Alma Mater Studiorum · Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

PROFILAZIONE DI SOFTWARE MEDICALE IN CUDA

Elaborato in:
High Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Marco Sangiorgi

Correlatore:
Dott. Claudio Landi,
Dott. Leonardo Sassi

Sessione III
Anno Accademico 2023-2024

A mia nonna, Gianna.

Indice

Sommario

Le GPU sono al centro dell'innovazione tecnologica caratterizzando il cuore dei supercomputer più potenti al mondo in ambito di HPC (High Performance Computing). Il loro utilizzo caratterizza le applicazioni informatiche che hanno maggiore impatto nella vita quotidiana di ognuno di noi, spaziando dall'Intelligenza Artificiale ad algoritmi di visualizzazione a monitor.

Le immagini mediche derivanti dai dispositivi a raggi X sono spesso elaborate dalle GPU, sia per motivazioni di tempo che di risorse. Il difficile compito di identificare eventuali malattie, salvando potenzialmente una vita, necessita elaborazioni near-real-time e di alta qualità, solitamente caratterizzate da algoritmi dall'elevato livello di complessità computazionale [6].
La progettazione e la modellazione di tali algoritmi per l'utilizzo delle GPU, non è un processo banale. Richiede competenze sulla struttura e sul funzionamento interno delle GPU, nonché solide conoscenze di programmazione parallela, talvolta anche matematiche al fine di comprendere il linguaggio utilizzato nei report scientifici.

In questa tesi, si affronta l'utilizzo della GPU nella sua declinazione in ambito medico, grazie alla gentile collaborazione dell'azienda See Through s.r.l [16] specializzata, tra le altre cose, nello sviluppo di software per dispositivi medici a raggi X.
Nello specifico, si affronta la profilazione e l'ottimizzazione di un algoritmo di rimozione del rumore (in inglese, *Denoising*).

Capitolo 1

Introduzione

1.1 Applicazione

L'applicazione analizzata è gentile concessione dell'azienda See Through s.r.l [16] ed è utilizzata in ambito medico per svolgere operazioni di rimozione del rumore dalle immagini di dispositivi medici a raggi X. Questi ultimi, come ogni altro strumento di misurazione, introducono rumore nelle loro osservazioni che può avere natura aleatoria o seguire distribuzioni di frequenza dipendenti dal meccanismo di funzionamento dei dispositivi stessi. Ne consegue che le immagini risultati appaiono meno chiare recando potenziali impedimenti all'identificazione di eventuali malattie.

Il funzionamento dell'applicazione si basa sull'utilizzo di una tecnica di confronto tra gruppi di pixel dell'immagine, chiamati *Patch*, per trovare quelle simili tra loro ed usarle nell'algoritmo di denoising [17]. A tal fine, l'applicazione sfrutta il parallelismo massivo delle GPU NVIDIA mediante il linguaggio di programmazione CUDA [10].

1.2 Architettura GPU

L'architettura delle GPU NVIDIA sfrutta il paradigma SIMT (Single Instruction Multiple Thread) [4], una combinazione tra il paradigma SIMD [1] e multithreading,

che mira a massimizzare il throughput delle operazioni, quindi la quantità di dati processati nell'unità di tempo come funzione della grandezza dell'input.

Dal punto di vista computazionale, la GPU è strutturata nel seguente modo:

- I threads, sono l'unità minima di lavoro in grado di eseguire tutti una stessa funzione, chiamata *Kernel* nel linguaggio di programmazione CUDA, permettendo tecniche di maschermento delle latenze delle operazioni di fetch [2] dalla memoria. Il loro numero dipende dalla configurazione di lancio del Kernel.

- I warps, sono l'unità minima di schedulazione contenenti 32 threads eseguiti secondo il paradigma SIMD con i propri registri e program counter.

- Gli Streaming Multiprocessor (SM) (vedi Figura 1.1), sono un'unità di lavoro con più unità di lavoro interne, chiamate Streaming Processor, ed uno scheduler incaricato di impartire le istruzioni di uno o più warp. A livello logico, si traduco in blocchi di threads.

Dal punto di vista della memoria, la GPU è strutturata nel seguente modo:

- La memoria Cache L1 (vedi Figura 1.1), risiede all'interno degli Streaming Multiprocessor permettendo alta velocità di comunicazione con le unità di lavoro in esecuzione al costo di avere piccole dimensioni. Nelle moderne GPU, la cache L1 condivide lo stesso chip con la memoria *Shared*, utilizzata per comunicazioni inter-thread a livello di SM.

- La memoria Cache L2 (vedi Figura 1.1), utilizzata da ogni SM.

- La memoria DRAM (vedi Figura 1.1), contenente la memoria Globale utilizzata dalle applicazioni per copiare i dati dall'host (memoria RAM dell'elaboratore) al device (memoria DRAM della GPU).

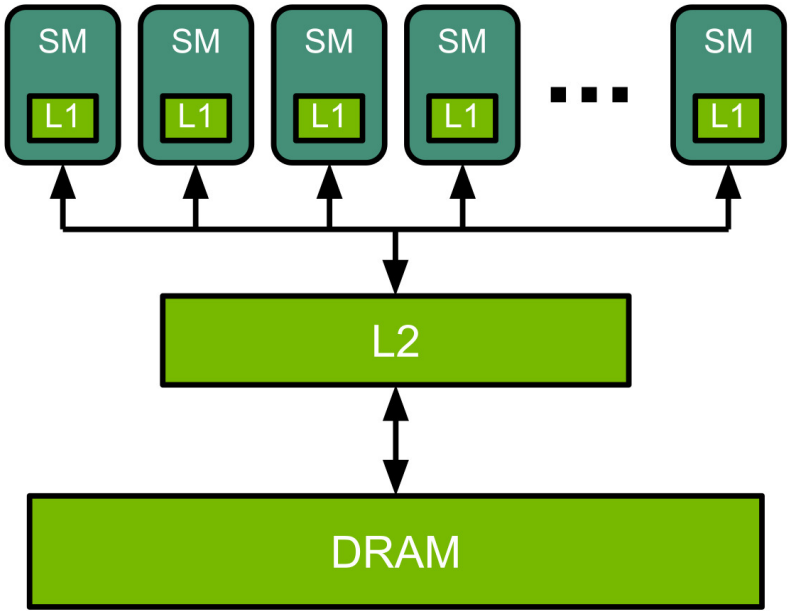


Figura 1.1: Struttura semplificata delle GPU NVIDIA. [11]

.

1.2.1 Scheduler e Warp

Gli scheduler di ogni SM possono impartire un'istruzione alla volta in diversi warp sulla base degli stati degli stessi [15]. Per questo, i warp si suddivino in:

- Warp Teorici, limite superiore dei warp schedulabili vincolato dalla configurazione di lancio del kernel.
- Warp Attivi, insieme dei warp allocati ad ogni ciclo.
- Warp Idonei, sottoinsieme dei warp attivi che non sono in stallo, quindi pronti per emettere le prossime istruzioni.
- Warp Emessi, sottoinsieme dei warp idonei che hanno impartito una o più istruzioni nel ciclo di scheduling.

1.2.2 Pratiche migliori

Le migliori pratiche da utilizzare per mantenere alti livelli di throughput computazionali e sfruttare a pieno l'architettura della GPU sono:

- Effettuare accessi contigui in memoria tra threads dello stesso warp, sfruttando a pieno il throughput della memoria.
- Utilizzare le memorie L1 e *Shared*, al fine di ridurre al minimo la latenza degli accessi in memoria.
- Evitare ramificazioni del codice, minimizzando i flussi del codice così da sfruttare a pieno il mascheramento di latenza del fetch di istruzioni. Al fine di evitare rami condizionali (come IF-ELSE), il compilatore utilizza tecniche di predicazione quando possibile.

1.3 Hardware Utilizzato

Nella tabella 1.1 sono mostrate le caratteristiche dell'hardware utilizzato, denominato WinRTX3060, e gentilmente messo a disposizione dall'azienda See Through s.r.l [16].

	WinRTX3060
Descrizione	Windows Server
CPU	Intel i5-12400 2.50 GHz
N. di Core	6 core + HyperT
RAM	16GB
S.O.	Windows 10
GPU	NVIDIA RTX 3060
Compilatore Host	MSVC 19.29.30151
Compilatore CUDA	NVCC V11.8.89

Tabella 1.1: Caratteristiche della macchina utilizzate

1.4 Obiettivo

L'obiettivo della tesi è rilevarne i punti critici dell'applicazione di denoising su GPU, tramite strumenti di profilazione CUDA.

Il fine è permettere l'attuazione di interventi mirati sulla base dei risultati delle analisi per l'ottimizzazione del codice CUDA fornito.

Capitolo 2

Profilazione

La profilazione è una forma di analisi del codice in grado di alleviare il compito del programmatore nell'incarico di ottimizzare un'applicazione di terzi. Il suo utilizzo non richiede informazioni sulla struttura del codice eseguito, ma solide conoscenze delle risorse messe a disposizione in un elaboratore (come CPU, GPU e memorie). Queste ultime, una volta accuratamente monitorate, sono il punto di partenza dell'ottimizzazione svolta (vedi Capitolo 3) e il punto di arrivo delle risultati conseguiti (vedi Capitolo 4).

2.1 Profilazione generica

Un primo livello di analisi si pone l'obiettivo di rilevare i principali punti critici dell'intera applicazione mediante Nsight System [14], uno strumento di profilazione NVIDIA che utilizza un profilatore statistico [5] in grado di tracciare le attività dell'esecuzione di codice CUDA campionando accuratamente gli stati delle risorse interne delle GPUs NVIDIA.

In Figura 2.1 è mostrata un grafico dei principali risultati ottenuti nel tempo profilando l'applicazione di denoising, e si nota che:

- Il numero di istruzioni impartite da ogni Streaming Multiprocessor è largamente inferiore (sotto il 50%) alla massima capacità computazionale degli stessi (vedi

voce "*SM Througput*").

- Il throughput della memoria L1 è superiore a quelli di L2 e VRAM (vedi voce "*L1 Throughput*", "*L2 Throughput*" e "*VRAM Throughput*"), indicando un maggiore utilizzo della memoria L1.

- Il numero di Warps Idonei è largamento inferiore alla massima capacità di ciascun scheduler (vedi voce "*Warps Eligible*").

- Il numero di warp in stato di stallo per dipendenza dall'esecuzione di istruzioni precedenti in media coinvolge gli interi Streaming Multiprocessors (vedi voce "*FE Stalls*").

- La percentuale di Warps Attivi per Streaming Multiprocessor è mediamente sotto il 50% (vedi "*SM Occupancy*") indicando un basso utilizzo delle risorse computazionali a disposizione.

