UNIVERSITÀ DI PISA

Computer Engineering

Computer Architecture

2D Convolutional Algorithm

Group Project Report

TEAM MEMBERS:
Federico Casu
Daniel Deiana

Academic Year: 2022/2023

Contents

Chapter 1

Introduzione

Il seguente lavoro nasce con lo scopo di studiare il comportamento dell'algoritmo della Convoluzione in 2 dimensioni analizzando le performance ottenute e considerando possbili ottimizzazioni. L'analisi è divisa in 2 parti:

- Analisi dell'algoritmo utilizzando il Multithreading della CPU.
- Analisi dell'algoritmo su una implementazione in CUDA utilizzando la GPU.

1.1 Background

La *Convoluzione 2D* è un algoritmo fondamentale in diversi ambiti: principalemente, essi riguardano la processazione ed il filtraggio di immagini e l'estrazione di feature da queste ultime nelle reti *Convolutional Neural Networks*. Date queste considerazioni abbiamo considerato interessante studiarne il comportamento in relazione al parallelismo ottenibile su CPU e GPU.

L'algoritmo consiste nel filtraggio di un immagine, rappresentata tramite una matrice di interi [1] applicando un kernel in movimento su di essa. Ogni elemento dell'immagine di output è calcolato come prodotto scalare fra elementi del kernel ed elementi della immagine di partenza. Questo rende l'algoritmo altamente parallellizabile: idealmente, si potrebbe assegnare un singolo elemento di uscita ad un thread, di fatto rendendo l'algoritmo ideale per un implementazione che sfrutta il parallelismo messo a disposizione dalle GPU.

[1] Consideriamo il caso di un solo canale

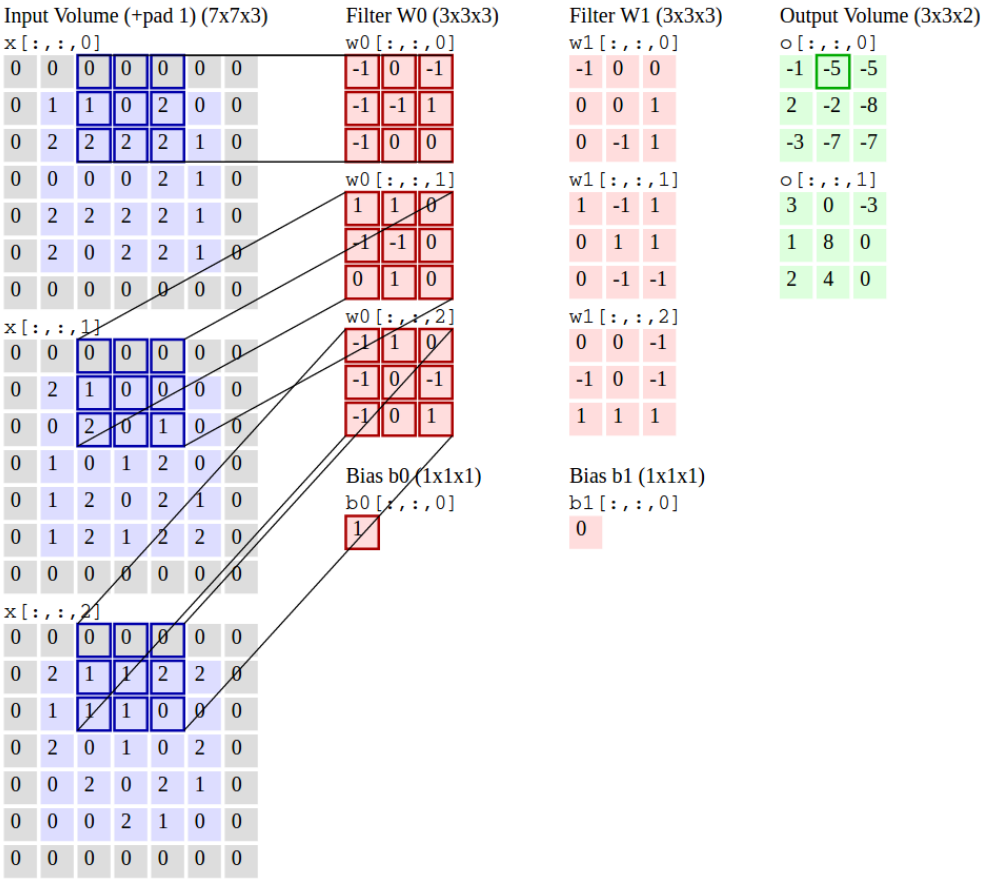

Figure 1.1: Esempio di applicazione della convoluzione.

Chapter 2

Analisi dell'implementazione su CPU

2.1 Implementazione dell'algoritmo

Il codice (sequenziale) che implementa l'algoritmo è il seguente:

```
void convolution(int* output, int* input, int* kernel, int input_rows,
                 int input_columns, int kernel_size)
{
    auto output_rows    = input_rows - kernel_size + 1;
    auto output_columns = input_columns - kernel_sise + 1;

    for (auto x = 0; x < output_rows; x++) {
        for (auto y = 0; y < output_columns; y++) {
            for (auto kx = 0; kx < kernel_size; kx++) {
                for (auto ky = 0; ky < kernel_size; kyy++) {
                    // Convolute here
                    output[x*output_columns + y] +=
                            input[(x + kx) * input_columns + (y + ky)] *
                            kernel[kx * kernel_size + ky];
                }
            }
        }
    }
}
```

Per quanto riguarda l'implementazione multithread, abbiamo deciso di suddividere il calcolo dell'immagine di output per righe. In particolare:

- Ad ogni thread è assegnato un id univoco (`thread_id`) che va da 0 a `n_thread`, dove quest'ultimo parametro corrisponde al numero di thread che il programma istanzia per svolgere la convoluzione.
- Un thread calcola gli elementi dell'immagine di output appartenenti alle righe

  $$[\texttt{thread\_id} * \texttt{task\_size},\ (\texttt{thread\_id} + 1) * \texttt{task\_size}]$$

  dove il parametro `task_size` è calcolato come `ceil`(`output_rows` / `n_thread`).

Il codice che si occupa della gestione dei thread è il seguente:

```
struct task {
    int *output, *input, *kernel;
    int start_row, end_row;
    int output_columns;
    int input_rows;
    int kernel_size;
};

void convolute(void*);

void convolution_pthread(int *output, int *input, int *kernel,
                         int input_rows, int input_columns,
                         int kernel_size, int n_thread)
{
    auto out_size_x = input_rows - kernel_size + 1;
    auto out_size_y = input_columns - kernel_size + 1;

    pthread_t thread_id[n_thread];
    struct task thread_task[n_thread];

    void (*convolute_ptr)(void *) = &convolute;

    auto task_size = (int)ceil(((float)out_size_x) / ((float)n_thread));

    for (auto i = 0; i < n_thread; i++)
    {
        thread_task[i].output         = output;
        thread_task[i].kernel         = kernel;
        thread_task[i].input          = input;
        thread_task[i].start_row      = i * task_size;

        thread_task[i].end_row = (out_size_x > (i + 1) * task_size) ?
                                 ((i + 1) * task_size) :
                                out_size_x;

        thread_task[i].output_columns = out_size_y;
        thread_task[i].input_rows     = input_rows;
        thread_task[i].kernel_size    = kernel_size;

        auto ret = pthread_create(&thread_id[i],
                                  NULL,
                                  (void *(*)(void *))convolute_ptr,
                                  (void *)&thread_task[i]);
        if (ret)
            exit(-1);
    }

    for (auto i = 0; i < n_thread; i++)
        pthread_join(thread_id[i], NULL);
}
```

```
void convolute(void *argument)
{
    struct task *__task = (struct task *)argument;

    auto output = __task->output;
    auto input  = __task->input;
    auto kernel = __task->kernel;
    auto start  = __task->start_row;
    auto end    = __task->end_row;

    auto out_size_y  = __task->output_columns;
    auto in_size_x   = __task->input_rows;
    auto kernel_size = __task->kernel_size;

    for (auto x = start; x < end; x++) {
        for (auto y = 0; y < out_size_y; y++) {
            for (auto kx = 0; kx < kernel_size; kx++) {
                for (auto ky = 0; ky < kernel_size; ky++)
                    // Convolute here.
                    output[x * out_size_y + y] +=
                            input[(x + kx) * in_size_x + (y + ky)] *
                            kernel[kx * kernel_size + ky];
            }
        }
    }
}
```

2.2 Analisi delle prestazioni

L'obiettivo della nostra analisi è studiare il **tempo di esecuzione** dell'implementazione sequenziale e dell'implementazione multithread. In particolare, abbiamo ritenuto opportuno studiare il tempo di esecuzione rispetto ad altre metriche (esempio: throughput) visto che questo algoritmo è di notevole importanza nell' ambito del machine learning. Nello specifico, uno dei problemi da prendere in considerazione durante lo sviluppo di un modello di machine learning è il costo della fase di *training*. Ridurre il tempo di esecuzione significa ridurre i costi della fase di training. Inoltre, ridurre i costi della fase training significa poter allenare il modello su dataset di dimensioni maggiori e, di conseguenza, ottenere un modello che gode di maggiore accurattezza rispetto ad un modello allenato su un dataset più piccolo.

