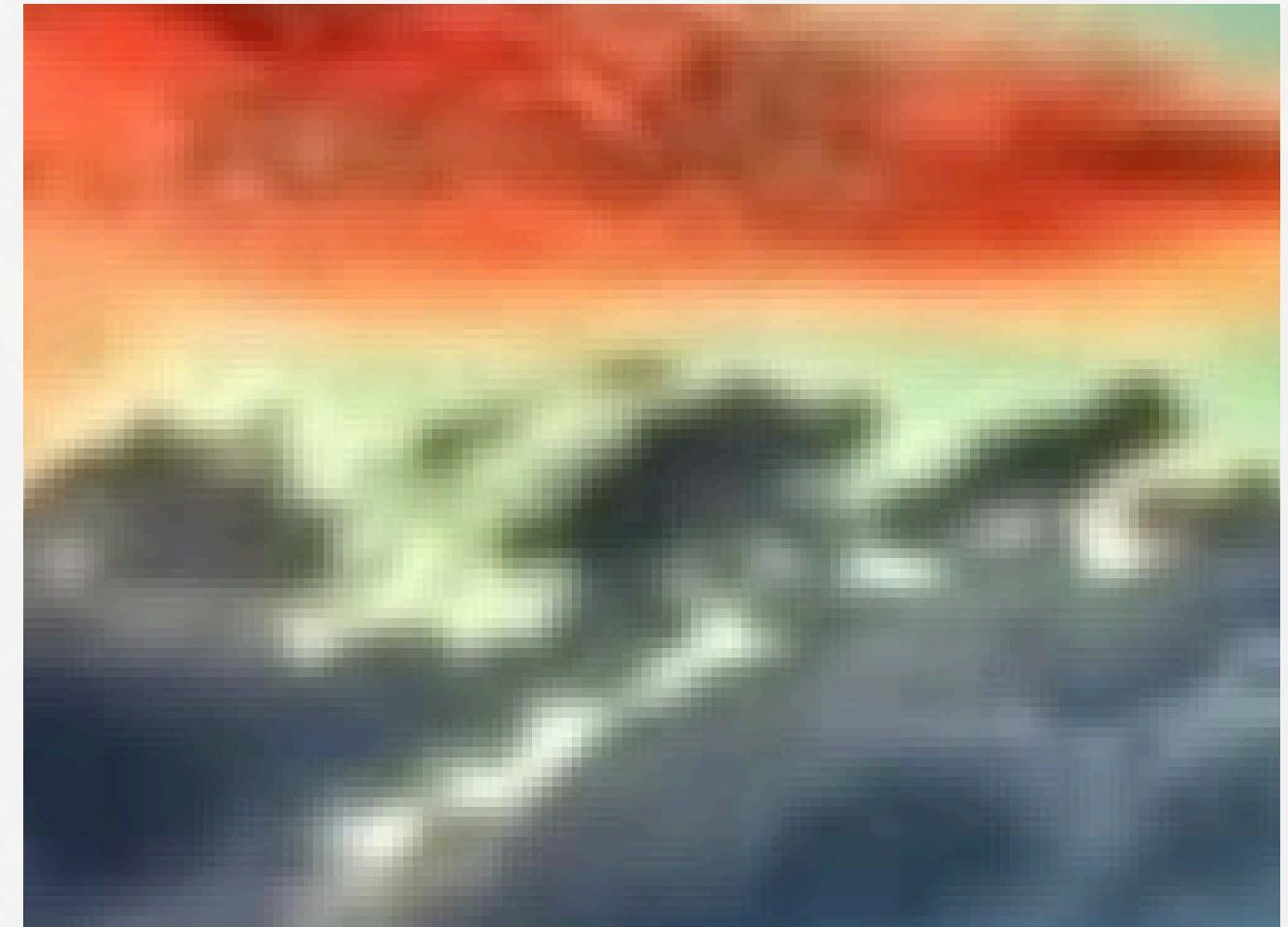


Risultati Visivi: Qualità della Ricostruzione (1920 x 1080)

Prima



Dopo



Risultati Visivi: Qualità della Ricostruzione (3992 x 2242)