- Le Cache Hit in memoria L1 e L2 (vedi voce "*L1 Hit Rate*" e "*L2 Hit Rate*") mostrano un corretto utilizzo di accessi contigui o di variabili in memoria condivisa (*Shared Memory*).

- Le percentuali di scritture/letture effettuate rispetto alle massime capacità teoriche delle memorie risultano nel complesso basse (vedi voce "*L1 Throughput*", "*L2 Throughput*", "*VRAM Throughput*", "*VRAM Throughput*"). A fronte di questo e dei punti precedenti, si può dedurre la presenza di operazioni principalmente limitate dagli accessi in memoria [8].

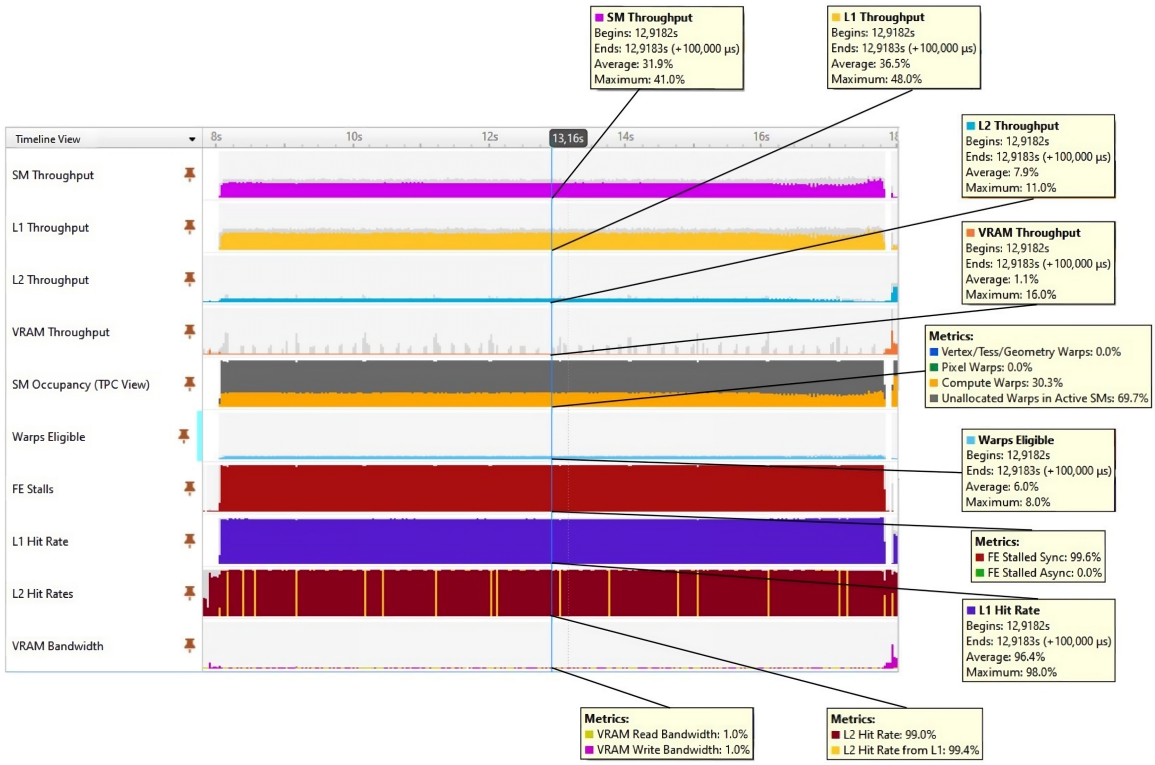


Figura 2.1: Profilazione NSight System dell'applicazione.

2.2 Profilazione mirata

La profilazione mirata si focalizza su un unico punto critico dell'applicazione mediante NVIDIA Nsight Compute [12], un profiler interattivo di kernel capace di fornire metriche dettagliate e strumenti di debug delle API CUDA utilizzate.

Analizzeremo il principale protagonista del punto critico, il kernel *K1*.
Per motivi di stabilità, l'input utilizzato dall'applicazione in questa fase della profilazione ha dimensioni minori rispetto all'originale.

2.2.1 Speed Of Light Throughput

NVIDIA SOL (Speed Of Light) Throughput [13] riporta il throughput della parte di calcolo e di memorizzazione come percentuale di utilizzo raggiunta rispetto al massimo teorico, chiamato *Speed Of Light*.

Dalla Figura 2.2, si nota che:

- Il kernel *K1* esibisce basse percentuali (sotto il 60%) per il throughput computazionale e per l'utilizzo della banda della memoria rispetto alle prestazioni di picco della GPU utilizzata (vedi campo "*Compute(SM) Throughput*" e "*Memory Throughput*").

- Il throughput della memoria L1 é largamente superiore, mantenendo in media basse percentuali (52.15%, vedi campo "*L1/TEX Cache Throughput*"), rispetto al throuput della memoria L2 e Globale (vedi campo "*L2 Cache Throughput*" e "*DRAM Throughput*"). Tali dati sono riconducibili ad un maggiore utilizzo della memoria L1, come constatato nella Sezione 2.1.

Tali risultati sono indagati approfonditamente nelle Sezioni 2.2.4 e 2.2.5, in quanto rappresentano tipici di latenza.

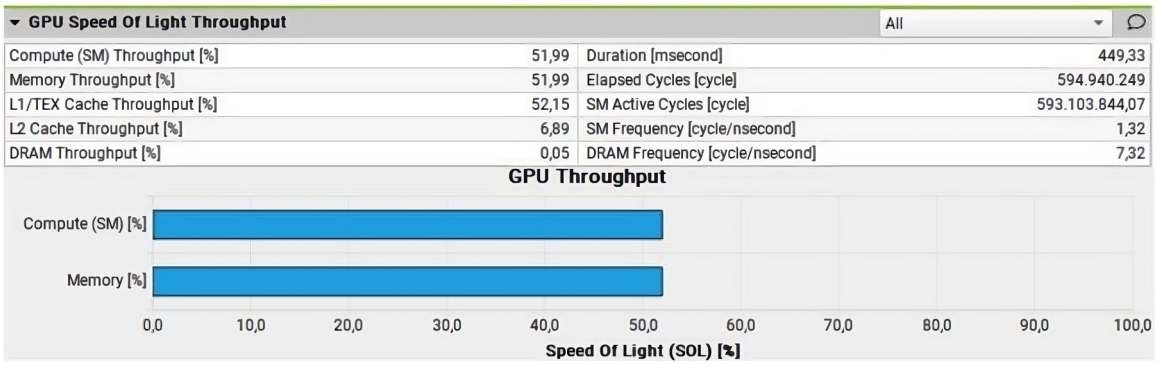


Figura 2.2: Metriche di Throughput della GPU.

2.2.2 Occupazione degli Streaming Multiprocessor

L'occupazione degli Streaming Multiprocessor [13] é il rapporto tra il numero di warp attivi rispetto al numero totale di warp disponibili per multiprocessore. È definita anche come la misura della capacità dell'hardware di processare i warp che sono attivamente in uso, in percentuale.

Una maggiore occupazione non sempre si traduce in prestazioni più elevate. Tuttavia, una bassa occupazione riduce sempre l'abilità di nascondere le latenze comportando una complessiva degradazione delle prestazioni.

Dai risultati in Figura 2.3, si nota che:

- Il kernel *K1* utilizza una configurazione di thread in grado di occupare quasi compleatamente lo spazio di ogni Streaming Multiprocessors in termini di warp (95.8% di occupazione, vedi campo "*Theoretical Occupancy*").

- La differenza tra l'occupazione teorica calcolata (95.8%, vedi campo "*Theoretical Occupancy*") e l'occupazione effettivamente raggiunta (29.97%, vedi campo "*Achieved Occupancy*") indica un possibile caso di overhead nella schedulazione dei warps oppure di carichi di lavoro non correttamente bilanciati tra warp dello stesso blocco o di blocchi diversi durante l'esecuzione del kernel.

Occupancy

Theoretical Occupancy [%]	95,83	Block Limit Registers [block]	2
Theoretical Active Warps per SM [warp]	46	Block Limit Shared Mem [block]	5
Achieved Occupancy [%]	29,97	Block Limit Warps [block]	2
Achieved Active Warps Per SM [warp]	14,38	Block Limit SM [block]	16
Max Cluster Size	n/a	Max Active Clusters	n/a
Cluster Occupancy [%]	n/a	Overall GPU Occupancy [%]	n/a

Figura 2.3: Occupazione degli Streaming Multiprocessors.

2.2.3 Analisi delle memorie GPU

L'analisi delle memorie GPU [13] fornisce informazioni dettagliate sul carico di lavoro delegato alle unità di memoria, al fine di dedurre se possa risultare in un fattore limitante per le prestazioni complessive del kernel.

Tale fattore limitante, si traduce in un collo di bottiglia nel momento in cui si utilizzano completamente le unità hardware coinvolte esaurendo la banda di comunicazione disponibile tra tali unità, o raggiungendo il throughput massimo di emissione di istruzioni di scrittura/lettura.

Dalla Figura 2.4, si nota che:

- Le percentuali di Cache Hit in memoria L1 e L2 sono massime (vedi campo "*L1/TEX Hit Rate*" e "*L2 Hit Rate*"), confermando possibili utilizzi degli accessi contigui in memoria o della memoria condivisa (*Shared Memory*) constatati nella Sezione 2.1.

- Il throughput delle attività interne delle unità hardware a disposizione non è elevato (35.8%, vedi campo "*Mem Busy*").

- La banda di comunicazione disponibile è sfruttata solo per metà (51.9%, vedi campo "*Mem Bandwidth*").

- Il throughput di emissione di istruzioni di scrittura/lettura di ogni Streaming Multiprocessor non è elevato (51.9%, vedi campo "*Mem Bandwidth*").

Memory Workload Analysis

Memory Throughput [Mbyte/second]	177,07	Mem Busy [%]	35,80
L1/TEX Hit Rate [%]	97,30	Max Bandwidth [%]	51,99
L2 Hit Rate [%]	99,59	Mem Pipes Busy [%]	51,99
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Figura 2.4: Analisi delle memorie GPU.

2.2.4 Statistiche degli scheduler

Le statische degli scheduler mostrano come le istruzioni sono schedulate tra i vari Warp Idonei (vedi Sezione 1.2.2) permettendo l'analisi quantitativa del numero di Warp Emessi come indicatore di scarsa capacità di nascondere la latenza. Minore è il numero di Warp Emessi, maggiore è latenza introdotta.

Dalla Figura 2.5, si nota che:

- Su un massimo di 12 warp per scheduler, il kernel *K1* alloca una media di 3.72 Warp Attivi per scheduler (vedi campo "*Active Warps Per Scheduler*"), ma solo una media di 0,74 sono Idonei per ciclo di schedulazione (vedi campo "*Eligible Warps Per Scheduler*").

