UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di laurea in Ingegneria Elettronica

“Classificazione di immagini tramite tecnica di Monte Carlo su GPU NVIDIA 1080 GTX”

“Classification of images by Monte Carlo technique on NVIDIA 1080 GTX GPU”

Relatore
Prof. Claudio Turchetti

Laureando
Giacomo Turchi

Correlatore
Dott.ssa Laura Falaschetti

Anno Accademico 2018/2019

INDICE

INTRODUZIONE

Negli ultimi anni si è assistito ad un forte sviluppo di applicazioni che fanno uso di tecniche di apprendimento automatico. L'uso di alcune di queste applicazioni è entrato a far parte della nostra vita quotidiana senza che in realtà ce ne rendessimo conto. Si pensi ad esempio alla fruizione di contenuti internet per mezzo di motori di ricerca, digitando una o più parole chiave questi restituiscono liste di risultati che sono il prodotto dell'implementazione di algoritmi di Machine Learning. Sulla base della ricerca effettuata, vengono fornite informazioni ritenute attinenti come conseguenza dell'analisi di schemi, modelli e strutture dati. Anche la realizzazione di filtri anti-spam delle e-mail sono basati su sistemi di apprendimento automatico che imparano ad intercettare messaggi di posta elettronica sospetti o fraudolenti e si occupano magari di eliminarli prima che vengano distribuiti sulle caselle personali degli utenti. Sistemi simili ma più sofisticati vengono impiegati per la prevenzione di frodi e di furti di dati o di identità. Applicazioni interessanti riguardano anche il settore della ricerca scientifica in campo medico. In questo ambito gli algoritmi imparano a fare previsioni sempre più accurate per la diagnosi di tumori e altre patologie in modo tempestivo, fornendo uno strumento valido a disposizione del personale medico. Suscitano attenzione anche applicazioni legate al riconoscimento vocale o all'identificazione della scrittura manuale nonché quelle relative allo sviluppo dell'auto a guida autonoma che proprio attraverso il Machine Learning imparano a riconoscere l'ambiente circostante e ad adattare la guida del veicolo alle specifiche situazioni che si presentano.

L'apprendimento automatico è una branca dell'intelligenza artificiale che permette ai computer di "imparare dall'esperienza". Lo scopo del Machine Learning è quello di emulare in qualche modo quei processi, insiti naturalmente nell'uomo, di apprendimento, riconoscimento e capacità decisionale, derivanti da un'esperienza. La definizione forse più nota di questo fenomeno è quella formulata da Tom Michael Mitchell, direttore del dipartimento di Machine Learning della Carnegie Mellon University:

"*Si dice che un programma apprende dall'esperienza E con riferimento ad alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E*."

Dal punto di vista informatico un software che possiamo definire tradizionale è caratterizzato da istruzioni eseguite dal calcolatore per svolgere determinati compiti. La logica definita dal programmatore deve essere in grado di prevedere tutti i possibili casi per poter svolgere una determinata attività e il calcolatore

esegue semplicemente ciò che gli viene impartito. L'approccio utilizzato nell'ambito del Machine Learning è basato sull'utilizzo di un set di dati forniti in input alla macchina. Sulla base di questi dati l'algoritmo sviluppa una propria logica durante una fase di allenamento (training) nota anche come fase di apprendimento. Una volta che il modello è stato addestrato può quindi svolgere le attività richieste al fine di raggiungere un obiettivo fissato. Gli algoritmi migliorano in modo adattivo le loro prestazioni all'aumentare del numero di campioni disponibili per il processo di addestramento facendo "esperienza", acquisendo in questo modo quella componente di "intelligenza" che permetta di assolvere attività mai affrontate in precedenza.

Possiamo riscontrare essenzialmente due tipologie di apprendimento automatico:
-apprendimento supervisionato (supervised learning);
-apprendimento non supervisionato (unsupervised learning).
Un algoritmo di apprendimento supervisionato è caratterizzato da un modello cui vengono forniti dei dati in ingresso di cui sono note le corrispondenti risposte in uscita. Il modello proposto viene addestrato per generare previsioni ragionevoli in risposta a nuovi dati forniti come input alla macchina. Nell'apprendimento senza supervisione i dati in ingresso vengono forniti senza nessuna informazione circa le loro caratteristiche ovvero viene fornito un set di dati alla macchina costituiti da dati di input senza risposte etichettate. In questo caso il computer deve riuscire a trovare, o meglio riconoscere, una struttura logica intrinseca nei dati fatta di schemi nascosti.

L'apprendimento supervisionato utilizza tecniche di classificazione e regressione per sviluppare modelli predittivi. La classificazione è un processo che prevede risposte discrete in uscita. I modelli di classificazione generano una corrispondenza dei dati in input alla macchina con due o più classi in uscita, i dati in input vengono cioè classificati in categorie discrete in uscita. Un esempio di classificazione è il filtraggio antispam per cui occorre stabilire se una e-mail è autentica o da considerarsi spam. Le tecniche di regressione prevedono risposte continue in uscita in corrispondenza di un set di dati in ingresso. In questo caso quindi la natura della risposta è un numero reale come ad esempio le variazioni di temperatura in corrispondenza di una serie di parametri forniti in ingresso alla macchina per un certo processo.
La tecnica di apprendimento senza supervisione più comune è il clustering. Questa viene utilizzata per trovare schemi o raggruppamenti nascosti nei dati forniti senza alcuna etichetta al calcolatore. In questo caso gli ingressi vengono suddivisi in gruppi che non sono noti a priori. Gli algoritmi di clustering si basano su misure relative alla somiglianza tra gli elementi che in genere è concepita in

termini di distanza, definita in vari modi, in uno spazio multidimensionale. Gli output vengono quindi raggruppati in base alla loro distanza reciproca in vari cluster o gruppi [2].

Le prestazioni degli algoritmi di Machine Learning sono legate in parte alla quantità di dati forniti come input alla macchina. Una richiesta computazionale elevata può essere soddisfatta dall'utilizzo di sistemi di elaborazione ad alte prestazioni HPC (high-performance computing).
Verso la fine degli anni '80 la diffusione di sistemi operativi basati su interfaccia grafica ha creato un mercato per gli acceleratori grafici 2D. Negli anni successivi questo tipo di hardware specializzato è diventato di uso comune anche grazie allo sviluppo di applicazioni consumer che sfruttavano grafica 3D. Il rilascio di GPU che possedevano pipeline programmabili ha creato la possibilità per i ricercatori di utilizzare questo tipo di hardware per scopi diversi da quelli del tradizionale rendering grafico. Vi era la possibilità di sfruttare le GPU esplorando il calcolo per scopi generici, che tradizionalmente veniva eseguito su CPU, ma con una capacità computazionale disponibile nettamente superiore. Il tentativo di eseguire calcoli arbitrari su GPU era comunque soggetto ai vincoli della programmazione all'interno dell'API grafica, la GPU veniva indotta ad eseguire tali calcoli facendo apparire queste attività come se fossero un rendering standard. Sussistevano però gravi limitazioni a questo approccio, interagire con la GPU richiedeva la conoscenza di librerie grafiche quali OpenGL e DirectX, sfruttare la potenza di calcolo della GPU imponeva ai ricercatori di scrivere i calcoli in speciali linguaggi di programmazione grafici. Questo unito al fatto che non era possibile eseguire calcoli in virgola mobile ha costituito un ostacolo troppo grande per un'ampia diffusione. Nel 2006 NVIDIA presentò la prima scheda grafica costruita con architettura CUDA. Questa architettura includeva nuovi componenti progettati appositamente per il GPU computing allo scopo di superare le limitazioni che caratterizzavano le precedenti generazioni di processori grafici. Le innovazioni introdotte a livello hardware permettevano di eseguire il calcolo in virgola mobile per scopi generici, l'accesso arbitrario in lettura e scrittura alla memoria del dispositivo oltre alla possibilità di utilizzare una cache, denominata memoria condivisa, gestita direttamente via software. Per non dover mascherare calcoli generici come problemi grafici è stato introdotto un nuovo linguaggio di programmazione, CUDA C. Questo costituisce un'estensione del linguaggio C. Tramite l'utilizzo di un insieme relativamente piccolo di parole chiave si è reso possibile sfruttare le caratteristiche speciali introdotte con l'architettura CUDA. Questo ha facilitato molto la possibilità di sfruttare il calcolo per scopi generici da parte di un ampio numero di sviluppatori, permettendone un'ampia diffusione [3].

Un sistema di calcolo eterogeneo è costituito da architetture differenti sulle quali è possibile eseguire calcoli per un utilizzo detto "general purpose". CPU e GPU presentano un hardware strutturato in modo differente ottimizzato per assolvere i compiti per cui è stato pensato. Il passaggio da un sistema di calcolo omogeneo, che utilizza uno o più processori della stessa architettura, ad uno eterogeneo permette di ottenere un incremento delle prestazioni per l'esecuzione di un certo tipo di applicazioni. Ciascuna architettura è più adatta ad eseguire determinate operazioni. La CPU è ottimizzata per carichi di lavoro dinamici caratterizzati da brevi sequenze di operazioni computazionali e attività ad alta intensità di controllo. La GPU è adatta invece a svolgere attività ad alta intensità di calcolo di dati in parallelo. Problemi che presentano una piccola dimensione dei dati, una logica di controllo sofisticata o un basso grado di parallelismo applicabile sono eseguiti in maniera ottimale sulla CPU. Se il problema invece prevede l'elaborazione di una grande quantità di dati e c'è la possibilità di eseguire questi calcoli in maniera indipendente in parallelo allora l'approccio su GPU produce risultati migliori [4].

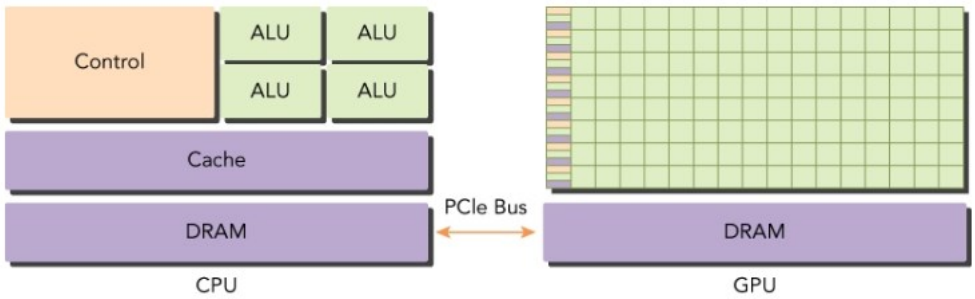

Figura 1 - Architettura di un sistema eterogeneo CPU-GPU [4]

L'attività di esecuzione di un programma, nell'ambito informatico, viene comunemente indicata col termine processo. Questo rappresenta un compito che il processore deve portare a termine su richiesta dell'utente. Un processo può essere suddiviso in più sotto processi (o istanze) denominati thread di esecuzione. Un'architettura CPU multithreading con più core di elaborazione è in grado di eseguire thread differenti a divisione di tempo su ogni core in maniera sequenziale e diversi thread effettivamente eseguiti allo stesso tempo in parallelo sui vari core fisici. Le CPU moderne possono essere caratterizzate anche da 64 core essendo quindi in grado di eseguire istruzioni relative a 64 thread contemporaneamente, 128 a livello logico se la CPU supporta il multithreading. Una GPU come quella presa in esame per questo lavoro è in grado di supportare migliaia di thread effettivamente attivi in parallelo essendo costituita da un numero molto superiore di core fisici che sono in grado di eseguire calcoli allo stesso tempo.

Nel caso di applicazioni come quelle relative al Machine Learning quantità considerevoli di dati sono poste come input alla macchina. La possibilità di assegnare tali dati a thread differenti, per effettuare calcoli indipendenti in parallelo su di essi, rende l'architettura GPU particolarmente adatta a svolgere tale compito. L'elaborazione degli stessi dati effettuata su CPU perde di efficacia dal momento che gran parte dell'esecuzione avviene in maniera sequenziale. Il lavoro proposto in questo elaborato prende come riferimento l'articolo "A GPU Parallel Algorithm for Non Parametric Tensor Learning" [1] realizzato dal Professor Claudio Turchetti in collaborazione con la Dottoressa Laura Falaschetti. Nel paper è stato sviluppato un modello matematico che implementa due algoritmi, uno di classificazione e l'altro di regressione. Tali algoritmi sfruttano la tecnica di Monte Carlo, per gestire direttamente un tensore di dati senza vettorizzazione degli stessi. Sono quindi stati eseguiti due esperimenti, uno relativo all'algoritmo di classificazione e l'altro a quello di regressione, utilizzando due data set differenti. Il fine era quello di dimostrare la possibilità di ottenere prestazioni migliori, in termini di runtime richiesto, applicando un modello MCT (Monte Carlo Tensor) piuttosto che un modello MCV (Monte Carlo Vector). Per mostrare la funzionalità della versione parallela tramite approccio tensoriale è stata realizzata un'implementazione scritta in CUDA C eseguita su GPU NVIDIA. Le prestazioni del codice proposto sono state testate su una macchina NVIDIA 1080 GTX con compilatore CUDA versione 8. Per fare un confronto tra le performance ottenute dagli approcci vettoriale e tensoriale al problema è stata sviluppata una versione del codice nel linguaggio di programmazione C ed eseguita su CPU. Il processore utilizzato a tal fine è una CPU Intel Core i7 6800K caratterizzata da una velocità di clock di 3,4 GHz e una dimensione della cache di 15 MB. Dai test effettuati si è potuto evincere che l'implementazione su GPU sovraperforma quella su CPU in termini di tempi di esecuzione del codice di un fattore 5. In questo elaborato si è preso in considerazione l'esperimento di classificazione di immagini proposto dal paper.

L'obiettivo che ci si pone è quello di ottimizzare il codice CUDA che permette di velocizzare il calcolo tramite l'impiego della GPU NVIDIA 1080 GTX. Si analizzerà la possibilità di ottenere prestazioni ancora migliori, in termini di tempi di esecuzione dell'algoritmo, dall'accelerazione del calcolo parallelo offerto dalla GPU stessa.

La trattazione si sviluppa come segue:

Nel primo capitolo vengono presentati alcuni concetti inerenti al modello di programmazione CUDA:
- Le modalità con cui viene allocata la memoria sul dispositivo sono mostrate nel primo paragrafo;
- Nel secondo paragrafo si spiega il concetto di kernel CUDA, la porzione di codice effettivamente eseguita su GPU;
- Il terzo paragrafo tratta i concetti di compilazione, come è possibile eseguire il codice e gli strumenti utilizzati per valutarne le prestazioni.

Il secondo capitolo riguarda la parte relativa all'implementazione del metodo Monte Carlo su GPU, questa parte si articola in quattro paragrafi:
- Nel primo paragrafo si espone la versione del codice utilizzata come riferimento, le correzioni apportate allo stesso per l'eliminazione di alcuni bug e la definizione delle prestazioni espresse dal codice per poterle poi confrontare con quelle ottenute dalle implementazioni del codice che verranno realizzate;
- Il concetto di parallelismo a livello logico è trattato nel secondo paragrafo, viene presentata una implementazione del codice per poter eseguire un numero maggiore di calcoli a livello logico sulla GPU;
- Il paragrafo tre di questo capitolo introduce le caratteristiche hardware del dispositivo GPU utilizzato per effettuare i test, sono introdotti anche alcuni concetti circa il modello di esecuzione CUDA e la gestione della memoria globale del dispositivo, assumere una visione complessiva dell'hardware è opportuno per sfruttare la capacità computazionale della GPU;
- L'ultimo paragrafo del capitolo due introduce il concetto di gerarchia dei thread la cui applicazione si concretizza in tre implementazioni caratterizzate da strutture differenti con cui sono organizzati i thread di esecuzione, queste implementazioni sviluppano prestazioni che saranno analizzate.

Le conclusioni riassumeranno brevemente il lavoro svolto e i risultati che sono stati raggiunti. Al termine dell'elaborato è presenta la sezione "Appendice" nella quale sono riportati il codice di riferimento in versione integrale, porzioni di codice relative ai kernel delle varie implementazioni e le relative sezioni di codice host per il lancio dei kernel stessi.