Le due implementazioni saranno messe a confronto analizzando lo **speedup** ottenuto dalla versione multithread. In particolare, calcoliamo lo speedup ottenuto dalla versione multithread, eseguita con n threads, attraverso la seguente formula:

$$\texttt{Speedup}(n) = \frac{\texttt{Sequential Execution Time}}{\texttt{Multithread Execution Time}(n)}$$

Le simulazioni sono state eseguite su una macchina avente una CPU `AMD Ryzen 5 3500U` con le seguenti caratteristiche:

- Threads per core: 2
- Cores per socket: 4

- Caches:
  - L1 data cache: 128 KiB (4 istanze, 32 KiB per core)
  - L1 instruction cache: 256 KiB (4 istanze, 64 KiB per core)
  - L2 (unificata): 2 MiB (4 istanze, 512 KiB per core)
  - L3 (unificata): 4 Mib (1 istanza)

Il tempo di esecuzione dell'algoritmo è stato misurato come segue:

```
// Start measuring time of conv2D
auto begin_conv2D = std::chrono::high_resolution_clock::now();

convolution_pthread(image_f.raw_data, image.raw_data,
                    kernel.raw_data, image.rows,
                    image.columns, kernel_size, threads);

// Stop measuring time and calculate the execution time of conv2D
auto end_conv2D = std::chrono::high_resolution_clock::now();
auto execution_time = std::chrono::duration_cast<std::chrono::nanoseconds>
                        (end_conv2D - begin_conv2D).count() * 1e-9;
```

Come è possibile vedere dal codice, abbiamo utilizzato il `wall time`, piuttosto che il `CPU time`, perchè abbiamo voluto orientare la nostra analisi anche sulla valutazione del tempo impiegato nelle operazioni di scheduling dei thread. La scelta, quindi, è ricaduta sul wall time perchè, a differenza del CPU time, tale misura tiene conto anche del tempo in cui i thread sono sospesi nell'attesa di essere schedulati.

Il programma, nella versione presentata nella sezione precedente, ha ottenuto i risultati riportati in Figura 2.1 e in Figura 2.2.

Dallo studio del grafico che riporta il tempo medio di esecuzione al variare del numero di threads (Figura 2.1) siamo giunti alle seguenti conclusioni:

- All'aumentare del numero di threads, il tempo di esecuzione diminuisce fino a raggiungere un valore costante.
- L'algoritmo trae maggiori benefici all'aumentare della dimensione dell'immagine. In particolare, il fenomeno è maggiormente visibile nelle simulazioni compiute su immagini di dimensione $\geq$ `1024x1024`.
- Abbiamo ritenuto opportuno limitare l'analisi del tempo di esecuzione fino ad un numero di thread pari a 16. La motivazione di questa scelta risiede nel fatto che i risultati ottenuti nelle simulazioni effettuate con un numero di thread maggiore a 16 sono poco significativi. I grafici ottenuti sono riportati in Figura 2.3 ed in Figura 2.4.

I risultati, in termini di speedup, sono i seguenti:

- A parità di numero di thread utilizzati, lo speedup ottenuto aumenta all'aumentare della dimensione dell'immagine.
- Nelle simulazioni compiute su immagini di dimensioni piccole (`64x64`, `256x256` e `521x512`) possiamo notare che lo speedup raggiunge un picco massimo e poi diminuisce all'aumentare del numero di threads. Tale fenomeno può essere sintomo di un aumento dell'overhead introdotto dallo scheduling dei thread: a parità di dimensione dell'immagine, all'aumentare del numero di thread aumenta il tempo

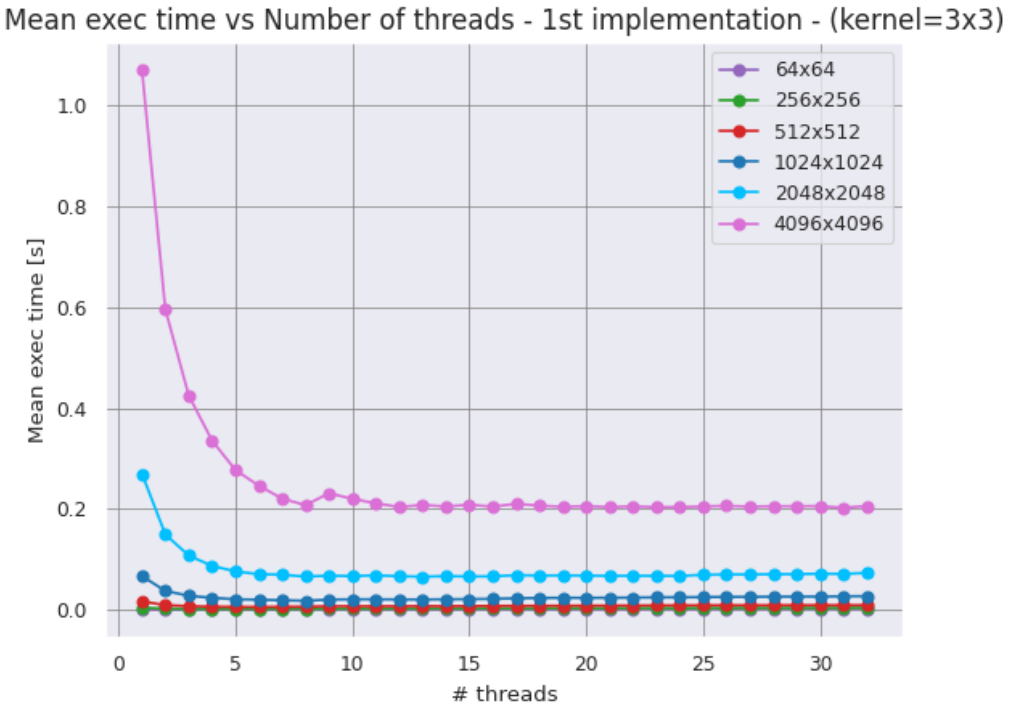

Figure 2.1: 1° versione - Tempo di esecuzione medio misurato su 100 simulazioni.