Prima

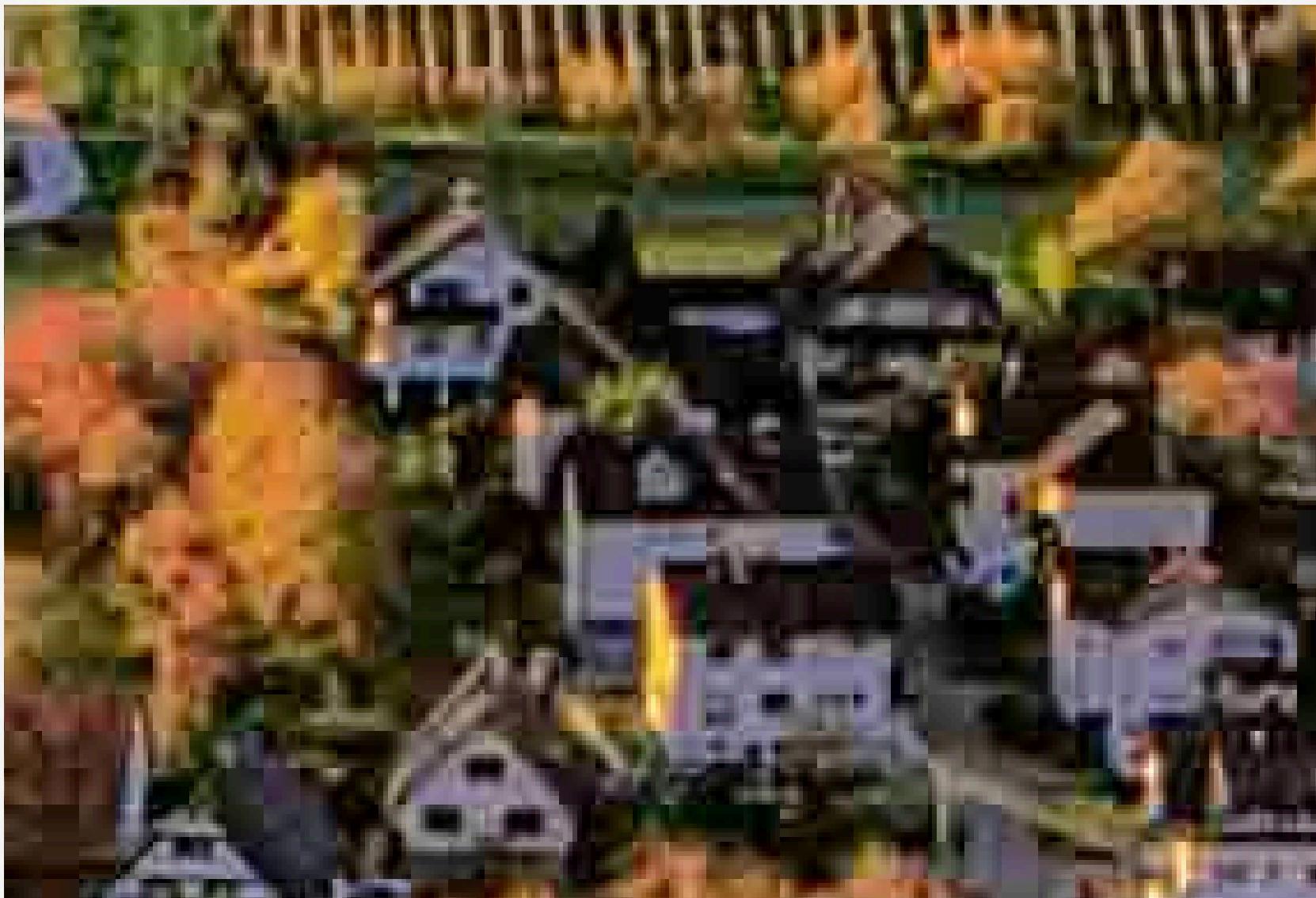


Dopo



Risultati Visivi: Qualità della Ricostruzione (5472 x 3648)

Prima



Dopo



Sviluppi Futuri

1

Potremmo scomporre il calcolo matematico in due passaggi più semplici, per ridurre i tempi di elaborazione, specialmente su immagini grandi, e sfruttare componenti hardware specifici della GPU (Texture Unit) per gestire i pixel in modo più efficiente rispetto alla memoria standard.

2

Potremmo implementare algoritmi più avanzati (come l'interpolazione Bicubica) per mantenere i bordi più nitidi e i testi più leggibili, e far scegliere al software l'algoritmo migliore in base al contenuto dell'immagine (es. uno per i paesaggi, uno per le scritte).

3

Si potrebbe passare da un approccio matematico a uno basato sull'Intelligenza Artificiale e utilizzare i nuclei specializzati delle moderne schede video per “ricostruire” i dettagli mancanti ottenendo risultati fotorealistici.

Conclusioni



Evoluzione dei Colli di Bottiglia

Siamo passati da un kernel Compute-Bound (bloccato dal calcolo ridondante della funzione `exp`) a un regime ottimizzato che satura il throughput della memoria.

La Vectorization

Su architettura Turing, l'allineamento dei dati si è rivelato più efficace della gestione manuale della Shared Memory. Il kernel vettorizzato sfrutta meglio la cache L1, evitando l'overhead di sincronizzazione del Tiling.

Sintesi Finale

L'algoritmo è altamente scalabile: il profiling ha trasformato un processo Compute-Bound in uno ottimizzato che satura la banda della GPU, riducendo i tempi quasi del 40%.