1. CONCETTI BASE DELLA PROGRAMAZIONE CUDA

CUDA consente di eseguire applicazioni su sistemi di elaborazione eterogenei costituiti da CPU integrate da GPU NVIDIA. Tale tipo di sistemi è caratterizzato da un host e da un device. L'host è costituito dalla CPU e dalla sua memoria (host memory), il device comprende la GPU e la sua memoria (device memory). Host e device sono caratterizzati ciascuno dal proprio spazio di memoria al quale CPU e GPU possono accedere rispettivamente per elaborare i propri dati. Il linguaggio CUDA C è contraddistinto da una piccola serie di estensioni al linguaggio C. Il codice che viene eseguito sulla GPU viene chiamato kernel ed è scritto usando CUDA C. Tale codice è lanciato dal codice host scritto in C o C++. Tutto il codice può essere inserito in un unico file di sorgente e può essere compilato tramite NVIDIA C Compiler (nvcc). Questo genera poi il codice eseguibile sia per l'host che per il device. Un tipico flusso di elaborazione di un programma CUDA è caratterizzato dalla copia dei dati dalla memoria dell'host alla memoria del device, dall'invocazione del kernel per operare sui dati memorizzati nella memoria del device e infine dalla copia dei dati elaborati dalla GPU dalla memoria del device stesso alla memoria dell'host.

1.1 Allocazione della memoria

Solamente i dati contenuti nella memoria del device possono essere elaborati dai kernel CUDA. Per tale motivo il runtime CUDA fornisce delle funzioni di allocazione della memoria per allocare la memoria del dispositivo, trasferire i dati tra la memoria host e quella del device e rilasciare la memoria lato device una volta terminata l'esecuzione del kernel. Queste funzioni sono del tutto analoghe a quelle definite per il C standard ma caratterizzate da una sintassi leggermente differente.

Funzioni C standard	Funzioni CUDA C
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Per allocare memoria sul device viene utilizzata la funzione cudaMalloc, questa presenta la seguente sintassi:

*cudaMalloc (void** puntatoreDevice, size_t dimensione)*

Un intervallo lineare di memoria del dispositivo viene allocato da cudaMalloc con la dimensione specificata in byte e la memoria allocata viene restituita tramite il puntatore puntatoreDevice. La funzione cudaMemcpy viene utilizzata per il trasferimento dei dati tra l'host e il device. La sintassi per questa funzione è specificata a seguito:

*cudaMemcpy (void** destinazione, const void** sorgente, size_t dimensione, cudaMemcpyKind tipo)*

La copia dei byte specificati dall'area di memoria di origine all'area di memoria di destinazione viene realizzata tramite la funzione cudaMemcpy, con la direzione specificata dal tipo di trasferimento. Il tipo può assumere quattro diverse connotazioni a seconda della direzione di trasferimento dei dati: cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost e cudaMamcpyDeviceToDevice.
Osserviamo anche che questa funzione mostra un comportamento sincrono ovvero l'host si blocca fino a quando cudaMemcpy non termina e il trasferimento è completato.

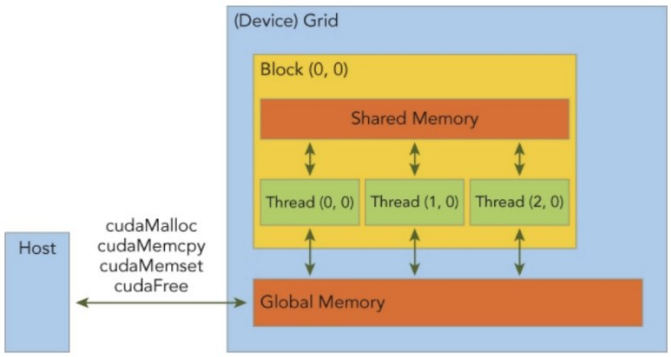

Figura 2 - Gestione della memoria del device [4]

1.2 Kernel

Un kernel CUDA può essere lanciato dal codice host analogamente a come può essere chiamata una funzione in C dal programma principale main. La sintassi con cui si lancia una funzione C è nota come segue:

nome_funzione (lista_argomenti);

Un kernel CUDA può essere lanciato con la seguente sintassi:

kernel <<< grid, block >>> (lista_argomenti);

La definizione di un kernel segue una specifica dichiarazione:

__global__ void nome_kernel (lista_argomenti);

I kernel CUDA sono delle funzioni che presentano delle limitazioni. Alcune di esse sono: la possibilità di accedere solamente alla memoria del dispositivo, devono avere un tipo di ritorno void, non supportano un numero variabile di argomenti né variabili statiche e neanche puntatori a funzione. Ciascun kernel esibisce un comportamento asincrono ovvero una volta lanciato dall'host quest'ultimo riprende ad eseguire le istruzioni successive ad esso.

I thread mandati in esecuzione da un kernel sono organizzati in una struttura che definisce un'astrazione gerarchica degli stessi. Tale gerarchia di thread è scomposta su due livelli. Tutti i thread generati da un singolo lancio del kernel sono collettivamente chiamati griglia e condividono lo stesso spazio di memoria globale. Una griglia è composta da molti blocchi di thread e ciascun blocco rappresenta un insieme di thread che possono cooperare tra di loro usando una sincronizzazione locale e una memoria condivisa a livello di blocco.

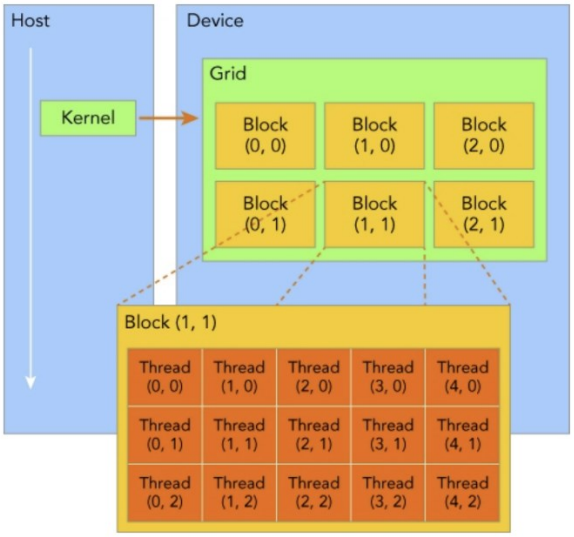

Figura 3 - Gerarchia dei thread [4]

CUDA permette di organizzare griglie di blocchi e blocchi di thread in strutture a 3 dimensioni. Queste dimensioni possono essere impostate lato host dalle variabili predefinite block (dimensione del blocco misurata in thread) e grid (dimensione della griglia misurata in blocchi). Tali variabili sono di tipo dim3, un

tipo vettoriale basato su uint3 incorporato in CUDA che rappresenta una struttura contenente 3 numeri interi senza segno. Ogni componente di questo tipo è accessibile tramite i campi x, y e z che rappresentano rispettivamente le dimensioni lungo le 3 componenti (block.x, block.y, block.z, grid.x, grid.y, grid.z). Qualsiasi componente lasciata non specificata viene inizializzata a 1. Analogamente sono definibili delle variabili di tipo uint3 lato dispositivo come blockDim (blockDim.x, blockDim.y, blockDim.z) e gridDim (gridDim.x, gridDim.y, gridDim.z). In sostanza è possibile definire le variabili per griglia e blocco sull'host prima di avviare un kernel e utilizzare le corrispettive variabili predefinite all'interno del kernel.

Ogni thread all'interno della griglia è identificato da due coordinate uniche rappresentate da blockIdx (blockIdx.x, blockIdx.y, blockIdx.z) che definisce l'indice del blocco in una griglia e threadIdx (threadIdx.x, threadIdx.y, threadIdx.z) che corrisponde all'indice del thread all'interno del blocco. Anche queste variabili sono di tipo uint3 e vengono assegnate a ciascun thread dal runtime CUDA al momento del lancio di un kernel.

Nella funzione del kernel si definisce il calcolo per un singolo thread e l'accesso ai dati per quel thread. Quando viene chiamato il kernel molti thread CUDA eseguono lo stesso calcolo in parallelo. Dal momento che i dati vengono archiviati linearmente nella memoria globale del dispositivo con un approccio di tipo riga è importante comprendere come ciascun thread, facente parte di una struttura organizzata gerarchicamente definita per un dato kernel, venga mappato su una locazione della memoria globale contenente il dato da elaborare. Consideriamo ad esempio una matrice di dati di dimensione 8 x 6. Questa verrà archiviata in maniera lineare in memoria nel seguente modo:

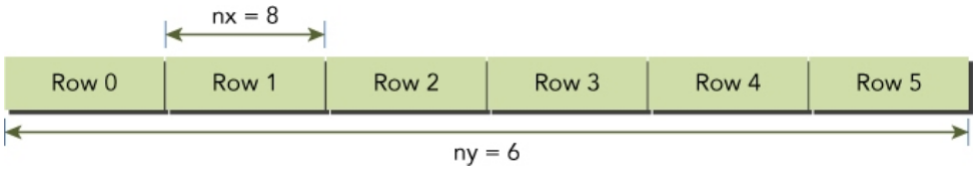

Figura 4 - Archiviazione lineare di una matrice in memoria [4]

Occorre prima mappare l'indice di thread e blocco sulle coordinate della matrice con le seguenti formule:

*ix = threadIdx.x + blockIdx.x * blockDim.x*
*iy = threadIdx.y + blockIdx.y * blockDim.y*

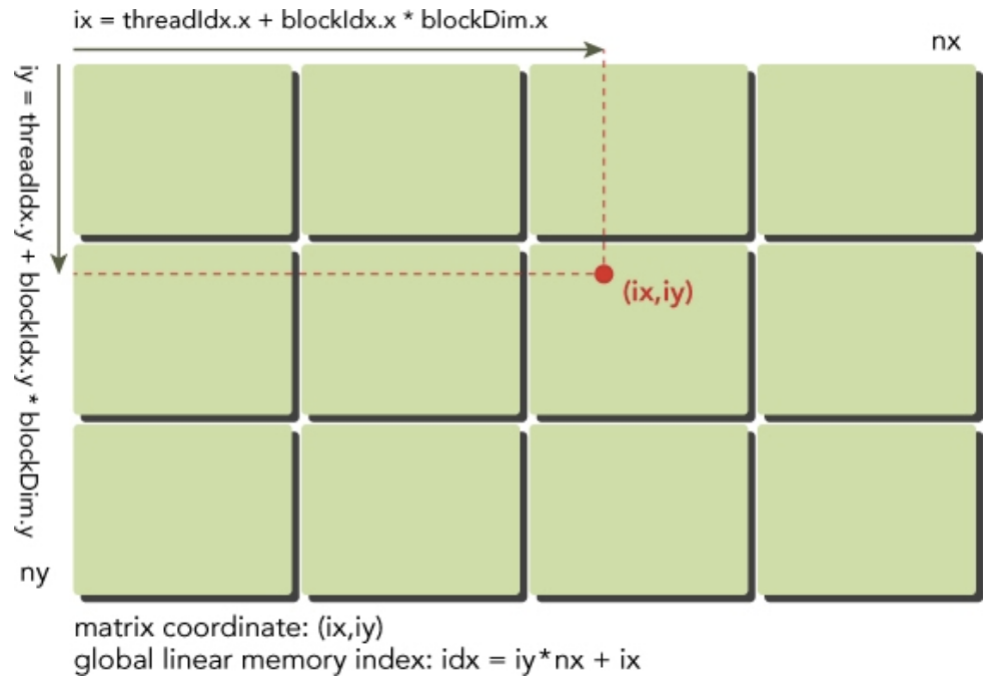

Figura 5 - Indice di memoria globale per una matrice [4]

Successivamente è possibile mappare le coordinate della matrice sull'indice di memoria. In questo modo è possibile definire una corrispondenza tra ogni thread e l'indice di memoria globale lineare del dispositivo ovvero i dati su cui opera il thread [4]:

*idx = iy *nx + ix*

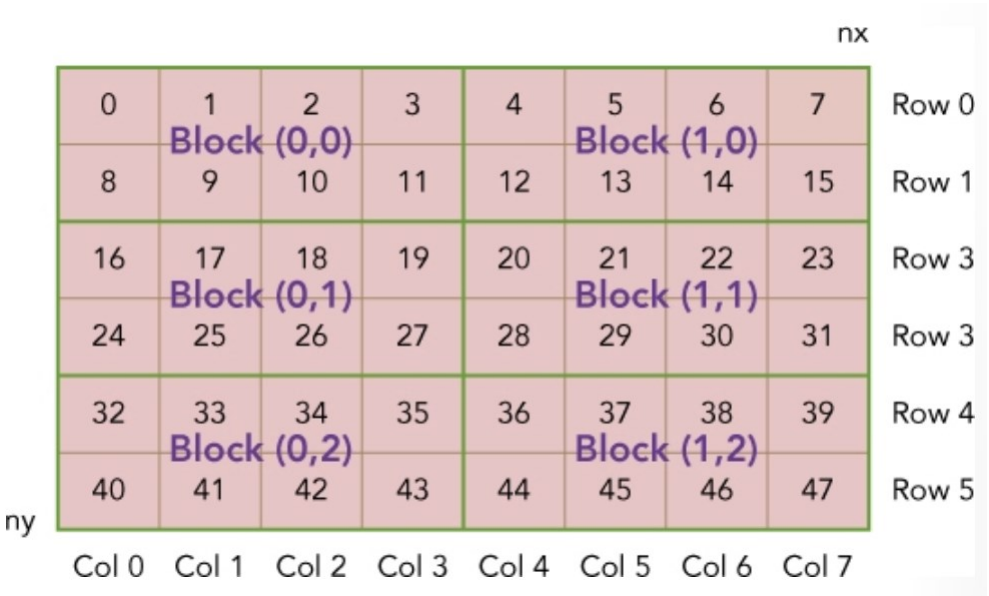

Figura 6 – Mapping di thread su elementi di memoria globale lineare da 0 a 47 [4]

1.3 Compilazione, esecuzione e strumenti di valutazione delle prestazioni

Il codice CUDA comprensivo sia della parte relativa all'host scritta in C che di quella che viene eseguita sul device scritta in CUDA C può essere inserito in un unico file sorgente di estensione .cu e compilato tramite NVIDIA C Compiler da linea di comando digitando:

nvcc nome_sorgente.cu –gpu-architecture=sm_60 -o nome_eseguibile

Viene specificata l'architettura hardware GPU utilizzata (sm_60) per eseguire il codice e le opzioni di ottimizzazione (-o). Una volta compilato il codice questo può essere eseguito semplicemente digitando da linea di comando:

./nome_eseguibile

Conoscere i tempi di esecuzione del kernel oltre che quelli del codice corrispondente implementato per l'host è essenziale per misurare le prestazioni e valutare le ottimizzazioni apportate. Ciò è stato reso possibile utilizzando un timer CPU per misurare il runtime dal lato host che sfrutta la chiamata di sistema gettimeofday per ottenere l'ora dell'orologio di sistema. Per fare questo è necessario includere nell'header file sys/time.h e quindi utilizzare il seguente codice per misurare i tempi sia del kernel che della corrispettiva funzione implementata sul codice host:

```
double iStart = cpuSecond();
kernel_name<<<grid, block>>>(lista_argomenti);
cudaDeviceSynchronize();
double iElaps = cpuSecond() – iStart;
```

Come detto in precedenza la chiamata di un kernel è asincrona rispetto all'host. Nel caso del kernel CUDA è necessario utilizzare cudaDeviceSynchronize per attendere il completamento di tutti i thread GPU, la variabile iElaps restituisce il tempo di esecuzione impiegato.
Un'alternativa per misurare i tempi di esecuzione del kernel è nvprof. Esso è uno strumento di profilatura utilizzabile dalla riga di comando che consente di raccogliere oltre ai tempi di esecuzione del kernel anche altri parametri come i trasferimenti di memoria e le chiamate API CUDA. Può essere lanciato come segue:

nvprof [argomenti_nvprof] ./nome_eseguibile [argomenti_applicazione]

2. IMPLEMENTAZIONE DEL METODO MONTE CARLO SU GPU

In riferimento all'articolo "A GPU Parallel Algorithm for Non Parametric Tensor Learning" si è preso in considerazione il primo esperimento, quello riguardante il problema di classificazione di immagini tramite tecnica di Monte Carlo. Le immagini utilizzate per la classificazione appartengono al dataset MNIST, una vasta base di dati costituita da immagini di cifre scritte a mano comunemente impiegata come insieme di addestramento e di testing nel campo del Machine Learning. Questo database contiene 60000 immagini di addestramento e 10000 immagini di test. Ciascuna immagine rappresenta una cifra da 0 a 9, definendo 10 possibili classi come output.