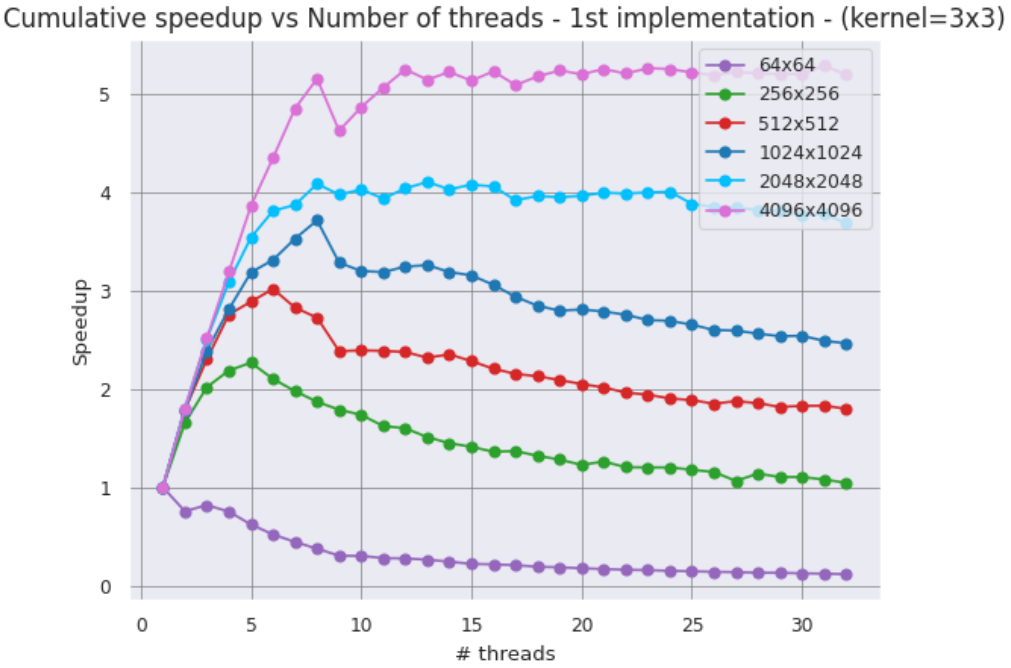

Figure 2.2: 1° versione - Speedup

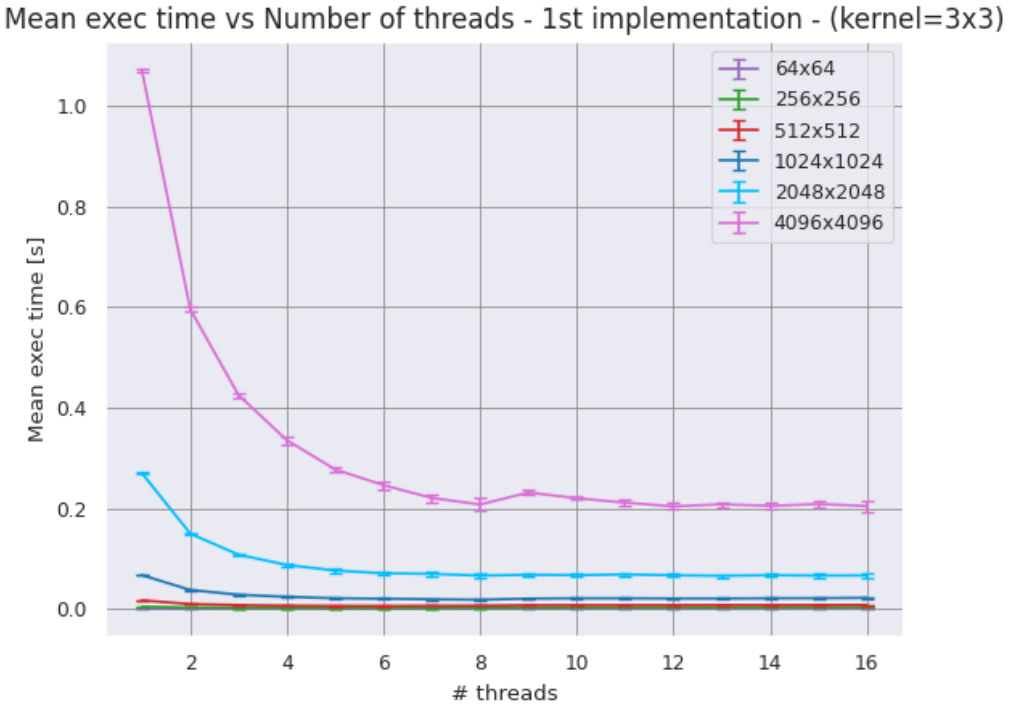

Figure 2.3: 1° versione, `n_thread` ≤ 16 - Tempo di esecuzione medio misurato su 100 simulazioni.

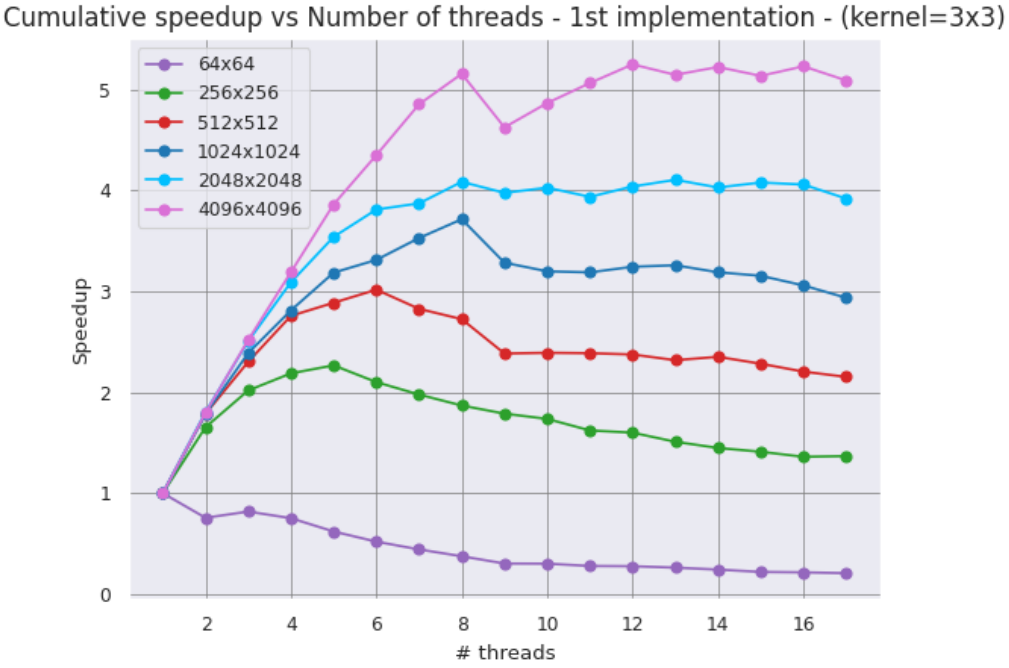

Figure 2.4: 1° versione, `n_thread` ≤ 16 - Speedup

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	7 994 955 698	7 994 955 698	Ir	
L1 Instr. Fetch Miss	7	7	I1mr	
LL Instr. Fetch Miss	7	7	ILmr	
Data Read Access	3 955 577 810	3 955 577 810	Dr	
L1 Data Read Miss	4 191 748	4 191 748	D1mr	
LL Data Read Miss	2 096 130	2 096 130	DLmr	
Data Write Access	217 894 986	217 894 986	Dw	
L1 Data Write Miss	4	4	D1mw	
LL Data Write Miss	4	4	DLmw	
L1 Miss Sum	4 191 759	4 191 759	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	2 096 141	2 096 141	LLm	= ILmr + DLmr + DLmw

Figure 2.5: 1° versione - Risultati ottenuti tramite il profiling della funzione `convolute(void*)` con lo strumento `Cachegrind`.

```
        // Fill output matrix: rows and columns are i and j respectively.
12 288  for (auto x = start; x < end; x++)
        {
50 290 696      for (auto y = 0; y < out_size_y; y++)
                {
                    // Kernel rows and columns are kx and ky respectively.
184 369 196         for (auto kx = 0; kx < kernel_size; kx++)
                    {
553 107 588             for (auto ky = 0; ky < kernel_size; ky++)
                            // Convolute here.
3 167 798 004               output[x * out_size_y + y] += (input[(x + kx) * in_size_x + (y + ky)] * kernel[kx * kernel_size + ky]);
                    }
                }
        }
```

Figure 2.6: 1° versione - Profiling della funzione `convolute(void*)` - Numero di accessi in lettura alla memoria (sul lato sinitro sono riportati gli accessi in memoria compiuti campionati dallo strumento di profiling).

speso dal kernel nelle operazioni di scheduling (quest'ultimo è tempo speso a compiere attività non utili ai fini dell'elaborazione dell'immagine) e, conseguentemente, aumenta il tempo di esecuzione.

Lo speedup massimo (circa 5.2) è stato ottenuto con immagini aventi dimensione pari a `4096x4096` nelle simulazioni svolte con 12 threads. Visto il risultato, abbiamo ritenuto oppurtuno indagare su eventuali problemi dati da un programma scritto in modo non ottimale.
L'analisi eseguita con il tool di profiling ha messo in evidenza i seguenti problemi:

1. In riferimento alla Figura 2.6, 80.01% degli accessi (in lettura) alla memoria compiuti dalla funzione `convolute(void*)` sono generati da `output[x * out_size_y +`

```
4 191 747   for (auto ky = 0; ky < kernel_size; ky++)
                // Convolute here.
                output[x * out_size_y + y] += (input[(x + kx) * in_size_x + (y + ky)] * kernel[kx * kernel_size + ky]);
```

Figure 2.7: 1° versione - Profiling della funzione `convolute(void*)` - Numero di miss in lettura sulla cache L1 dei dati (sul lato sinitro sono riportati le miss campionate dallo strumento di profiling).

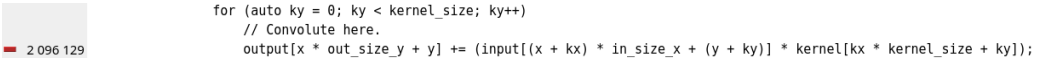

Figure 2.8: 1° versione - Profiling della funzione `convolute(void*)` - Numero di miss in lettura sulla cache L3 (sul lato sinitro sono riportati le miss campionate dallo strumento di profiling).

`y] += (input[(x + kx) * in_size_y + (y + ky)] * kernel[kx * kernel_size + ky])`.

2. In riferimento alla Figura 2.7, la quasi totalità delle **miss** in lettura sulla cache L1 dei dati sono causate dall'istruzione `output[x * out_size_y + y] += (input[(x + kx) * in_size_y + (y + ky)] * kernel[kx * kernel_size + ky])` (99.99%).

3. In riferimento alla Figura 2.8, la quasi totalità delle **miss** in lettura sulla cache L3 sono causate dall'istruzione `output[x * out_size_y + y] += (input[(x + kx) * in_size_y + (y + ky)] * kernel[kx * kernel_size + ky])` (99.99%).