- Su 0.74 Warp Idonei (vedi campo "*Eligible Warps Per Scheduler*") solo 0.33 sono Emessi (vedi campo "*Issued Warp Per Scheduler*"), portando ogni scheduler ad impartire un'istruzione ogni 3.1 cicli. Tali dati si traduco in latenza.

- La maggioranza dei Warp Attivi (il 67.7%) è in stallo rivelandosi non Idoneo (vedi campo "*No Eligible*").

Figura 2.5: Statistiche degli scheduler.

2.2.5 Statistiche dei Warp

Le statistiche dei warp mostrano metriche sull'analisi degli stati (vedi Sezione 1.2.2) in cui tutti i warp hanno trascorso i cicli di schedulazione durante l'esecuzione del kernel *K1*.

Dalla Figura 2.6, si nota che:

- Ogni warp spende 11.35 cicli tra l'emissione/esecuzione di due instruzioni (vedi campo "*Warp Cycles Per Issued Instruction*" e "*Warp Cycles Per Executed Instruction*").

- Ogni warp spende 2.69 cicli in stallo per dipendenza dall'esecuzione (vedi voce "*Stall Wait*" del grafico).

- Il kernel *K1* raggiunge una media di 23.64 thread attivi per ciclo (vedi campo "*Avg. Active Threads Per Warp*"). Questo valore viene ulteriormente ridotto a 22.5 thread per warp a causa della predicazione adottata dal compilatore per evitare una ramificazione del codice (vedi campo "*Avg. Not Predicated Off Threads Per Warp*").

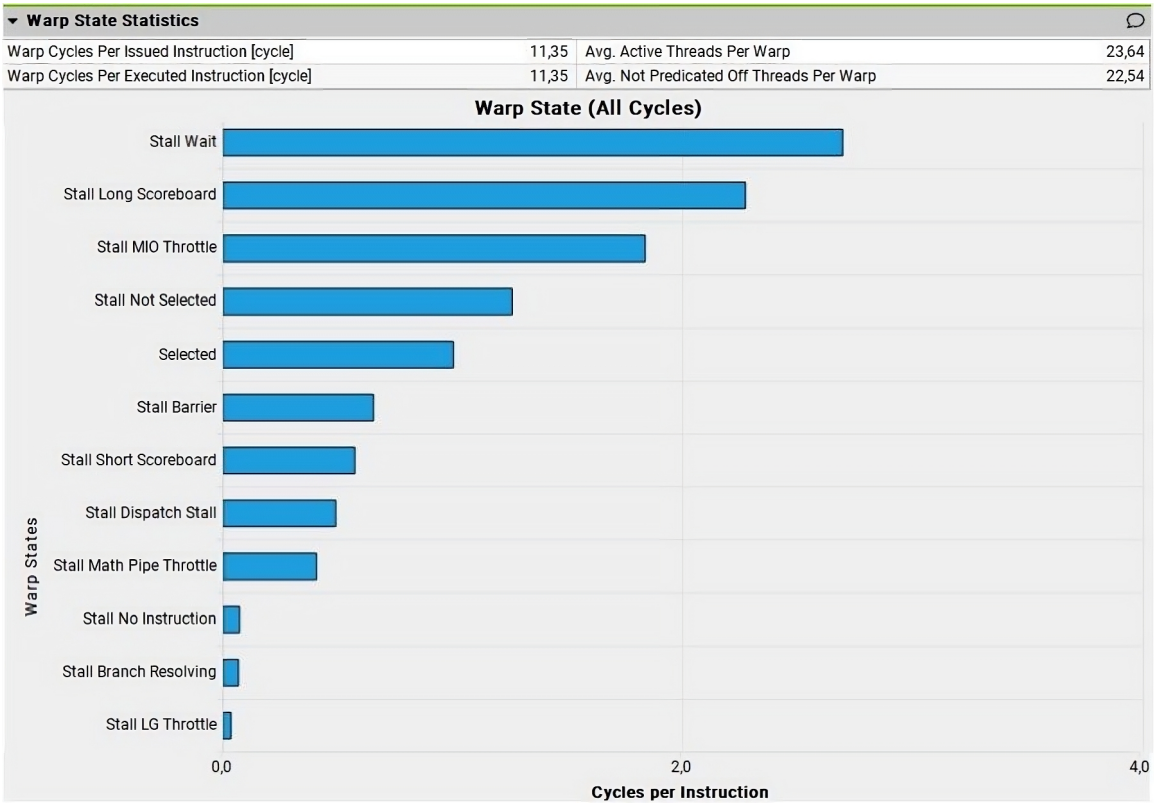


Figura 2.6: Statistiche dei Warp.

2.3 Analisi del codice

L'algoritmo utilizzato per la rimozione del rumore si basa sulla ricerca e il confronto di Patch simili nell'immagine di input (vedi Sezione 1.1). A tale scopo, ogni blocco di thread parte da una Patch di riferimento per confrontare le altre.

In Algoritmo 2 è mostrato il codice del kernel *K1* profilato precedentemente nella Sezione 2.2, mentre l'Algoritmo 1 contiene la struttura delle Patch utilizzata.
Si denotano le seguenti variabili dall'Algoritmo 2:

- La variabile *img* (riga 1) rappresenta l'immagine di input in memoria globale. La sua struttura non è rilevante ai fini dell'ottimizzazione.
- La variabile *maxN* (riga 1) rappresenta il numero di Patch simili da trovare.
- La variabile *d_nstacks* rappresenta l'array in memoria globale dei conteggi delle Patch simili trovate per ogni Patch di riferimento.
- La variabile *d_stacks*(riga 1) rappresenta l'array in memoria globale contenente *MaxN* Patch simili per ogni Patch di riferimento.
- Le variabili *patchSize* e *sizeWindow* (riga 1) rappresentano rispettivamente la grandezza della Patch di riferimento e la grandezza della finestra di ricerca.
- La variabile in memoria condivisa (*Shared Memory*) *referencePatch* (riga 2) rappresenta la Patch di riferimento.
- La variabile in memoria condivisa (*Shared Memory*) *distances* (riga 3) rappresenta l'array di distanze di ogni Patch nell'intorno di ricerca.
- La variabile *thisStackVal* (riga 16) rappresenta una variabile Patch temporanea caricata con un valore della distanza specifico e l'indirizzo della patch di riferimento del bloccco.

Analizzando l'Algoritmo 2, si nota che:

- La funzione *GetThreadId* (riga 5) recupera l'id del thread corrente all'interno del blocco di thread.

- La funzione *GetReferencePatchIdInsideVolume* (riga 6) recupera l'id della Patch di riferimento all'interno dell'immagine sulla base dell'indice del blocco nella griglia.
- La funzione *InitReferencePatch* (riga 8) carica i valori della Patch di riferimento dall'immagine di input.
- La funzione *GetPatchId* (riga 17) calcola l'indirizzo della Patch di riferimento.
- La funzione *Compute_distance* (riga 12) confronta la Patch di riferimento con la Patch corrispettiva al thread in esecuzione. Il suo valore di ritorno è minore di $REAL\_MAX$) (il più grande float rappresentabile) se simili, altrimenti $REAL\_MAX$).
  Il costo computazionale di questa funzione è $O(sizeWindow * sizeWindow * sizeWindow)$.
- La funzione *Add_stack* (riga 22) opera un inserimento ordinato della Patch *thisStackVal* nell'array di Patch simili *d_stacks* in memoria globale. Allo stesso tempo, aggiorna il numero *d_nstacks* di patch simili trovate, se necessario.
  Il costo computazionale di questa funzione è $O(MaxN)$.
- Il ramo condizionale da riga 15 a 26 è eseguito in ogni blocco da un solo thread (l'unico con threadId a 0). Quest'ultimo è incaricato di iterare sull'array *distances* ed eseguire l'inserimento ordinato di *Add_stack*.
  Considerando N come la grandezza dell'array *distances* di $patchSize*patchSize*patchSize$ elementi, il costo computazione complessivo di questo ramo è $O(N * maxN)$.

Algorithm 1 Struttura della Patch

```
1 struct Patch
2 {
3     unsigned long ind;
4     float val;
5 };
```

Algorithm 2 Kernel K1

```
1  __global__ void K1(Image* img, Patch* d_stacks, int* d_nStacks, int
       patchSize, int sizeWindow) {
2      __shared__ float referencePatch[patchSize * patchSize * patchSize];
3      __shared__ float distances[sizeWindow * sizeWindow * sizeWindow];
4
5      const int threadId =  GetThreadId();
6      const int referencePatchId = GetReferencePatchIdInsideVolume(
           blockIdx);
7      if (threadId < patchSize*patchSize*patchSize) {
8          InitReferencePatch(referencePatch, img, referencePatchId);
9      }
10     __syncthreads();
11     if (threadId < sizeWindow*sizeWindow*sizeWindow) {
12         distances[threadId] = Compute_distance(img, size, referencePatch
               , referencePatchId);
13     }
14     __syncthreads();
15     if (threadId == 0) {
16         Patch thisStackVal;
17         thisStackVal.ind = getPatchId(referencePatchId, threadId);
18         for (auto x = 0; x < sizeWindow; ++x) {
19             for (auto y = 0; y < sizeWindow; ++y) {
20                 for (auto z = 0; z < sizeWindow; ++z) {
21                     thisStackVal.val  = distances[x+sizeWindow*y+
                           sizeWindow*sizeWindow*z];
22                     Add_stack(d_stacks, d_nStacks, thisStackVal, maxN);
23                 }
24             }
25         }
26     }
27 }
```

Capitolo 3

Ottimizzazione

I pattern di programmazione parallela sono il migliore punto di partenza per tentare l'ottimizzazione di codice di terzi quando il dominio di applicazione esula dalle competenze informatiche del programmatore.

Reduce, Scan, Stencil e Partition sono alcuni dei pattern di programmazione più noti e studiati nei corsi di High Performance Computing.
Nel nostro caso, studieremo come eliminare l'overhead di computazione introdotto dal singolo thread (threadId=0) incaricato di trovare $maxN$ Patch con la distanza minore dalla Patch di riferimento e caricarle in ordine decrescente in memoria globale, eseguendo $O(N * maxN)$ operazioni.

3.1 Pattern Riduzione

Il Pattern di programmazione parallela Riduzione (in inglese, Reduce) è la chiave dell'ottimizzazione, ed è implementabile sui calcolatori con $O(log_2 N)$ passi paralleli. Nella sua definizione vi è l'applicazione di un operatore binario (esempio: somma, prodotto, min, max...) a tutti gli elementi di un array.

