# Relazione progetto Cuda

Jacopo Orlandini,

UNIVERSITÀ DI PARMA

**Sommario**—Il progetto consegue un ruolo di ottimizzazione parallela di operazioni su matrici , dove una lettura di un EEG con 64 sensori comporta la scrittura di una grande mole di dati.

## 1 Introduction

CUDA, acronimo di Compute Unified Device Architecture - ormai in disuso a favore del solo termine CUDA - identifica un'architettura hardware per l'elaborazione parallela progettata da Nvidia Corporation. Tramite l'ambiente di sviluppo, il programmatore è in grado di implementare applicazioni con un forte grado di parallelismo. Durante lo sviluppo di questa tesina, mi sono avvalso di una scheda gpu 940mx, versione per laptop, con driver Nvidia binary driver - version 396.54 (nvidia-396). Il linguaggio di programmazione usato per lo sviluppo del codice sorgente è stato CUDA-C e CUDA-C++, mentre per la visualizzazione grafica e verifica dei risultati, sono stati usati script matlab forniti dal prof. Cagnoni. Il sistema operativo utilizzato è stato sia Ubuntu 16.04 che Window10.

### 1.1 Nvidia GPU

A partire dal 2003, l'industria dei semiconduttori ha imposto due trend principali per la progettazione di microprocessori. Il primo chiamato *multicore* cerca di mantenere la velocità di esecuzione dei programmi sequenziali. In contrasto la casa Nvidia sviluppa la seconda corrente, chiamata *many-core*, che si focalizza sull'esecuzione in parallelo di applicazioni. La corrente del manycore incominciò la progettazione di schede grafiche con core sempre più piccoli (ad esempio GTX 280 (GPU) possiede 240 core), ma incentrati pesantemente con il multithreading. Un gap prestazionale (Figura

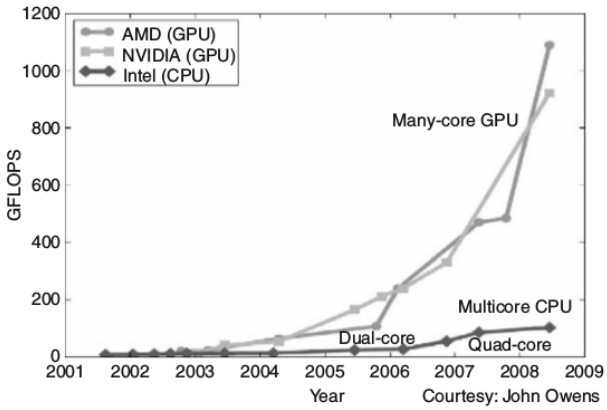



Figura 1. Differenza computazionale fra CPUs e GPUs

1), così marcato tra l'esecuzione parallela e quella sequenziale, ha mosso lo sviluppo delle applicazioni di computazione intensiva verso le GPU. Il concetto basilare è la divisione del lavoro con multipli "worker" in parallelo.
In questo documento sono presentate le principali difficoltà e esperienze che sono state acquisite durante lo sviluppo della tesina. Il documento viene organizzato secondo un approccio incrementale del problema, costruendo mano a mano i vari componenti. La prima fase di setup del sistema e installazione di cuda Toolkit non viene trattata in quanto ritengo si ritiene non siano necessarie conoscenze approfondite. Il problema è stato spezzato in tre fasi principali: la fase di acquisizione dei dati e delle

• *Jacopo Orlandini, matr. 286416,*
*E-mail: jacopo.orlandini@studenti.unipr.it, Università di Parma.*

conoscenze tecniche per poter svolgere le varie operazioni su di essi; La fase di pianificazione, in cui si è scelta una strategia conforme al progetto di parallelizzazione ed in ultimo l'implementazione dei metodi utilizzati.