2.2.1 Ottimizzazione (1): private variable

I risultati ottenuti grazie all'attività di profilazione ci guidano verso un ottimizzazione semplice ma efficace: introduciamo una variabile temporanea e locale (quindi non condivisa tra i thread) su cui memorizzare i risultati parziali. Il codice della funzione `convolute(void*)` è stato modificato come segue:

```
void convolute(void *argument)
{
    /*
     * Variable inizialization
     */

    int convolute = 0;

    for (auto x = start; x < end; x++) {
        for (auto y = 0; y < out_size_y; y++) {
            for (auto kx = 0; kx < kernel_size; kx++) {
                for (auto ky = 0; ky < kernel_size; ky++)
                    convolute += input[(x + kx) * in_size_x + (y + ky)] *
                                 kernel[kx * kernel_size + ky];
            }
            output[x * out_size_y + y] = convolute;
            convolute = 0;
        }
    }
}
```

L'introduzione della variabile temporanea `convolute` è giustificata dalle seguenti ragioni:

- Per ogni iterazione dei due cicli più interni (ovvero i cicli i cui indici sono `kx` e `ky`) l'elemento dell' immagine di output su cui la funzione lavora è sempre lo stesso. Sommare i risultati parziali all'elemento dell'immagine di output oppure assegnare il risultato alla fine dell'elaborazione non influisce sulla correttezza dell'output.

- Eliminiamo la possibilità che il fenomeno del *false sharing* possa degradare le prestazioni del programma.

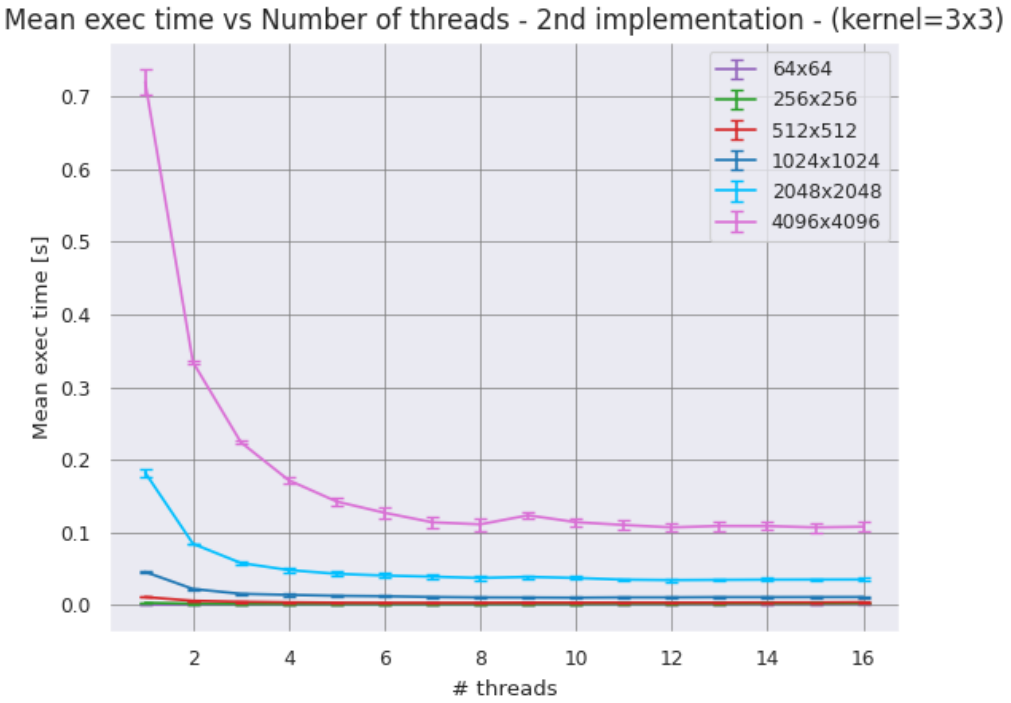

Figure 2.9: 2° versione - Tempo di esecuzione medio misurato su 100 simulazioni.

- In generale, eliminiamo qualsiasi problema che può scaturire dal fatto che la matrice `output` è condivisa tra tutti i thread (ad ogni iterazione dei due cicli più interni gli accessi sono compiuti sulla variabile `convolute` che è privata).

I risultati ottenuti dalla nuova implementazione sono riportati in Figura 2.9 e in Figura 2.10.
Dai risultati ottenuti possiamo trarre le seguenti conclusioni:

1. Il numero di accessi alla memoria è diminuito del 28.39% (2.832.601.798 accessi contro i 3.955.577.810 della prima versione).

2. Il numero di read miss sulla cache L1 dei dati è diminuito del 25% (3.144.200 read miss contro le 4.191.748 read miss della prima versione).

3. Il numero di read miss sulla cache L3 è diminuito del 49.95% (1.049.092 read miss contro le 2.096.130 read miss della prima versione).

4. Il numero totale[1] di miss sulla cache L3 è rimasto praticamente invariato perchè sono raddoppiate le miss in scrittura sulla cache L3. Questo fenomeno trova spiegazione nel fatto che la scrittura del risultato della convoluzione (`output[x * out_size_y + y] = convolute;`) è la prima ed unica volta a cui si accede all'elemento della matrice `output[x][y]`. Nella versione precedente l'elemento della matrice veniva prima acceduto in lettura e poi aggiornato con il risultato parziale: gli accessi che compivano una miss in lettura sull'ultimo livello ora sono diventati accessi in scrittura.

[1] LL miss = Instruction miss read + Data miss read + Data miss write

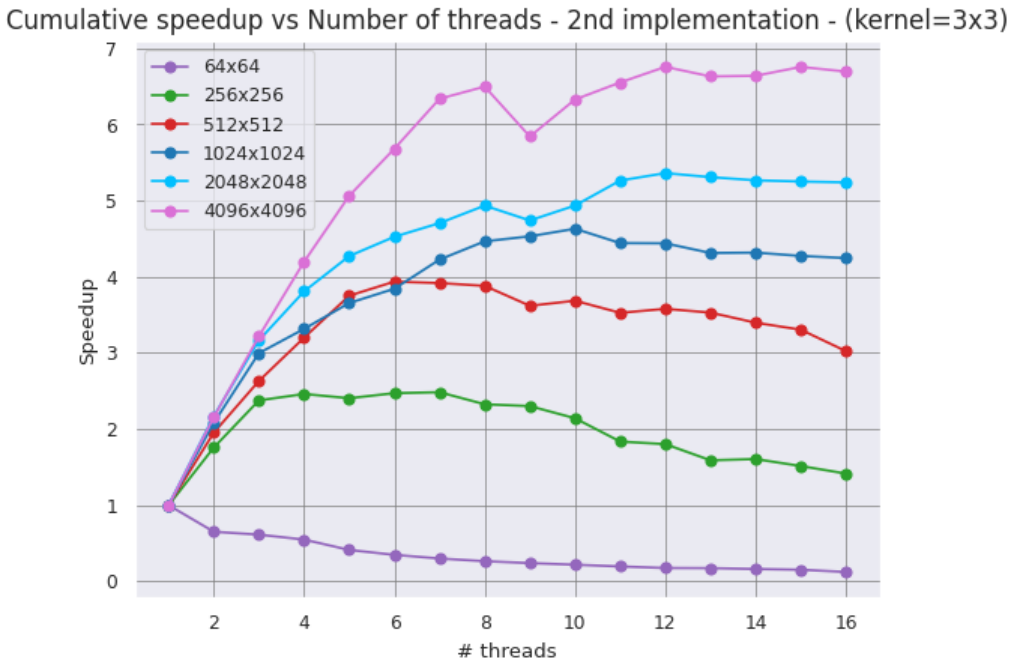

Figure 2.10: 2° versione - Speedup

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	5 179 135 252	5 179 135 252	Ir	
L1 Instr. Fetch Miss	7	7	I1mr	
LL Instr. Fetch Miss	7	7	ILmr	
Data Read Access	2 832 601 798	2 832 601 798	Dr	
L1 Data Read Miss	3 144 200	3 144 200	D1mr	
LL Data Read Miss	1 049 092	1 049 092	DLmr	
Data Write Access	100 569 136	100 569 136	Dw	
L1 Data Write Miss	1 047 558	1 047 558	D1mw	
LL Data Write Miss	1 047 558	1 047 558	DLmw	
L1 Miss Sum	4 191 765	4 191 765	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	2 096 657	2 096 657	LLm	= ILmr + DLmr + DLmw

Figure 2.11: 2° versione - Risultati ottenuti tramite il profiling della funzione `convolute(void*)` con lo strumento `Cachegrind`.

```
  12 288    for (auto x = start; x < end; x++)
            {
  50 290 696        for (auto y = 0; y < out_size_y; y++)
                    {
                        // Kernel rows and columns are kx and ky respectively.
 184 369 196            for (auto kx = 0; kx < kernel_size; kx++)
                        {
 553 107 588                for (auto ky = 0; ky < kernel_size; ky++)
                                // Convolute here.
1 961 017 812                   convolute += (input[(x + kx) * in_size_x + (y + ky)] * kernel[kx * kernel_size + ky]);
                        }
  83 804 180            output[x * out_size_y + y] = convolute;

                        convolute = 0;
                    }
            }
        4 }
```

