University of Calabria

Department of Mathematics and Computer Science

Master's Degree in Computer Science

Massive Parallel Programming on GPUs - 6 CFU

MPPGPU

6 FU

Daniele Avolio

A.Y. 2022/2023

Contents

0.1 Introduzione

In questo PDF c'è un sunto della teoria e della pratica del corso di GPGPU tenuto dal professore Donato D'Ambrosio.

Non sono molto fan del corso quindi sarà fatto un pochino all'acqua di rosa.

Chapter 1

Computazione parallela eterogenea

1.1 Architettura di una GPU moderna

Le GPU hanno un architettura che permette di runnare un numero elevatissimo di threads. Solitamente sono organizzati in **Streaming Multiprocessor** (SM). Ogni SM ha un numero di **Streaming Processors** (SPs)

$$16SMs, 128SPs = 2048SPs \tag{1.1}$$

Cioè circa 2048 cores.

Dobbiamo dire che i Threads sono divisi in **blocchi** e sono processati dagli SMs. L'insieme di blocchi viene chiamata **Grid**.

Ogni blocco viene diviso successivamente in **warps** (32 threads). I warps sono eseguiti in modo **SIMD** (Single Instruction Multiple Data).

1.2 Modello di memoria fisica di CUDA

Dobbiamo parlare delle seguenti nomencalture:

- Registri: Per threads veloce sulla chip memory
- L1 Cache / Memoria condivisa: Per SM veloce sulla chip memory
- Memoria di sola lettura: Per SM, per istruzioni e dati costanti
- L2 Cache: per GPU, visibile globalmente
- DRAM: per GPU visibile globalmente e accessibile dall'host

Dobbiamo parlare anche dei diversi tipi di memoria:

- Memoria locale: varaibili private, salvate sui registri o DRAM
- Memoria condivisa: per blocchi, salvata sulla Shared memory
- Memoria costante: dati globalmente read only, ssalvata in DRAM e cachata in L1
- Memoria globale: dati globalmente accessibili, salvata in DRAM

Chapter 2

Data Parallel Computing

2.1 CUDA C

Si parla di funzioni **host** e **device**. Le funzioni host sono eseguite sulla CPU mentre le funzioni device sono eseguite sulla GPU.

Parliamo di **kernels** che sono funzioni device che possono essere eseguite da un numero elevato di threads.

```
1   __global__ void kernel_name(int *a, int *b, int *c) {
2       int tid = blockIdx.x * blockDim.x + threadIdx.x;
3       if (tid < N) {
4           c[tid] = a[tid] + b[tid];
5       }
6   }
```

CUDA mantiene una coda di esecuzione (stream) che e' transparente all'utente. Ogni volta che una task viene aggiunta alla coda, viene eseguita quando tutte le task precedenti sono state eseguite.

Le task possono essere sincrone o asincrone.

2.1.1 Esempio di codici CUDA

Addizione di vettore sequenziale

```
1   void vecAdd(float* h_A, float* h_B, float* h_C, int n)
2   {
3       for (int i = 0; i < n; i++)
4       h_C[i] = h_A[i] + h_B[i];
5   }
6   int main()
7   {
8       // Memory allocation for h_A, h_B, and h_C, I/O, etc...
9       vecAdd(h_A, h_B, h_C, N);
10  }
```

Praticamente, si prendono due vettori e si sommano in un terzo vettore. Questo viene fatto in maniera sequenziale.

Il codice CUDA corrispondente e' il seguente:

```
1  #include <cuda.h>
2  void vecAdd(float* A, float* B, float* C, int n)
3  {
4      float *d_A *d_B, *d_C; int size = n* sizeof(float);
5      int block_size = 32, number_of_blocks = ceil(n/block_size);
6
7      cudaMalloc((void**)&d_A, size);
8      cudaMalloc((void**)&d_B, size);
9      cudaMalloc((void**)&d_C, size);
10
11     cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
12     cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
13
14     vecAddKernel<<<number_of_blocks, block_size>>>(d_A, d_B, d_C, n);
15
16     cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
17
18     cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
19 }
```

In pratica, la memoria viene allocata in modo asincrono con **cudaMalloc** e i dati vengono trasferiti sul device in modo sincrono con **cudaMemcpy**.

Il kernel viene aggiunto alla coda con **vecAddKernel** <<<>>> e viene eseguito in modo asincrono.

Il numero totale di threads, cioe, $numeroblocchi \times blocksize$ deve necessariamente essere $\geq$ al numero di elementi da processore (*esecuzione monolitica*)

Eventualmente, i dati vengono copiati in modo sincrono all'host e viene rilasciata la memoria.

- cudaMemcpy: copiare sulla CPU
- cudaFree: rilasciare la memoria

2.2 Funzioni Kernel

Un kernel specifica il codice da eseguire sui threads. Viene eseguito da una griglia uniforme di threads blocks.

Un 0-base index viene assegnato ad ogni blocco nella griglia e ad ogni thread nel blocco.

Un global thread index, i, puo' essere ottenuto e utilizzato per accedere ai dati.

La griglia prende il posto del loop seriale.

Sia la griglia che i blocchi possono essere **1D, 2D, o 3D**.

Cuda ha delle variabili built in:

- blockIdx: contiene l'indice del blocco corrente
- blockDim: contiene le dimensioni del blocco corrente
- threadIdx: contiene l'indice del thread corrente

Hanno tutte e 3 le variabili un membro x, y, e z.

Vengono calcoalte cosi:

- ix = blockIdx.x * blockDim.x + threadIdx.x
- iy = blockIdx.y * blockDim.y + threadIdx.y
- iz = blockIdx.z * blockDim.z + threadIdx.z

Ogni device ha un limite di threads per blocco e di threads per griglia. Ad esempio, la GTX 980 permette 1024 threads per blocco e 2^{31} blocchi nella griglia, nel caso 1D.