## 2 Fase di acquisizione

### 2.1 Contesto della lettura EEG

Il materiale fornito sono file testuali rappresentanti la EEG di un paziente. Il paziente, con deficit motori, è stato sottoposto a stimoli luminosi. Un caschetto contenente 64 sensori trascriveva il segnale prodotto dal cervello umano soggetto agli stimoli in forma comprensibile, ovvero in una matrice. I file utilizzati durante le elaborazioni sono denominati "s00_nontarget.txt" per i casi nonTarget, dove non sono presenti gli stimoli luminosi, e "s00_target.txt" per il caso target, ove presenti gli stimoli. Per la parte di analisi della matrice si ricorda che: ogni riga contiene i valori delle acquisizioni nella stessa finestra temporale delle 64 derivazioni o sensori.La finestra temporale base è di 800 unità, se quindi si usa la derivazione 6 (offset = 6) si otterranno le finestre con inizio a t=600,1400,2200, ecc. Nel file nonTarget sono presenti tutti gli offset tranne l'offset 6. La matrice target è quindi composta da 358 righe per 5248 colonne.

### 2.2 Acquisizione

Il file fornito come input al programma è di formato testuale di grande dimensione. La riga rappresenta diverse finestre temporali, dove i singoli valori rappresentano un'acquisizione del segnale del paziente. Le colonne contengono le diverse finestre temporali di tutti i sensori (64*82) per un totale di 5248. Il punto iniziale è consinstito nel trasferimento del file testuale all'interno della memoria del calcolatore.
Uno degli aspetti principali durante questa fase è stata la scelta della tipologia della matrice. Conoscendo in anticipo il formato input richiesto dai metodi CUDA, la matrice deve essere un array monodimensionale. Per la natura 2D del file testuale, ho optato in primo luogo per un array bidimensionale e successivamente per la conversione dello stesso in 1D. Il principio

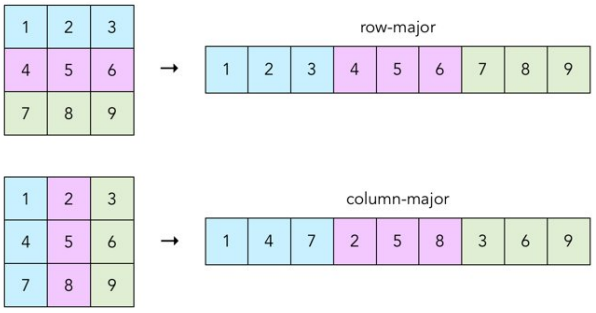



Figura 2. Row major and Color

di base per far ciò è la rimozione di una dimensione della matrice secondo la convenzione row-major order [3].
Per la creazione di un array dinamico 2D si impone la sequenzialità di indirizzi fra le varie porzioni di matrice allocate nel calcolatore [2]. In tal modo si potrebbe utilizzare la stessa matrice allocata dinamicamente come array monodimensionale.

```
/*
* Implementazione della matrice in C++ con
    fscanf C
*/
  float** A = new float*[ROW];
    A[0] = new float[ROW * COL];
  for (int i = 1; i < ROW; ++i)
    A[i] = A[i - 1] + COL;
  for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
      char *string = (char *)malloc(...);
      if (fscanf(file, "%s ", string) <= 0)
        perror("Error reading from file\n");
      A[i][j] = convesionFloat(string);
    }
  }
```

Il codice riporta uno dei possibili metodi per la soluzione. Il linguaggio C classicamente è di tipo row-major (come lo sono C++, Pascal e Python) Matlab è column-major.
La convenzione del row-major consiste nel fare storage della matrice riga per riga in maniera continua.

```
  for (int i = 0; i < subCOL; i++) {
    for (int j = 0; j < ROW; j++) {
      subMatrix[j + i*ROW] = A[j][i];
    }
  }
```