Figure 2.12: 2° versione - Profiling della funzione `convolute(void*)` - Numero di accessi in lettura alla memoria (sul lato sinitro sono riportati gli accessi in memoria compiuti campionati dallo strumento di profiling).

```
        1   auto output = __task->output;
            auto input  = __task->input;
        1   auto kernel = __task->kernel;
            auto start  = __task->start_row;
            auto end    = __task->end_row;

    ...
                {
                    for (auto ky = 0; ky < kernel_size; ky++)
                        // Convolute here.
3 144 198               convolute += (input[(x + kx) * in_size_x + (y + ky)] * kernel[kx * kernel_size + ky]);
```

Figure 2.13: 2° versione - Profiling della funzione `convolute(void*)` - Numero di miss in lettura sulla cache L1 dei dati (sul lato sinitro sono riportati le miss campionate dallo strumento di profiling).

```
        1   auto output = __task->output;
            auto input  = __task->input;
        1   auto kernel = __task->kernel;
            auto start  = __task->start_row;
            auto end    = __task->end_row;

    ...
                {
                    for (auto ky = 0; ky < kernel_size; ky++)
                        // Convolute here.
1 049 090               convolute += (input[(x + kx) * in_size_x + (y + ky)] * kernel[kx * kernel_size + ky]);
```

Figure 2.14: 2° versione - Profiling della funzione `convolute(void*)` - Numero di miss in lettura sulla cache L3 (sul lato sinitro sono riportati le miss campionate dallo strumento di profiling).

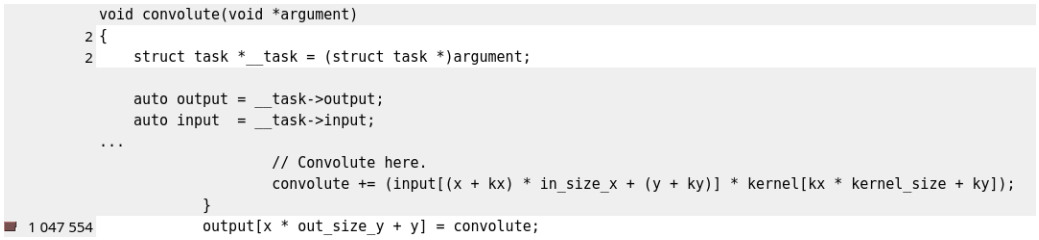

Figure 2.15: 2° versione - Profiling della funzione `convolute(void*)` - Numero di miss in scrittura sulla cache L3 (sul lato sinitro sono riportati le miss campionate dallo strumento di profiling).

L'introduzione della variabile privata `convolute`, in generale, ha permesso la diminuzione dei tempi di esecuzione: questa affermazione è dimostrata dal grafico in Figura 2.16. Il grafico riporta lo speedup relativo, calcolato come segue:

$$\texttt{Relative Speedup}(n) = \frac{1^{st}\ \texttt{version Execution Time}(n)}{2^{nd}\ \texttt{version Execution Time}(n)}$$

Seppur non è possibile individuare un trend, possiamo notare che lo speedup relativo è sempre maggiore di 1.4

2.2.2 Ottimizzazione (2): loop unrolling e variabili `register`

L'obiettivo delle ottimizzazioni introdotte è ridurre il numero di accessi in memoria della funzione `convolute(void*)`. Per raggiungere l'obiettivo ci siamo concentrati sull'average use case: filtro con dimensione pari a `3x3`, `5x5` e `7x7`.

Il codice della funzione è stato modificato come segue:

```
void convolute_opt_3x3(void *argument);
{
    struct task *__task = (struct task *)argument;

    register auto output = __task->output;
    register auto input = __task->input;
    register auto kernel = __task->kernel;
    auto start = __task->start_row;
    auto end = __task->end_row;
    auto out_size_y = __task->output_columns;
    register auto in_size_x = __task->input_rows;
    register auto kernel_size = __task->kernel_size;
    register int convolute = 0;

    for (register auto x = start; x < end; x++) {
        for (register auto y = 0; y < out_size_y; y++) {
            for (register auto kx = 0; kx < kernel_size; kx++) {
                convolute += input[(x + kx) * in_size_x + (y + 0)] *
                                    kernel[kx * kernel_size + 0];
                convolute += input[(x + kx) * in_size_x + (y + 1)] *
                                    kernel[kx * kernel_size + 1];
                convolute += input[(x + kx) * in_size_x + (y + 2)] *
                                    kernel[kx * kernel_size + 2];
```

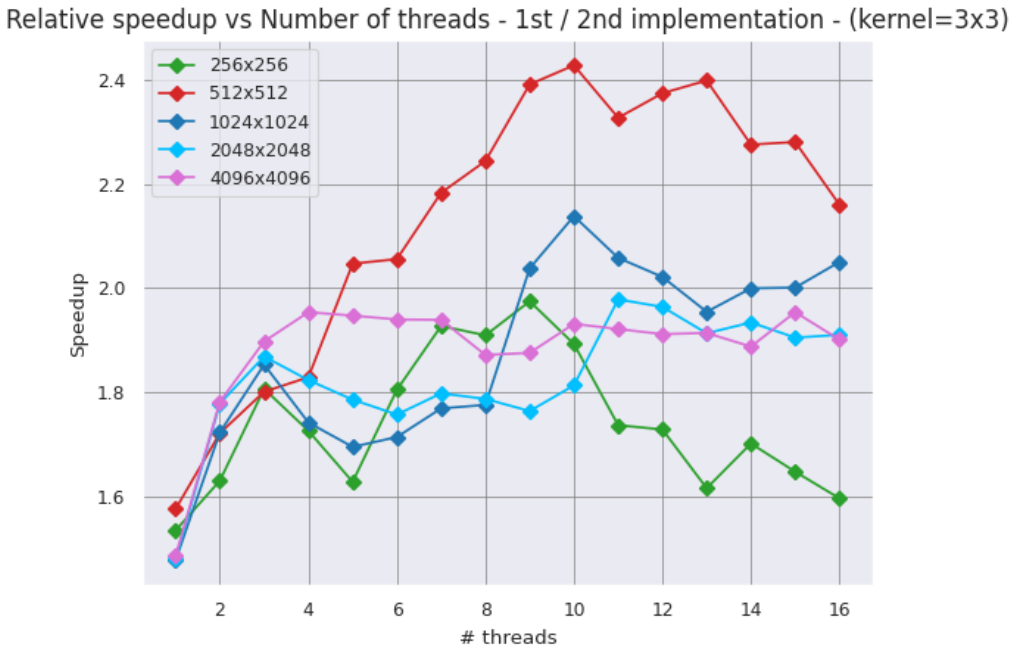

Figure 2.16: Speedup relativo - 1° versione vs 2° versione

```
            }
            output[x * out_size_y + y] = convolute;
            convolute = 0;
        }
    }
}

void convolution_pthread(int *output, int *input, int *kernel,
                         int input_rows, int input_columns,
                         int kernel_size, int n_thread)
{
    auto out_size_x = input_rows - kernel_size + 1;
    auto out_size_y = input_columns - kernel_size + 1;

    pthread_t thread_id[n_thread];
    struct task thread_task[n_thread];

    void (*convolute_ptr)(void *) = ((kernel_size == 3) ?
                                &convolute_opt_3x3 :
                                ((kernel_size == 5) ? &convolute_opt_5x5 :
                                ((kernel_size == 7) ? &convolute_opt_7x7 :
                                &convolute)));

    auto task_size = (int)ceil(((float)out_size_x) / ((float)n_thread));

    for (auto i = 0; i < n_thread; i++)
    {
```