Altre informazioni importanti sono **Il massimo numero di threads per SM** e il **massimo numero di blocchi per SM**

- GTX 980: 2048 threads per SM, 32 blocchi per SM

  Il massimo numero di threads per SM va massimizzato per avere le migliori perfromance

```
1 __global__ void vecAddKernel(float* A, float* B, float* C, int n){
2
3     //i is a private variable
4     int i = blockDim.x*blockIdx.x + threadIdx.x;
5
6     if (i < n) // to be sure to "intercept" data
7         C[i] = A[i] + B[i];
8 }
```

Alcune nomenclature importanti:

- __global__: indica una funzione kernel che viene eseguita sul device e puo' essere chiamata solo dall'host
- __device__: indica una funzione che viene eseguita sul device e puo' essere chiamata solo dal device
- __host__: indica una funzione che viene eseguita sull'host e puo' essere chiamata solo dall'host

Notiamo che la varaibile privata i e' privata per ogni threads.

Utilizziamo quel if perche' potrebbe essere che il numero di threads non sia multiplo del numero di blocchi. Questo significa che alcuni threads non verrebbero eseguiti.

- Assumiamo che prendiamo 32 come block size
- 4 blocchi, per un totale di 128 threads sono necessari per processare 100 elementi
- Siccome tutti i threads eseguono lo stesso codice, i threads in eccesso devono essere bloccati
- Quindi 28 threads non vengono eseguiti

2.2.1 Gestione degli errori

Errori sincroni: Succedeono quando il thresad host capisce che il kernel e' invalido. Ad esempio, quando la block size o la grid size di un kernel e' troppo grande. Un errore viene mandato immediatamente dopo che il kernel viene lanciato.

Questo errore viene catturato a runtime con le chiamate API, come **cudaGetLastError**, esattamente dopo che viene lanciato il kernel.

Errori asincroni: Succedono quando il kernel viene eseguito. Alcuni esempi sono:

- Accesso a memoria non valido
- Accesso a meoria non valido durante cudaMemcpyAsync

Ci sono 2 tipi di errori:

Errori sticky: sono quegli errori che praticamente vengono riportati da tutte le chiamate API di cuda. Ad esempio, se il kernel accedesse un indirizzo di memoria non valido ritornerebbe un errore sticky. Allora, meglio controlalre

```
1     cudaError_t err = cudaGetLastError();
2     if (err != cudaSuccess) {
3         printf("%s␣in␣%s␣at␣line␣%d\n", cudaGetErrorString(err),
4             __FILE__,__LINE__);
5         exit(EXIT_FAILURE);
6     }
```

Errori non sticky: Sono quegli errori che non sono riportati da tutte le chiamate CUDA di seguito. Cioe, il contesto CUDA non e' corrotto.

```
1     cudaError_t err = cudaMalloc((void **) &d_A, size);
2     if (err != cudaSuccess) {
3         printf("%s␣in␣%s␣at␣line␣%d\n", cudaGetErrorString(err),
4             __FILE__,__LINE__);
5         exit(EXIT_FAILURE);
6     }
```

Chapter 3

Esecuzione parallela scalabile

3.1 Organizzazione dei thread CUDA

In pratica i threads CUDA eseguono la stessa funzione ma su memoria diversa.

In generale, una griglia e' un array 3D di blocchi e ogni blocco ha un array 3D di threads.

$$\text{Grid} \rightarrow \text{Block} \rightarrow \text{Thread} \tag{3.1}$$

La griglia viene definita quando il kernel viene lancaito aggiugnendo al kernel i seguenti parametri:

- number_of_blocks: numero di blocchi nella griglia
- block_size: numero di threads in ogni blocco

```
1    dim3 block_size(256, 1, 1);
2    dim3 number_of_blocks(ceil(n/(float)block_size.x), 1, 1);
3    vecAddKernel<<<number_of_blocks, block_size>>>(...);
```

Entrambi i parametri sono di tipo **dim3** che e' una struct C che permette dia vere 3 integer.

E' possibile avere meno dimensioni mettendo le altre a 1.

IMPORTANTE: il numero di thread per blocco viene fissato a 256, mentre il numero di blocchi e; una funzione che dipende da:

- numero di elementi da processare (n)
- numero di thread per blocco (256)

Si prende la divisione intera.

Importante: il limite per la block size e' in generale $2^{10} = 1024$.

3.2 Mappare thread a dati multidimensionali

Ad esempio, prendimao un immagine 76 x 62: Se dovessimo usare un blocco 16x16 avremmo bisogno di 5 blocchi nella direzione X e 4 blocchi nella direzione Y, per un totale di 20 blocchi.

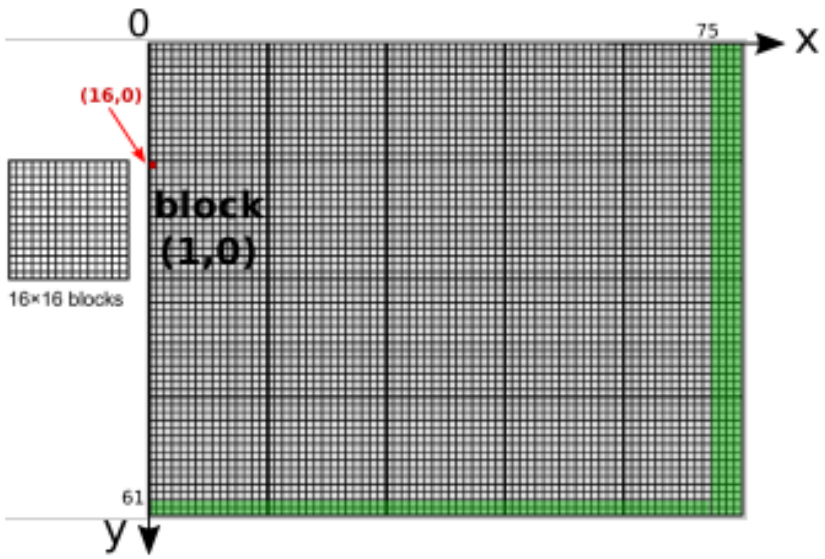

Figure 3.1: Mappare thread a dati multidimensionali

Codice che permetterebbe di processare l'immagine

```
dim3 number_of_blocks(ceil(m/16.0), ceil(n/16.0), 1);
dim3 block_size(16, 16, 1);
colorToGrey<<<number_of_blocks,block_size>>>(d_Pin,d_Pout,m,n);
```

Prima di capire come fare pero dobbiamo capire anche come funziona l'accesso alla memoria in cuda.