Durante la fase di acquisizione dei dati, tutte le istanze del file (stringa) sono convertite in

numero e allocate in memoria della scheda grafica per la successiva elaborazione parallela con Cuda. Per la parte di lettura ci si è avvalsi di chiamate a funzioni classiche implementate dal C (ad esempio fscanf), mentre per la trasformazione da testo a numero è stata implementato una semplice funzione chiamata conversionFloat o conversionDouble. La funzione convesion[Type] prende come input un'array di caratteri rappresentate il numero in notazione esponenziale e come output restituisce il formato richiesto - float o double. Per quanto riguarda l'implementazione, la funzione nella prima fase separa la mantissa dal suo esponente; in seguito crea il valore effettivo e lo restituisce. Una problematica riscontrata nella prima versione della funzione è stato valutare tutti i numeri con la stessa quantità di informazione. I numeri all'interno del file presentano diverse precisioni nella parte decimale, ovvero alcuni hanno un numero maggiore di cifre decimali. Nella prima versione della funzione è stato predisposto un numero fissato di cifre per dividere la mantissa dall'esponente. Questa implementazione funzionava per la maggior parte dei casi dove la precisione era quella impostata, al contrario era possibile che alcune cifre andassero perse o nel caso peggiore che il numero convertito fosse di ordini di grandezza differente.

```
/*
* Esempio di conversione
*/
 La stringa da lettura file: -1.32063(...)e+01
float convesionFloat(char *str){
  float result = -99;
  int exponent = +99;
  char *s_mantissa = strsep(&str, "e");
  exponent = atoi(str);
  result = strtof(s_mantissa,NULL)*pow(10,
    exponent);
  return result;
}

Risultato = -13.2063(...);
```

La soluzione è stato separare l'esponente dalla mantissa attraverso la funzione strsep con delimitatore "e". In particolar modo cito la funzione strtof. La funzione prende come input una stringa che rappresenta un float e in caso di successo restituisce floating-point number [1].

# 3 PROGETTAZIONE

Un programma CUDA consiste in una o più fasi, eseguite tra l'architettura della CPU detta Host e l'architettura della GPU chiamata Device. Le fasi che non richiedono parallelismo saranno implementate sull'Host. Le fasi che mostrano parallelismo su dati o operazioni aritmetiche verrano implementate sul device. In questo progetto il programma che svolge le operazioni è prevalentemente sviluppato in C-language. Per esaltare la possibilità di integrare il C++ ho implementato la parte di costruzione e inizializzazione della matrice dati e altre funzionalità. Il linguaggio sarà quindi chiamato CUDA C/C++ che viene correttamente compilato . Il codice device è implementato anch'esso in C ma esteso con keywords per le funzioni di parallelismo fornite da CUDA.

## 3.1 Memoria e trasferimento dati

In CUDA, le memorie del device e di host sono separate fisicamente. Questo riflette la natura del differente hardware utilizzato. Il device corrisponde in genere ad una scheda grafica con la propria memoria dedicata (DRAM), mentre l'Host funziona con la memoria allocata attraverso la CPU.
Un ciclo classico dell'esecuzione di un programma CUDA è: una prima fase di inizializzazione e allocazione della memoria da Host e Device, un lancio delle istruzioni parallele attraverso un kernel ed infine il trasferimento dei risultati nella memoria Host. Per allocare la memoria necessaria al device sono possibili diverse modalità: quella che è stata utilizzata sarà la cudaMalloc(devPtr, size).
Parametri:

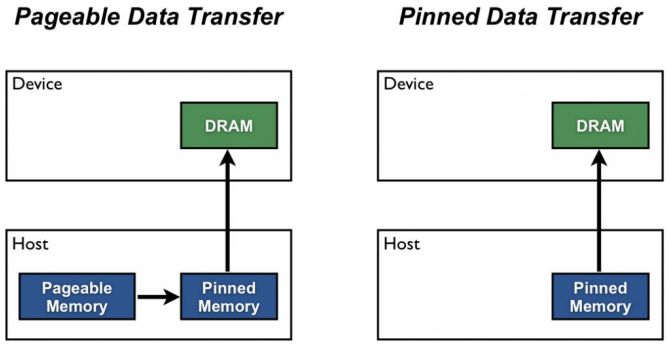
- devPtr: puntatore alla memoria device allocata.
- size: dimensione richiesta di allocazione in bytes.