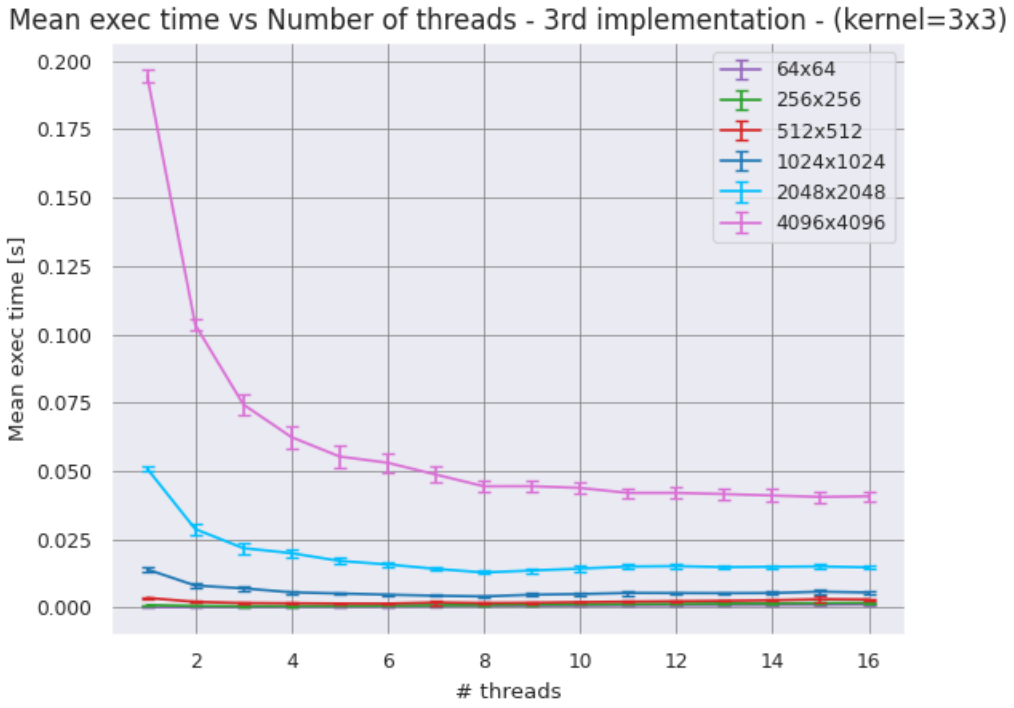

Figure 2.17: 3° versione - Tempo di esecuzione medio misurato su 100 simulazioni.

```
        thread_task[i].output         = output;
        thread_task[i].kernel         = kernel;
        thread_task[i].input          = input;
        thread_task[i].start_row      = i * task_size;
        thread_task[i].end_row        = (out_size_x > (i + 1) * task_size) ?
                                        ((i + 1) * task_size) :
                                        out_size_x;
        thread_task[i].output_columns = out_size_y;
        thread_task[i].input_rows     = input_rows;
        thread_task[i].kernel_size    = kernel_size;

        auto ret = pthread_create(&thread_id[i],
                                  NULL,
                                  (void *(*)(void *)) convolute_ptr,
                                  (void*) &thread_task[i]);
        if (ret) exit(-1);
    }

    for (auto i = 0; i < n_thread; i++)
        pthread_join(thread_id[i], NULL);
}
```

I risultati ottenuti dalla nuova implementazione sono riportati in Figura 2.17 e in Figura 2.18.
L'analisi compiuta mediante il tool di profiling (Figura 2.19) ci permette di affermare che l'introduzione delle ottimizzazioni ha fatto si che il numero di accessi in memoria sia

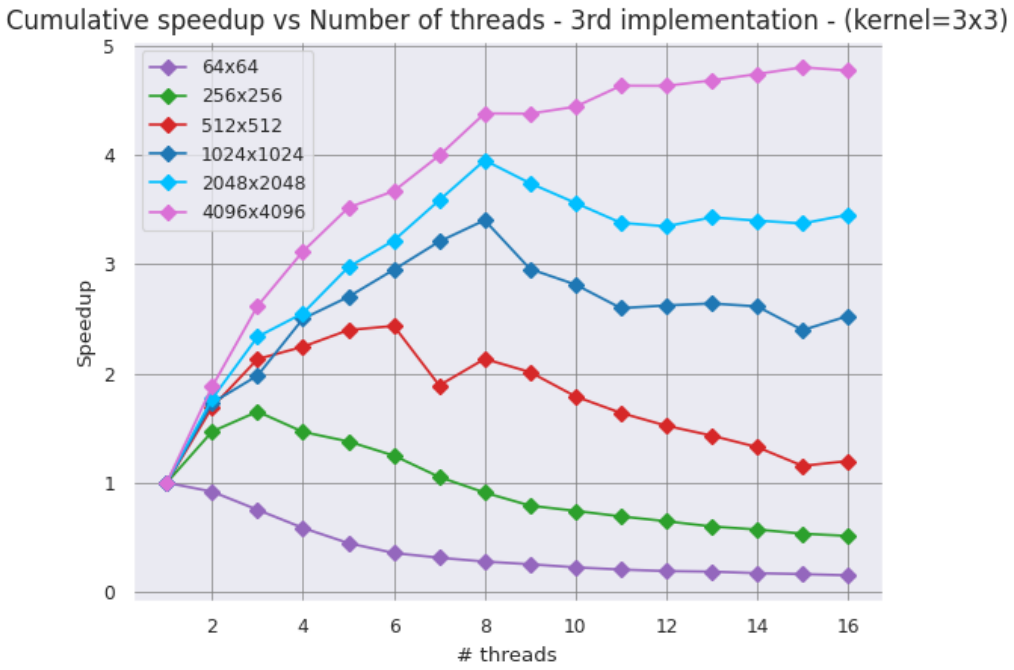

Figure 2.18: 3° versione - Speedup

diminuito. Di conseguenza anche il tempo di esecuzione è dimuito visto che gli accessi compiuti (in lettura) in memoria sono solamente quelli necessari a prelevare gli elementi dell'immagine di input e gli elementi del kernel.

Event Type	Incl.	Self	Short	Formula
Instruction Fetch	2 028 089 892	2 028 089 892	Ir	
L1 Instr. Fetch Miss	5	5	I1mr	
LL Instr. Fetch Miss	5	5	ILmr	
Data Read Access	335 224 944	335 224 944	Dr	
L1 Data Read Miss	3 144 204	3 144 204	D1mr	
LL Data Read Miss	1 049 096	1 049 096	DLmr	
Data Write Access	16 760 856	16 760 856	Dw	
L1 Data Write Miss	1 047 558	1 047 558	D1mw	
LL Data Write Miss	1 047 558	1 047 558	DLmw	
L1 Miss Sum	4 191 767	4 191 767	L1m	= I1mr + D1mr + D1mw
Last-level Miss Sum	2 096 659	2 096 659	LLm	= ILmr + DLmr + DLmw

Figure 2.19: 3° versione - Risultati ottenuti tramite il profiling della funzione `convolute(void*)` con lo strumento `Cachegrind`.

```
50 282 508          movl    (%rax), %edx    # %edx = input[(x + kx)*in_size_x + (y + 0)]
                    movl    %r13d, %eax     # %eax = kx
                    imull   %r15d, %eax     # %eax = kx*kernel_size
                    cltq
                ...
                    movq    %rsi, %rax          # %rax = &kernel[0]
                    addq    %rcx, %rax        # %rax = &kernel[kx*kernel_size + 0]
                    movq    %rax, %r9       # %r9 = &kernel[kx*kernel_size + 0]
50 282 508          movl    (%rax), %eax        # %eax = kernel[kx*kernel_size + 0]
                    imull   %edx, %eax        # %eax = input[(x + kx)*in_size_x + (y + 0)] * kernel[kx*kernel_size + 0]
                    addl    %eax, %r11d        # convolute += input[(x + kx)*in_size_x + (y + 0)] * kernel[kx*kernel_size + 0]
                    xor     %rcx, %rcx
                    addq    $1, %rcx
50 282 508          movl    (%r8,%rcx,4), %edx      # %edx = input[(x + kx)*in_size_x + (y + 1)]
50 282 508          movl    (%r9,%rcx,4), %eax      # %eax = kernel[kx*kernel_size + 1]
                    imull   %edx, %eax      # %eax = input[(x + kx)*in_size_x + (y + 1)] * kernel[kx*kernel_size + 1]
                    addl    %eax, %r11d     # convolute += input[(x + kx)*in_size_x + (y + 1)] * kernel[kx*kernel_size + 1]
                    addq    $1, %rcx
50 282 508          movl    (%r8,%rcx,4), %edx      # %edx = input[(x + kx)*in_size_x + (y + 2)]
50 282 508          movl    (%r9,%rcx,4), %eax      # %eax = kernel[kx*kernel_size + 2]
                    imull   %edx, %eax      # %eax = input[(x + kx)*in_size_x + (y + 2)] * kernel[kx*kernel_size + 2]
                    addl    %eax, %r11d     # convolute += input[(x + kx)*in_size_x + (y + 2)] * kernel[kx*kernel_size + 2]
                    addl    $1, %r13d       # kx++
```

Figure 2.20: 3° versione - Profiling della funzione `convolute(void*)` - Numero di accessi in lettura alla memoria (sul lato sinitro sono riportati gli accessi in memoria campionati dallo strumento di profiling). La somma degli accessi compiuti dalle istruzioni che eseguono l'accesso all'input, al kernel ed all'output coincide con la totalità degli accessi in memoria compiuti dalla funzione. In conclusione possiamo dire che gli accessi compiuti in memoria sono solo quelli necessari per l'elaborazione.