Matematicamente una Riduzione f è definibile come:

$f : R^n \longrightarrow R$ Con $X = [x1, x2, \ldots, xn] \in R^n$,

t.c. $f(X) = x1 * x2 * \cdots * xn$ $*$ un operatore binario.

Alcuni esempi di riduzione sono:

- min-reduce(X) = $\min(x1, x2, \ldots, xn)$
- sum-reduce(X) = $x1 + x2 + \cdots + xn$

3.1.1 Pattern Riduzione NVIDIA

NVIDIA propone un algoritmo [3] efficiente per GPUs che esegue la riduzione di un array di N elementi in memoria globale in $O(log_2 N)$ passi paralleli.

L'idea principale è rappresentata nella Figura 3.1. Consiste nell'applicare l'operatore binario (in Figura 3.1 la somma) sfruttando accessi contigui tramite indirizzamento sequenziale, dimezzando la lunghezza del passo (stride) ad ogni iterazione. Si tratta di un algoritmo in-place il cui risultato converge nella prima posizione dell'array.

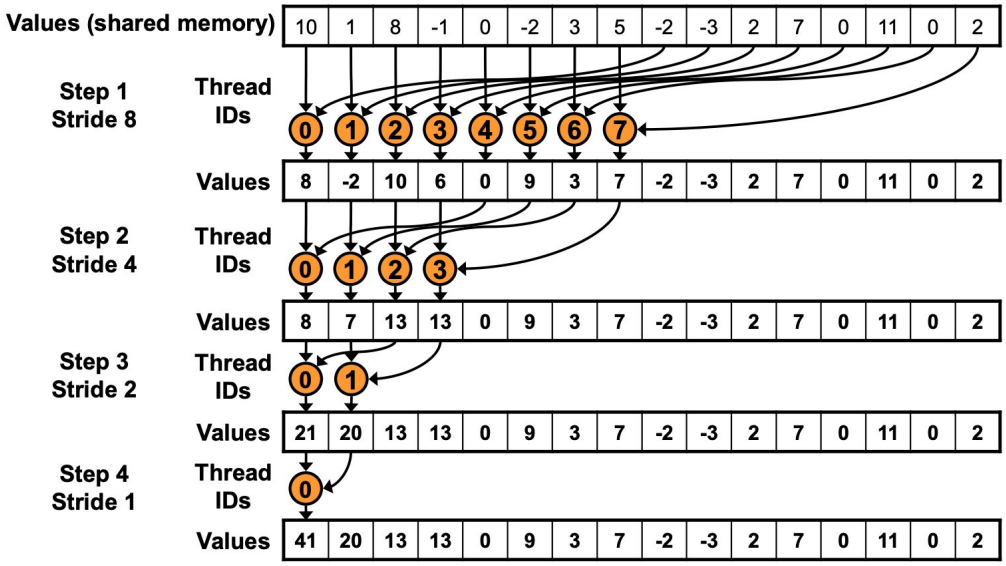


Figura 3.1: Indirizzamento sequenziale in Sum-Reduce. [3]

3.1.2 Implementazione

NVIDIA propone un'implementazione della riduzione [3] derivante da diversi raffinamenti successivi ben documentati nella guida. Nel nostro caso, alcuni raffinamenti non sono applicabili o complicherebbero il codice rendendolo meno manutenibile. Per questo, il mio punto di partenza sarà il codice NVIDIA dell'Algoritmo 3 (comprensivo di 5 raffinamenti su 7 proposti da NVIDIA).
Nello specifico, dall'Algoritmo 3 si nota che:

- Ogni blocco della griglia di thread effettua una riduzione su una partizione dell'array *g_idata*.
- Il risultato di ogni riduzione è memorizzato in *g_odata* all'indice del blocco corrispondente. Per completare la riduzione è necessario richiamare la stessa su *g_odata*, sfruttando una tecnica chiamata Kernel Decomposition [3].
- La funzione *warpReduce* (riga 1) effettua l'unrolling delle iterazioni dell'ultimo warp del blocco, sfruttando il fatto che le operazioni a livello di warp sono SIMD sincrone e non necessitano sincronizzazioni con *__syncthreads()*.
- La direttiva *volatile* [9] alla riga 1 è necessaria per evitare che il compilatore ottimizzi le letture/scritture della variabile *sdata* (ad esempio salvando le letture da memoria condivisa in cache L1 o registri) causando potenziali errori semantici.

Algorithm 3 Riduzione NVIDIA

```
1  __device__ void warpReduce(volatile int *sdata, unsigned int tid) {
2      sdata[tid] += sdata[tid + 32];
3      sdata[tid] += sdata[tid + 16];
4      sdata[tid] += sdata[tid + 8];
5      sdata[tid] += sdata[tid + 4];
6      sdata[tid] += sdata[tid + 2];
7      sdata[tid] += sdata[tid + 1];
8  }
9
10 __global__ void reduce(int *g_idata, int *g_odata, unsigned int n) {
11     extern __shared__ int sdata[];
12     unsigned int tid = threadIdx.x;
13     unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
14     sdata[tid] = g_idata[i];
15     for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
16         if (tid < s) { sdata[tid] += sdata[tid + s]; }
17         __syncthreads();
18     }
19     if (tid < 32) warpReduce(sdata, tid);
20     if (tid == 0) g_odata[blockIdx.x] = sdata[0];
21 }
```

3.2 Algoritmo MinAndSwap

Il cuore dell'ottimizzazione consiste nell'adattare l'algoritmo proposto da NVIDIA (vedi Algoritmo 3) al caso d'uso dell'applicazione di denoising.

Per farlo, la considerazione principale da me adottata si basa sul fatto che, per ordinare un qualsiasi array di N elementi, è possibile ricondursi ad un particolare caso di **estrazione del minimo** replicato N volte sullo stesso array.

Tale nozione declinata al caso d'uso di interesse, equivale a ripetere *MaxN* volte una riduzione ad indirizzamento sequenziale ottenendo una selezione ordinata delle prime *MaxN* Patch che meno differiscono dalla Patch di riferimento.

3.2.1 Descrizione

L'algoritmo sviluppato, chiamato *MinAndSwap*, utilizza un estrazione **in-place** delle *MaxN* Patch più simili. A tale scopo, l'operatore binario di riduzione utilizzato tra gli elementi dell'array è il minimo con scambio. Ovvero, ogni coppia di elementi è confrontata e scambiata mantenendo il minimo nelle posizioni più basse dell'array.

Ne consegue che, ogni iterazione di *MinAndSwap* modella l'array sulla base delle iterazioni precedenti senza considerare i minimi estratti fino a quel momento (e posizionati negli indici più bassi dell'array). Algoritmicamente, si traduce nel considerare l'array dall'indice successivo ad ogni iterazione.

Nella Figura 3.2 che segue sono mostrate due iterazioni dell'algoritmo *MinAndSwap* rappresentando con stessi colori gli elementi dell'array scambiati di posizione. Dalla Figura 3.2 si nota che:

- Il minimo della prima iterazione (in questo caso l'elemento di valore -3) non è considerato nella seconda iterazione dell'algoritmo.
- L'algoritmo è composto da *MaxN* chiamate ad una riduzione con indirizzamento sequenziale. Considerando *MaxN* una costante dell'algoritmo, il costo computazionale di *MinAndSwap* è $O(log_2 N)$.

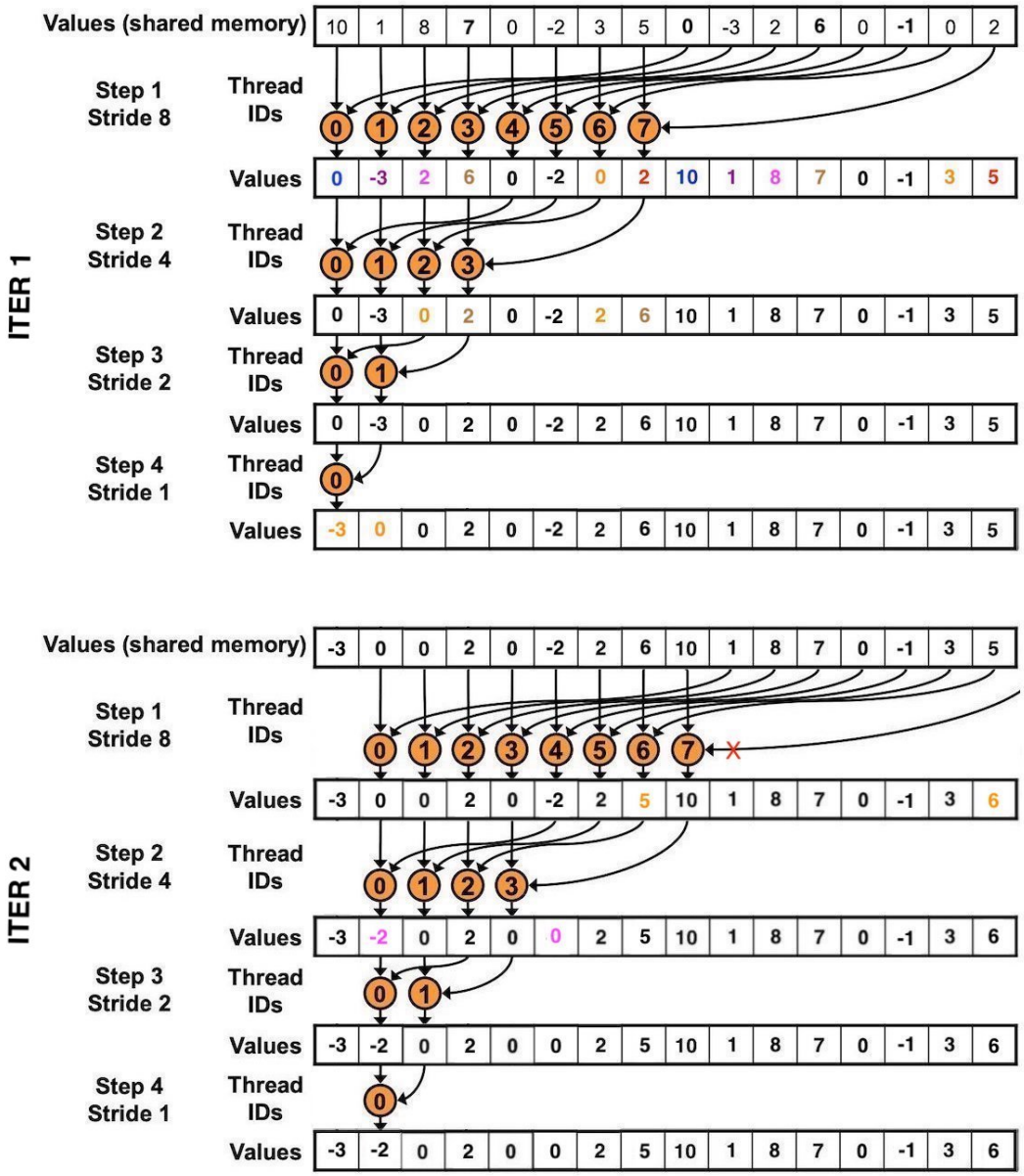


Figura 3.2: Algoritmo *MinAndSwap* basato su Pattern di riduzione NVIDIA ad indirizzamento sequenziale.