Alloca *size* bytes di memoria lineare sul device e ritorna in **devPtr* un puntatore della memoria allocata. Una volta terminata l'esecuzione sulla memoria deve essere deallocata con cudaFree().

```
/*
* Allocazione di memoria su device:
*/
  cudaMalloc((void **)&d_A, m_size);
```

Per convenzione nel progetto si utilizza un prefisso h_ o d_ per esplicitare la natura del puntatore in questione. Non si deve solamente considerare il tempo di esecuzione di un kernel relativo ad un'implementazione CPU o GPU. Il programmatore deve anche considerare il costo di movimentazione dei dati. I dati Host sono allocati attraverso la tecnica della paginazione [7].

Figura 3. Trasferimento di memoria su device




Di norma il device non può accedere direttamente ai dati dalla memoria Host paginata dalla cpu (e.g malloc). Quando avviene un trasferimento dati tra Host e device, il driver CUDA deve, in anticipo, allocare una memoria temporanea con page-lock e trasferirla nel device. Un ottimizzazione sulle prestazioni si può avere allocando direttamente la memoria pinned su Host attraverso la chiamata cudaMallocHost. Riguardo all'allocazione di memoria nell'Host sono quindi possibili due metodi principali:

- malloc: alloca size_t bytes e restituisce un puntatore al byte più basso in memoria.
- cudaMallocHost: Alloca size_t bytes di memoria Host che risulta essere page-locked e accessibile direttamente dal device [6].

```
cudaError_t status =
cudaMallocHost( (void**)&h_aPinned, bytes );
if (status != cudaSuccess)
  printf("Error allocating pinned buff");
```

Una volta allocata la memoria, la fase successiva impone il trasferimento dei dati da Host a device attraverso il metodo cudaMemCpy(). CudaMemCpy copia una quantità determinata di byte dalla memoria puntata (src) alla memoria di destinazione (dst).

## 3.2 Kernel

In CUDA, una funzione kernel specifica il codice che sarà eseguito da tutti i thread durante la fase parallela. La programmazione CUDA rientra nella categoria della SPMD (single program, multiple data), un famoso stile di programmazione per il calcolo intensivo parallelo. Ulteriore estensione del C sono le keywords. Per indicare un kernel ci avvaliamo della dicitura __global__. Una volta caricata una matrice all'interno del device si deve poter accedere in modo parallelo ai suoi dati. Invocando un kernel si genera una griglia di blocchi thread attraverso cui si può accedere alla memoria ed eseguire algoritmi paralleli. La sfida è la gestione dei thread. Tutti i thread eseguono le stesse struzioni.

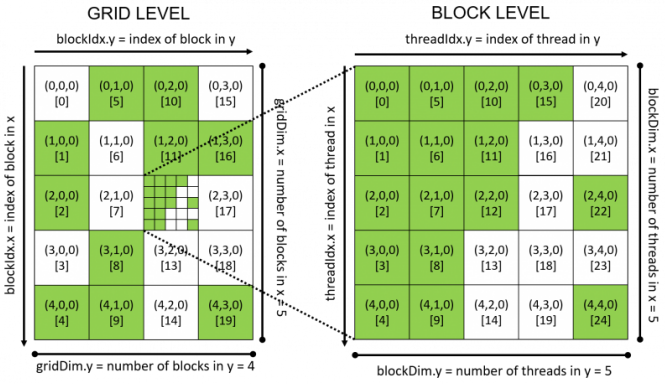



Figura 4. Esempio di mappatura CUDA

### 3.2.1 *Layout of thread*

Ricapitolando un thread block è composto da un array tridimensionale di thread con un massimo di 1024 (32*32) threads per la scheda grafica. Le coordinate del thread utilizzabili sono threadIdx.x, threadIdx.y, threadIdx.z. In questo progetto userò solamente la threadIdx.x. I thread sono racchiusi in un blocco, organizzato anch'esso in un array tridimensionale, blockIdx.x, blockIdx.y, blockIdx.z. Quindi si ottiene una griglia di blocchi ed ogni blocco è una griglia di thread. Quando nel codice host viene invocato un kernel, è buona regola impostare i parametri di gridSize e blockSize in anticipo. I parametri sono struct di tipo *dim3* e contengono i 3 assi. Il lancio del kernel viene attuato con la sintassi:

```
const unsigned int block_x_threads = 1;
const unsigned int block_y_threads = 1;
const unsigned int block_z_threads = 1;

const unsigned int tot_blocks_x = 1;
//Di default le dimensioni mancanti vengono impostate ad 1 se non impostate.
dim3 dimBlock( block_x_threads , block_y_threads);
dim3 dimGrid( tot_blocks_x ,1 ,1);
nome_kernel<<<dimGrid , dimBlock >>>(flaot * d_A, float *d_b);
```