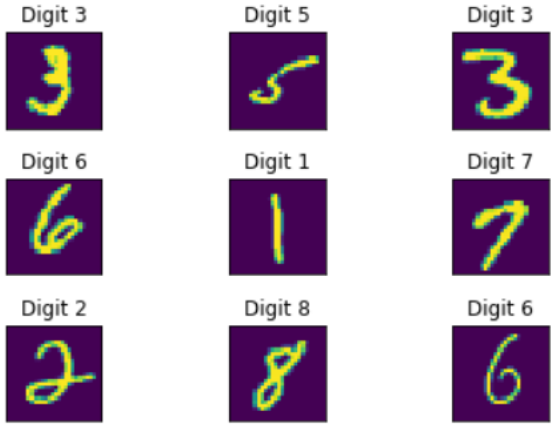

Figura 7 - Esempi di immagini appartenenti al dataset MNIST con rispettive etichette

Ogni immagine presenta una dimensione di 28 x 28 pixel ed è memorizzata come un'unica sequenza di 784 elementi. Le immagini appartenenti al training set e al test set sono acquisite dal programma tramite i file di testo "X_train.txt" e "X_test.txt" che contengono i valoro corrispondenti ai pixel delle immagini archiviate in sequenza. Nei file "y_train.txt" e "y_test.txt" sono memorizzate le etichette corrispondenti alle possibili classi cui ogni immagine è associata.

2.1 Versione originale del codice

La struttura del codice preso come riferimento viene brevemente riepilogata a seguito. La parte iniziale prevede la definizione di quattro funzioni. La prima, denominata "initialFileData", è necessaria per l'acquisizione dei dati dai file di tipo txt archiviati nella memoria di massa della macchina. Vengono poi definite la funzione "montecarloMatrixOnHost" che implementa l'algoritmo di Monte Carlo eseguito su CPU e il kernel CUDA "montecarloMatrixOnGPU1D" che rappresenta la corrispondente versione della funzione C precedentemente

menzionata ma che verrà poi eseguita su GPU. Segue il codice relativo alla funzione "checkResult" utilizzata per confrontare i risultati ottenuti dalle due implementazioni dell'algoritmo fatte girare su CPU e GPU. Nel programma principale main sono definite le dimensioni delle strutture di dati quindi il numero di immagini appartenenti al training set che si decide di adoperare, il numero di immagini del test set e il numero di pixel per immagine. Sulla base di queste si ottengono delle matrici in cui i dati vengono organizzati a livello logico e si fissano le relative dimensioni in byte necessarie poi per l'allocazione in memoria. Segue l'allocazione dinamica della memoria tramite malloc per la memori relativa all'host, il caricamento dei dati dalla memoria di massa a quella dell'host e il lancio della funzione che implementa l'algoritmo su CPU. Una porzione di codice permette il calcolo dell'accuratezza che l'algoritmo che sfrutta la tecnica di Monte Carlo su CPU è in grado di raggiungere. Tramite cudaMalloc viene allocata la memoria relativa al device sulla quale vengono trasferiti i dati mediante cudaMemcpy dalla memoria host. La sezione di codice successiva è quella relativa all'invocazione del kernel CUDA dal codice host del programma principale. Sono definiti il numero di blocchi per la griglia e il numero di thread per blocco per la configurazione di lancio del kernel. Quest'ultimo viene quindi mandato in esecuzione realizzando il calcolo su GPU. Il kernel viene lanciato tramite un ciclo for tante volte quante sono le immagini del test set, producendo ad ogni esecuzione un valore predetto. La corrispondente funzione su CPU è richiamata una sola volta integrando il corrispondente ciclo for all'interno del codice della funzione stessa. I risultati conseguiti da questa operazione vengono copiati tramite cudaMemcpy sulla memoria host. Seguono le istruzioni relative al calcolo dell'accuratezza ottenuta dall'algoritmo eseguito su GPU, il check dei risultati con quelli ottenuti dalla corrispettiva versione fatta girare su CPU e il rilascio della memoria allocata sul dispositivo tramite cudaFree e sull'host per mezzo della funzione free. Tramite cudaDeviceReset può essere resettato il dispositivo. Il codice versione integrale è riportato in appendice, corretto in alcune parti come spiegato nel paragrafo successivo.

Il cuore del codice è rappresentato dalla funzione C e dal kernel CUDA che implementano l'algoritmo di predizione ottenuto tramite tecnica di Monte Carlo. In riferimento al paper [1] tale funzione sfrutta un kernel di tipo gaussiano caratterizzato da un ordine m e può essere espressa come segue:

$$f_m(X) \approx \frac{\sum_{i=1}^{N} f(t_i) \exp\left(-\frac{1}{2} m^2 \|X - t_i\|^2\right)}{\sum_{i=1}^{N} \exp\left(-\frac{1}{2} m^2 \|X - t_i\|^2\right)}$$

La funzione f(t_i) rappresenta l'etichetta corrispondente all'immagine i-esima del training set, X l'immagine di test considerata, t_i l'immagine i-esima del training set e N il numero di immagini del training set complessivamente utilizzate per la fase di addestramento. L'ordine m del kernel gaussiano utilizzato è pari a 10.

Le immagini di 28 x 28 pixel vengono allocate in memoria in maniera lineare come vettori composti da 784 elementi posti in sequenza. Ogni elemento corrisponde ad un pixel dell'immagine. Il dataset prevede un totale di 60000 immagini di training e 10000 immagini di test. Nella seguente trattazione utilizzeremo solamente una parte delle immagini di test pari a 1000. Ne segue che concettualmente possiamo pensare di avere due matrici, una relativa al training set di 60000 x 784 elementi e una relativa al test set di 1000 x 784 elementi. Ogni riga di ciascuna delle due matrici rappresenta un'immagine che appartiene rispettivamente al training set e al test set. Per il calcolo della norma tra immagini di test e di training viene allocato spazio in maniera lineare in memoria per contenere una matrice differenza di dimensione 60000 x 784 elementi. Viene realizzata la differenza tra la prima riga della matrice di test (prima immagine del training test) e tutte le righe in sequenza della matrice di training (immagini del training set). Risulta così riempita la matrice differenza per il calcolo di un valore predetto dall'algoritmo. Questa operazione deve essere iterata tante volte quante sono le immagini del test set che si decide di utilizzare. Per questo motivo è utilizzato un ciclo for che esegue 1000 iterazioni, una per ogni riga della matrice di test. Nel caso della funzione C eseguita su host il ciclo for in questione è inserito nel codice della funzione stessa che è lanciata una sola volta dal main. Per il kernel CUDA eseguito su GPU invece il ciclo for è inserito nel programma principale lanciando il kernel tante volte quante sono le immagini del test set.

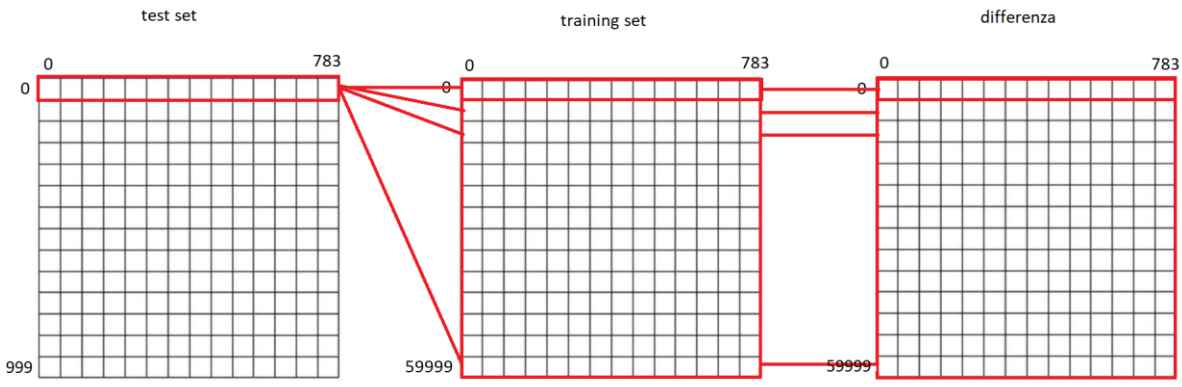

Figura 8 - Calcolo della matrice differenza per un'immagine del test set

Nel codice eseguito su CPU sono inseriti complessivamente tre cicli for. Quello più interno scorre i 784 elementi che caratterizzano ciascuna riga permettendo di calcolare la differenza pixel per pixel tra una riga della matrice di test e una riga della matrice di training, di memorizzare il risultato su una riga della matrice differenza e calcolarne la norma sui 784 elementi. Il secondo ciclo più esterno al primo consente il calcolo della funzione esponenziale a numeratore e denominatore della funzione $f_m(X)$. Anche in questo caso le operazioni sono eseguite in maniera sequenziale tramite 60000 iterazioni pari al numero di righe della matrice di training. Ad ogni iterazione le variabili csum e fsum che consentono di mantenere il valore delle sommatorie rispettivamente corrispondenti a denominatore e numeratore della funzione $f_m(X)$ vengono incrementate del valore che deriva dal calcolo corrispondente alla riga di training considerata e viene incrementato un puntatore per passare ad un'altra riga del training set alla successiva iterazione. Il ciclo for più esterno consente di scorrere le 1000 righe relative al test set e ripetere tutte le operazioni precedentemente descritte per un totale di 1000 valori predetti dalla funzione $f_m(X)$. Prima di uscire dal ciclo ad ogni iterazione sono incrementati due puntatori, uno per riportarsi alla prima riga della matrice di training e l'altro per puntare alla successiva riga della matrice di test. Chiaramente il calcolo eseguito su CPU è caratterizzato esclusivamente dall'esecuzione in sequenza delle varie operazioni.

Il codice eseguito sulla GPU fa uso di due soli cicli for nel complesso. All'interno del kernel un ciclo for è utilizzato per scorrere le 784 elementi di ogni riga delle matrici ma i 60000 calcoli relativi ad ogni colonna sono realizzati in parallelo, questo consente un primo incremento computazionale rispetto alla versione eseguita su CPU. Ottenuta la matrice differenza viene calcolata la norma relativa ad ogni riga costituita da 784 elementi per un totale di 60000 valori. Nel codice host è utilizzato un elevamento a potenza e la funzione sqrt() per la radice quadrata mentre dal lato device si utilizza la funzione norm() delle librerie matematiche CUDA che calcola direttamente la norma di una riga della matrice differenza. Successivamente è possibile calcolare l'esponenziale dunque denominatore e numeratore della funzione $f_m(X)$. Per la versione eseguita su GPU vengono impiegati gli stessi 60000 thread impiegati per il calcolo di una colonna della matrice differenza per eseguire i calcoli relativi alla norma, all'esponenziale e alle sommatorie in parallelo. Per il calcolo delle sommatorie si rende necessario l'utilizzo dell'istruzione atomicAdd che consente di memorizzare il valore di denominatore e numeratore nelle rispettive locazioni di memoria mantenendo la coerenza dei dati. Questo perché i thread eseguono operazioni simultaneamente, la scrittura sulla locazione deve essere coerente per ottenere un risultato corretto. Viene fatto poi il rapporto tra numeratore e denominatore per ottenere il valore predetto dalla funzione $f_m(X)$, questo

rappresenta una cifra da 0 a 9 corrispondente all'immagine del test set presa in considerazione. Il secondo ciclo for situato nel programma principale permette di lanciare il kernel CUDA tante volte quante sono le righe della matrice di test che si decide di utilizzare, nel caso trattato 1000 per un totale di 1000 valori predetti da $f_m(X)$.

Correzione di bugs

Nonostante la versione originale del codice relativo al kernel CUDA permettesse di valutare i tempi di esecuzione lato device, dal punto di vista della correttezza dei calcoli presentava imprecisioni. Nello specifico veniva effettuato il calcolo dell'intera matrice differenza, la norma riga per riga di tutte le righe relative alla matrice differenza, le sommatorie corrispondenti a numeratore e denominatore della funzione predittiva e il valore predetto dalla funzione stessa ma il tutto solo per la prima riga della matrice di test, ripetendo i calcoli sempre utilizzando tale riga. Per questo motivo si è reso necessario apportare essenzialmente due modifiche al codice originale. La prima è quella di passare l'indice di incremento utilizzato nel ciclo for del main come ulteriore argomento del kernel (test_row), per scrivere locazioni di memoria progressive nelle quali memorizzare i valori calcolati per le sommatorie di numeratore e denominatore della funzione relative alle varie righe di test e i valori predetti dalla funzione relativi sempre alle singole righe di test:

```
atomicAdd((double*)&csum[test_row], exp_value[ix]);
atomicAdd((double*)&fsum[test_row], (Vect_YTrain[ix]*exp_value[ix]));
```

in luogo di:

```
atomicAdd(csum, exp_value[ix]);
atomicAdd(fsum, (Vect_YTrain[ix]*exp_value[ix]));
```

```
Vect_Predict[test_row] = __double2int_rn(fsum[test_row]/csum[test_row]);
```

in luogo di:

```
Vect_Predict[0] = __double2int_rn(fsum[0]/csum[0]);
```

La seconda modifica corrisponde all'inserimento di un puntatore i_xtest all'interno del kernel che puntasse ad ogni lancio progressivo dello stesso ad una

riga successiva della matrice di test con cui effettuare tutti i calcoli precedentemente descritti:

*double * i_xtest = Mat_XTest;*
*i_xtest += ny * test_row;*

La variabile ny rappresenta il numero di pixel di un'immagine. Questo ha permesso di ottenere un codice che producesse risultati corretti dal punto di vista logico.

Definizione delle prestazioni di riferimento

I risultati ottenuti per l'implementazione dell'algoritmo di classificazione eseguito su CPU e GPU possono essere tratti dall'articolo [1]:

Train dim	Test dim	Runtime [s]	
		CPU	GPU
60000	1	0.36	0.07
60000	10	3.65	0.67
60000	100	36.58	6.60
60000	1000	365.70	66.12
60000	10000	3662.03	664.16

Figura 9 - Prestazioni di riferimento

La tabella mostra i tempi di esecuzione del codice relativo all'algoritmo applicato al dataset MNIST utilizzando l'intero training set di immagini e un numero variabile di immagini prese dal test set. Nel paper si è potuto dimostrare che il codice eseguito su GPU sovraperforma quello eseguito su CPU di un fattore di circa 5 in tutte le condizioni di test. Questo è dovuto al fatto che mentre il codice host è eseguito in maniera sequenziale, nella versione corrispondente su device parte dei calcoli sono eseguiti in parallelo.
Per la trattazione successiva andremo a prendere come riferimento la configurazione che sfrutta 60000 immagini per l'addestramento e 1000 immagini di test. Si sarebbero potute considerare tutte le immagini del test set ma ciò avrebbe richiesto dei tempi superiori per testare le varie configurazioni del codice. Nella condizione presa in esame il runtime che è possibile raggiungere su CPU è di circa 365 secondi mentre l'implementazione su GPU permette di fermarsi a 66 secondi. L'accuratezza che si può raggiungere lato CPU e GPU è pari al 95%. Prendiamo questi riferimenti per andare a modificare il kernel CUDA

eseguito sulla GPU e tentare di migliorare ulteriormente le prestazioni che è possibile ottenere dal calcolo parallelo.

2.2 Parallelismo a livello logico

Analizzando le operazioni che devono essere eseguite per implementare l'algoritmo si può osservare che la maggior parte dei calcoli da effettuare è localizzata nella porzione iniziale del kernel. Per effettuare il calcolo della matrice differenza di dimensioni 60000 x 784 elementi, sono necessari 47040000 calcoli solo per ottenere i risultati per tale matrice per ciascuna immagine di test. Nel kernel eseguito dalla GPU parte dei calcoli relativi alla matrice differenza è realizzata in parallelo. Questo ha consentito di incrementare le prestazioni rispetto alla versione su CPU. Numerosi thread (60000) eseguono gli stessi calcoli su dati diversi. In particolare nella versione originale del software vengono sfruttati 60000 blocchi, ciascuno caratterizzato da un singolo thread, che permettono di processare 60000 calcoli in parallelo e ottenere i relativi risultati per una colonna della matrice differenza. Questi calcoli poi vengono ripetuti impiegando gli stessi thread in maniera sequenziale per 784 volte relative a ciascuna riga della matrice differenza utilizzando un ciclo for che scorre le varie colonne della matrice.