3.2.2 Implementazione

Kernel Reduce

Il codice in Algoritmo 4 implementa il kernel di riduzione dell'algoritmo *MinAndSwap* sfruttando il patter riduzione di NVIDIA.

Le variabili utilizzate adempiono gli stessi scopi di quelle presenti nel codice NVIDIA, ma hanno nomi diversi per meglio avvicinarsi a quanto effettivamente programmato e poter dare riferimenti migliori a chi leggerà questa tesi come documentazione.

Dal codice in Algoritmo 4 si nota che:

- La funzione *min_and_swap* (riga 1) si occupa di applicare l'operatore binario della riduzione, trovando il minimo tra due elementi dell'array di Patch *sdata* mantenendolo nell'indice più basso.

- La funzione *warpReduce* (riga 14) si occupa dell'unrolling delle iterazioni nell'ultimo warp. Per motivi di semplicità, non è mostrato il kernel *min_and_swap* richiamato da *warpReduce* che fa uso della direttiva *volatile* [9] per la variabile *sdata*, in quanto contenente il medesimo codice dell'omonimo kernel citato al punto precedente.

- La funzione *findMinOfArray* (riga 26) si occupa di operare la min-reduce inplace sull'array di Patch *minChunk* di lunghezza *arrayLength*.

Algorithm 4 Kernel di riduzione di MinAndSwap.

```
1  __device__ void min_and_swap(Patch* sdata, unsigned int tid1, unsigned
       int tid2)
2  {
3      unsigned int idMax = (data[tid1].val >= sdata[tid2].val) ? tid1 :
           tid2;
4      unsigned int idMin = (idMax == tid1) ? tid2 : tid1;
5      Patch tmp;
6      tmp.val = sdata[idMax].val;
7      tmp.ind = sdata[idMax].ind;
8      sdata[tid1].val = sdata[idMin].val;
9      sdata[tid1].ind = sdata[idMin].ind;
10     sdata[tid2].val = tmp.val;
11     sdata[tid2].ind = tmp.ind;
12 }
13
14 __device__ void warpReduce(volatile Patch* sdata, unsigned int tid)
15 {
16     min_and_swap(sdata, tid, tid + 32);
17     min_and_swap(sdata, tid, tid + 16);
18     min_and_swap(sdata, tid, tid + 8);
19     min_and_swap(sdata, tid, tid + 4);
20     min_and_swap(sdata, tid, tid + 2);
21     min_and_swap(sdata, tid, tid + 1);
22 }
23
24 __device__ void findMinOfArray(Patch* minChunk, int arrayLength)
25 {
26     unsigned int tid = GetThreadId();
27     for(unsigned int s = (arrayLength/2); s > 32 ; s /=2){
28         if(tid < s) min_and_swap(minChunk, tid, tid + s);
29         __syncthreads();
30     }
31     if(tid < 32) warpReduce(minChunk, tid);
32 }
```

Kernel K1

Il Codice 5 implementa l'algoritmo *MinAndSwap* nel kernel K1 (vedi Algorirmo 2) dell'applicazione, e delinea i principali cambiamenti:

- L'array *d_stacks_shared* (riga 3) assolve le stesse funzioni dell'array *distances* nell'Algoritmo 2, ma essendo un array di Patch permette di poter applicare l'ordinamento basato su *MinAndSwap* senza farsi carico di ulteriori stratagemmi per ricostruire le informazioni mancanti delle Patch una volta ordinate, come il campo *ind*.

- L'algoritmo *MinAndSwap* è implementato nel *for* a riga 16. Da notare, ad ogni iterazione si considera porzioni dell'array sempre minori partendo dall'indice successivo (vedi riga 17), al fine di non considerare i minimi precedentemente estratti.

- Al fine di mantenere la stessa semantica dell'algoritmo originale, l'aggiornamento dell'array in memoria globale contenente le *MaxN* Patch più simili è effettuato mantenendo l'ordine decrescente, quindi leggendo gli elementi di *d_stacks_shared* nell'ordine inverso (vedi riga 22).

- L'aggiornamento del conteggio delle Patch simili trovate è effettuato in una funzione a parte, *updateCountStacks* (vedi riga 24).

Algorithm 5 Kernel K1 con riduzione MinAndSwap.

```
1  __global__ void K1(Image* img, Patch* d_stacks, int patchSize, int
       sizeWindow, int maxN) {
2      __shared__ float referencePatch[patchSize * patchSize * patchSize];
3      __shared__ Patch d_stacks_shared[sizeWindow * sizeWindow *
           sizeWindow];
4      const int threadId =  GetThreadId();
5      const int referencePatchId = GetReferencePatchIdInsideVolume(
           blockIdx);
6      if (threadId < patchSize*patchSize*patchSize) {
7          InitReferencePatch(referencePatch, img, referencePatchId);
8      }
9      __syncthreads();
10     d_stacks_shared[tid].ind = getPatchId(referencePatchId, threadId);
11     if (threadId < sizeWindow*sizeWindow*sizeWindow) {
12         d_stacks_shared[threadId].val = Compute_distance(img, size,
               referencePatch, referencePatchId);
13     }
14     // selection of first maxN stacks - ascending order
15     __syncthreads();
16     for(int i = 0; i < maxN; i++) {
17         findMinOfArray(d_stacks_shared+i, sizeWindow * sizeWindow *
               sizeWindow-i);
18         __syncthreads();
19     }
20     // update stacks in global memory - descending order
21     if (tid < maxN && d_stacks_shared[tid].val < REAL_MAX) {
22         d_stacks[maxN * ind + tid] = d_stacks_shared[maxN - 1 - tid];
23     }
24     updateCountStacks(d_stacks_shared, d_nStacks, maxN);
25 }
```

Kernel updateCountStack

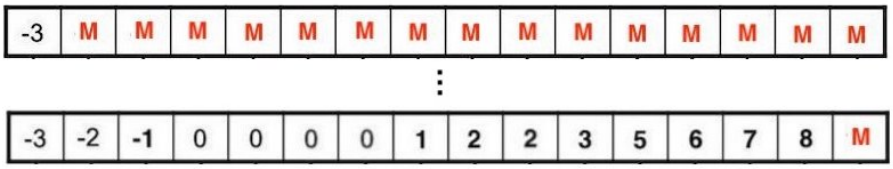
Il meccanismo di conteggio delle Patch simili, e quindi del kernel *updateCountStacks* in Algoritmo 6, è basato sulla scrittura in memoria globale da parte di un solo thread, l'unico il cui indice i assolve uno dei seguenti stati mostrati in Figura 3.3 (dove la M rossa è un alias per il valore $REAL\_MAX$) relativi all'array X di Patch ordinate:

1. Se i è 0 e $X[i]$ non è una Patch simile (campo *val* della Patch uguale a $REAL\_MAX$), allora il conteggio è 0 (vedi Figura 3.3a e riga 4 dell'Algoritmo 6).

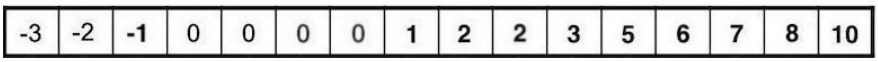
2. Se i è minore di $MaxN$ e si verifica la condizione per la quale $X[i]$ non è una Patch simile mentre $X[i-1]$ lo è (campo *val* della Patch minore di $REAL\_MAX$), allora il conteggio equivale a i (vedi Figura 3.3b e riga 9 dell'Algoritmo 6).

3. Se i è $MaxN$-1 e $X[i]$ è una Patch simile, allore il conteggio è $MaxN$ (vedi Figura 3.3c e riga 14 dell'Algoritmo 6).

(a) Caso in cui nessuna Patch simile sia stata trovata.



(b) Caso in cui da 1 a $MaxN$-1 Patch simili siano state trovate.



(c) Caso in cui $MaxN$ Patch simili siano state trovate.

Figura 3.3: Casistiche sfruttate dal kernel *updateCountStacks*.

Algorithm 6 Kernel updateCountStack - conteggio Patch simili.

```
1  __device__ updateCountStacks(Patch* d_stacks_shared, int* d_nStacks, int
       maxN) {
2      const int tid = getThreadId();
3      // count stacks
4      if(tid == 0 && d_stacks_shared[tid].val == REAL_MAX) {
5          d_nStacks[ind] = 0;
6          // skip next branches
7          return;
8      }
9      if(tid < maxN
10         &&  d_stacks_shared[tid].val == REAL_MAX
11         &&  d_stacks_shared[tid-1].val < REAL_MAX) {
12         d_nStacks[ind] = tid;
13     }
14     if(tid == maxN-1 && d_stacks_shared[tid].val < REAL_MAX) {
15         d_nStacks[ind] = maxN;
16     }
17 }
```

3.2.3 Parziale Rimozione Unrolling in WarpReduce

L'unrolling adottato da NVIDIA in *WarpReduce* (vedi l'Algoritmo 3) presuppone che ogni riga del kernel sia eseguita da tutti i 32 thread dell'ultimo warp, apportando modifiche ridondanti all'array ad un costo di schedulazione $O(1)$ (in quanto si parla di operazioni SIMD a livello di Warp). Tali modifiche a *sdata* non hanno ripercussioni sulle iterazioni successive dell'algoritmo, grazie al fatto che l'algoritmo non è in-place.

Al contrario, un algoritmo in-place come *MinAndSwap* non permette di modificare l'array a piacere. Per questo, come mostrato di seguito in Algoritmo 7, è necessario modificare il kernel *WarpReduce* regolando i thread attivi con condizioni aggiuntive, di fatto eliminando parzialmente l'unrolling adottato da NVIDIA (le operazioni non richiedono comunque alcun *__syncthreads()*).