Chapter 3

Analisi dell'implementazione su GPU

3.1 Implementazione dell'algoritmo su GPU - Baseline

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

// baseline 2d convolution
// Only use odd kernel sizes
__global__ void convolution_2d(int *A,int *F, int p, int n, int *C) {

  int tmp = 0;

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int row = blockIdx.x*blockDim.x + tx;
  int col = blockIdx.y*blockDim.y + ty;

  int offset_k = p/2;

  int row_offset = row - offset_k;
  int col_offset = col - offset_k;

  for(int kx = 0; kx < p; kx++) {
    for(int ky = 0; ky < p; ky++) {
      if(row_offset + kx >= 0 && row_offset + kx < n) {
        if(col_offset + ky >= 0 && col_offset + ky < n)
          tmp += A[(row_offset + kx)*n + col_offset + ky]
                * F[kx * p + ky];
      }
    }
  }

  if(row < n && col < n){
```

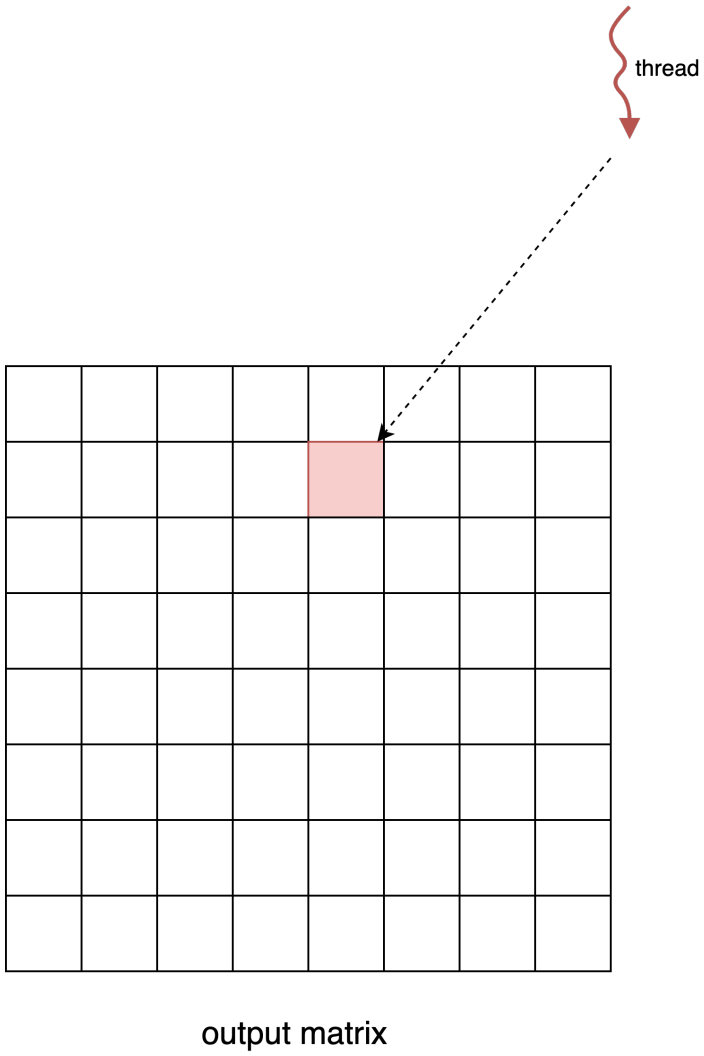

Figure 3.1: Calcolo elementi di output su GPU

```
            C[row*n + col] = tmp;
    }
}
```

L'implementazione su GPU ci permette di poter usare la caratteristica di queste macchine di avere un parallelismo sull'ordine di migliaia di thread, quindi la scelta piu naturale è stata quella di utilizzare per ogni elemento di uscita un thread che ne eseguisse il calcolo.

I parametri utilizzati per queste analisi sono:

- Dimensione del kernel: $7x7$.
- Dimensione dell'input: $4096x4096$.

Abbiamo utilizzato `CudaEvent` come metodo per misurare il tempo di esecuzione del kernel in quanto l'utilizzo di un metodo basato su eventi della CPU (i.e. `std::chrono` o le funzione della libreria `time.h`) non ci avrebbero permesso di misurare correttamente il tempo speso nell'elaborazione dal kernel. Il motivo risiede nel fatto che il kernel è eseguito

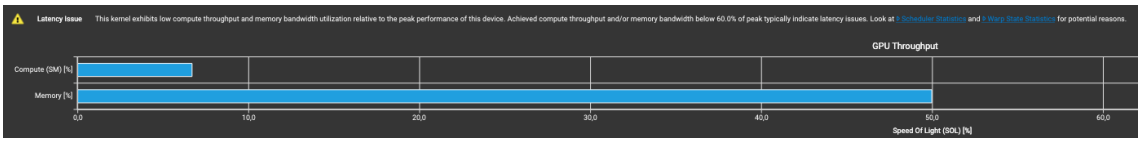

Figure 3.2: Report NVIDIA NSIGHT - *Throughput*

sul device (GPU) in modo asincrono all'esecuzione del main (quest'ultimo eseguito sulla CPU).

```
cudaEvent_t start, stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
convolution_2d<<<grid,block>>>(c_A, c_F, MASK_SIZE, N, c_C);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

3.1.1 Indici di performance

Gli indici di performance per questa analisi si concentrano sul *Throughput* in quanto è inutile fare delle comparazioni sui tempi di esecuzione rispetto al caso CPU e, inoltre, ci permette di capire quanto il codice scritto sfrutti le potenzialità teoriche messe a disposizione dalla macchina (facendo un confronto fra throughput massimo raggiungibile e quello ottenuto).

3.1.2 Profiling del codice baseline

Per poter analizzare le prestazioni del codice CUDA abbiamo usato il tool NVIDIA NSIGHT che offre uno strumento di profiling dove vengono messi a disposizione statistiche sull'utilizzazione delle risorse SM e di indici di performance quali throughput, execution time, etc..

Dopo una prima run del profile i risultati ottenuti sono stati i seguenti:

La statistica piu importante è quella che riguarda il *computation throughput* (espresso in GFLOPs). Questa metrica indica quale è l'utilizzo delle risorse computazionali svolte dalla GPU. Il valore ottenuto nella versione baseline è molto basso, **riusciamo ad ottenere solamente il 10% del throghput massimo ottenibile**. I motivi sono legati alle latenze introdotte dalle load dalla memoria come indicato dal profiler. Dopo questa prima analisi ci poniamo l' obiettivo di sfruttare elementi come la *memoria condivisa* per poter ridurre la latenza del dover prelevare gli operandi dalla memoria globale.

3.2 Implementazione dell'algoritmo su GPU - Memoria Condivisa

```
__global__ void convolution_2d_tiled(int *A, int  width, int height, int *F, int *P) {

  int const MASK_OFFSET = MASK_SIZE/2;

  __shared__ float i_shmem[TILE_SIZE + MASK_SIZE-1][TILE_SIZE + MASK_SIZE-1];

  // local position of thread inside the block
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // boundaries
  int o_row = blockIdx.x * TILE_SIZE + tx;
  int o_col = blockIdx.y * TILE_SIZE + ty;

  // position of the thread in the input space
  int i_row = o_row - MASK_OFFSET;
  int i_col = o_col - MASK_OFFSET;

  if ((i_row >= 0 && i_row < height)
  && (i_col >= 0 && i_col < width))
    i_shmem[tx][ty] = A[i_row * width + i_col];
  else
    i_shmem[tx][ty] = 0;

  // wait that all threads have loaded into shared memory
  __syncthreads();

  int tmp = 0;

    if (tx < TILE_SIZE && ty < TILE_SIZE) {
      // compute output element
      for(int i = 0; i < MASK_SIZE; i++) {
        for (int j = 0; j < MASK_SIZE; j++) {
          tmp += F[i*MASK_SIZE+j] * i_shmem[i+tx][j+ty];
        }
      }

      if(o_row < height && o_col < width){
              P[o_row * width + o_col] = tmp;
      }
    }
}
```

3.3 Tecnica del tiling

Un metodo per poter sfruttare le basse latenze della shared memory è quello di avere un buffer di memoria convidisa fra i thread di un blocco che contiene tutti gli elementi