Un primo approccio potrebbe essere quello di aumentare il grado di parallelismo a livello logico per il calcolo su GPU eliminando il ciclo for all'interno del kernel. E' lecito pensare quindi di utilizzare in luogo di 60000 thread un numero di thread pari a 60000 x 784 per eseguire tutti i 47040000 calcoli in un solo passaggio in parallelo al fine di ottenere la matrice differenza. Dal momento che per la rimanente parte del codice sono sufficienti 60000 thread per eseguire le operazioni restanti si è deciso di suddividere il kernel CUDA in due parti. Il primo kernel si occupa dunque del calcolo della matrice differenza. Per questo kernel è previsto l'utilizzo di una gerarchia 2D di blocchi (60000 x 784) e un solo thread per blocco per poterne confrontare le prestazioni con la versione originale del codice che utilizzava anch'essa un thread per blocco. Questo tipo di struttura viene denominata 2D1D in quanto la griglia di blocchi è organizzata in una forma bidimensionale mentre i blocchi in una sola dimensione. Per ottenere questo, dal lato host, sono state inizializzate le variabili predefinite di tipo dim3 come segue:

```
dim3 grid (60000, 784);
dim3 block (1);
```

Il secondo kernel si occupa di eseguire la restante parte dei calcoli. Presenta una struttura monodimensionale sia per la griglia di blocchi che per i blocchi di thread quindi di tipo 1D1D:

dim3 grid (60000);
dim3 block (1);

In questo caso si utilizza un solo thread per ognuno dei 60000 blocchi, per un totale di 60000 thread. Occorre apportare delle modifiche all'interno del codice relativo al primo kernel per poter scorrere i 60000 blocchi nella dimensione x e i 784 blocchi nella dimensione y in modo da mappare ciascun thread nella giusta posizione della matrice differenza:

unsigned int ix = blockIdx.x;
unsigned int iy = blockIdx.y;

Questo perché nella versione originale del codice la griglia di blocchi era monodimensionale quindi si sviluppava solo lungo la componente x, ora abbiamo anche una componente lungo y. Occorre inoltre effettuare un controllo anche nella dimensione y per fare in modo che i thread non eccedano le dimensioni della matrice di dati lungo y oltre che lungo x:

if (ix < nx && iy < ny)

Si è dunque eseguito il codice modificato e misurato il tempo complessivamente impiegato dai due kernel utilizzati (GPU 2 runtime). Di seguito sono riepilogati i risultati ottenuti a confronto con le prestazioni di riferimento (CPU e GPU1):

CPU runtime [s]	GPU 1 runtime [s]	GPU 2 runtime [s]
365.70	66.12	132.47

A livello logico l'ambiente CUDA permette di eseguire il calcolo parallelo lanciando un numero di thread elevato a piacere. Esistono però dei limiti sul numero di thread simultanei che è possibile effettivamente avere in esecuzione. Aumentando il numero di thread per eseguire i calcoli nel primo kernel si è potuto constatare che non necessariamente questo porti a un incremento di velocità complessiva. Sebbene le prestazioni siano ancora superiore alla versione implementata su CPU questa versione del codice non è chiaramente ottimizzata per sfruttare al meglio l'hardware della GPU. Occorre effettuare un'analisi più approfondita su come venga utilizzato effettivamente l'hardware e sul modello di esecuzione CUDA al fine di ottenere l'obiettivo desiderato.

2.3 Parallelismo a livello hardware

Architettura della GPU

Per comprendere come ottimizzare il codice eseguito sul device è opportuno analizzare l'architettura hardware e il modello di esecuzione implementato in CUDA. Questo fornisce una visione operativa di come vengono eseguite le istruzioni sulla specifica architettura informatica. L'architettura della GPU è costituita da un'unita hardware elementare chiamata streaming multiprocessor (SM) che viene replicata per ottenere l'architettura complessiva della GPU. Ciò permette di ottenere un parallelismo a livello hardware molto potente ed efficace per il calcolo parallelo.

Figura 10 - Architettura NVIDIA 1080 GTX [5]

Nello specifico la scheda grafica utilizzata per effettuare i test relativi a questo lavoro è una GeForce 1080 GTX che utilizza il chip GP104. Tale chip integra 20 streaming multiprocessor ciascuno dotato di 128 CUDA core al proprio interno per un totale di 2560 CUDA core complessivi. Gli SM sono raccolti in 4 GPC (Graphics Processing Clusters). Sono presenti 8 interfacce di memoria GDDR5 DRAM a 256 bit che supportano 8 GBytes di memoria globale interna.

Un'interfaccia host collega la GPU alla CPU tramite il bus PCI Express. Il motore GigaThread è uno scheduler globale che distribuisce i blocchi di thread ai warp scheduler degli streaming multiprocessor. I componenti chiave di uno SM sono rappresentati dalle unità di calcolo fondamentali chiamate CUDA core. Sono presenti altre risorse hardware all'interno dello SM: una memoria condivisa (shared memory), una memoria cache L1, dei file register, delle unità load/store, delle unità per il calcolo di funzioni speciali (special function unit) e dei warp scheduler.

Figura 11 – Architettura di uno streaming multiprocessor (SM) [5]

Nell'architettura GPU presa in considerazione per questo lavoro ogni SM contiene 32 unità load/store che consentono di calcolare gli indirizzi di origine e destinazione di 32 thread per ciclo di clock. Le unità a funzione speciale (SFU) eseguono istruzioni per il calcolo di funzioni specifiche tra la quali seno, coseno, radice quadrata e interpolazione. Per ogni SM sono presenti 4 warp scheduler e 8 unità di invio istruzioni.

Modello di esecuzione CUDA

Quando viene lanciato un kernel dall'host, dal punto di vista logico, tutti i thread del kernel funzionano in parallelo. Come è stato sperimentato con la precedente versione del codice è possibile mandare in esecuzione un numero grande a piacere di thread. Da una prospettiva hardware non tutti i thread possono essere eseguiti fisicamente in parallelo contemporaneamente. Al momento dell'avvio del kernel si lancia una griglia di blocchi di thread e i blocchi di thread vengono distribuiti tra le varie unità streaming multiprocessor disponibili. Una volta che un blocco è assegnato a uno SM i thread relativi a quel blocco vengono ulteriormente partizionati in warp di 32 thread. Un warp è l'unità base di esecuzione su uno streaming multiprocessor. Esso è costituito da 32 thread consecutivi e i thread appartenenti a un warp vengono eseguiti in modalità Single Instruction Multiple Thread (SIMT). Tutti i thread eseguono la stessa istruzione e ogni thread porta a termine tale operazione sui propri dati privati.

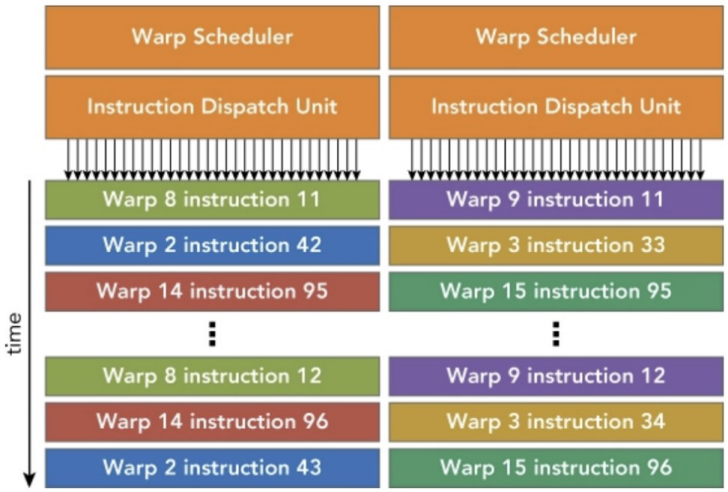

Figura 12 - Esecuzione del warp [4]

Mentre i warp all'interno di un blocco possono essere programmati in qualsiasi ordine, il numero di warp attivi è limitato dalle risorse dello SM. Quando un warp diventa inattivo per qualsiasi motivo lo SM è libero di pianificare un altro warp disponibile da qualsiasi blocco residente nello SM tramite il warp scheduler. Il passaggio da un warp all'altro non crea sovraccarico perché le risorse hardware sono suddivise tra tutti i thread e i blocchi su uno SM quindi lo stato del warp appena programmato è già memorizzato sullo SM. CUDA suddivide le risorse di calcolo in uno SM tra più blocchi di thread residenti. Questo partizionamento fa sì che alcune risorse diventino limitatori di prestazioni. Un blocco thread viene chiamato blocco attivo quando sono state allocate risorse di calcolo come registri e memoria condivisa. I warp che contiene sono chiamati warp attivi. I warp attivi possono essere ulteriormente classificati nei seguenti tre tipi: warp

selezionato, warp bloccato, warp idoneo. I warp scheduler sugli SM selezionano warp attivi per ogni ciclo e li inviano alle unità di esecuzione. Un warp che viene eseguito attivamente è chiamato warp selezionato. Se un warp attivo è pronto per l'esecuzione ma non è attualmente in esecuzione è detto warp idoneo. Se un warp non è pronto per l'esecuzione è un warp bloccato. Un warp è idoneo all'esecuzione se 32 core CUDA sono disponibili per l'esecuzione e se tutti gli argomenti dell'istruzione corrente sono pronti.

Il numero massimo di warp che possono essere lanciati su uno SM è determinato dall'architettura utilizzata. CUDA mette a disposizione delle API per visionare alcune proprietà del device richiamabili da codice. Le informazioni sulle caratteristiche e limitazioni relative all'hardware specifico sono state raccolte lanciando il codice simpleDeviceQuery.cu:

```
turchi@machlearn:~$ nvcc simpleDeviceQuery.cu --gpu-architecture=sm_60 -o q
turchi@machlearn:~$ nvprof ./q
==31723== NVPROF is profiling process 31723, command: ./q
Device 0: GeForce GTX 1080
  Number of multiprocessors:                     20
  Total amount of constant memory:               64.00 KB
  Total amount of shared memory per block:       48.00 KB
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum number of threads per multiprocessor:  2048
  Maximum number of warps per multiprocessor:    64
==31723== Profiling application: ./q
==31723== Profiling result:
No kernels were profiled.
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
     API calls:   33.60%  761.99us         1  761.99us  761.99us  761.99us  cuDeviceTotalMem
                  30.97%  702.32us         1  702.32us  702.32us  702.32us  cudaGetDeviceProperties
                  30.95%  701.84us        97  7.2350us     292ns  306.85us  cuDeviceGetAttribute
                   4.01%  90.886us         1  90.886us  90.886us  90.886us  cuDeviceGetName
                   0.23%  5.1250us         1  5.1250us  5.1250us  5.1250us  cuDeviceGetPCIBusId
                   0.14%  3.1850us         3  1.0610us     338ns  1.9090us  cuDeviceGetCount
                   0.08%  1.8840us         2     942ns     383ns  1.5010us  cuDeviceGet
                   0.03%     661ns         1     661ns     661ns     661ns  cuDeviceGetUuid
turchi@machlearn:~$
```

Figura 13 - Query device CUDA

Ogni streaming multiprocessor è in grado di ospitare 64 warp di cui solo 4 attivi selezionabili dal warp scheduler. L'hardware è costituito da 20 unità SM per un totale di 1280 warp eleggibili. Dal momento che un warp è un insieme di 32 thread potenzialmente è possibile avere 40960 thread idonei all'esecuzione. I warp idonei ma non attivi in eccesso sono gestiti dallo scheduler globale GigaThread che invia gruppi di 32 thread ai warp scheduler dei vari SM. Dei 40960 thread solamente 2560 possono essere effettivamente attivi, pari al numero di CUDA core presenti. Se un warp si blocca il warp scheduler raccoglie un warp idoneo da eseguire al suo posto. Poiché le risorse di calcolo sono suddivise tra i warp e mantenute su chip durante l'intera vita del warp, il passaggio da un contesto all'altro è comunque molto rapido. Si deduce che solamente un numero limitato di thread è in volo contemporaneamente, occorre ottimizzare altri parametri piuttosto che aumentare il numero di thread lanciati per una dato kernel come testato al passaggio precedente.

La scheda grafica in questione permette di utilizzare un numero massimo di 1024 thread per blocco, caratteristica di cui bisogna tener conto al momento della definizione del numero di thread per blocco utilizzati per il lancio di un kernel. I thread possono essere organizzati in blocchi di struttura monodimensionale, bidimensionale o tridimensionale ma dal punto di vista hardware sono ordinati in modo unidimensionale. Se i thread sono organizzati in un blocco ad una dimensione l'identificativo univoco relativo del thread è archiviato nella variabile incorporata CUDA threadIdx.x e i thread con valori consecutivi di questa variabile sono raggruppati in warp. Per blocchi bidimensionali il layout logico del blocco può essere convertito in layout fisico monodimensionale utilizzando la dimensione x come dimensione più interna e la dimensione y come dimensione più esterna. L'identificatore univoco del thread in questo caso può essere ottenuto utilizzando oltre alla variabile threadIdx anche la variabile blockDim tramite l'espressione threadIdx.y * blockDim.x + threadIdx.x.
L'hardware alloca sempre un numero discreto di warp per un blocco thread. Un warp non viene mai suddiviso tra diversi blocchi di thread. Se la dimensione del blocco di thread non è un multiplo pari della dimensione del warp alcuni thread del warp rimangono inattivi [4].

Gestione della memoria globale

Il modello di esecuzione CUDA prevede che le istruzioni siano emesse ed eseguite per warp. Anche le operazioni di memoria vengono eseguite per warp. Quando viene attuata un'istruzione di memoria ogni thread in un warp fornisce un indirizzo di memoria che sta caricando o memorizzando. I 32 thread in un warp presentano un'unica richiesta di accesso alla memoria composta dagli indirizzi richiesti che è servita da una o più transazioni di memoria dal dispositivo. A seconda della distribuzione degli indirizzi di memoria all'interno di un warp gli accessi alla memoria possono essere classificati in diversi schemi. Le letture e le scritture di memoria globale sono organizzate utilizzando la memoria cache. La memoria globale è uno spazio di memoria logica a cui è possibile accedere dal proprio kernel. Tutti i dati dell'applicazione risiedono inizialmente nella DRAM, la memoria del dispositivo fisico. Le richieste di memoria del kernel vengono in genere servite tra DRAM del dispositivo e memoria sullo streaming multiprocessor utilizzando transazioni di memoria a 128 o 32 byte. Tutti gli accessi alla memoria globale passano attraverso la cache L2 e numerosi accessi passano anche attraverso la cache L1. Questo dipende del tipo di accesso e dell'architettura della GPU. Se vengono utilizzate entrambe le cache L1 e L2 un

accesso alla memoria viene gestito da una transazione di memoria a 128 byte. Una riga della cache L1 è di 128 byte e viene mappata su un segmento allineato di 128 byte nella memoria del dispositivo. Gli accessi alla memoria possono essere caratterizzati come accessi di memoria allineati (o non allineati) e accessi di memoria coalizzati (o non coalizzati). Gli accessi alla memoria allineati si verificano quando il primo indirizzo di una transazione di memoria del dispositivo è un multiplo pari della granularità della cache utilizzata per servire la transazione (32 byte per la cache L2 o 128 byte per la cache L1). L'esecuzione di un carico disallineato causerà uno spreco di larghezza di banda. Gli accessi alla memoria coalizzati si verificano quando tutti e 32 i thread in un warp accedono a un pezzo contiguo di memoria [4].

2.4 Gerarchie di thread per sfruttare l'hardware

Per utilizzare al meglio la capacità computazionale della GPU occorre tenere occupati gli streaming multiprocessor nella misura maggiore possibile. Questo si traduce nell'avere un numero di warp attivi potenzialmente elevato. Un altro fattore da prendere in considerazione è l'utilizzo della memoria DRAM del dispositivo dalla quale vengono letti i dati da elaborare. In alcuni casi queste ultime operazioni possono avere un effetto impattante sulle prestazioni complessive che è possibile ottenere dal codice. Esplorare varie gerarchie con cui sono organizzati i thread nella griglia permette di ottenere delle configurazioni di esecuzione del kernel che sviluppano performance differenti. Sarà possibile poi misurare parametri che rendano conto del livello di occupazione delle unità di calcolo e dell'utilizzo della memoria per ciascuno di questi casi e analizzare come questo influisca sui tempi di esecuzione del codice fatto girare sulla GPU.