Siccome i dati sono salvati in un buffer lineare, un indice singolo va usato.

CUDA C rappresenta gli array a due dimensioni in memoria row-major order, cioe' per righe una dopo l'altra.

$$\text{index} = \text{row} \cdot \text{width} + \text{column} \tag{3.2}$$

Esempio di codice che lo fa:

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGrey(unsigned char* Pout, unsigned char* Pin, int width, int height) {
    int Col = threadIdx.x + blockIdx.x * blockDim.x;
    int Row = threadIdx.y + blockIdx.y * blockDim.y;
    if (Col < width && Row < height) {
    // get 1D coordinate for the grayscale image
    int greyOffset = Row*width + Col;
    // one can think of the RGB image having
    // CHANNEL times columns than the grayscale image
    int rgbOffset = greyOffset*CHANNELS;
    unsigned char r = Pin[rgbOffset ]; // red value for pixel
    unsigned char g = Pin[rgbOffset + 1]; // green value for pixel
    unsigned char b = Pin[rgbOffset + 2]; // blue value for pixel
```

```
16          // perform the rescaling and store it
17          // We multiply by floating point constants
18          Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
19      }
20  }
```

3.2.1 Convoluzione

La convoluzione non e' altro che una funzione che aggiorna il valore di un pixel basandosi su una somma pesata dei valori dei pixel vicini.

Consideriamo un valore medio di NxN pixel intorno al pixel, incluso lui.

Codice Kernel per blurrare immagine:

```
1       #define BLUR_SIZE 5
2   __global__ void blurKernel(unsigned char* in, unsigned char* out, int w, int h)
3       {
4           int Col = blockIdx.x * blockDim.x + threadIdx.x;
5           int Row = blockIdx.y * blockDim.y + threadIdx.y;
6           if (Col < w && Row < h)
7           {
8               int pixR = 0; int pixG = 0; int pixB = 0;
9               int pixels = 0;
10              for (int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; blurRow++)
11              for (int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; blurCol++)
12              {
13                  int curRow = Row + blurRow;
14                  int curCol = Col + blurCol;
15                  if (curRow > -1 && curRow < h && curCol > -1 && curCol < w)
16                  {
17                      pixR += in[3*(curRow * w + curCol) ];
18                      pixG += in[3*(curRow * w + curCol) + 1];
19                      pixB += in[3*(curRow * w + curCol) + 2];
20                      pixels++;
21                  }
22              }
23              out[3*(Row * w + Col) ] = (unsigned char)(pixR / pixels);
24              out[3*(Row * w + Col) + 1] = (unsigned char)(pixG / pixels);
25              out[3*(Row * w + Col) + 2] = (unsigned char)(pixB / pixels);
26          }
27      }
```

Da notare che l'if a riga 15 viene usato per gestire i bordi dell'immagine.

I thread possono essere sincronizzati **solamente** all'interno dello stesso blocco. Si utilizza la funzione **__syncthreads()**. Questo e' utile perche' in questo modo si evita che un thread possa leggere dati non ancora aggiornati da un altro thread.

In questo modo i blocchi possono essere eseguiti in qualsiasi ordine e aumenta la scalabilita'.

3.3 Scheduling e tolleranza della latenza

Nella maggior parte delle implementazioni i glocchi di threads sono divis in WARPS. I warps sono gruppi di 32 threads che vengono eseguiti in modo sincrono. E' l'unita del thread Scheduling negli SMs.

Gli SMs eseguono tutti i thread di un warp con la stessa istruzione. (SIMD) single instruction multiple data.

Assegnare tanti warps ad un SM e' utile. Questo perche' quando una istruzione e' in attesa di un dato, l'hardware puo' eseguire un altro warp e non eseguire quello che e' in attesa. Questo meccanismo permette di tollerare la latenza della memoria.

Viene utilizzata una metrica per parlare di questo: **la occupancy**

$$occupancy = \frac{TpSM}{TpSN_{max}} = \frac{WpSM}{WpSM_{max}} \tag{3.3}$$

dove:

- $TpSM$ e' il numero di thread per SM che stanno effettivamente eseguendo (warps)
- $TpSM_{max}$ e' il numero massimo di thread per SM (capacity dei warps del SM)

E' chiaro che **massimizzare l'occupancy** massimizza anche la tolleranza della latenza.
Consideriamo una GTX 980:

- 1024 threads per block (32 warps)
- 2048 threads per SM (64 warps)
- 32 blocchi per SM

Una configurazione ottima sarebbe:

- 32x32 = 1024 threads per block. Schedulerebbe 2048/1024=2 blocchi per SM, avendo occupancy = 1
- 8x8 = 64 threads per blocco. Schedulerebbe 2048/64=32 blocchi per SM, avendo occupancy = 32/32 = 1

Non ottime:

- 64x64 = 4096 threads per block. Si supera sia block che SM thread capacity
- 4x4 = 16 threads per blocco. Schedulerebbe 2048/16=128 blocchi per SM. Cuda runnerebbe 32 blocchi, per 512 threads totali, avendo occupancy 1/4.

Chapter 4

Importanza dell'efficienza di accesso alla memoria

4.1 Memoria e data locality

Le risorse hardware come la potenza computazionale e la memoria possono divenrare **colli di bottiglia** a lungo andare.

Dato un dispositivo di computazione, di questo dispositivi un applicazione potrebbe essere:

- **compute-bound**: se il bottleneck e' limitata dalla potenza computazionale
- **memory-bound**: se il bottleneck e' limitata dalla banda di memoria

Per il primo caso, se possibile, la soluzione e' quella di fare **uno switch** ad un dispositivo piu performante.

Per il secondo caso, una **differente implementazine** potrebbe portare ad avere un boost alel performance, anche considereveole.

Un altro bottleneck e' assolutamente la **latenza della DRAM**, con migliaia di clock per ciclo e la limitata banda di memoria.

- Anche avendo a disposizione tantissimi thread per l'esecuzione, **una congestione di traffico** puo' portare ad avere un **bottleneck** sulla memoria, avendo pochi thread che lavorano rispetto a quelli che sono fermi, rendendo cosi **SMs idle.**

La chiave per la soluzione: bisogna quindi minimizzare il numero di accesso alla RAM per ogni operazione computazionale.

- Una soluzione e' quella di usare le memorie on-chip
- Le GPU moderne hanno on-chip una memoria di cache L1, che e' trasparente al programmatore
- CUDA fornisce al programmatore una memoria L1-like, cioe la **shared memory** che puo' essere usata quando il caching non basta

Metrica: si ha una metrica per definire l'efficienza di un kernel. Questa prende il nome di **intenstita' operazionale**.

$$IO = \frac{W}{Q} \tag{4.1}$$

- W: numero di operazioni in virgola mobile performate

- Q: Bytes accessi

FP64 significa floating point a 64 bit, *FP32* a 32 bit, *FP16* a 16 bit.

Esempio: Una GPU con con 10 FP64 TFLOPS come performance massima e 1.5TB/s di banda di DRAM.

$$IO_{FP64_{best}} \approx 6.67 \tag{4.2}$$

Se 1 perazione FP64 e' performata per operandi FP64 lunghi 8 bytesL

$$IO = \frac{1}{8} = 0.125 \rightarrow W = IO \cdot Q = 0.125 \cdot 1.5TB/s = 0.1875TFLOPS \tag{4.3}$$

Cioe'1.875% della performance massima.

FP32 Single precision arithmetic. Le performance in questo caso sono generalmente piu alte.

La stessa GPU di prima, cioe' la Nvidia A100, ha circa 20 FP32 TFLOPS peak performance, con 1.5TB/s di banda di DRAM.

Se prendiamo le stesse operazioni di prima, ma con FP32:

$$IO_{FP32_{best}} = \frac{20 \cdot 10^9}{1.5 \cdot 10^9} \approx 13.34 \tag{4.4}$$

Se 1 operazione a virgola mobile 32 e' fatta per operandi FP32 lunghi 4 bytes:

$$IO = \frac{1}{4} = 0.25 \rightarrow W = IO \cdot Q = 0.25 \cdot 1.5TB/s = 0.375TFLOPS \tag{4.5}$$

Diciamo che quindi si ha il 3.75% della performance massima.

4.2 Esempio con Matrix Multiplication

$$P = M \cdot N \tag{4.6}$$

Da questo esempio possiamo imparare una tecnica che riduce il numero di accessi alla memoria globale.

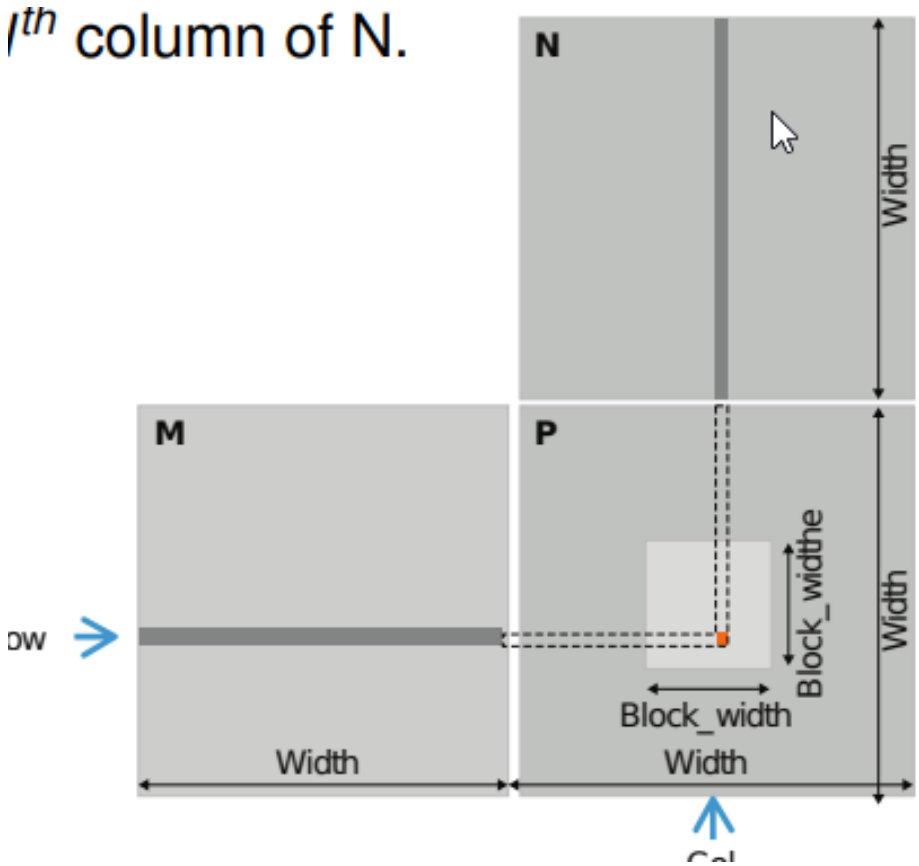

Figure 4.1: Matrix Multiplication

Per capirci, l'elemento $P_{row,col}$ e' dato dal prodotto della i-esima Riga di M con la i-esima colonna di N. Vediamo direttamente l'impostazione del problema:

```
1   dim3 dimGrid(Width/16.0, Width/16.0, 1);
2   dim3 dimBlock(16, 16, 1);
3   MatrixMulKernel<<<dimGrid,dimBlock>>>(M, N, P, Width);
4
5   Row = blockIdx.y*blockDim.y+threadIdx.y;
6   Col = blockIdx.x*blockDim.x+threadIdx.x;
7
8   Pvalue = 0;
9   for (int k = 0; k < Width; ++k)
10      Pvalue += M[Row*Width+k] * N[k*Width+Col];
```

Dove abbiamo inizialmente una matrice quadrata di dimensione potenza di 2. Ogni thread calcola Row e Col come coordinate utilizzando il suo ID e le dimensioni del blocco e del grid.

Quando si effetuta $Row \cdot Width + K$ si muove all'inizio della riga Row mentre il K si muove all k-esimo elemento della stessa riga.

Allo stesso modo $K \cdot Width + Col$ si muove all'inizio della k-esima riga e va all'offset dell'indiec di col.

Kernel semplice:

```
1   __global__ void MatrixMulKernel(float* M, float* N, float* P,int Width) {
```

```
2         int Row = blockIdx.y*blockDim.y+threadIdx.y;
3         int Col = blockIdx.x*blockDim.x+threadIdx.x;
4         if ((Row < Width) && (Col < Width)) {
5             float Pvalue = 0;
6             for (int k = 0; k < Width; ++k) {
7                 Pvalue += M[Row*Width+k] * N[k*Width+Col];
8             }
9         P[Row*Width+Col] = Pvalue;
10        }
11    }
```

Per ogni iterazione del ciclo for abbiamo 2 *accessi* alla memoria globale.

Abbiamo un'intensita' operazionale di: $IO = \frac{1}{4} = 0.25$

Dobbiamo sapere che il kernel e' memory bound in circa la maggior parte delle GPU di oggi.

4.2.1 Il roofline performance model

Quello che abbiamo detto prima sull'analisi delle performance e' una cosa solamente teorica. E' probabile che ci possano essere risultati diversi in base a operazioni di cache e altri aspetti.

Il roofline performance model e' un modello che permette di dire se un kernel e' memory bound oppure compute bound, dato un dispositivo di computazione.

$$\frac{FLOP}{s} = \frac{FLOP}{byte} \cdot \frac{byte}{s} \tag{4.7}$$

da questa analisi dimensionale si puo' vedere che il numero di FLOP per secondo e' dato dal numero di FLOP per byte moltiplicato per il numero di byte per secondo.

Se si applica il logaritmo ad ambo i membri si ottiene:

$$\log(FLOPS) = \log(AI) + \log(BW) \tag{4.8}$$

- AI: Arithmetic Intensity
- BW: Bandwidth
- FLOPS: Floating Point Operations per Second

Notiamo che abbiamo l'equazione di una retta nella forma $y = mx + c$ con:

- y = $\log(FLOPS)$
- x = $\log(BW)$
- m = 1

nel grafico Roofline, il picco di FLOPS e' un punto nell'asse verticlae, mentre il picco di banda di memoria

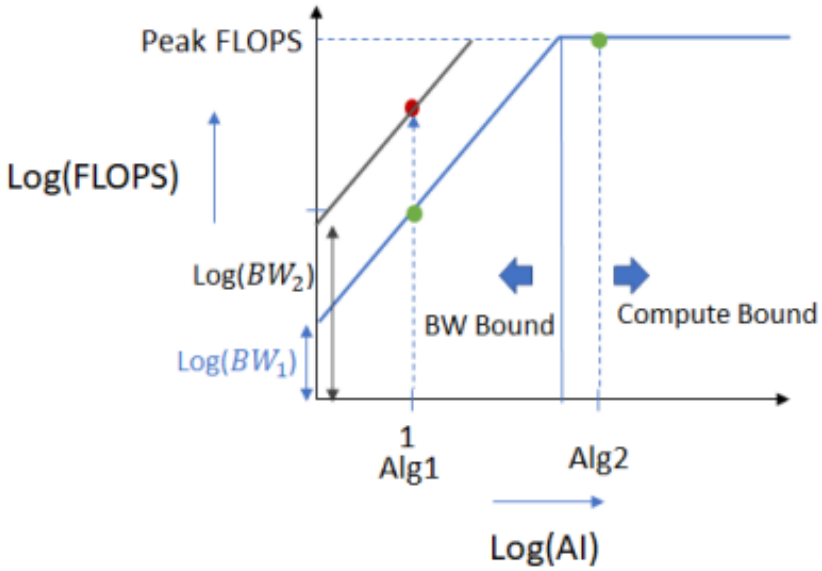

Figure 4.2: Roofline Performance Model

Vediamo che in questo caso abbiamo Alg1 e Alg2 che sono i kernel, e BW1 e BW2 che sono le bande di memoria.

- Alg1 e' memory bound, poiche' si posizione nel punto di salita della curva, e non al picco. Questo praticamente ci dice che la memoria gli impedisce di avere delle performance migliori.
- Alg2 e' compute bound, poiche' si posiziona al picco della curva, e non al punto di salita. Questo praticamente ci dice che la memoria non e' un problema per questo kernel.

Possiamo dire quindi che quando un kernel si trova nella parte precedente al picco e' memory bound, se si trova nella parte successiva al picco e' compute bound.

4.2.2 Roofline Performance Model per la Matrix Multiplication

Dobbiamo conoscere i particolari della GPU per farlo.

- GPUI fp32 peak Performance
- DRAM larghezza di banda
- Larghezza di banda della shared memory
- Numero di bytes accessi per ogni operazione, contanto le transazionid i memoria performate
- Numero di operazioni fp32 performate
- tempo passato

comandi utili:

- FLOPS e OI di un kernel: nvprof
- Quante Floating point operations e 32 bytes long transaction effettua il kernel:

```
nvprof --log-file log_oi.csv --csv --metrics flop_count_dp --
metrics flop_count_sp --metrics flop_count_hp --metrics
gld_transactions --metrics gst_transactions --metrics
atomic_transactions --metrics local_load_transactions --
metrics local_store_transactions --metrics
shared_load_transactions --metrics shared_store_transactions
--metrics l2_read_transactions --metrics l2_write_transactions
--metrics dram_read_transactions --metrics
dram_write_transactions ./mm
```

- Tempo passato:

```
nvprof --log-file log_flop.csv --csv --print-gpu-summary ./mm
```

4.3 Esempio di tiled matrix multiplication

Un altro modo per altro e' organizzare una strategia diversa per ottimizzare il kernel.

Una di queste e' quella di organizzare il problema in **tiles** che possono essere calcolate in maniera indipendente tra loro. In questo modo ogni tile puo' essere caricata in shared memory e calcolata da un blocco di thread.

Consideriamo 4 blocchi e dimensione del tile guale a quella dei blocchi.

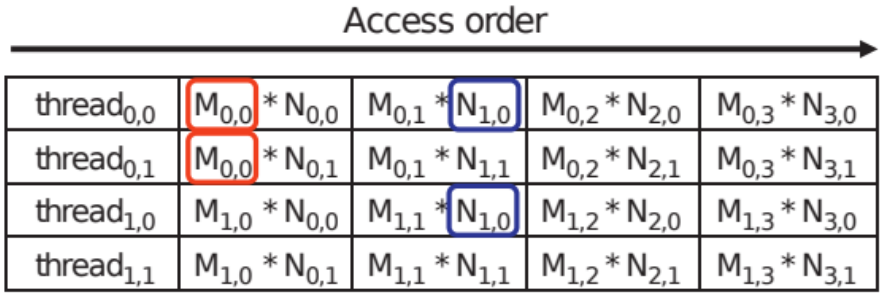

Figure 4.3: Ordine accesso

In questo esempio ogni thread accede a 4 elementi di M 4 elementi di N. Notiamo che dei thread accedono alla stessa area di memoria durante l'esecuzione. Se tutti e 4 i thread potessero collaborare, gli accessi delle memoria potrebbero ridursi di meta'.

Se abbiamo $Width \times Width$, la riduzione potenziale dell'accesso alla memoria gloale sarebbe Width.

$$16 \times 16 \; blocks = \frac{1}{6} \tag{4.9}$$

questo se collaborassero.

L'idea e' quella che i thread caricassero parti di M e N nella memoria condivisa prima di utilizzare gli elementi nel calcolo del prodotto.

Bisogna fare attenzione perche' molte volte la memoria condivisa delle GPU non e' cosi grande.

- Il prodotto viene diviso in fasi
- In ogni fase, tutti i thread del blocco collaborano per caricare una tile di M e una tile di N nella memoria condivisa.
- Ogni thread nel blocco carica un elemento di M e un elemento di N nella memoria condivisa

```
1     __global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
2     __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
3     __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
4     int tr = threadIdx.y;
5     int tc = threadIdx.x;
6     int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
7     int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;
8     float Pvalue = 0;
9     for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
10        Mds[tr][tc] = M[Row*Width + (ph*TILE_WIDTH + tc)];
11        Nds[tr][tc] = N[(ph*TILE_WIDTH + tr)*Width + Col];
12        __syncthreads();
13        for (int k = 0; k < TILE_WIDTH; ++k)
14        Pvalue += Mds[tr][k] * Nds[k][tc];
15        __syncthreads();
16    }
17    P[Row*Width + Col] = Pvalue;
18 }
```

ph e' la variabile che indica la fase. In ogni fase:

- Si carica l'elemento di M nella memoria condivisa
- Si carica l'elemento di N nella memoriacondivisa
- la barriera __syncthreads()__ assicura che tutti i thread abbiano finito di caricare i dati nella memoria condivisa

Purtroppo anche questo algoritmo e' memory bound, ma e' molto piu efficiente del precedente.
Come superiamo questo limite? Bisogna superare il limite della shared memory.

Chapter 5

Convolution

La convoluzione viene usata in molti algoritmi. Solitamente la convoluzione viene chiamata anche **computazione stencil**.

Un pro della convoluzione e' che ogni elemento del dato viene calcolato in modo indipendente dagli altri.

Il contro e' che ci possono essere sovrapposizioni tra i dati di input con possibili problemi ai boundaries.

Solitamente una convoluzione e' un operazione su un array dove ogni elemento di output viene fuori dalla somma pesata degli elementi intorno a lui. I pesi sono spesso identificati come **kernel convoluzioale**.

Problemino: Cosa succede quando siamo ai bordi dell'array? I vicini come vengono calcolati? In questo caso, si parla di **celle fantasma**, cioe' celle che non danno alcun contributo alla computazione.

Chiaramente questo problema esiste anche in 2D e 3D.

Ecco un esempio di convoluzione in 1D. La maschera viene calcolata come un numero dispari uguale a $2n + 1$.