L'obiettivo dell'applicazione è estrarre in modo parallelo la somma delle colonne e calcolarne la media . Un primo approccio è stato lanciare un thread per ogni colonna della matrice e con un loop for, estrarne la media. Per questo motivo la divisione della matrice in blocchi risultava avere una forte impronta sequenziale.

```
__global__ void matrixMeanFLOAT(int rows, int cols, float *matrix, float *dest) {
  unsigned int iter =  blockDim.x*blockIdx.x + threadIdx.x;
  for (int  i = 0;  i < rows; ++i ) {
    float iterCol = matrix[cols*i + iter]; //scorro i valori della colonna per riga
    dest[iter] += iterCol; //sommo nella destinazione in device memory.
  }
  dest[iter] = dest[iter]/rows; //media della somma
}
```

### 3.2.2 Shared Memory

CUDA C rende disponibile una regione di memoria chiamata *shared memory*. Questa regione di memoria tratta le variabili in maniera particolare. Crea una copia delle variabili per ogni blocco che viene lanciato sulla GPU. Ogni thread nel blocco condivide quella memoria e i threads di altri blocchi non possono modificare le shared memory di altri blocchi. La memoria shared consente la collaborazione e comunicazione fra thread di un blocco. In aggiunta la shared memory risiede fisicamente sulla GPU quindi comporta tempi di accesso minori rispetto ai tipici buffer. Ovviamente nulla è gratuito quindi paghiamo i benefici della memoria condivisa nella sincronizzazione dei threads.

### 3.2.3 Riduzione array

Uno dei pattern più comuni è la riduzione. Una riduzione combina tutti gli elementi in una collezioni dati in un singolo elemento, usando un operatore che associa a due input un output. Data una collezzione di n elementi, due elementi adiacenti possono essere selezionati e combinati per formare una collezione con n-1 elementi. Questa procedura può essere reiterata finchè non rimane un singolo elemento. Nel mio caso essendo l'operatore la somma, la riduzione compie la somma di tutti gli elementi sulla colonna. [10].

$$reduce : output_j = \frac{\sum_{i=0}^{358}(x_{i,j})}{rows}$$

L'ottimizzazione è stata quindi il trasporto dei calcoli aritmetici in un algoritmo di riduzione per le colonne e l'utilizzo della memoria shared,[ [9]]. E' stato utilizzato un approccio ad albero per la somma degli elementi nella memoria shared. Ogni thread carica un elemento dalla memoria globale nella memoria shared. Viene assegnato un indice ad ogni thraed e si impone una barriera. Si esegue l'algoritmo di riduzione degli elementi nella memoria condivisa e il thread con indice 0 scrive nella memoria globale il risultato media.

```
__global__ void reduce(float *g_idata, float *g_odata) {
  extern __shared__ float sdata[ROW];
  unsigned int i = blockIdx.x *blockDim.x + threadIdx.x;
  unsigned int tid = threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  for (int s = 1; s < blockDim.x; s *= 2) {
    int index = 2 * s * threadIdx.x;
    if (index < blockDim.x && (index+s) <= blockDim.x) {
        if ((index + s) >= blockDim.x)
        sdata[index + s] = 0;
      sdata[index] += sdata[index + s];}
    __syncthreads();}
  if (tid == 0) g_odata[blockIdx.x] = sdata[tid]/ROW;
}
```

Quello che realmente succede all'interno dell'algorimo è rappresentato dalla seguente figura. Sorge un problema di divisibilità della riduzione, in quanto la prima fase riduce gli elementi da 358 a 179. La seconda iterazione quindi si trova a non poter funzionare correttamente, l'elemento di output non riesce più ad associare due elementi di input. La soluzione implementata è stata di sommare degli 0 nel momento in cui si vada ad accedere a locazioni di memoria proibite.