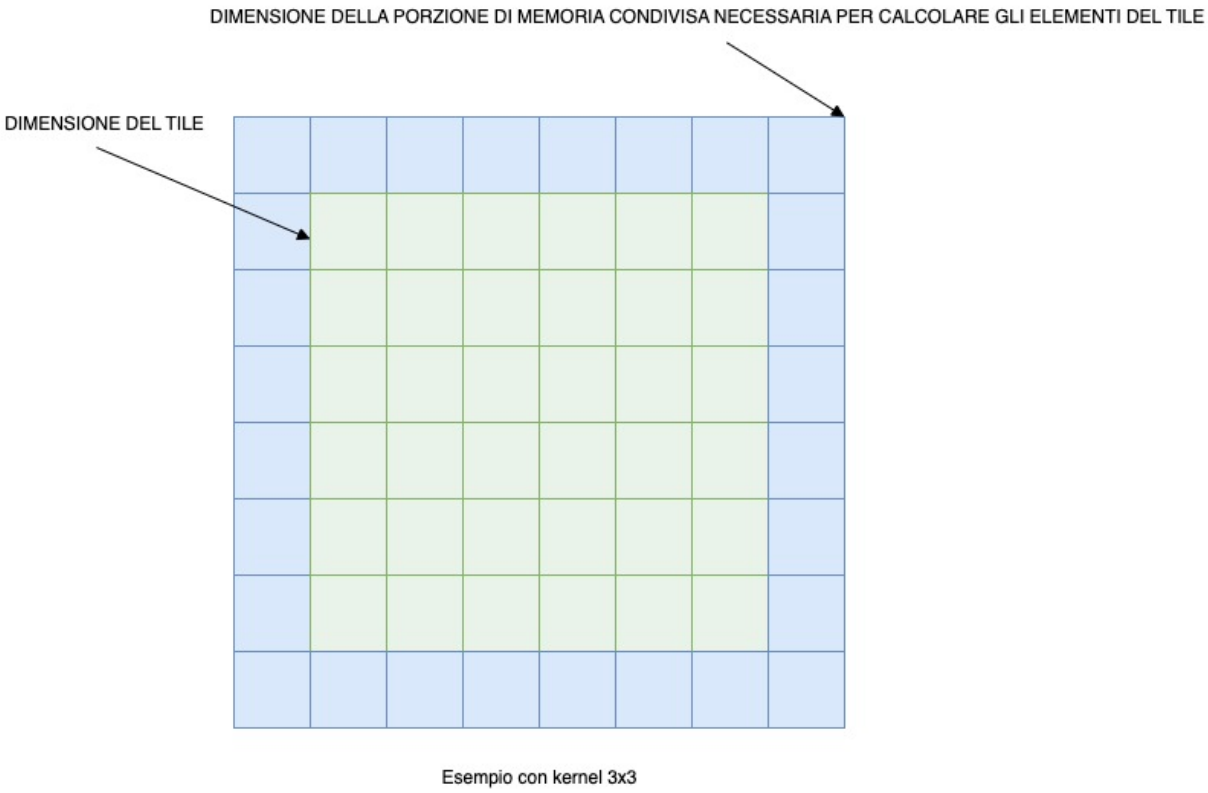

Figure 3.3: Struttura di un tile della matrice di output

di input necessari per calcolare un TILE [1]. Ne consegue che la dimensione del blocco di input debba essere maggiore in quanto sui bordi vi siano gli elementi del cosiddetto halo, ovvero quegli elementi necessari per poter calcolare gli elementi ai bordi del blocco.

Quindi avremo che in un blocco i thread al bordo del tile effettuano solamente il caricamento in memoria condivisa di un elemento di input, mentre i thread che effettivamente fanno parte del tile (zona evidenziata in verde nella immagine precedente) oltre a fare il caricamento si occuperanno anche di eseguire il calcolo. Nel codice di questa versione tiled quindi:

1. Si caricano tutti gli elementi in memoria convdivisa (utilizzando la `__syncthreads()` per poter fermare l'esecuzione fin quando tutti gli elementi della memoria condivisa sono stati inizializzati ed evitare che un thread cominci ad effettuare calcoli su input non validi).

2. I thread appartenenti al TILE calcolano gli elementi di output.

Questa tecnica ci ha permesso di poter riutilizzare elementi precedentemente caricati in shared memory e poter sfruttare delle basse latenze di quest'ultima, al posto di dover caricare tutti gli elementi dalla global memory ogni volta che un singolo thread dovesse calcolare un elemento di uscita.

I risultati del profiling su quest'ultima versione confermano quanto ci si aspettava sulle prestazioni del programma, il *Throughput* è salito fino a raggiungere l'89% del valore

[1] Porzione della matrice di output associata ad un blocco

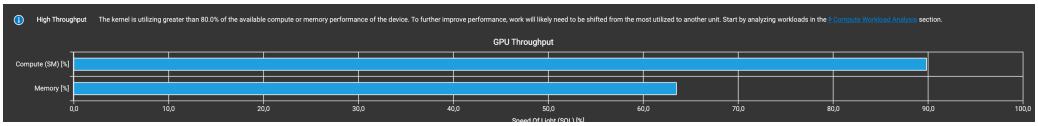

Figure 3.4: Struttura di un tile della matrice di output

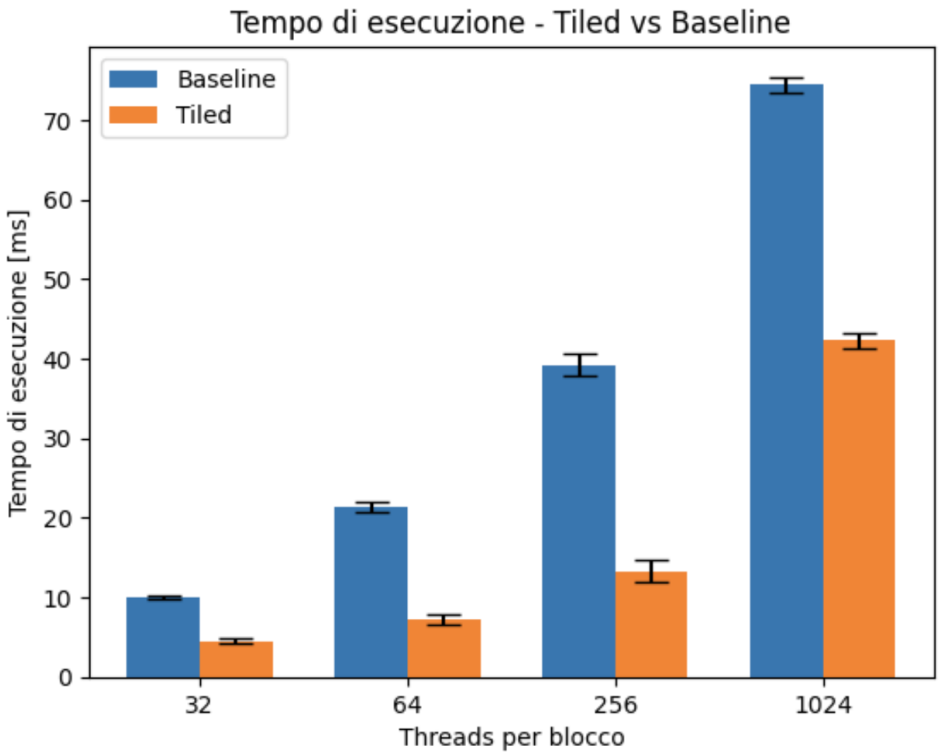

Figure 3.5: Confronto tempi di esecuzione al variare dei threads per blocco.

teorico ottenibile dalla GPU, indice di quanto lo sfruttare la memoria in modo ottimale possa sbloccare le potenzialità offerte dalla macchina.

3.3.1 Confronto tempi di esecuzione

Dopo le analisi del throghput abbiamo confrontato i tempi di esecuzione delle 2 versioni al variare della dimensione del blocco. Possiamo notare che la versione tiled gode di tempi di esecuzione minori e risulta essere almeno 2x più veloce di quella di partenza, la differenza maggiore in termini di speedup è quella relativa a blocchi di dimensione di 32 thread per blocco dove la tiled raggiunge il 2.8x rispetto alla baseline. Possiamo notare anche come i tempi di esecuzione salgano aumentando il numero di thread per blocco e di conseguenza il numero di blocchi.

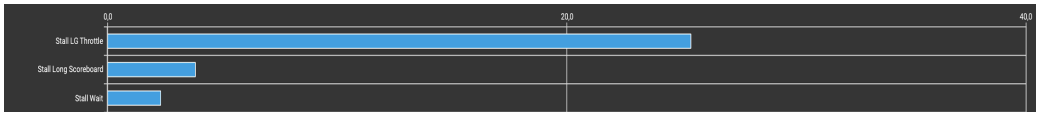

Figure 3.6: Statistiche sugli stalli della versione base.

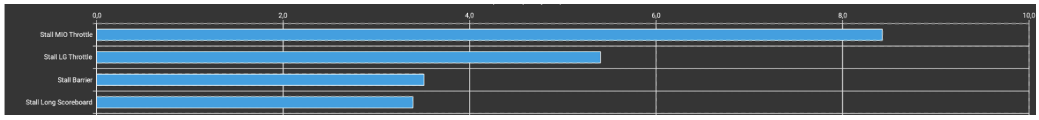

Figure 3.7: Statistiche sugli stalli della versione tiled.

3.3.2 Perchè all'aumentare del numero di thread per blocco le prestazioni peggiorano

Come evidenziato aumentando il numero di threads per blocco le prestazioni peggiorano. Questo è dovuto principalmente alla **Concorrenza sulla memoria**, nel caso della versione di base questo tipo di conflitto lo abbiamo su una memoria lenta come quella globale, nella versione tiled invece il fenomeno è sempre presente ma è mitigato dalle latenze basse della shared memory, in seguito alcuni dati che confermano questa ipotesi.

Possiamo notare come nella versione tiled (Figura 3.7) gli stalli di un warp su operazioni di load della memoria globale (in Figura 3.7 sono rappresentati dalla voce `LG Throttle`) siano diminuti di 5x volte rispetto alla versione di base (**??**), vengono introdotti nuovi stalli come quello sulla sincronizzazione e sulla memoria condivisa ma il loro effetto resta molto meno pesante di quelli sulla global memory, questo ci mostra come la shared memory riesce a mitigare le latenze sulla memoria globale.