```
1     __global__
2 void convolution_1D_basic_kernel(float *N, float *M, float *P, intMask_Width, int Width)
3 {
4     int i = blockIdx.x*blockDim.x + threadIdx.x;
5     float Pvalue = 0;
6     int N_start_point = i - (Mask_Width/2);
7     for (int j = 0; j < Mask_Width; j++) {
8         if (N_start_point + j >= 0 && N_start_point + j < Width) {
9             Pvalue += N[N_start_point + j]*M[j];
10        }
11    }
12    P[i] = Pvalue;
13 }
```

Per i blocchi interni senza celle fantasma, ogni thread accede $Mask\_Width$ volte alla memoria globale, per u totale di: $blockDim.x * Mask\_Width$ accessi alla memoria globale.

La cella fantasma piu a sinistra e' usata da un solo thread, mentre quella piu a destra e' usata da n thread. Abbiamo un totale di accessi evitati di $n * (n + 1)$. **Quindi, il numero totale di accessi alla memoria globale e':**

$$(blockDim.x * 2n + 1) - n * (n + 1) \tag{5.1}$$

Notiamo che i costi dipendono dalla $Width$ ovvero la size dell'array in input e dalla $Mask\_Width$ la grandezza della maschera.

Ci sono 3 proprieta' nel modo in cui l'array mask M e' usato **memoria costante e caching**

- La grandezza di M e' spesso piccola
- Il contenuto di M non cambia mai
- Tutti i threads accedono gli elementi di M nello stesso ordines

Il rapporto di operazioni in virgola mobile e' circa 1:1 con gli accessi alla memoria globale., con un intensita operazionale di 1/4, cioe' 1 operazione ogni 4 accessi alla memoria globale.

La memoria costante e' come la memoria globale, ma e' **64kb**, read-only e cached in L1.

5.0.1 Come usare la memoria costante?

Si dichiara un array M come **variabile globale**, cioe fuori da una funzione

```
1    #define MAX_MASK_WIDTH 10
2    __constant__ float M[MAX_MASK_WIDTH];
```

Se abbiamo che M_h salva la maschera in memoria host, allora possiamo copiare la maschera in memoria device con:

```
1    cudaMemcpyToSymbol(M, M_h, Mask_Width * sizeof(float));
```

dove M e' destinazione, M_h e' sorgente, Mask_Width e' la size della maschera.

Allora il kernel accede a M come un oggetto globale, ma in realta' accede alla memoria costante.

Non ci sara' bisogno di passare la maschera come parametro al kernel.

Il rapporto di operazioni in virgola mobile e' passato da 1 a 2, con un intensita operazionale di 1/2, cioe' 1 operazione ogni 2 accessi alla memoria globale.

5.0.2 Convoluzione 1D con Halo Cells

Immaginiamo di avere un array di 16 elementi. Ogni thread calcola un elemento in output in P. Abbiamo 4 blocchi di 4 threads. Ogni blocco calcola 4 elementi in P.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Table 5.1: Array with 16 elements

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Table 5.2: 4 tiles with a block of 4

Un algoritmo tiled caricherebbe tutti gli elementi per calcolare tutti quelli in output nella shared memory per ogni blocco.

Immaginiamo di avere una mask size come numero dispari uguale a $2n + 1$.

Se immaginassimo di avere un $n = 2$, avremmo una situazione del genere:

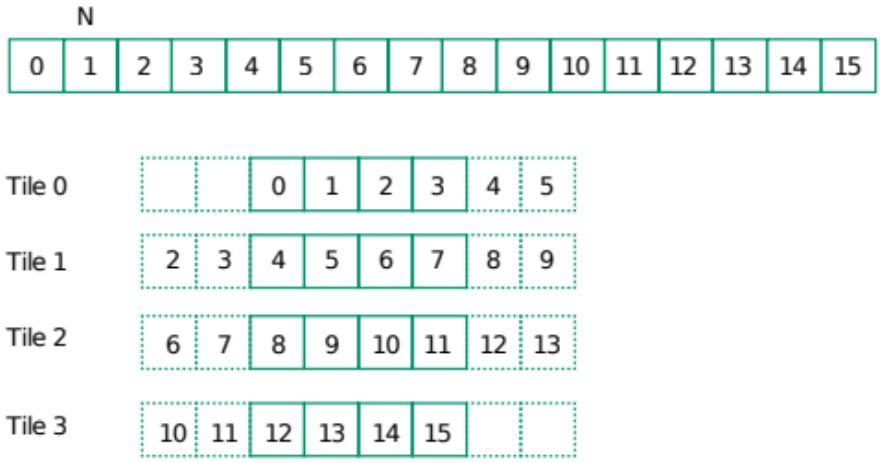

Figure 5.1: Convolution 1D with Halo Cells

Cioe, le celle che vediamo tratteggiate sono considerate **celle halo**. Solitamente si fa riferimento alle celle halo come celle che che sono coinvolte in piu tiles.

Consideriamo i tiles 0 e 3 come tiles esterni, mentre 1 e 2 come tiles interni.

Se vediamo, ogni halo appartiene al prossimo tile, quindi praticamente viene caricato **due volte** quell'elemento.

Consideriamo nel prossimo algoritmo le seguenti proprieta:

- The mask has size $2 * n + 1$;
- $TILE\_SIZE$ = blockDim.x;
- I primi n threads, cioe' quelli che hanno indice minore di n, e gli ultimi n threads, cioe quelli che hanno indice maggiore di blockDim-n, faranno piu lavoro caricando le celle halo nel buffer condiviso N_ds

```
1  __global__ void convolution_1D_tiled_kernel(float *N, float *P,
2      int Mask_Width, int Width) {
3      int i = blockIdx.x*blockDim.x + threadIdx.x; int n = Mask_Width/2;
4      __shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
5      if (threadIdx.x < n) {
6          int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
7          N_ds[n + blockDim.x + threadIdx.x] =
8              (halo_index_right >= Width) ? 0 : N[halo_index_right];
9      }
10     N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];
11     if (threadIdx.x >= blockDim.x - n) {
12         int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
13         N_ds[threadIdx.x - (blockDim.x - n)] =
14             (halo_index_left < 0) ? 0 : N[halo_index_left];
15     }
16     __syncthreads();
17     float Pvalue = 0;
```