Algorithm 7 Kernel WarpReduce con Unrolling Parziale.

```
1  __device__ void warpReduce(volatile Patch* sdata, unsigned int tid)
2  {
3      min_and_swap(sdata, tid, tid + 32);
4      if(tid<16)
5          min_and_swap(sdata, tid, tid + 16);
6      if(tid<8)
7          min_and_swap(sdata, tid, tid + 8);
8      if(tid<4)
9          min_and_swap(sdata, tid, tid + 4);
10     if(tid<2)
11         min_and_swap(sdata, tid, tid + 2);
12     if(tid<1)
13         min_and_swap(sdata, tid, tid + 1);
14 }
```

3.2.4 Generalizzazione a potenze di 2

Il codice NVIDIA presuppone che la lunghezza dell'array di input sia una potenza di 2 (vedi l'Algoritmo 3). Al fine di rilassare tale vincolo e avvicinarsi al caso d'uso reale, è stato ricondotto il kernel *findMinOfArray* a lavorare sempre con la minor potenza di 2 maggiore della lunghezza dell'array di input.
Nell'Algoritmo 8 viene implementa questa idea, e si nota che:

- La potenza di 2 è calcolata trovando il primo bit a 1 che precede la rappresentazione binaria di *arrayLength* (vedi riga 5).
  A tale scopo, il primo bit a 1 di un intero a 32 bit (in esadecimale equivale a 0x80000000) è stato shiftato a destra di una posizione in meno rispetto al numero di leading-zeros (zeri a sinistra del primo bit a 1) in *arrayLength* tramite la funzione intrinseca *__clz* delle Math API CUDA [7].

- Una ulteriore condizione è stata aggiunta per evitare che gli indici eccedano la reale lunghezza dell'array (vedi riga 8).

Algorithm 8 Kernel findMinOfArray - Generalizzazione a potenza di 2.

```
1  __device__ void findMinOfArray(Patch* minChunk, int arrayLength)
2  {
3      unsigned int tid = GetThreadId();
4      int realArrayLength = arrayLength;
5      arrayLength = (0x80000000>>(__clz (arrayLength)-1));
6      for(unsigned int s = (arrayLength/2); s > 32 ; s /= 2)
7      {
8          if(thread_id < s && thread_id + s < realArrayLength)
9                  min_and_swap(minChunk, thread_id, thread_id + s);
10         __syncthreads();
11     }
12     if(tid < 32) warpReduce(minChunk, tid);
13 }
```

Capitolo 4

Risultati

4.1 Prestazioni

Le prestazioni dell'algoritmo ottimizzato sono migliori rispetto all'originale in termini di speedup.
In Figura 4.1 sono mostrati le medie di una decina di esecuzioni dei due differenti algoritmi sulla macchina di test WinRTX3060 (vedi Sezione 1.3).
Su quest'ultimo hardware, l'algoritmo ottimizzato ottiene mediamente uno speedup di 1.9x rispetto all'algoritmo originale.

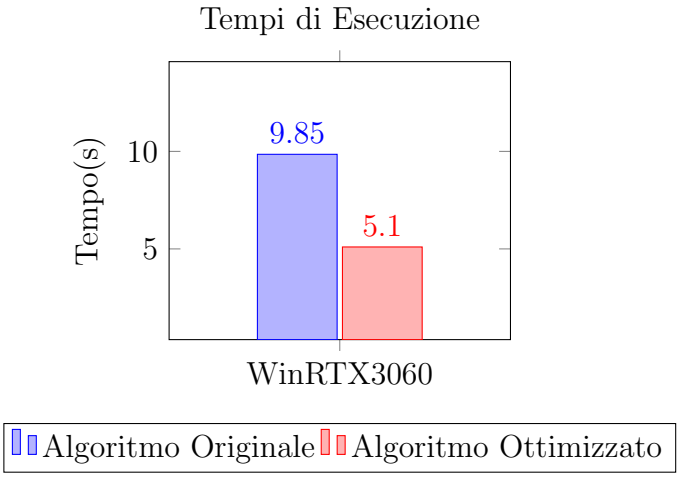


Figura 4.1: Tempi di esecuzione a confronto.

4.2 Qualità dell'immagine

La Figura 4.2 confronta le immagini dell'algoritmo originale e di quello ottimizzato, mostrando che il lavoro di questa tesi migliora le prestazioni in termini di speedup senza apportare cambiamenti al risultato finale.

Per dimostrare tale affermazione, la Figura 4.3 mostra una matrice di differenza tra le immagini risultanti dall'algoritmo originale (vedi prima riga della matrice) e da quello ottimizzato (vedi prima colonna della matrice della matrice), utilizzando istogrammi della distribuzione dei pixel.

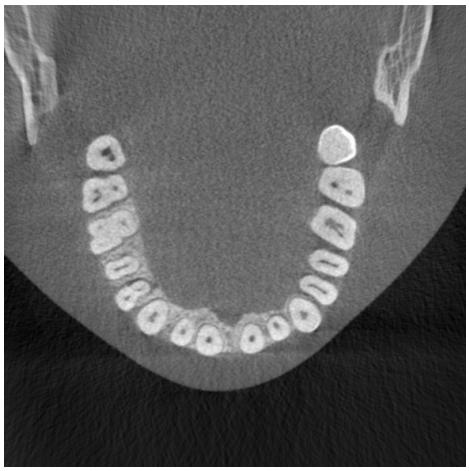
Ogni istogramma centrale (vedi secondo riga e terza riga, ignorando la prima colonna) rappresenta la differenza tra il risultato dell'algoritmo originale nella stessa colonna e quello dell'algoritmo ottimizzato nella stessa riga.

Per motivi di analisi, gli istogrammi si riferiscono ad una stessa sotto-regione delle immagini normalizzate rispetto all'immagine di input, quindi divisi per essa.

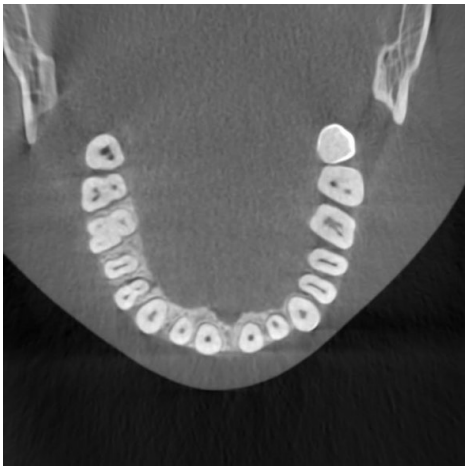
Analizzando la Figura 4.3, si nota che:

- I risultati dell'algoritmo ottimizzato sono identici a quelli dell'algoritmo originale dal punto di vista della distribuzione dei pixel in qualsiasi zona dell'immagine (vedi prima colonna e prima riga della matrice).

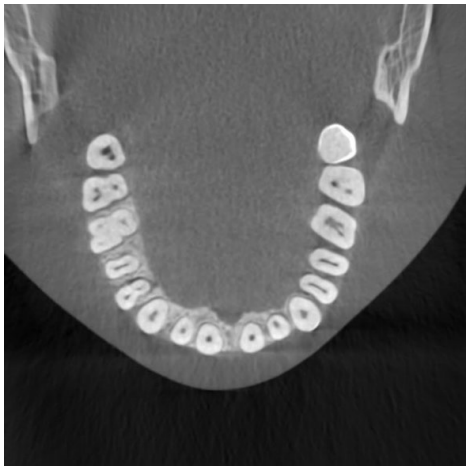
- Le differenze tra le immagini (vedi secondo riga e terza riga, ignorando la prima colonna) risultanti dall'algoritmo originale e quello originale sono conseguenza del diverso ordine di scelta delle Patch con stessa distanza di similarità dalla Patch di riferimento, introdotto da *MinAndSwap*.



(a) Immagine Originale.



(b) Immagine Denoised.



(c) Immagine Denoised ottimizzata.

Figura 4.2: Risultati a confronto.

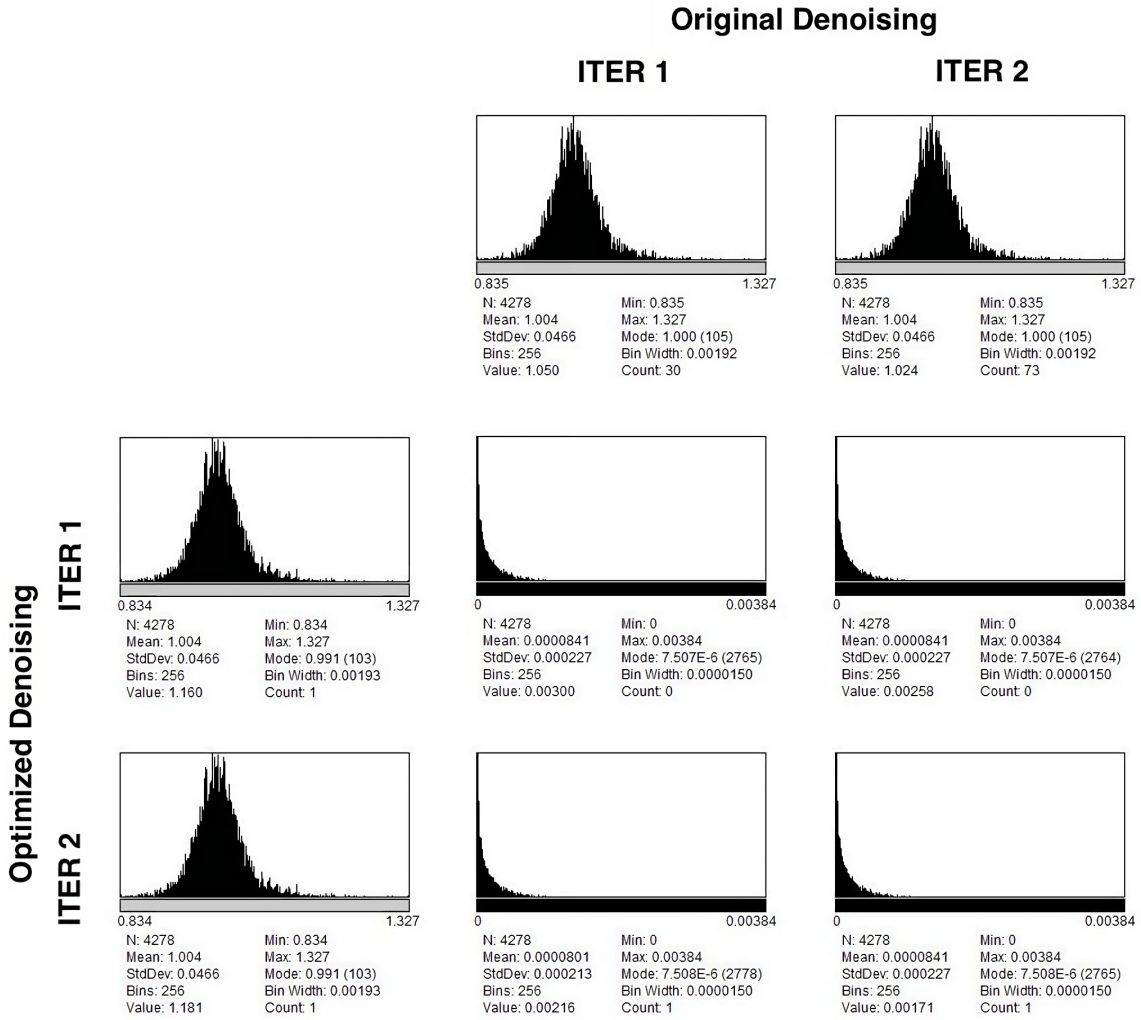


Figura 4.3: Distribuzione dei pixel.