Il modello di programmazione CUDA offre un modo per organizzare i thread sulla GPU attraverso una struttura gerarchica. Dal momento che le risorse hardware assegnate ad un blocco quando questo viene caricato su uno SM vengono suddivise tra i thread del blocco si potrebbe pensare che con la configurazione ad un thread per blocco descritta comporti un sottoutilizzo di tali risorse quali registri e memoria condivisa. In effetti configurazioni di lancio estreme quali piccoli blocchi costituiti da tanti thread o grandi blocchi formati da pochi thread, come nelle configurazioni del kernel analizzate fino a questo punto, non generi quasi mai le prestazioni migliori. Troppi pochi thread per blocco comportano limiti hardware sul numero di warp per SM da raggiungere prima che tutte le risorse siano completamente utilizzate. Nel caso opposto troppi thread per blocco portano ad un numero inferiore di risorse hardware per SM disponibili per ciascun thread. Le risorse hardware necessarie ad ogni thread dipendono dalla natura del codice per cui occorre sperimentare diverse gerarchie di organizzazione dei thread per poter verificare quale produca le prestazioni migliori.
La procedura che verrà utilizzata per definire una struttura gerarchica in cui organizzare i thread lanciati dal kernel è spiegata di seguito. Per una data dimensione dei dati:
-si fissa la dimensione dei blocchi;
-si calcola la dimensione della griglia in base alla dimensione dei dati dell'applicazione e alla dimensione del blocco.
Per determinare la dimensione del blocco occorre considerare:
-le caratteristiche prestazionali del kernel;
-le limitazioni sulle risorse GPU.

Prima implementazione (kernel 1D1D)

Nella versione di riferimento del codice viene utilizzato un solo kernel CUDA che sfrutta 60000 thread per eseguire i calcoli. All'interno del kernel vengono eseguite sia le istruzioni che permettono il calcolo della matrice differenza che le restanti operazioni di norma, sommatorie e rapporto tra queste, utilizzando gli stessi thread in sequenza. La griglia è costituita da 60000 blocchi ciascuno dei quali contenente un thread. Questa si sviluppa soltanto lungo la componente x con un numero di elementi pari al numero dei blocchi, ogni blocco ha un solo elemento lungo x pari al numero di thread. Per questo motivo la gerarchia dei thread definita è di tipo 1D1D.

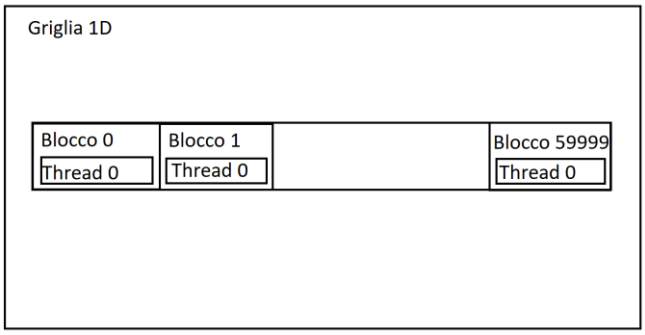

Figura 14 - Griglia di blocchi da un thread

Applicando la procedura precedentemente descritta andiamo a definire diverse strutture di organizzazione dei thread. La prima operazione da effettuare è quella di fissare la dimensione dei dati. Nel caso in esame questa è data dal numero di immagini del training set utilizzate che complessivamente sono 60000.
Decidiamo di testare configurazioni di lancio del kernel con blocchi monodimensionali costituiti da 2, 4, 8, 16, 32, 64, 128, 256, 512 threads. La variabile dimx che rappresenta il numero di thread per blocco lungo x andrà ad assumere uno di questi valori in ciascun caso, a seconda della configurazione del kernel lanciata.
Stabiliti dimensione dei dati e il numero di thread per blocco si può determinare, come diretta conseguenza, la dimensione della griglia ovvero il numero di blocchi per griglia utilizzati per mezzo della seguente formula:

(numero_di_dati + numero_thread_per_blocco – 1)/ numero_thread_per_blocco

Tuttavia in luogo di 60000 righe della matrice di training ne sono state considerate 59904. Questa scelta è stata fatta essenzialmente per due motivi. Il primo è fare in modo che tutte le configurazioni del kernel lanciate presentino il medesimo numero complessivo di thread utilizzati effettivamente per i calcoli.

Per ottenere questo occorre considerare una dimensione dei dati che sia un numero divisibile per tutte le dimensioni del blocco che abbiamo deciso di testare. Il secondo motivo è legato al modello di esecuzione CUDA per cui l'hardware alloca sempre un numero discreto di warp per blocco di thread. Un warp non viene mai suddiviso tra diversi blocchi di thread. Se la dimensione del blocco non è un multiplo pari della dimensione del warp alcuni thread nell'ultimo warp relativo a un blocco rimangono inattivi. Ciascuno blocco viene assegnato ad uno streaming multiprocessor, quando il blocco viene suddiviso in warp, se la sua dimensione è inferiore a 32 thread, alcuni di essi rimangono inattivi. Per tutte le configurazioni che utilizzano un numero di thread per blocco inferiore a 32 questo fenomeno è inevitabile. Ad esempio una configurazione caratterizzata da 4 thread per blocco sfrutterà solamente 4 thread all'interno di un warp selezionato dallo scheduler per l'esecuzione.

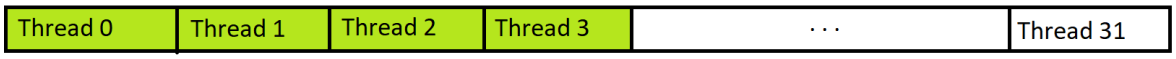

Figura 15 - Thread attivi all'interno di un warp per un blocco di 4 thread

La scelta di utilizzare un numero leggermente inferiore di immagini appartenenti al training set non andrà ad influire in modo significativo sulle prestazioni dell'algoritmo dal punto di vista dell'accuratezza di previsione. L'effetto è del tutto trascurabile come è possibile verificare dall'esecuzione del codice.

Consideriamo il codice di riferimento. La variabile ix, definita nel codice del kernel, stabilisce la corrispondenza univoca tra l'indice di ogni thread e l'elemento del vettore di dati su cui il thread va ad operare. Per una gerarchia ad un thread per blocco è sufficiente assegnare a tale variabile l'identificatore di blocco blockIdx.x che assume valori da 0 a 59903 al variare di x.

unsigned int ix = blockIdx.x;

Nel caso si utilizzi un'organizzazione a più thread per blocco è necessario modificare il valore della variabile dal momento che l'indice relativo a un blocco non permette di identificare in maniera univoca ciascun thread della griglia. Occorre pertanto definire ix prendendo in considerazione anche l'indice di thread threadIdx.x all'interno di ogni blocco nonché la dimensione di ogni blocco blockDim.x :

unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;

Consideriamo come esempio la gerarchia che sfrutta 4 thread per blocco. In questo caso la variabile threadIdx.x assume valori da 0 a 3, blockIdx.x varia da 0 a 14975 mentre blockDim.x vale 4:

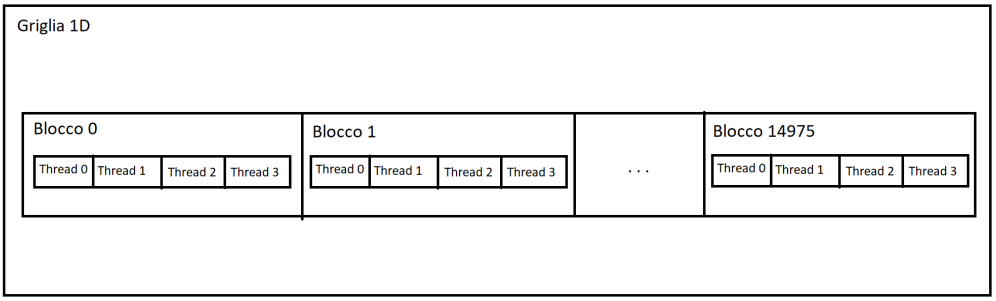

Figura 16 - Griglia di blocchi da 4 thread

La seconda modifica da apportare riguarda il programma principale, nel codice host. Qui si definisce la configurazione di lancio del kernel, impostando le variabili predefinite grid e block è possibile selezionare le diverse modalità di esecuzione del kernel:

dim3 grid ((59904 + dimx - 1) / dimx);
dim3 block (dimx);

Per fare questo ad ogni esecuzione del codice occorre variare il valore di dimx che rappresenta il numero di thread lungo la componente x di ogni blocco. Assegnando i valori menzionati in precedenza da 1 a 512 si ottengono le seguenti configurazioni di lancio del kernel:

kernel 1 <<<(59904), (1)>>> (versione originale del codice)
kernel 2 <<<(29952), (2)>>>
kernel 4 <<<(14976), (4)>>>
kernel 8 <<<(7488), (8)>>>
kernel 16 <<<(3744), (16)>>>
kernel 32 <<<(1872), (32)>>>
kernel 64 <<<(936), (64)>>>
kernel 128 <<<(468), (128)>>>
kernel 256 <<<(234), (256)>>>
kernel 512 <<<(117), (512)>>>

Moltiplicando il numero di blocchi nella griglia per il numero di thread per blocco si ottiene sempre un numero di thread complessivo pari a 59904 come preventivato.

Osserviamo inoltre che nel definire le varie configurazioni di esecuzione si è tenuto conto dei limiti imposti dal modello di programmazione CUDA e dall'hardware utilizzato circa le dimensioni della griglia e del blocco. La grandezza massima della griglia per ogni dimensione x, y e z è pari a 65535. Questa limitazione è imposta per ogni tipo di hardware. Per il dispositivo in questione è possibile utilizzare fino a un massimo di 1024 thread per blocco, se ne sono utilizzati un massimo di 512. Possiamo quindi constatare che tutte queste limitazioni sono rispettate dalle configurazioni di lancio che sono state prese in esame. La porzione di codice relativa al kernel e quella relativa all'invocazione del kernel stesso dall'host sono riportati in appendice.

Prestazioni ottenute dalla prima implementazione

Tramite lo strumento di profilazione nvprof messo a disposizione dall'ambiente CUDA è possibile visionare diversi parametri relativi all'esecuzione del codice sulla GPU NVIDIA. Il primo parametro che andiamo a prendere in considerazione è il tempo di esecuzione del kernel per le varie configurazioni di lancio. Ricordiamo che il runtime di riferimento per il codice eseguito lato host dalla CPU ammonta a 365 secondi, la versione originale del codice eseguiva gli stessi calcoli su GPU in un tempo di 66 secondi. Per i motivi spiegati in precedenza si è reso necessario considerare 59904 immagini del training set in luogo di 60000. Per questa ragione anche la configurazione ad un thread per blocco è stata rieseguita per verificarne il tempo impiegato e prenderlo come riferimento per valutare le prestazioni delle altre configurazioni di lancio del kernel. I tempi che si sono ottenuti sono riportati di seguito:

Configurazione del kernel	Runtime [s]
kernel 1 <<<(59904),(1)>>>	61.004
kernel 2 <<<(29952),(2)>>>	32.489
kernel 4 <<<(14976), (4)>>>	27.480
kernel 8 <<<(7488), (8)>>>	30.229
kernel 16 <<<(3744), (16)>>>	65.901
kernel 32 <<<(1872), (32)>>>	100.908
kernel 64 <<<(936), (64)>>>	107.197
kernel 128 <<<(468), (128)>>>	107.538
kernel 256 <<<(234), (256)>>>	108.603
kernel 512 <<<(117), (512)>>>	109.096

Possiamo notare che la configurazione che sfrutta una struttura a 4 thread per blocco impiega un tempo inferiore rispetto a quella dell'implementazione originale ad un solo thread. Anche il kernel 2 e il kernel 8 permettono di sovraperformare la configurazione di riferimento.

Per comprendere il motivo per il quale alcune gerarchie di thread permettano di ottenere tempi di esecuzione minori rispetto ad altre possiamo analizzare altri indicatori sempre tramite lo strumento nvprof da riga di comando. Uno di questi parametri è l'occupazione. Questa rappresenta il rapporto tra warp attivi e il numero massimo di warp per streaming multiprocessor:

occupazione = warps attivi / massimo numero di warps

La situazione ideale sarebbe avere un numero più elevato possibile di warp attivi in rapporto al numero totale per mantenere i nuclei del dispositivo maggiormente occupati. I risultati ottenuti sono qui mostrati con riferimento ai tempi di esecuzione:

Configurazione del kernel	Runtime [s]	Occupazione
kernel 1 <<<(59904),(1)>>>	61.004	0.498539
kernel 2 <<<(29952),(2)>>>	32.489	0.496956
kernel 4 <<<(14976), (4)>>>	27.480	0.495101
kernel 8 <<<(7488), (8)>>>	30.229	0.491185
kernel 16 <<<(3744), (16)>>>	65.901	0.491286
kernel 32 <<<(1872), (32)>>>	100.908	0.488266
kernel 64 <<<(936), (64)>>>	107.197	0.945143
kernel 128 <<<(468), (128)>>>	107.538	0.951734
kernel 256 <<<(234), (256)>>>	108.603	0.964588
kernel 512 <<<(117), (512)>>>	109.096	0.976751

Ci si aspetterebbe che un'occupazione maggiore degli streaming multiprocessor, relativa a certe configurazioni di esecuzione, produca tempi di esecuzione migliori. Si osserva invece che il kernel 512 che utilizza 512 thread per blocco esprime le prestazioni peggiori in assoluto in termini di runtime nonostante presenti il valore più alto di occupazione. Quest'ultima rende conto esclusivamente sul numero di warp simultanei per SM. Sebbene cercare di ottenerne un valore elevato sia un criterio importante per migliorare le prestazioni, vi sono altri fattori che influiscono sulle stesse.

Prendiamo ora in considerazione alcuni parametri relativi all'utilizzo della memoria del device. L'efficienza di utilizzo in lettura della memoria globale può essere rilevata tramite la metrica "memory load efficiency". Lanciando nvprof per visionare questo parametro relativo ad ogni kernel si ottiene:

Configurazione del kernel	Runtime [s]	Memory load efficiency
kernel 1 <<<(59904),(1)>>>	61.004	25.01%
kernel 2 <<<(29952),(2)>>>	32.489	25.01%
kernel 4 <<<(14976), (4)>>>	27.480	25.02%
kernel 8 <<<(7488), (8)>>>	30.229	25.02%
kernel 16 <<<(3744), (16)>>>	65.901	25.02%
kernel 32 <<<(1872), (32)>>>	100.908	25.02%
kernel 64 <<<(936), (64)>>>	107.197	25.02%
kernel 128 <<<(468), (128)>>>	107.538	25.02%
kernel 256 <<<(234), (256)>>>	108.603	25.02%
kernel 512 <<<(117), (512)>>>	109.096	25.02%

Quello che si può notare è che l'efficienza di utilizzo della memoria globale in lettura è piuttosto scarso per tutte le gerarchie di thread testate. I risultati non si discostano molto da una configurazione del kernel all'altra. Analizziamo il codice della prima parte del kernel. Qui viene implementata la differenza tra una riga della matrice di test e tutte le righe della matrice di training, i risultati vengono poi memorizzati nella matrice differenza.