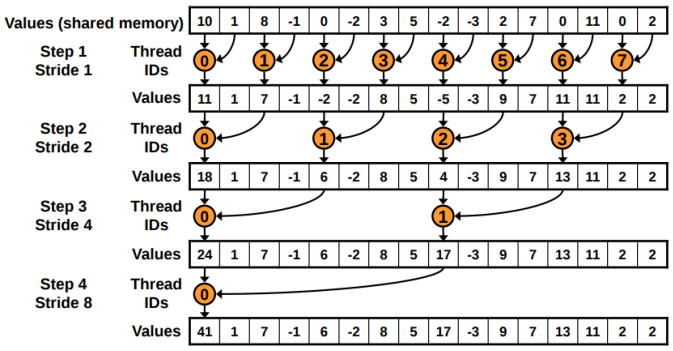



Figura 5. Algoritmo di riduzione

### 3.2.4 syncthreads

Cuda permette il coordinamento di attività di thread nello stesso blocco usando la funzione di sincronizzazione barriera, chiamata __syncthreads(). Quando una funzione kernel richiama __syncthreads(), il thread chiamante è mantenuto nella stessa locazione fino a che ogni thread nel blocco raggiunge la stessa locazione. Questo assicura che tutti i thread in un blocco abbiano completato una fase della loro esecuzione prima di eseguire successive operazioni. In CUDA, un chiamata __syncthreads deve essere eseguita da tutti i componenti di un blocco. Quando quest'ultima funzione viene eseguita in un percorso contenente un if-else statement, o tutti i threads nel blocco eseguono la chiamata __syncthreads o nessuno. Nel momento in cui i threads eseguono percorsi diversi a causa di if-else o altri statement aspettano su barriere differenti. Questo comportamento potrebbe portare a creare un'attesa infinita fra di loro. La capacità di sincronizzarsi impone alcuni vincoli sui thread di uno stesso blocco. Quindi i thread dovrebbero eseguire in prossimità temporale con altri thread per evitare lunghi tempi di attesa. CUDA runtime system soddisfa questo vincolo assegnando le risorse a tutti i thread in un blocco unico. Quindi tutti i thread di un blocco vengono assegnati alla stessa risorsa di esecuzione, tutti gli altri thread nello stesso blocco sono assegnati alla stessa risorsa.

### 3.2.5 Ottimizzazione algoritmo di riduzione

L'algoritmo di riduzione possiede una bassa intensità di computazione aritmetica, quindi per ottenere la massima efficienza dobbiamo raggiungere il massimo flusso di trasmissione, ovvero il picco di banda [11]. Per massimizzare la banda di memoria globale è necessario minimizzare il numero di transazioni sul bus (mantenendo una accesso alla memoria contiguo "coalesce memory"). La gpu preferisce di gran lunga un approccio coalesced rispetto ad uno strided in cui si va ad accedere alla memoria attraverso un offset. Il codice ottimizzato quindi risulta essere:

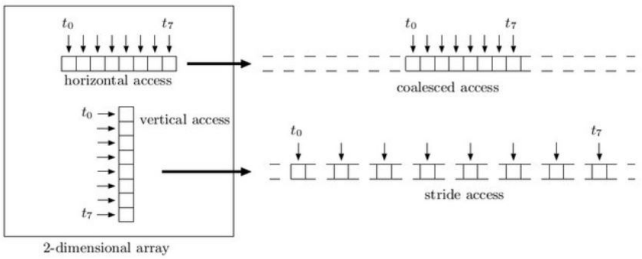



Figura 6. coalesced vs strided access

```
template <unsigned int blockSize> __global__
    void reduce(float *g_idata, float *g_odata
    ) {
  __shared__ float sdata[blockSize];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockSize *2 +
    threadIdx.x;
  sdata[tid] = g_idata[i] + g_idata[i +
    blockSize];
  __syncthreads();
  if (blockSize >= 256) { if (tid < 128) {
    sdata[tid] += sdata[tid + 128]; }
    __syncthreads(); }
  if (blockSize >= 128) { if (tid < 64) {
    sdata[tid] += sdata[tid + 64]; }
    __syncthreads(); }
  if (tid < 32) {
    if (blockSize >= 64) sdata[tid] += sdata[
    tid + 32];
    if (blockSize >= 32) sdata[tid] += sdata[
    tid + 16];
    if (blockSize >= 16) sdata[tid] += sdata[
    tid + 8];
    if (blockSize >= 8) sdata[tid] += sdata[
    tid + 4];
    if (blockSize >= 4) sdata[tid] += sdata[
    tid + 2];
    if (blockSize >= 2) sdata[tid] += sdata[
    tid + 1];
  }
  if (tid == 0)
      g_odata[blockIdx.x] = sdata[0]/ROW;
```

```
}

template <unsigned int blockSize> __device__
    void warpReduce(volatile float* sdata, int
     tid) {
  if (blockSize >= 64) sdata[tid] += sdata[tid
     + 32];
  if (blockSize >= 32) sdata[tid] += sdata[tid
     + 16];
  if (blockSize >= 16) sdata[tid] += sdata[tid
     + 8];
  if (blockSize >= 8) sdata[tid] += sdata[tid
    + 4];
  if (blockSize >= 4) sdata[tid] += sdata[tid
    + 2];
  if (blockSize >= 2) sdata[tid] += sdata[tid
    + 1];
}
```

Quello che ora succede è simile al seguente schema. Il trucco utilizzato per usare al meglio i warps e l'algoritmo di riduzione è stato creare un blocco dati non di 358 elementi (non divisibile per 32) ma di 512 elementi. In questo modo è stato possibile sfruttare al 100% i processori grafici e la divisibilità per warps. Viene implementata anche la funzione warp

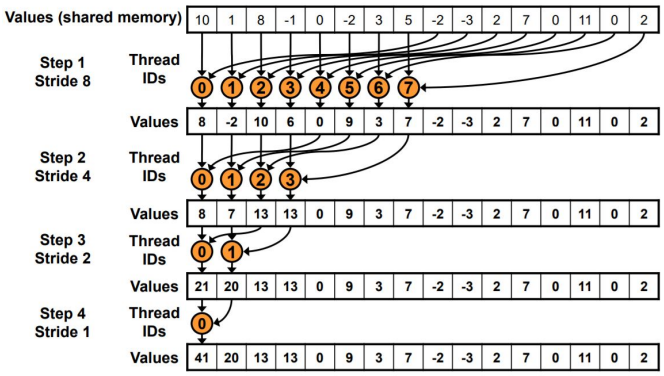



Figura 7. Parallel Reduction: Sequential Addressing

reduce per rendere molto efficienti le ultime iterazioni dell'algoritmo, quando per esempio si raggiunge un block size.

## 4 Conclusioni

Vengono ora riepilogati i risultati ottenuti durante le varie versioni del progetto. L'allocazione della memoria viene eseguita attraverso cudaMallocHost in quanto rende più rapido il processo di trasferimento dei dati da host a device (da 1300 a 1650 MB/s) con una riduzione del 20% sul tempo.

| | Source | Destination | Size (bytes) | Rate (MB/s) | Duration (µs) |
|---|---|---|---|---|---|
| 1 | Host Pinned | Device | 7515136 | 1634.2 | 4,385.538 |
| 2 | Device | Host Pinned | 20992 | 1850.9 | 10.816 |

Figura 8. Transmission Rate (MB/s) pinned to device

| | Source | Destination | Size (bytes) | Rate (MB/s) | Duration (µs) |
|---|---|---|---|---|---|
| 1 | Host Unpinned | Device | 7515136 | 1297.4 | 5,524.323 |
| 2 | Device | Host Pinned | 20992 | 1834.6 | 10.912 |

Figura 9. Transmission Rate (MB/s) unpinned to device

In conclusione un approccio secondo riduzione non è valorizzato con una mole di dati così ridotta. Nella versione del kernel con riduzione, il costo di sincronizzazione e di creazione di molti thread possiede un peso troppo elevato in relazione alla quantità di dati ed alla ridotta computazione. Al contrario la versione con il ciclo for non contiene né forti divergenze né sincronizzazioni complesse.