4.3 Profilazione Generica - Post Ottimizzazione

Come visto in Sezione 2.1, l'applicazione ottimizzata è profilata con Nsight System per ottenere un'analisi di alto livello dell'algoritmo ottimizzato.

Confrontando la Figura 4.4 con la Figura 2.1 su stessa scala temporale, si nota che:

- Il throughput di computazione è quasi raddoppiato (vedi voce "*SM Throughput*") grazie alla capacità di *MinAndSwap* ordinare le Patch ad un costo computazionale minore.

- Il throughput della memoria L1 è quasi raddoppiato (vedi voce "*L1 Throughput*"), contrariamente ai throughput di L2 e VRAM che sono dimezzati o rimasti invariati (vedi voce "*L2 Throughput*", "*VRAM Throughput*"). Tali dati, sono la conseguenza di un maggiore utilizzo della memoria condivisa (*Shared Memory*) in *MinAndSwap*.

- L'occupazione degli Streaming Multiprocessor è massima nell'algoritmo ottimizzato (vedi voce "*SM Occupancy*"), indicando un alto utilizzo delle risorse computazionali a disposizione grazie all'utilizzo di un elevato numero di warp in *MinAndSwap*.

- I Warp Idonei (vedi Sezione 1.2.2) sono praticamente raddoppiati rispetto all'algoritmo originale (vedi voce "*Warps Eligible*"), conseguenza del maggiore numero di warp coinvolti in *MinAndSwap* e quindi di una maggiore probabilità di trovare warp disponibili ad impartire l'istruzione successiva.

- Lo stallo per dipendenza dall'esecuzione riguarda tutti i Warp Attivi anche dopo l'ottimizzazione (vedi voce "*FE Stalls*"), indicando uno stallo insito nelle istruzioni dell'algoritmo o una modellazione dell'input che causa un collo di bottiglia negli accessi in memoria globale.

- Le Cache Hit per le memorie L1 e L2 non variano rispetto all'algoritmo originale.

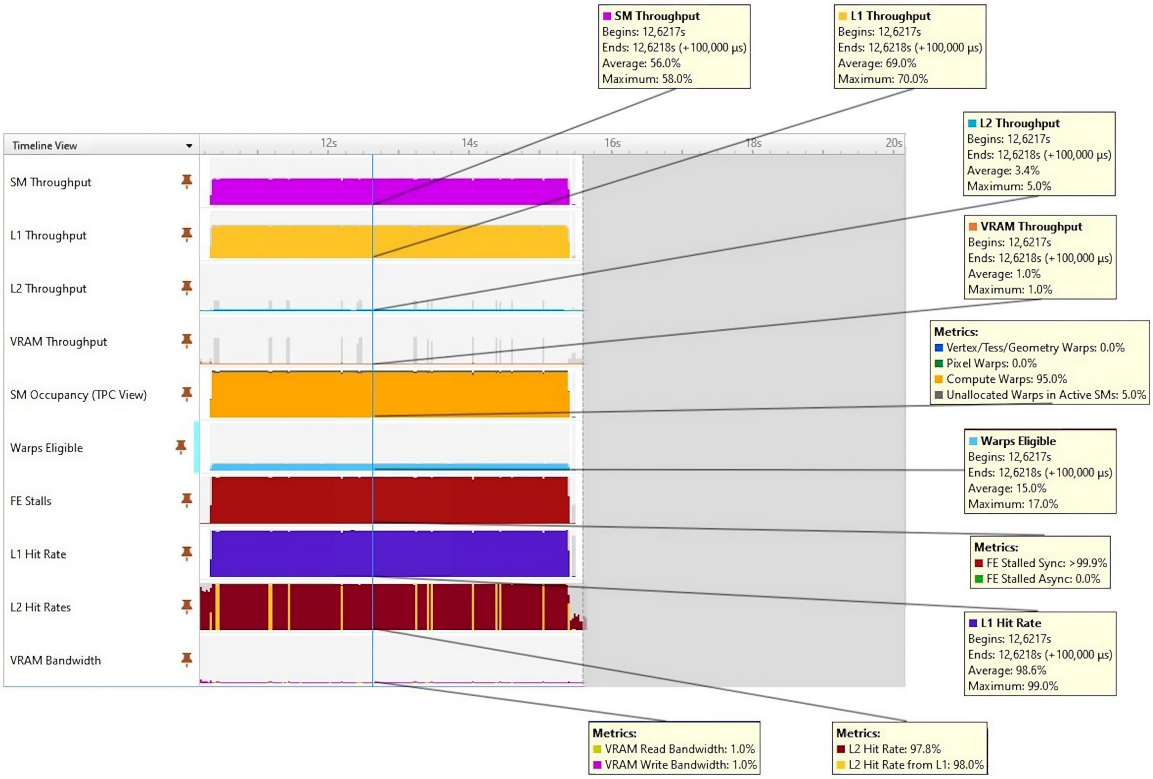


Figura 4.4: Profilazione NSight System dell'applicazione.

4.4 Profilazione Mirata - Post Ottimizzazione

Come visto in Sezione 2.2 profileremo il kernel *K1*, utilizzando un input ridotto, per ottenere un'analisi dettagliata delle prestazioni dell'algoritmo ottimizzato.

4.4.1 SpeedOfLight Throughput

In Figura 4.5 sono raffigurati i risultati della NVIDIA SOL Throughput (vedi Sezione 2.2.1).
Confrontando la Figura 4.5 con la Figura 2.2, si nota che:

- L'algoritmo ottimizzato ha prestazioni migliori sia in termini di throughput computazionale che di utilizzo della banda a disposizione (vedi campo "*Com-*

pute(SM) Throughput" e "*Memory Throughput*").

- Come constatato nella Sezione 4.3, la memoria L1 è ampiamente utilizzata nell'algoritmo ottimizzato rispetto all'algoritmo originale ed alle memorie L2 e DRAM (vedi campo "*L1/TEX Cache Throughput*", "*L2 Cache Throughput*", "*DRAM Throughput*").

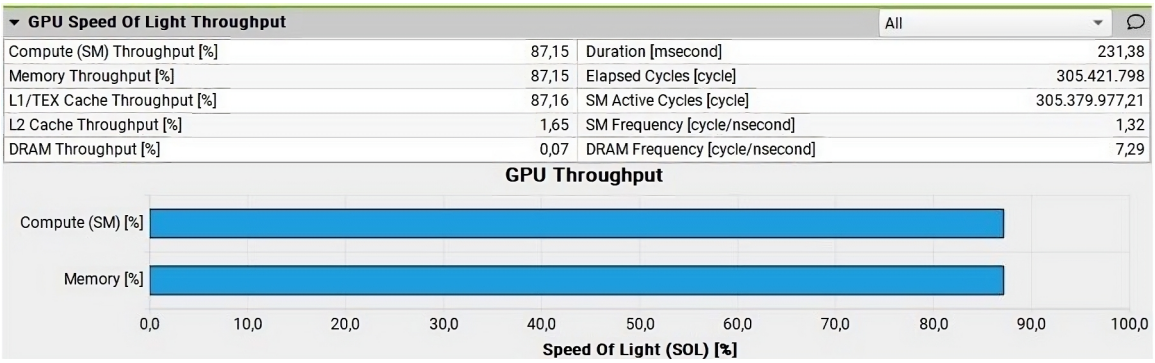


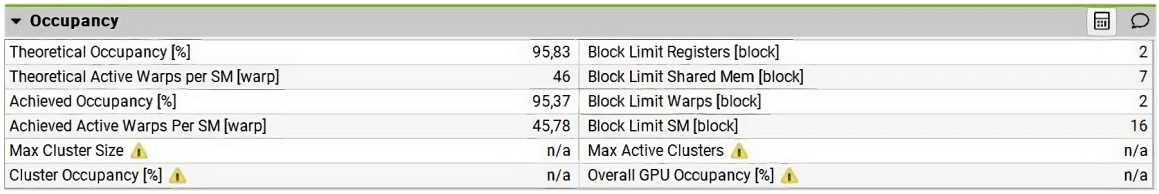
Figura 4.5: Metriche di Throughput della GPU.

4.4.2 Occupazione degli Streaming Multiprocessor

In Figura 4.6 sono mostrate le metriche di occupazione (spiegata nella Sezione 2.2.2) degli Streaming Multiprocessor nell'algoritmo ottimzzato.
Confrontando la Figura 4.6 con la Figura 2.3, si nota che:

- Come constatato nella Sezione 4.3, l'algoritmo ottimizzato raggiunge un'occupazione effettiva (vedi campo "*Achieved Occupancy*") che sfrutta quasi completamente (95.37%) l'occupazione teorica a disposizione (vedi campo "*Theoretical Occupancy*"), a differenza dell'algoritmo originale.

- Il numero di Warp Attivi per SM aumenta rispetto all'algoritmo originale arrivando ad una media di 45.78 warp sui 46 teoricamente disponibili (vedi campo "*Achieved Active Warps Per SM*" e "*Theoretical Active Warps Per SM*"), comportando il possibile raggiungimento di un maggiore numero di Warp Idoeni come constatato nella Sezione 4.3.



Occupancy			
Theoretical Occupancy [%]	95,83	Block Limit Registers [block]	2
Theoretical Active Warps per SM [warp]	46	Block Limit Shared Mem [block]	7
Achieved Occupancy [%]	95,37	Block Limit Warps [block]	2
Achieved Active Warps Per SM [warp]	45,78	Block Limit SM [block]	16
Max Cluster Size	n/a	Max Active Clusters	n/a
Cluster Occupancy [%]	n/a	Overall GPU Occupancy [%]	n/a

Figura 4.6: Occupazione degli Streaming Multiprocessors.

4.4.3 Analisi delle memorie GPU

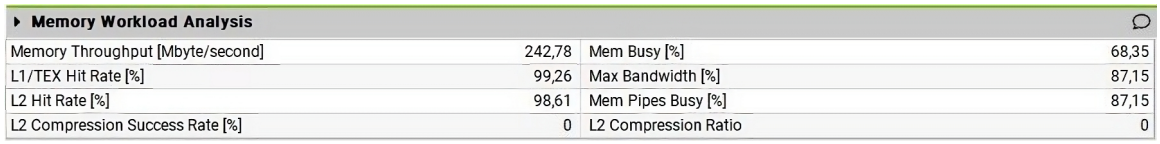
In Figura 4.7 sono rappresentate le analisi delle risorse di memorizzaione della GPU nell'algoritmo ottimizzato.
Confrontando la Figura 4.7 con la Figura 2.4, si nota che:

- Dopo l'ottimizzazione, il throughput della memoria raggiunge 242.78 MB/s rispetto i 177.07 MB/s dell'algoritmo originale (vedi campo "*Memory Throughput*").