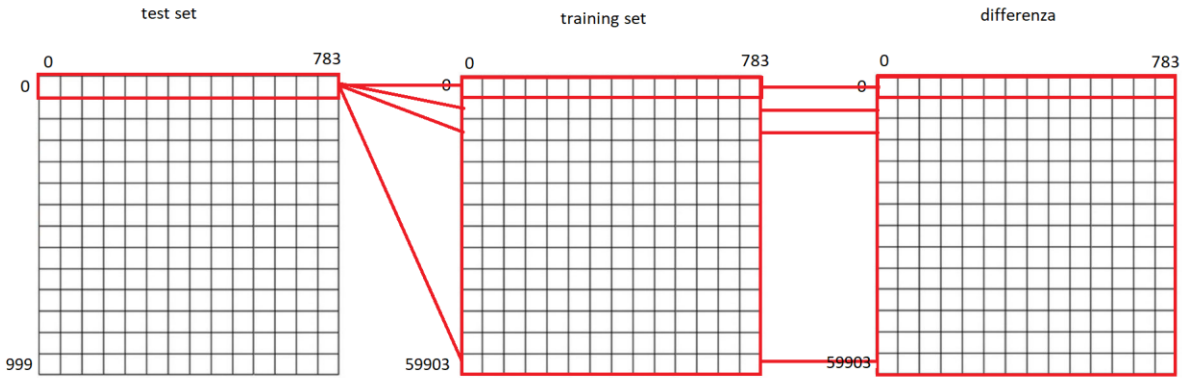

Figura 17 - Calcolo della matrice differenza

Si può verificare che per come sono memorizzati i dati in memoria e come vengono definite le operazioni, gli accessi alla memoria globale non seguono uno schema coalescente. L'istruzione eseguita da un thread per la differenza tra le righe realizza due letture in memoria, il calcolo di sottrazione e una scrittura nella memoria allocata alla matrice differenza:

*idx = ix * ny + iy;*
Mat_diff[idx] = Mat_XTest[iy] – Mat_XTrain[idx]

La variabile ix identifica a livello logico i thread tra 0 e 59903, ny rappresenta il numero di pixel di un'immagine assumendo come valore 784 e iy acquisisce un valore tra 0 e 784 tramite il ciclo for nel kernel, ad ogni passo viene incrementato di uno. Considerando come esempio il primo step del ciclo iy vale 0, possiamo riscontrare una situazione come la seguente:

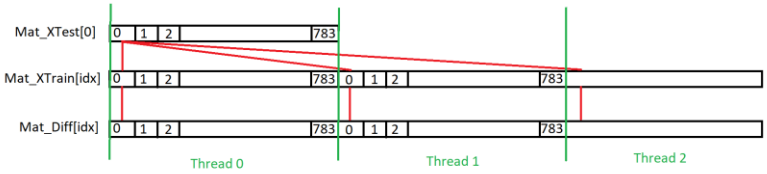

Figura 18 - Lettura memoria non coalescente

I thread sono organizzati al momento del lancio in warp da 32 thread che vanno ad effettuare letture e scritture non coalescenti in memoria globale, distanziate tra loro di 784 locazioni. Questo determina un utilizzo non efficiente della memoria globale sia in lettura ma anche in scrittura. Tale caratteristica non spiega le diverse prestazioni ottenute dalle configurazioni di lancio dei kernel perché si verifica per ognuna di esse.

La struttura con la quale vengono organizzati i thread nella griglia invece determina quante richieste di lettura della memoria avvengono simultaneamente in volo. Questo fenomeno può essere rilevato tramite profilazione osservando la velocità di lettura effettiva della DRAM del device. Lanciando da linea di comando nvprof per rilevare la metrica "dram read throughput" si ottiene:

Configurazione del kernel	Runtime [s]	DRAM read throughput [GB/s]
kernel 1 <<<(59904),(1)>>>	61.004	17.085
kernel 2 <<<(29952),(2)>>>	32.489	32.426
kernel 4 <<<(14976), (4)>>>	27.480	38.316
kernel 8 <<<(7488), (8)>>>	30.229	35.507
kernel 16 <<<(3744), (16)>>>	65.901	25.104
kernel 32 <<<(1872), (32)>>>	100.908	22.985
kernel 64 <<<(936), (64)>>>	107.197	24.282
kernel 128 <<<(468), (128)>>>	107.538	24.257
kernel 256 <<<(234), (256)>>>	108.603	24.178
kernel 512 <<<(117), (512)>>>	109.096	24.152

In questo caso si può osservare che la configurazione corrispondente al kernel 4 ovvero quella che sfrutta 4 thread per blocco è caratterizzata dal throughput più elevato. Si può verificare anche che le configurazioni a 2 e 8 thread per blocco, le cui prestazioni in termini di velocità di esecuzione si avvicinano a quella a 4 thread, sviluppano una velocità di lettura molto buona.

Le prestazioni complessive che è possibile ottenere in termini di tempo di esecuzione sono determinate sia dalla quantità di parallelismo esposto a livello di warp ma anche dal modo in cui viene sfruttata la memoria del dispositivo. Esporre più parallelismo è sempre importante al fine di aumentare la capacità computazionale. Tuttavia, come si è potuto verificare nel caso preso in esame, le operazioni di lettura dalla memoria DRAM relativa alla GPU incidono in maniera pesante sul tempo di esecuzione. Sebbene la configurazione del kernel che utilizza 4 thread per blocco non ottenga il risultato migliore in termini di occupazione degli SM dal punto di vista dell'utilizzo della memoria offre le prestazioni migliori e complessivamente consente di ottenere un miglioramento in termini di tempo di esecuzione di 2.2 volte rispetto all'implementazione originale del kernel che sfruttava la configurazione ad un thread per blocco. Questa realizzava già un miglioramento di circa 5 volte rispetto all'implementazione su CPU.

Seconda implementazione (kernel1 2D1D – kernel2 1D1D)

In questa versione del codice sono stati adottati due kernel per eseguire le istruzioni su GPU. Il primo si occupa esclusivamente del calcolo della matrice differenza, il secondo kernel realizza norma, sommatorie e calcolo della funzione di predizione come rapporto tra le sommatorie stesse. Per il primo kernel definiamo una gerarchia 2D1D, la griglia si sviluppa in una struttura di blocchi lungo le componenti x e y, ciascun blocco è monodimensionale, costituito cioè da un certo numero di thread disposti solo lungo x. Per il secondo kernel si adotta una struttura 1D1D costituita da una griglia e da blocchi entrambi ad una dimensione. Possiamo applicare la medesima procedura adottata per l'implementazione precedente al fine di esplorare differenti strutture di organizzazione e analizzare le prestazioni per ciascuna di esse. Fissiamo la dimensione dei dati considerando un numero di immagini relative al training set pari a 59904.

La parte del codice che permette di ottenere la matrice differenza realizza a livello logico un numero di calcoli pari a 59904 x 784. Per arrivare a questo si è eliminato il ciclo for che permetteva di scorrere le 784 colonne del vettore di test, della matrice di training e della matrice differenza. In questo caso per stabilire una corrispondenza tra i thread e gli elementi di dati delle due matrici occorre ridefinire non solo la variabile ix che agisce lungo la componente x delle matrici ma anche la variabile iy per la dimensione lungo y. La modifica apportata al codice all'interno del kernel rispetto alla prima implementazione è la seguente:

```
unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
unsigned int iy = blockIdx.y;
```

Dal momento che i blocchi sono ad una dimensione per identificare i thread lungo x è necessario definire ix così mentre lungo y il numero di thread corrisponde col numero di blocchi lungo y. Abbiamo un solo thread per blocco lungo y e un numero variabile di thread per blocco lungo x in relazione alla configurazione del kernel considerata. In questo caso occorre inoltre effettuare un controllo anche lungo la dimensione y per fare in modo che i thread non eccedano la dimensione delle tre strutture di dati lungo y oltre che lungo x:

```
if (ix < nx && iy < ny)
```

La variabile nx rappresenta il numero complessivo di immagini del training set utilizzate (59904) e ny il numero di pixel di un'immagine (784).

Consideriamo come esempio la gerarchia che sfrutta 4 thread per blocco. In questo caso la variabile threadIdx.x assume valori da 0 a 3, blockIdx.x varia da 0 a 14975 mentre blocDim.x vale 4. Lungo y blockIdx.y varia da 0 a 783. Tramite la variabile idx i thread vengono mappati sui corrispettivi elementi di dati in memoria globale:

*idx = ix * ny + iy;*

Si applica la medesima procedura descritta per l'implementazione precedente circa il dimensionamento della griglia sulla base della dimensione dei dati utilizzati e del numero di thread per blocco scelti. Anche in questo caso si è scelto di testare strutture a 1, 2, 4, 8, 16, 32, 64, 128, 256 e 512 thread per blocco. Occorre quindi definire le impostazioni per invocare il kernel dal programma principale nel codice host. Impostando le variabili predefinite grid e block è possibile selezionare le diverse modalità di esecuzione del kernel:

dim3 grid ((59904 + dimx -1) / dimx, 784);
dim3 block (dimx);

Ad ogni lancio del codice la variabile dimx va impostata con i valori citati precedentemente ottenendo le configurazioni del kernel che seguono:

kernel 1A <<<(59904, 784), (1)>>>
kernel 2A <<<(29952, 784), (2)>>>
kernel 4A <<<(14976, 784), (4)>>>
kernel 8A <<<(7488, 784), (8)>>>
kernel 16A <<<(3744, 784), (16)>>>
kernel 32A <<<(1872, 784), (32)>>>
kernel 64A <<<(936, 784), (64)>>>
kernel 128A <<<(468, 784), (128)>>>
kernel 256A <<<(234, 784), (256)>>>
kernel 512A <<<(117, 784), (512)>>>

Il secondo kernel esegue la restante parte dei calcoli. Questo utilizza un totale di 59904 thread organizzati in una struttura gerarchica di tipo 1D1D già adottata nella prima implementazione del codice. La variabile ix all'interno del codice del kernel è definita come segue:

*unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;*

Nel programma principale si definiscono:

dim3 grid ((59904 + dimx -1) / dimx);
dim3 block (dimx);

La variabile dimx assume anche in questo caso i valori da 1 a 512 ottenendo le configurazioni di seguito elencate:

kernel 1B <<<(59904), (1)>>>
kernel 2B <<<(29952), (2)>>>
kernel 4B <<<(14976), (4)>>>
kernel 8B <<<(7488), (8)>>>
kernel 16B <<<(3744), (16)>>>
kernel 32B <<<(1872), (32)>>>
kernel 64B <<<(936), (64)>>>
kernel 128B <<<(468), (128)>>>
kernel 256B <<<(234), (256)>>>
kernel 512B <<<(117), (512)>>>

Le porzioni di codice relative ai kernel e quella relativa all'invocazione dei kernel, per questa seconda implementazione, sono riportate in appendice.

Prestazioni ottenute dalla seconda implementazione

Tramite lo strumento di profilazione nvprof si sono raccolti i dati circa i tempi di esecuzione, l'occupazione e la velocità di lettura della memoria del dispositivo per il primo e per il secondo kernel. I risultati relativi al primo kernel per il calcolo della matrice differenza sono riportati nella tabella a seguito:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 1A <<<(59904, 784), (1)>>>	80.5888	0.401801	17.492
kernel 2A <<<(29952, 784), (2)>>>	79.7016	0.467069	17.673
kernel 4A <<<(14976, 784), (4)>>>	79.6074	0.487878	17.689
kernel 8A <<<(7488, 784), (8)>>>	79.5617	0.494879	17.703
kernel 16A <<<(3744, 784), (16)>>>	79.3418	0.497423	17.751
kernel 32A <<<(1872, 784), (32)>>>	79.0579	0.498663	17.813
kernel 64A <<<(936, 784), (64)>>>	79.0135	0.764664	17.822
kernel 128A <<<(468, 784), (128)>>>	79.0621	0.838229	17.812
kernel 256A <<<(234, 784), (256)>>>	79.0835	0.816298	17.806
kernel 512A <<<(117, 512), (512)>>>	79.0763	0.758771	17.801

La configurazione del kernel che produce il tempo di esecuzione inferiore è quella corrispondente all'utilizzo di 64 thread per blocco. Possiamo anche osservare che la differenza in termini di runtime tra un caso e l'altro assume un valore abbastanza piccolo.

Il kernel più veloce inoltre non corrisponde a quello con l'occupazione più elevata. Sebbene il numero di warp attivi sugli streaming multiprocessor in rapporto a quelli totali raggiunga un buon valore per il kernel 64A, quello che sfrutta 128 thread per blocco presenta un'occupazione più elevata.

Si può inoltre notare una correlazione tra la velocità effettiva di lettura della memoria globale e i tempi di esecuzione, la configurazione del kernel più veloce è quella che presenta il throughput in lettura della DRAM più elevato.

Analizziamo ora le metriche raccolte per il secondo kernel che si occupa del calcolo della norma, esponenziale, sommatorie e rapporto tra queste ultime. La tabella mostra i risultati ottenuti:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 1B <<<(59904),(1)>>>	50.1258	0.498524	12.066
kernel 2B <<<(29952),(2)>>>	25.2292	0.498782	27.904
kernel 4B <<<(14976), (4)>>>	14.8165	0.496326	47.305
kernel 8B <<<(7488), (8)>>>	17.2782	0.493151	40.655
kernel 16B <<<(3744), (16)>>>	17.7810	0.490117	39.805
kernel 32B <<<(1872), (32)>>>	22.9765	0.488086	39.362
kernel 64B <<<(936), (64)>>>	32.3527	0.964452	39.130
kernel 128B <<<(468), (128)>>>	32.2967	0.972065	39.191
kernel 256B <<<(234), (256)>>>	32.3794	0.969312	39.149
kernel 512B <<<(117), (512)>>>	32.3293	0.972894	39.210

Per questo kernel che sfrutta gerarchia 1D1D la struttura che offre prestazioni migliori è quella a 4 thread per blocco. In questo caso l'utilizzo di varie configurazioni di esecuzione mostra prestazioni complessive molto diverse tra loro. Rispetto all'utilizzo di blocchi a un solo thread il tempo di esecuzione è inferiore di circa 3 volte per il kernel 4B.

Ancora una volta la condizione più performante non è determinata da un valore di occupazione tanto elevato, è il kernel a 512 thread per bocco ad ottenere il valore migliore.

La velocità di lettura dalla memoria si dimostra anche in questo caso fattore incisivo per ottenere il tempo migliore.

Scegliendo la gerarchia a 64 thread per blocco per il primo kernel e quella a 4 thread per blocco per il secondo kernel si ottengono le prestazioni migliori in termini di runtime per questa implementazione. Sommando i tempi ottenuti risulta che il calcolo su GPU può essere effettuato in circa 93 secondi. Come riferimento il codice eseguito su CPU impiega 365 secondi mentre la versione originale su GPU richiede 61 secondi. Non si è ottenuto quindi un miglioramento delle prestazioni rispetto al codice originale fatto girare su GPU.

Terza implementazione (kernel1 2D2D – kernel2 1D1D)

Per questa implementazione si utilizza l'impostazione vista per il caso descritto al passo precedente. Vengono utilizzati due kernel, il primo per il calcolo della matrice differenza e il secondo per i restanti calcoli. Per esplorare la possibilità di ottenere prestazioni migliori, per il primo kernel si adotta una gerarchia 2D2D. Questo equivale ad utilizzare una griglia composta da blocchi che si sviluppano lungo la componente x e lungo la componente y ma prevede anche che i blocchi siano costituiti da thread lungo entrambe le componenti x e y. Il secondo kernel è il medesimo dell'implementazione precedente caratterizzato da gerarchia 1D1D.

All'interno del codice relativo al primo kernel occorre ridefinire le variabili ix e iy che generano il mapping della griglia di thread sulle strutture di dati come segue:

*unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;*
*unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;*

Mentre per l'invocazione del kernel lato host, nel programma principale, le variabili predefinite grid e block sono state impostate nel seguente modo:

dim3 grid ((59904 + dimx -1) / dimx, (784 + dimy - 1) / dimy);
dim3 block (dimx, dimy);

In questo caso la variabile dimx consente di definire il numero di thread per blocco lungo x, la variabile dimy il numero di thread per blocco lungo y. Il numero totale di thread all'interno di un blocco bidimensionale è dato dal prodotto del valore assunto dalle due variabili. Si è deciso di assegnare alle variabili dimx e dimy le seguenti coppie di valori: (1, 1), (2, 2), (4, 4), (8, 8), (16, 16), (32, 16), (64, 16), (128, 8), (256, 4). Nella scelta delle coppie si è tenuto conto della limitazione sul numero massimo di thread per blocco pari a 1024, il prodotto degli elementi di ciascuna coppia non supera in nessun caso questo limite.