| | Function Name | Grid Dimensions | Block Dimensions | Start Time (µs) | Duration (µs) | Occupancy | Registers per Thread | Static Shared Memory per Block (bytes) |
|---|---|---|---|---|---|---|---|---|
| 1 | reduce | {128, 1, 1} | {41, 1, 1} | 49,985,362.151 | 2,582.547 | 100.00% | 19 | 0 |
| 2 | reduce | {128, 1, 1} | {41, 1, 1} | 50,042,198.664 | 2,601.108 | 100.00% | 19 | 0 |
| 3 | reduce | {128, 1, 1} | {41, 1, 1} | 50,099,414.000 | 2,572.275 | 100.00% | 19 | 0 |
| 4 | reduce | {128, 1, 1} | {41, 1, 1} | 50,154,611.693 | 2,563.859 | 100.00% | 19 | 0 |
| 5 | reduce | {128, 1, 1} | {41, 1, 1} | 50,210,721.224 | 2,596.499 | 100.00% | 19 | 0 |
| 6 | reduce | {128, 1, 1} | {41, 1, 1} | 50,269,965.568 | 2,582.962 | 100.00% | 19 | 0 |
| 7 | reduce | {128, 1, 1} | {41, 1, 1} | 50,325,454.307 | 2,575.667 | 100.00% | 19 | 0 |

Figura 10. kernel basato su ciclo for

| | Function Name | Grid Dimensions | Block Dimensions | Start Time (µs) | Duration (µs) |
|---|---|---|---|---|---|
| 1 | reduce | {5248, 1, 1} | {358, 1, 1} | 52,887,292.979 | 18,211.495 |
| 2 | reduce | {5248, 1, 1} | {358, 1, 1} | 52,987,076.400 | 18,222.343 |
| 3 | reduce | {5248, 1, 1} | {358, 1, 1} | 53,099,355.339 | 18,222.502 |
| 4 | reduce | {5248, 1, 1} | {358, 1, 1} | 53,235,122.512 | 18,214.757 |
| 5 | reduce | {5248, 1, 1} | {358, 1, 1} | 53,335,991.242 | 18,209.638 |
| 6 | reduce | {5248, 1, 1} | {358, 1, 1} | 53,415,366.366 | 18,211.590 |
| 7 | reduce | {5248, 1, 1} | {358, 1, 1} | 53,495,444.888 | 18,207.461 |

Figura 11. Prima versione del kernel con array dispari

Questo si rispecchia nei risultati ottenuti durante i vari test eseguiti, la versione con il ciclo for risulta essere la più veloce pur mantenendo un approccio sequenziale.

| Function Name | Grid Dimensions | Block Dimensions | Duration (µs) | Occupancy |
|---|---|---|---|---|
| reduce<uint=256> | {5248, 1, 1} | {256, 1, 1} | 3,163.298 | 100.00% |

Figura 12. Versione ottimizzata del kernel attraverso warp

## RIFERIMENTI BIBLIOGRAFICI


[1] IBM, strtof() — Convert character string to float, https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxbd00/strtof.htm.

[2] Trevor Simonton, Transfer 2d array memory to cuda, http://www.trevorsimonton.com/blog/2016/11/16/transfer-2d-array-memory-to-cuda.html.

[3] Scratchapixel 2.0, Row Major vs Column Major Vector, https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/geometry/row-major-vs-column-major-vector.

[4] Cuda C/C++ Basic, Introduction to Cuda, https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf.

[5] .David B Kirk, Wen-mei W. Hwu, *Programming massively parallel processor*. [*A Hands-on Approach*]. 2010.

[6] Optimize data transfer, pinned data transfer, https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc.

[7] Paginazione, https://it.wikipedia.org/wiki/Paginazione.

[8] Cuda mapping, https://www.pelagos-consulting.com/?p=503.

[9] Cuda redcution, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[10] parallel pattern reduce, http://www.drdobbs.com/architecture-and-design/parallel-pattern-7-reduce/222000718.

[11] ottimizzazione riduzione, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.