- Le risorse di memoria sono più utilizzate nell'algoritmo ottimizzato rispetto all'algoritmo originale (vedi campo "*Mem Busy*").

- La banda di comunicazione disponibile è meglio sfruttata nell'algoritmo ottimizzato rispetto all'algoritmo originale (vedi campo "*Mem Bandwidth*").

- Il throughput di emissione di istruzioni di scrittura/lettura di ogni Streaming Multiprocessor è maggiore nell'algoritmo ottimizzato rispetto all'algoritmo originale (vedi campo "*Mem Bandwidth*").



Memory Workload Analysis			
Memory Throughput [Mbyte/second]	242,78	Mem Busy [%]	68,35
L1/TEX Hit Rate [%]	99,26	Max Bandwidth [%]	87,15
L2 Hit Rate [%]	98,61	Mem Pipes Busy [%]	87,15
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Figura 4.7: Analisi delle memorie GPU.

4.4.4 Statistiche degli scheduler

In Figura 4.8 sono rappresentate le statistiche degli scheduler nell'algoritmo ottimizzato.
Confrontando la Figura 4.8 con la Figura 2.5, si nota che il numero di Active Warp aumenta sensibilmente nell'algoritmo ottimizzato passando ad 11.44 (vedi campo "*Active Warps Per Scheduler*"), mentre diminuisce la percentuale di Warp non-Idonei passando a 44.26% (vedi campo "*No Eligible*").
Complessivamente, si può dire che il Kernel *K1* ottimizzato ha una migliore capacità di nascondere la latenza.

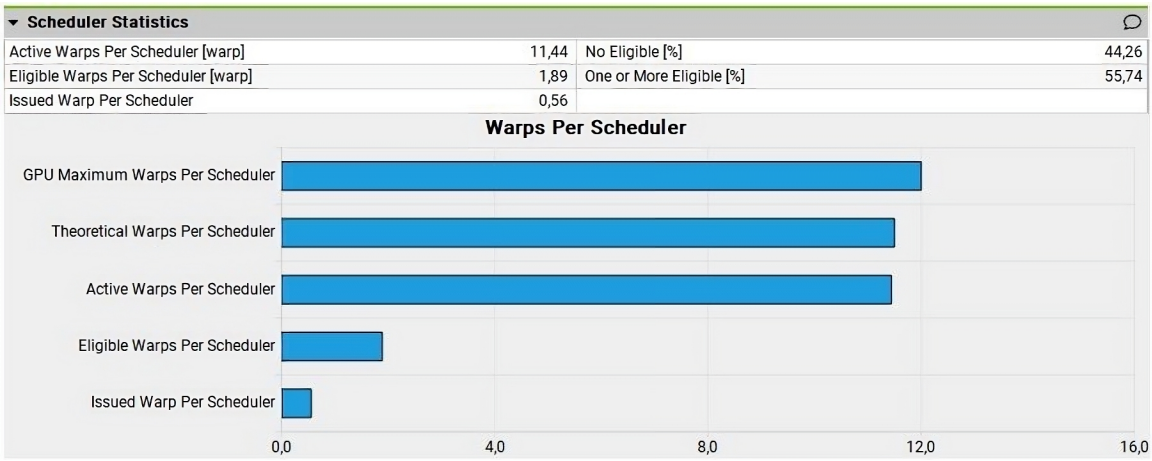


Figura 4.8: Statistiche degli scheduler.

4.4.5 Statistiche dei Warp

In Figura 4.9 sono mostrate l'analisi complessiva degli stati dei warp nell'algoritmo ottimizzato.
Confrontando la Figura 4.9 con la Figura 2.6, si nota che:

- Aumentano considerevolmente i warp in stallo per sincronizzazione nell'algoritmo ottimizzato (vedi voce "*Stall Barrier*" nel grafico), pertanto aumentano tutte le metriche analizzate (vedi tabella).
  Tale comportamento, è causato dal funzionamento del pattern di riduzione in *MinAndSwap*.
  Infatti, i rami condizionali nelle iterazioni del *for* dell'Algoritmo 4 creano divergenze nell'esecuzione dei threads che necessitano di aspettarsi nella direttiva *__syncthreads()*.

- Gli stati dei warp constatati nell'algoritmo originale, eccetto gli stalli per sincronizzazione, sono i medesimi una volta ottimizzato (vedi grafico).

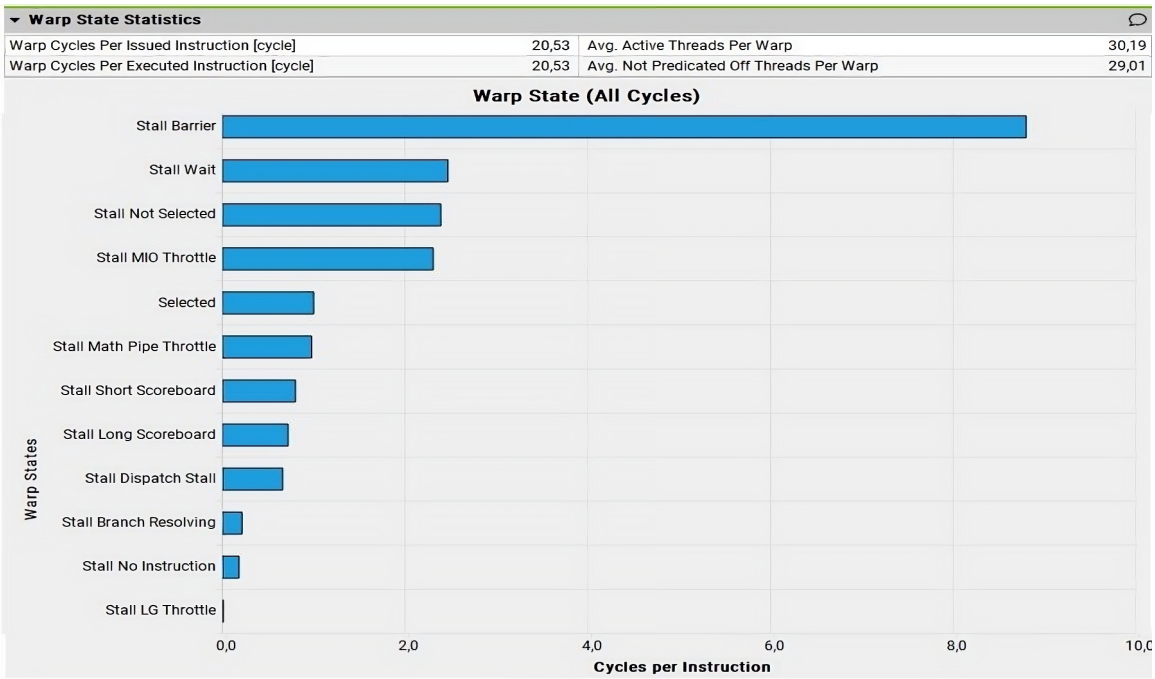


Figura 4.9: Statistiche dei Warp.

Conclusioni

La tesi ha raggiunto gli obiettivi prefissati (vedi sezione Sezione 1.4) profilando l'applicazione di denoising con strumenti NVIDIA e permettendo l'attuazione di ottimizzazioni mirate del codice CUDA.
Superando gli obiettivi prefissati, è stata proposta un'ottimizzazione del kernel *K1* raggiungendo 2x in termini di speedup rispetto al codice originale e potendo approfondire le competenze sull'architettura ed il funzionamento interno delle GPU.

Nel contesto d'uso dell'applicazione, ho assimilato conoscenze sugli algoritmi di elaborazioni delle immagini e sulle metodologie per valutare la qualità dei risultati sulla base delle immagini originali.

Tali considerazioni portano a ritenermi soddisfatto dei risultati conseguiti, non solo per il superamento degli obietti prefissati ma anche per la formazione acquisita.

Ringraziamenti

Desidero ringraziare la mia famiglia per il supporto costante e incondizionato che mi ha consentito una serena dedizione allo studio.
Un sentito ringraziamento al Prof. Moreno Marzolla per i suoi consigli e l'oppurtunità datami di svoglere una tesi in linea con i miei interessi in un settore tanto importante come quello medico.
Ringrazio i dottori Claudio Landi e Leonardo Sassi di SeeThrough s.r.l per aver messo a disposizione risorse e competenze con professionalità e pazienza nei miei confronti.
Infine, ringrazio tutti coloro che mi hanno sostenuto fino a questo momento.

Bibliografia

[1] M.J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[2] Bosky Agarwal. *Instruction Fetch Execute Cycle*. 2004. URL: https://web.archive.org/web/20090611211308/http://www.cs.montana.edu/~bosky/cs518/ife/IFE.pdf.

[3] Mark Harris. *Optimizing Parallel Reduction in CUDA*. 2007. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[4] Erik Lindholm et al. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. DOI: 10.1109/MM.2008.31.

[5] *Statistical Inaccuracy of gprof Output*. 2012. URL: https://web.archive.org/web/20120529075000/http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html).

[6] The Editors of Encyclopaedia. "Computational Complexity". In: Encyclopedia Britannica, 2023. URL: https://www.britannica.com/topic/computational-complexity.

[7] Nvidia. *CUDA Math API - Integer Intrinsics*. 2023. URL: https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html.

[8] Nvidia. *GPU Performance Background User's Guide - Understanding Performance*. 2023. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf.

[9] Nvidia. *CUDA C++ Programming Guide - Volatile Qualifier*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#volatile-qualifier.

[10] Nvidia. *CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/.

[11] Nvidia. *GPU Performance Background User's Guide - GPU Architecture Fundamentals*. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#gpu-arch.

[12] Nvidia. *Nsight Compute*. URL: https://docs.nvidia.com/nsight-compute/.

[13] Nvidia. *Nsight Compute Profile Guide - Sections and Rules*. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#sections-and-rules.

[14] Nvidia. *Nsight System*. URL: https://docs.nvidia.com/nsight-systems/.

[15] Nvidia. *Profiler User's Guide - Warp State*. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#warp-state.

[16] *See Through S.r.l.* URL: https://www.seethrough.one/.

[17] L. Fan, F. Zhang e H. Fan. *Brief review of image denoising techniques. Vis. Comput. Ind. Biomed.* Art 2, 7 (2019). URL: https://doi.org/10.1186/s42492-019-0016-7.