Le configurazioni di lancio del kernel che si ottengono sono di seguito riportate:

kernel 1A <<<(59904, 784), (1, 1)>>>
kernel 4A <<<(29952, 392), (2, 2)>>>
kernel 16A <<<(14976, 196), (4, 4)>>>
kernel 64A <<<(7488, 98), (8, 8)>>>
kernel 256A <<<(3744, 49), (16, 16)>>>
kernel 512A <<<(1872, 49), (32, 16)>>>
kernel 1024A <<<(936, 49), (64, 16)>>>
kernel 1024A <<<(468, 98), (128, 8)>>>
kernel 1024A <<<(234, 196), (256, 4)>>>

Il secondo kernel è lo stesso utilizzato nella seconda implementazione, riportiamo le configurazioni di lancio del kernel:

kernel 1B <<<(59904), (1)>>>
kernel 2B <<<(29952), (2)>>>
kernel 4B <<<(14976), (4)>>>
kernel 8B <<<(7488), (8)>>>
kernel 16B <<<(3744), (16)>>>
kernel 32B <<<(1872), (32)>>>
kernel 64B <<<(936), (64)>>>
kernel 128B <<<(468), (128)>>>
kernel 256B <<<(234), (256)>>>
kernel 512B <<<(117), (512)>>>

Le porzioni di codice relative ai kernel e quella relativa all'invocazione dei kernel, per questa terza implementazione, sono riportate in appendice.

Prestazioni ottenute dalla terza implementazione

Nuovamente tramite strumento di profilazione sono state raccolte le metriche relative al primo kernel 2D2D. Queste sono riportate di seguito:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 1A <<<(59904, 784), (1, 1)>>>	80.5888	0.401801	17.492
kernel 4A <<<(29952, 392), (2, 2)>>>	39.8444	0.467106	17.676
kernel 16A <<<(14976, 196), (4, 4)>>>	19.5504	0.489233	19.007
kernel 64A <<<(7488, 98), (8, 8)>>>	12.1305	0.802298	28.904
kernel 256A <<<(3744, 49), (16, 16)>>>	6.1837	0.839385	56.403
kernel 512A <<<(1872, 49), (32, 16)>>>	7.00658	0.837853	49.243
kernel 1024A <<<(936, 49), (64, 16)>>>	7.82139	0.845956	46.482
kernel 1024A <<<(468, 98), (128, 8)>>>	13.3314	0.768125	26.569
kernel 1024A <<<(234, 196),(256, 4)>>>	21.1364	0.730705	17.484

I kernel che presentano struttura dei thread per blocco (16, 16), (32, 16) e (64,16) utilizzano rispettivamente 256, 512 e 1024 thread all'interno di un blocco. Queste configurazioni sono in grado di esprimere un runtime ottimo. Presentano tutte un livello di occupazione degli streaming multiprocessor molto buono.
Il kernel 256A esprime il secondo valore di occupazione più elevato tra le configurazioni testate.
Per quanto riguarda la velocità effettiva di lettura dalla memoria del dispositivo la configurazione a 256 thread per blocco conferma ancora una volta che questo parametro influisce molto sulle prestazioni complessive che è possibile ottenere dal kernel. Il valore associato a tale configurazione è il più elevato e pari a 56,4 GB/s. Osserviamo inoltre che aver scelto la dimensione lungo y in termini di thread in un blocco pari a 16 determina un effetto sulle prestazioni in lettura dalla memoria rilevante.

Di seguito riportiamo anche i risultati ottenuti per il secondo kernel:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 1B <<<(59904),(1)>>>	50.1258	0.498524	12.066
kernel 2B <<<(29952),(2)>>>	25.2292	0.498782	27.904
kernel 4B <<<(14976), (4)>>>	14.8165	0.496326	47.305
kernel 8B <<<(7488), (8)>>>	17.2782	0.493151	40.655
kernel 16B <<<(3744), (16)>>>	17.7810	0.490117	39.805
kernel 32B <<<(1872), (32)>>>	22.9765	0.488086	39.362
kernel 64B <<<(936), (64)>>>	32.3527	0.964452	39.130
kernel 128B <<<(468), (128)>>>	32.2967	0.972065	39.191
kernel 256B <<<(234), (256)>>>	32.3794	0.969312	39.149
kernel 512B <<<(117), (512)>>>	32.3293	0.972894	39.210

L'analisi dei risultati per questo kernel è già stata effettuata per la seconda implementazione del codice dal momento che si è utilizzato esattamente lo stesso codice.

In definitiva considerando la combinazione del kernel 256A con il kernel 4B, per eseguire tutte le operazioni necessarie ad implementare l'algoritmo, si ottiene un tempo di esecuzione complessivo pari a 21 secondi. Questo è il risultato migliore che si è potuto ottenere utilizzando una gerarchia 2D2D per il primo kernel e 1D1D per il secondo dalle varie configurazioni di prova.

Prendiamo in considerazione le configurazioni migliori in termini di tempi di esecuzione per le tre implementazioni e analizziamone i risultati. Per la prima implementazione il kernel che ha permesso di ottenere le prestazioni complessivamente migliori è quello che utilizzava 4 thread per blocco. Il codice è strutturato all'interno di un unico kernel con gerarchia dei thred 1D1D:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 4 <<<(14976), (4)>>>	27.480	0.495101	38.316

Per la seconda implementazione si è diviso il codice eseguito su GPU in due porzioni esplorando una struttura 2D1D per la griglia nella prima parte e una gerarchia 1D1D per la seconda:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 64A <<<(936, 784), (64)>>>	79.0135	0.764664	17.822
kernel 4B <<<(14976), (4)>>>	14.8165	0.496326	47.305

Il runtime complessivo del codice eseguito su GPU in questo caso è dato dalla somma dei tempi ottenuti dai due kernel ed è pari a 93.83 secondi.

La terza implementazione prevedeva lo stesso approccio della seconda ma applicando una struttura 2D2D per il primo kernel, lasciando invariato il secondo kernel:

Configurazione del kernel	Runtime [s]	Occupazione	DRAM read throughput [GB/s]
kernel 256A <<<(3744, 49), (16, 16)>>>	6.1837	0.839385	56.403
kernel 4B <<<(14976), (4)>>>	14.8165	0.496326	47.305

In questo caso il runtime totale 21 secondi.

La miglior configurazione relativa alla seconda implementazione mostra un tempo di esecuzione complessivo del codice su GPU di 93.83 secondi. Questo è addirittura superiore al runtime della versione originale del codice pari a 61 secondi.

La prima implementazione presenta una struttura con la quale sono organizzati blocchi nella griglia e thread nei blocchi di tipo 1D1D analoga a quella utilizzata per la versione di riferimento. Nella versione originale del codice però si utilizzava un solo thread per blocco. Gerarchie con troppi pochi thread per blocco non permettono di massimizzare l'utilizzo di registri e altre risorse hardware a disposizione di un blocco di thread assegnato ad uno streaming multiprocessor per l'esecuzione. Inoltre la velocità di lettura dalla memoria del device risultava piuttosto scarsa. L'impiego di un numero pari a 4 thread per blocco ha permesso in questo caso un tempo di esecuzione di circa 27 secondi comportando un miglioramento in termini di runtime di 2.2 volte rispetto al codice di riferimento eseguito su GPU.

Complessivamente la terza implementazione è quella che offre il risultato migliore in assoluto. L'utilizzo della gerarchia 2D2D per il primo kernel ha consentito un'ottima occupazione delle unità di calcolo e un throughput in lettura dalla memoria DRAM elevato. Il secondo kernel che sfrutta una struttura 1D1D potrebbe essere ulteriormente migliorato, magari scindendolo in altri due kernel ed esplorando per questi varie strutture con cui sono organizzati i thread. Il runtime complessivo in questo caso è di 21 secondi e costituisce un miglioramento di circa 3 volte rispetto al codice originale fatto girare su GPU che a sua volta sovraperformava la versione implementata su CPU di circa 5 volte.

CONCLUSIONI

L'obiettivo di questo elaborato era quello di ottimizzare il codice CUDA per vedere se fosse possibile eseguire l'algoritmo di classificazioni di immagini su GPU in un tempo inferiore a quello espresso dal codice preso come riferimento. Il calcolo su sistemi eterogenei è uno strumento molto potente per poter elaborare grandi dataset messi a disposizione per applicazioni di Machine Learning. Il codice sviluppato per l'articolo preso come riferimento consentiva di dimostrare come sfruttando la capacità computazionale della GPU si riuscisse ad eseguire i calcoli necessari per l'esecuzione dell'algoritmo, basato su tecnica di Monte Carlo, in un tempo inferiore rispetto a quanto possibile su CPU. Analizzando le caratteristiche hardware del dispositivo fornito nonché il modello di esecuzione messo a disposizione dalle librerie CUDA si è riusciti a ottenere un ulteriore miglioramento delle prestazioni in termini di runtime. In particolare l'approccio che ha permesso di ottenere tale risultato è stato quello di esplorare varie gerarchie mediante le quali sono organizzati i thread nella griglia di esecuzione del kernel. Questo ha consentito di occupare le unità di calcolo fondamentali del dispositivo in misura maggiore e di ottenere un rendimento migliore circa l'utilizzo della memoria del dispositivo. In definitiva il codice nella configurazione di esecuzione più veloce tra quelle testate sovraperforma la versione del codice di riferimento fatta girare su GPU di un fattore pari a 3.

RINGRAZIAMENTI

Ringrazio il Professor Claudio Turchetti, Relatore di questo elaborato di Tesi, per avermi dato la possibilità di approcciarmi per la prima volta ad applicazioni inerenti al Machine Learning, permettendomi di sviluppare una conoscenza di base su questi argomenti di grande interesse per me. Questo lavoro mi ha permesso di approfondire in modo sostanzioso argomenti riguardanti la programmazione in generale e di acquisire conoscenze relative ad un linguaggio di programmazione del tutto nuovo per me, CUDA C, per sfruttare il calcolo parallelo su GPU. Ringrazio di cuore la Correlatrice di questo elaborato, la Professoressa Laura Falaschetti, per avermi proposto questo lavoro di Tesi e di tirocinio, per il supporto tecnico fornito grazie alla sua conoscenza approfondita circa la programmazione e il mondo del Machine Learning. La ringrazio in modo particolare per avermi indirizzato a questo lavoro e per l'aiuto alla risoluzione di problemi incontrati durante questo percorso. Ringrazio inoltre i miei genitori per avermi sostenuto sempre, aiutato nei momenti più difficili e per la speranza che hanno riposto in me. Un ringraziamento va anche a mia sorella Alessandra e ai miei amici per il loro sostegno e per essermi stati vicini.