```
18      for(int j = 0; j < Mask_Width; j++) {
19          Pvalue += N_ds[threadIdx.x + j]*M[j];
20      }
21      P[i] = Pvalue;
22  }
```

Notiamo che il kernel aumenta di complessita'. Questa complessita' riduce il numero di accessi alla DRAM per gli N elementi e questo aumenta l'aritmetica per ogni accesso alla DRAM.

- Rispetto al kernel base, tutti gli accessi alla memoria globale corrispondono al caricamento degli N elementi nella memoria condivisa
- Ogni elemento di N viene caricato da un solo thread. Pero', 2n celle halo saranno caricate, n da sinistra e n da destra, per bocchi che non gestiscono le celle fantasma.
- Abbiamoq uindi blockDim.x+2n elementi caricati dai blocchi interni e blockDim+n caricati dai blocchi esterni
- Per i blocchi interni, il rapporto di accessi di memoria tra il kernel base e il tiled 1d convolution e' $\frac{(blockDim.x*(2n+1))}{(blockDim.x+2n)}$
- Per i blocchi esterni, il rapporto e': $\frac{blockDim.x*(2n+1)}{blockDim.x} = 2n+1 = Mask\_Width$

A quanto pare le GPU recenti hanno L1 e L2 caches, dove L1 e' privata e L2 e' condivisa tra tutti gli SMs

Siccome CUDA ha delle policy di scheduling specifiche, c'e' una grande probabilita' che le celle halo di un dato blocco siano disponibili nella cache L2.

Allora, possiamo lasciare gli accessi a queste celle halo negli N elementi originali, invece di caricarli in N_ds.

```
1       __global__ void convolution_1D_tiled_kernel(float *N, float *P, int
2   Mask_Width, int Width)
3   {
4       __shared__ float N_ds[TILE_SIZE];
5       int i = blockIdx.x*blockDim.x + threadIdx.x;
6       N_ds[threadIdx.x] = N[blockIdx.x*blockDim.x+threadIdx.x];
7       __syncthreads();
8       int This_tile_start_point = blockIdx.x * blockDim.x;
9       int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
10      float Pvalue = 0;
11      for (int j = 0; j < Mask_Width; j++)
12      {
13          int N_index = i - (Mask_Width/2) + j;
14          if (N_index >= 0 && N_index < Width)
15              if ((N_index >= This_tile_start_point)
16              && (N_index < Next_tile_start_point))
17                  Pvalue += N_ds[threadIdx.x - (Mask_Width/2) + j]*M[j];
18              else
19              Pvalue += N[N_index] * M[j]; //N[N_index] is hopefully cached
```

```
20      }
21      P[i] = Pvalue;
22  }
```

Praticamente con l'if stiamo dicendo che se N_index e' interno al blocco corrente, l'elemento viene accesso in N_ds, altrimenti viene accesso in N, che si spera sia nella cache L2.

L'idea e' quella di definire la block size uguale alla $TILE\_WIDTH$ piu' il numero di celle necessarie dalle celle halo per ogni direazione.

- Nella prima fase i blocchi di thread copiano i valori di input nella input tile (incluse le halo) e poi sincronizzano
- Nella seconda fase, solamente $TILE\_WIDTH * TILE\_WIDTH$ threads aggiornano la tile output.
- La griglia CUDA viene determinata basandodsi sulla $TILE\_SIZE$ e non la block size.

```
1       __global__ void convolution_2D_tiled(float *P, float *N, int height, int
2   width, int Mask_Width, const float __restrict__ *M){
3       int i = blockIdx.y*TILE_WIDTH + threadIdx.y;
4       int j = blockIdx.x*TILE_WIDTH + threadIdx.x;
5       int i_halo = i - Mask_Width/2;
6       int j_halo = j - Mask_Width/2;
7       __shared__
8       float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1][TILE_SIZE+MAX_MASK_HEIGHT-1];
9       // Threads load the input tiles into the shared memory.
10      // If the input element the thread is attempting to load is actually
11      // a ghost element a 0.0 value is placed into the shared memory.
12      if((i_halo >= 0)&&(i_halo < height)&&(j_halo >= 0)&&(j_halo < width))
13          N_ds[threadIdx.y][threadIdx.x] = N[i_halo * width + j_halo];
14      else
15          N_ds[threadIdx.y][threadIdx.x] = 0.0f;
16      __syncthreads();
17      float output = 0.0f;
18      // Only the threads whose indices are both smaller than
19      // the TILE_WIDTH must compute
20      if(threadIdx.y < TILE_WIDTH && threadIdx.x < TILE_WIDTH)
21      {
22          for(i_mask = 0; i_mask < MASK_WIDTH; i_mask++)
23              for(j_mask = 0; j_mask < MASK_WIDTH; j_mask++)
24              output += M[i_mask][j_mask] * N_ds[threadIdx.y+i_mask]
25                  [threadIdx.x+j_mask];
26          if(i < height && j < width)
27              N[i*width + j] = output;
28      }
29  }
```

In un kernel base, ogni thread in un blocco di thread effettua un numero di accessi all'array immage pari a $Mask\_width^2$, per un totale di $Mask\_width^2 * TILE\_WIDTH^2$ accessi per blocco,

Nel kernel tiled, un blocco carica collettivamente una tile input, risultando in un numero di accessi pari a $(TILE\_WIDTH + Mask\_width - 1)^2$.

Il rapporto di accessi all'array immagine tra il kernel base e quello tiled e' quindi:

$$\frac{Mask\_width^2 * TILE\_WIDTH^2}{(TILE\_WIDTH + Mask\_width - 1)^2} \tag{5.2}$$

Maggiore e' $TILE\_WIDTH$, la grandezza della maschera diventa piu trascurabile comparata alla grandezza della tile. Quindi, il rapporto si avvicina a $(Mask\_width)^2$

Abbiamo bisogo di una grossa TILE_WIDTH per ottimizzare le performance, anche avendo una grande quantita' di shaed memory.