APPENDICE

Versione originale del codice corretta

```
#include "common.h"          // CUDA header file
#include <cuda_runtime.h>    // CUDA runtime library
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <string>
#include <cmath>
#include <vector>
#include "math.h"            // CUDA math apis

using namespace std;

// Funzione per caricamento dei dati da file

void initialFileData(string filename, double *ip, const int size)
{
  int x;
  ifstream in(filename.c_str());

  if (!in) {
    cout << "Cannot open file.\n";
    return;
  }

  for (x = 0; x < size; x++) {
      in >> ip[x];
  }

  in.close();
  return;
}

// Funzione che realizza operazioni matematiche lato host

void montecarloMatrixOnHost(double *Mat_XTrain, double *Vect_YTrain, double
*Mat_XTest, double *Vect_diff, double *Vect_predict, int kernel_value, int
num_rows_train, int num_rows_test, int num_columns)
{
    int N = num_rows_train;     // numero righe train
    int M = num_rows_test;      // numero righe test
    int P = num_columns;        // numero colonne
    double *i_xtrain = Mat_XTrain;
    double *i_xtest = Mat_XTest;
    double *i_ytrain = Vect_YTrain;
    double a = (double)(1/(double)kernel_value);
    double norm_value = 0;
    double k_exp = 0;
    double pow_value = 0;
    for (int j = 0; j < M; j++) {

        double fsum = 0;
        double csum = 0;
        norm_value = 0;
        k_exp = 0;
        pow_value = 0;

        for (int i = 0; i < N; i++) {
```

```
                for (int k = 0; k < P; k++) {
                    Vect_diff[k] = i_xtest[k] - i_xtrain[k];
                    norm_value += pow(Vect_diff[k],2);
              }
            norm_value = sqrt(norm_value);
            pow_value = 2*pow(a,2);
            k_exp = exp(-norm_value/pow_value);

            csum = csum + k_exp;
            fsum = fsum + i_ytrain[i] * k_exp;

            i_xtrain += P;

          }

        Vect_predict[j] = round((double)(fsum/csum));

        i_xtest += P;
        i_xtrain = Mat_XTrain;

    }

    std::cout << "Host Done!" << std::endl;

    return;
}


// Kernel che realizza operazioni matematiche lato device

__global__ void montecarloMatrixOnGPU1D(double *Mat_XTrain, double *Vect_YTrain,
double *Mat_XTest, double *Mat_Diff, double *Vect_Predict,
         int kernel_value, int nx, int ny, double *norm_value, double *exp_value,
double *csum, double *fsum, int test_row)
{
   unsigned int ix =  blockIdx.x;
   double * i_xtest = Mat_XTest;
   i_xtest += ny * test_row;

   if (ix < nx ) {
        for (int iy = 0; iy < ny; iy++)
        {
             int idx = ix * ny + iy;
            Mat_Diff[idx] = i_xtest[iy] - Mat_XTrain[idx];
        }
       norm_value[ix] = norm(double(ny), (double*)&Mat_Diff[ix*784]);
       double pow_value = 2*pow((double)(1/(double)kernel_value),(double)2);
       exp_value[ix] = exp(-norm_value[ix]/pow_value);

    atomicAdd((double*)&csum[test_row], exp_value[ix]);
    atomicAdd((double*)&fsum[test_row], (Vect_YTrain[ix]*exp_value[ix]));
    }

    Vect_Predict[test_row] = __double2int_rn(fsum[test_row]/csum[test_row]);


}



// Funzione per il check dei risultati ottenuti da host e device

void checkResult(double *hostRef, double *gpuRef, const int N)
```

```
{
    double epsilon = 1.0E-8;
    bool match = 1;

    for (int i = 0; i < N; i++)
    {
        if (abs(hostRef[i] - gpuRef[i]) > epsilon)
        {
            match = 0;
            printf("host %f gpu %f\n", hostRef[i], gpuRef[i]);
            break;
        }
    }

    if (match)
        printf("Arrays match.\n\n");
    else
        printf("Arrays do not match.\n\n");
}



int main(int argc, char **argv)
{
    printf("%s Starting...\n", argv[0]);

    // Impostazioni del device
    int dev = 0;
    cudaDeviceProp deviceProp;
    CHECK(cudaGetDeviceProperties(&deviceProp, dev));
    printf("Using Device %d: %s\n", dev, deviceProp.name);
    CHECK(cudaSetDevice(dev));

    // Definizione delle dimensioni delle strutture di dati
    int num_rows_train = 60000; // Numero di immagini del training set
    int num_rows_test = 1000;     // Numero di immagini del test set
    int num_columns = 784;        // Numero pixel di un'immagine
    int nxy_xtrain = num_rows_train * num_columns;
    int nBytes_XTrain = nxy_xtrain * sizeof(double);
    int nxy_xtest = num_rows_test * num_columns;
    int nBytes_XTest = nxy_xtest * sizeof(double);
    int nxy_ytrain = num_rows_train * 1;
    int nBytes_YTrain = nxy_ytrain * sizeof(double);
    int nxy_ytest = num_rows_test * 1;
    int nBytes_YTest = nxy_ytest * sizeof(double);


    // Allocazione dinamica della memoria tramite malloc lato host
    double *h_XTrain, *h_XTest, *h_YTrain, *h_YTest, *h_diff, *hostRef, *gpuRef,
*h_predict, *h_predict_gpu_copy, *h_norm, *h_exp, *h_csum, *h_fsum;
    h_XTrain = (double *)malloc(nBytes_XTrain);
    h_XTest = (double *)malloc(nBytes_XTest);
    h_YTrain = (double *)malloc(nBytes_YTrain);
    h_YTest = (double *)malloc(nBytes_YTest);
    h_diff = (double *)malloc(nBytes_XTrain);
    h_predict = (double *)malloc(nBytes_YTest);
    h_predict_gpu_copy = (double *)malloc(nBytes_YTest);
    h_norm = (double *)malloc(nBytes_YTrain);
    h_exp = (double *)malloc(nBytes_YTrain);
    h_csum = (double *)malloc(nBytes_YTest);
    h_fsum = (double *)malloc(nBytes_YTest);
    hostRef = (double *)malloc(nBytes_YTest);
    gpuRef = (double *)malloc(nBytes_YTest);
```

```
    std::cout << "Malloc done!" << std::endl;

    // Inizializzazione dei dati lato host
    double iStart = seconds();
    initialFileData("X_train.txt", h_XTrain, nxy_xtrain);
    initialFileData("X_test.txt", h_XTest, nxy_xtest);
    initialFileData("y_train.txt", h_YTrain, nxy_ytrain);
    initialFileData("y_test.txt", h_YTest, nxy_ytest);
    std::cout << "Initializa data done!" << std::endl;

    double iElaps = seconds() - iStart;
    printf("initialize matrix elapsed %f sec\n", iElaps);

    memset(hostRef, 0, nBytes_YTest);
    memset(gpuRef, 0, nBytes_YTest);
    memset(h_csum, 0, nBytes_YTest);
    memset(h_fsum, 0, nBytes_YTest);

    // Lancio della funzione di calcolo su CPU
    iStart = seconds();
    montecarloMatrixOnHost(h_XTrain, h_YTrain, h_XTest, h_diff, h_predict, 10,
num_rows_train, num_rows_test, num_columns);
    iElaps = seconds() - iStart;
    printf("\n montecarloMatrixOnHost elapsed %f sec\n\n", iElaps);
    printf("CPU first ypredict result = %f\n", *h_predict);


    // Copia dei risultati lato host per il check con i risultati del device
    memcpy(hostRef, h_predict, nBytes_YTest);

    // ML: Calcolo accuratezza algoritmo CPU
    int counter_cpu = 0;
    int test_size = num_rows_test;
    for(int i = 0; i < test_size; i++) {

        if(h_YTest[i] == hostRef[i]) {
            counter_cpu++;
           }
    }
    float accuracy_cpu = float((float)counter_cpu/(float)test_size);
    float accuracy_percentage_cpu = 100*accuracy_cpu;
    printf("\nACCURATEZZA CALCOLO CPU: %f [%%]\n\n", accuracy_percentage_cpu);


    // Allocazione dinamica della memoria tramite cudaMalloc lato device
    std::cout << "Cuda malloc starting.." << std::endl;
    double *d_MatXTrain, *d_MatXTest, *d_VectYTrain, *d_VectYTest, *d_VectDiff,
*d_VectPredict, *d_VectNorm, *d_VectExp, *d_Csum, *d_Fsum;
    CHECK(cudaMalloc((void **)&d_MatXTrain, nBytes_XTrain));
    CHECK(cudaMalloc((void **)&d_MatXTest, nBytes_XTest));
    CHECK(cudaMalloc((void **)&d_VectYTrain, nBytes_YTrain));
    CHECK(cudaMalloc((void **)&d_VectYTest, nBytes_YTest));
    CHECK(cudaMalloc((void **)&d_VectDiff, nBytes_XTrain));
    CHECK(cudaMalloc((void **)&d_VectPredict, nBytes_YTest));
    CHECK(cudaMalloc((void **)&d_VectNorm, nBytes_YTrain));
    CHECK(cudaMalloc((void **)&d_VectExp, nBytes_YTrain));
    CHECK(cudaMalloc((void **)&d_Csum, nBytes_YTest));
    CHECK(cudaMalloc((void **)&d_Fsum, nBytes_YTest));
    std::cout << "Cuda malloc done!" << std::endl;


    // Trasferimento dati dall'host al device
    CHECK(cudaMemcpy(d_MatXTrain, h_XTrain, nBytes_XTrain, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_MatXTest, h_XTest, nBytes_XTest, cudaMemcpyHostToDevice));
```

```
    CHECK(cudaMemcpy(d_VectYTrain, h_YTrain, nBytes_YTrain, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_VectYTest, h_YTest, nBytes_YTest, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_Csum, h_csum, nBytes_YTest, cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy(d_Fsum, h_fsum, nBytes_YTest, cudaMemcpyHostToDevice));
    std::cout << "Cuda memcopy done!" << std::endl;


    // Invocazione del kernel dal lato host
    int dimx = 1;   // Numero di thread per blocco
    dim3 block(dimx);
    dim3 grid(num_rows_train);
    int nx = num_rows_train;
    int ny = num_columns;

    iStart = seconds();
    for(int i = 0; i < num_rows_test; i++)  {
       montecarloMatrixOnGPU1D<<<grid, block>>>(d_MatXTrain, d_VectYTrain, d_MatXTest,
d_VectDiff, d_VectPredict, 10, nx, ny, d_VectNorm, d_VectExp, d_Csum, d_Fsum, i);
    }
    CHECK(cudaDeviceSynchronize());
    iElaps = seconds() - iStart;
    printf("\n montecarloMatrixOnGPU1D <<<(%d,%d), (%d,%d)>>> elapsed %f sec \n",
grid.x, grid.y, block.x, block.y, iElaps);

    CHECK(cudaMemcpy(h_XTrain, d_MatXTrain, nBytes_XTrain, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_XTest, d_MatXTest, nBytes_XTest, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_diff, d_VectDiff, nBytes_XTrain, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_norm, d_VectNorm, nBytes_YTrain, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_exp, d_VectExp, nBytes_YTrain, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_csum, d_Csum, nBytes_YTest, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_fsum, d_Fsum, nBytes_YTest, cudaMemcpyDeviceToHost));
    CHECK(cudaMemcpy(h_predict_gpu_copy, d_VectPredict, nBytes_YTest,
cudaMemcpyDeviceToHost));

    // Check kernel error
    CHECK(cudaGetLastError());

    // Copia i risultati del kernel lato host
    CHECK(cudaMemcpy(gpuRef, d_VectPredict, nBytes_YTest, cudaMemcpyDeviceToHost));


    //ML: calcolo accuratezza algoritmo GPU
    int counter_gpu = 0;
    int test_size_gpu = num_rows_test;
    for(int i = 0; i < test_size_gpu; i++) {

          if(h_YTest[i] == h_predict_gpu_copy[i]){
           counter_gpu++;
        }
    }
    float accuracy_gpu = float((float)counter_gpu/(float)test_size_gpu);
    float accuracy_percentage_gpu = 100*accuracy_gpu;
    printf("\nACCURATEZZA CALCOLO GPU: %f [%%]\n\n", accuracy_percentage_gpu);


    // Check risultati device
    checkResult(hostRef, gpuRef, nxy_ytest);

    // Libera la memoria globale del device
    CHECK(cudaFree(d_MatXTrain));
    CHECK(cudaFree(d_MatXTest));
    CHECK(cudaFree(d_VectYTrain));
    CHECK(cudaFree(d_VectYTest));
    CHECK(cudaFree(d_VectDiff));
```

```
    CHECK(cudaFree(d_VectPredict));
    CHECK(cudaFree(d_VectNorm));
    CHECK(cudaFree(d_VectExp));
    CHECK(cudaFree(d_Csum));
    CHECK(cudaFree(d_Fsum));

    // Libera la memoria lato host
    free(hostRef);
    free(gpuRef);
    free(h_XTrain);
    free(h_XTest);
    free(h_YTrain);
    free(h_YTest);
    free(h_diff);
    free(h_predict);
    free(h_predict_gpu_copy);
    free(h_norm);
    free(h_exp);
    free(h_csum);
    free(h_fsum);

    // reset device
    CHECK(cudaDeviceReset());

    return (0);
}
```

Codice kernel prima implementazione

```
// Kernel che realizza operazioni matematiche lato device

__global__ void montecarloMatrixOnGPU1D(double *Mat_XTrain, double *Vect_YTrain,
double *Mat_XTest, double *Mat_Diff, double *Vect_Predict, int kernel_value, int nx,
int ny, double *norm_value, double *exp_value, double *csum, double *fsum, int
test_row)
{
   unsigned int ix =  threadIdx.x + blockIdx.x * blockDim.x;

   double * i_xtest = Mat_XTest;
   i_xtest += ny * test_row;

   if (ix < nx ) {
        for (int iy = 0; iy < ny; iy++)
        {
            int idx = ix * ny + iy;
           Mat_Diff[idx] = i_xtest[iy] - Mat_XTrain[idx];
        }
       norm_value[ix] = norm(double(ny), (double*)&Mat_Diff[ix*784]);

       double pow_value = 2*pow((double)(1/(double)kernel_value),(double)2);
       exp_value[ix] = exp(-norm_value[ix]/pow_value);

       atomicAdd((double*)&csum[test_row], exp_value[ix]);
       atomicAdd((double*)&fsum[test_row], (Vect_YTrain[ix]*exp_value[ix]));
    }

    Vect_Predict[test_row] = __double2int_rn(fsum[test_row]/csum[test_row]);


}
```

Invocazione del kernel prima implementazione

```
// Invocazione del kernel lato host

int dimx = 1;      // Numero di thread per blocco
dim3 block(dimx);
dim3 grid((num_rows_train + dimx - 1) / dimx);
int nx = num_rows_train;
int ny = num_columns;

iStart = seconds();
for(int i = 0; i < num_rows_test; i++)  {
       montecarloMatrixOnGPU1D<<<grid, block>>>(d_MatXTrain, d_VectYTrain, d_MatXTest,
       d_VectDiff, d_VectPredict, 10, nx, ny, d_VectNorm, d_VectExp, d_Csum, d_Fsum,
       i);
    }
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("\n montecarloMatrixOnGPU1D <<<(%d,%d), (%d,%d)>>> elapsed %f sec \n", grid.x,
grid.y, block.x, block.y, iElaps);
```

Codice dei kernel seconda implementazione

```
// Kernel1: realizza il calcolo della matrice differenza lato device

__global__ void montecarloStep1OnGPU(double *Mat_XTrain, double *Mat_XTest, double
*Mat_Diff, int nx, int ny, int test_row)
{
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned int iy = blockIdx.y;
   int idx = ix * ny + iy;

   double * i_xtest = Mat_XTest;
   i_xtest += ny * test_row;

   if (ix < nx && iy < ny) {

        Mat_Diff[idx] = i_xtest[iy] - Mat_XTrain[idx];
       }
 }



// Kernel2: realizza i restanti calcoli lato device

__global__ void montecarloStep2OnGPU(double *Mat_Diff, double *Vect_YTrain, double
*Vect_Predict, int kernel_value, int nx, int ny, double *norm_value, double
*exp_value, double *csum, double *fsum, int test_row)
{
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;

   if (ix < nx) {
        norm_value[ix] = norm(double(ny), (double*)&Mat_Diff[ix*784]);

        double pow_value = 2*pow((double)(1/(double)kernel_value),(double)2);
        exp_value[ix] = exp(-norm_value[ix]/pow_value);

        atomicAdd((double*)&csum[test_row], exp_value[ix]);
        atomicAdd((double*)&fsum[test_row], (Vect_YTrain[ix]*exp_value[ix]));
    }

    Vect_Predict[test_row] = __double2int_rn(fsum[test_row]/csum[test_row]);
}
```

Invocazione dei kernel seconda implementazione

```
// Invocazione dei kernel lato host

int dimx1 = 1;      // Numero di thread per blocco lungo x
dim3 block1(dimx1);
dim3 grid1((num_rows_train + dimx1 - 1) / dimx1, 784);

int dimx2 = 1;      // Numero di thread per blocco lungo x
dim3 block2(dimx2);
dim3 grid2((num_rows_train + dimx2 - 1) / dimx2);

int nx = num_rows_train;
int ny = num_columns;

iStart = seconds();
for(int i = 0; i < num_rows_test; i++)  {
       montecarloStep1OnGPU<<<grid1, block1>>>(d_MatXTrain, d_MatXTest, d_VectDiff, nx,
ny, i);
       CHECK(cudaDeviceSynchronize());
       montecarloStep2OnGPU<<<grid2, block2>>>(d_VectDiff, d_VectYTrain, d_VectPredict,
10, nx, ny,  d_VectNorm, d_VectExp, d_Csum, d_Fsum, i);
       CHECK(cudaDeviceSynchronize());
    }
CHECK(cudaDeviceSynchronize());
iElaps = seconds() - iStart;
printf("\n montecarloMatrixOnGPU1 <<<(%d,%d), (%d,%d)>>> + montecarloMatrixOnGPU2
<<<(%d,%d), (%d,%d)>>> elapsed %f sec\n", grid1.x, grid1.y, block1.x, block1.y,
grid2.x, grid2.y, block2.x, block2.y, iElaps);
```

Codice dei kernel terza implementazione

```
// Kernel1: realizza il calcolo della matrice differenza lato device

__global__ void montecarloStep1OnGPU(double *Mat_XTrain, double *Mat_XTest, double
*Mat_Diff, int nx, int ny, int test_row)
{
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
   unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
   int idx = ix * ny + iy;

   double * i_xtest = Mat_XTest;
   i_xtest += ny * test_row;

   if (ix < nx && iy < ny) {

        Mat_Diff[idx] = i_xtest[iy] - Mat_XTrain[idx];

       }
  }



// Kernel2: realizza i restanti calcoli lato device

__global__ void montecarloStep2OnGPU(double *Mat_Diff, double *Vect_YTrain, double
*Vect_Predict, int kernel_value, int nx, int ny, double *norm_value, double
*exp_value, double *csum, double *fsum, int test_row)
{
   unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;

   if (ix < nx) {
        norm_value[ix] = norm(double(ny), (double*)&Mat_Diff[ix*784]);

        double pow_value = 2*pow((double)(1/(double)kernel_value),(double)2);
        exp_value[ix] = exp(-norm_value[ix]/pow_value);

        atomicAdd((double*)&csum[test_row], exp_value[ix]);
        atomicAdd((double*)&fsum[test_row], (Vect_YTrain[ix]*exp_value[ix]));
    }

    Vect_Predict[test_row] = __double2int_rn(fsum[test_row]/csum[test_row]);
}
```

Invocazione dei kernel terza implementazione

```
// Invocazione del kernel1 dal lato host

int dimx1 = 1;      // Numero di thread per blocco lungo x
int dimy1 = 1;      // Numero di thread per blocco lungo y
dim3 block1(dimx1, dimy1);
dim3 grid1((num_rows_train + dimx1 - 1) / dimx1, (num_columns + dimy1 -1) / dimy1);



// Invocazione del kernel2 dal lato host

int dimx2 = 1;      // Numero di threads per blocco lungo x
dim3 block2(dimx2);
dim3 grid2((num_rows_train + dimx2 - 1) / dimx2);

int nx = num_rows_train;
int ny = num_columns;

iStart = seconds();
for(int i = 0; i < num_rows_test; i++)  {
       montecarloStep1OnGPU<<<grid1, block1>>>(d_MatXTrain, d_MatXTest, d_VectDiff, nx,
ny, i);
       CHECK(cudaDeviceSynchronize());
       montecarloStep2OnGPU<<<grid2, block2>>>(d_VectDiff, d_VectYTrain,
d_VectPredict, 10, nx, ny,  d_VectNorm, d_VectExp, d_Csum, d_Fsum, i);
       CHECK(cudaDeviceSynchronize());
    }
    CHECK(cudaDeviceSynchronize());
    iElaps = seconds() - iStart;
    printf("\n montecarloMatrixOnGPU1 <<<(%d,%d), (%d,%d)>>> + montecarloMatrixOnGPU2
<<<(%d,%d), (%d,%d)>>> elapsed %f sec\n", grid1.x, grid1.y, block1.x, block1.y,
grid2.x, grid2.y, block2.x, block2.y, iElaps);
```

BIBLIOGRAFIA E SITOGRAFIA

[1] Claudio Turchetti, Falaschetti Laura, “A GPU Parallel Algorithm for Non Parametric Tensor Learning”

[2] Sito MathWorks
”https://www.mathworks.com/discovery/machine-learning.html”

[3] Jason Sanders, Edward Kandrot, “CUDA BY EXAMPLE An Introduction to General-Purpose GPU Programming”

[4] John Cheng, Max Grossman, Ty McKercher, “Professional CUDA C Programming”

[5] Sito HardwareUpgrade
“https://www.hwupgrade.it/articoli/skvideo/4643/nvidia-geforce-gtx-1080-founders-edition-la-nuova-top-di-gamma_2.html”