Università Politecnica delle Marche

Sistemi Embedded

Implementazione dell'algoritmo di One-Sided Jacobi Rotation per la decomposizione SVD tramite librerie NVIDIA CUDA C su piattaforma embedded Jetson TK1 (GPU).

Matteo Orlandini
Jacopo Pagliuca

16 aprile 2020

Indice

1 Introduzione **1**
1.1 SVD 1
1.2 One Sided Jacobi SVD 3
1.2.1 Jacobi rotation 3
1.2.2 Algoritmo One Sided Jacobi 4

2 CUDA **5**
2.1 Calcolo parallelo 5
2.2 Calcolo sequenziale e parallelo 6
2.3 Struttura di programmazione CUDA 7
2.4 Organizzazione della memoria 8
2.5 Organizzazione dei thread 10
2.6 Modello della memoria CUDA 12
2.6.1 Registri 13
2.6.2 Local Memory 14
2.6.3 Shared Memory 14
2.6.4 Constant Memory 15
2.6.5 Texture Memory 16
2.6.6 Global Memory 16

3 Implementazione sequenziale **18**

4 Implementazione parallela **22**
4.1 Global Memory 25
4.2 Semi Shared Memory 26
4.3 Shared Memory 27

5 Performance **30**

6 Conclusioni **36**

1 Introduzione

La seguente tesina descrive il lavoro svolto per la progettazione di un algoritmo di One-sided Jacobi per la decomposizione in valori singolari (SVD) di una matrice, tramite librerie NVIDIA CUDA C. Il codice è stato testato e pensato per una piattaforma embedded Jetson TK1.

La Jetson TK1 è una scheda grafica embedded realizzato dalla NVIDIA che contiene un processore Tegra K1 SoC nella variante T124. Inoltre essa possiede un sistema operativo Ubuntu Linux ed ha installata la versione di CUDA 6.5. La compute capability della GPU, cioè la capacità di calcolo che determina le specifiche generali e le funzionalità disponibili, è pari a 3.2.

In facoltà era già stato sviluppato un codice in C per la SVD ottimizzato per lavorare su CPU. Il nostro lavoro era quello di confrontare i nostri risultati con quelli precedentemente ottenuti e valutare la possibilità di una implementazione alternativa che sfruttasse appieno le potenzialità del calcolo parallelo della GPU della Jetson.

Come base per il nostro lavoro abbiamo studiato il linguaggio di programmazione e l'architettura CUDA in [3] e [6]. In seguito sono stati analizzati gli articoli [1], [2], [4] e [5] che descrivevano vari approcci per l'ottimizzazione dell'algoritmo su GPU, per poi realizzare la nostra versione, i cui risultati sono molto simili a quelli ottenuti negli articoli.

Per lavorare sulla scheda abbiamo sfruttato il protocollo SSH dal dipartimento, in modo da connettersi alla scheda da remoto. Alla fine della realizzazione del progetto non è però stato possibile accedere al dipartimento e quindi riportare i risultati esatti del tempo computazionale che necessita l'algoritmo sulla Jetson. Abbiamo quindi usato come riferimento una NVIDIA GTX 610M, i cui risultati sono proporzionali a quelli ottenuti con la Jetson.

1.1 SVD

In algebra lineare, la decomposizione ai valori singolari (SVD), è una fattorizzazione di una matrice in tre diverse matrici basata sull'uso di autovalori e autovettori.

La decomposizione di una matrice $\mathbf{A}$ si basa sul *teorema fondamentale dell'algebra lineare*:

Data una matrice $\mathbf{A} \in \mathbb{C}^{m \times n}$ di rango ρ, la decomposizione in valori singolari di $\mathbf{A}$ è rappresentata dal prodotto di due matrici unitarie $\mathbf{U} \in \mathbb{C}^{m \times m}$, $\mathbf{V} \in \mathbb{C}^{n \times n}$ e una matrice diagonale $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$, come mostrato in (1).

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{1}$$

dove $\mathbf{V}^*$ è la trasposta coniugata della matrice unitaria $\mathbf{V}$. Le colonne di $\mathbf{U}$ sono chiamate *vettori singolari sinistri* di $\mathbf{A}$ mentre le colonne di $\mathbf{V}$ sono i *vettori singolari destri* di $\mathbf{A}$. Inoltre, $\mathbf{\Sigma}$ è una matrice reale non negativa del tipo

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \sigma_\rho & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Gli elementi diagonali di $\mathbf{\Sigma}$ sono i *valori singolari* di $\mathbf{A}$ e sono solitamente in ordine decrescente: $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_\rho > 0$, $\sigma_{\rho+1} = ... = \sigma_n = 0$.

Nel caso in cui $\mathbf{A} \in \mathbb{R}^{m \times n}$ l'equazione (1) diventa:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{2}$$

dove $\mathbf{U} \in \mathbb{R}^{m \times m}$ e $\mathbf{V} \in \mathbb{R}^{n \times n}$ sono matrici ortonormali.

$$\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = I^{m \times m} \tag{3}$$
$$\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = I^{n \times n} \tag{4}$$

Unendo queste due proprietà all'equazione (2) si possono ricavare l'espressione di $\mathbf{A}\mathbf{A}^T$:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}\mathbf{U}^T = \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T$$

Allo stesso modo, per $\mathbf{A}^T\mathbf{A}$:

$$\mathbf{A}^T\mathbf{A} = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T$$

La SVD ha numerose applicazioni nel campo dell'algebra lineare. Innanzitutto fornisce delle informazioni importanti sulla matrice $\mathbf{A}$, come il suo rango, qual è il suo nucleo e qual è la sua immagine. Viene usata per definire la pseudo-inversa di una matrice rettangolare utile per la risoluzione del problema dei minimi quadrati. Trova utilizzo anche nella risoluzione di sistemi di equazioni lineari omogeneo.

Un'altra importante applicazione riguarda l'approssimazione della matrice $\mathbf{A}$, con una di rango inferiore (SVD troncata), utilizzata nell'elaborazione di immagini e nell'elaborazione dei segnali.

La SVD ha anche note applicazioni nel campo dell'analisi delle componenti principali.

1.2 One Sided Jacobi SVD

Per effettuare la SVD di una matrice, sono stati sviluppati numerosi algoritmi con lo scopo di ottimizzare il numero di operazioni svolte dalla macchina. Uno di quelli più usati è l'algoritmo di Jacobi con la sua variante One Sided Jacobi. L'approccio utilizzato è quello di applicare successive rotazioni alla matrice originale, in modo da azzerare le componenti che si trovino al di fuori della diagonale. Tramite diverse iterazioni, si ottiene come risultato finale una matrice diagonale contenente i valori singolari richiesti.

1.2.1 Jacobi rotation

La rotazione di Jacobi è una operazione che consente di azzerare selettivamente elementi specifici di una matrice. Tramite una rotazione in due dimensioni p e q vengono azzerati gli elementi (p, q) e (q, p) della matrice.

L'operazione si basa sull'utilizzo della matrice di Jacobi $\mathbf{J}(p, q, \theta)$ del tipo:

$$\mathbf{J}(p,q,\theta) = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & 0 & \\ & & c & \cdots & -s & & \\ & & \vdots & \ddots & \vdots & & \\ & & s & \cdots & c & & \\ & 0 & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \tag{5}$$

dove $c = \cos(\theta)$ e $s = \sin(\theta)$, vengono applicati solo nelle dimensioni p e q.

Premoltiplicare un vettore per $\mathbf{J}(p, q, \theta)^T$ equivale a ruotarlo in senso antiorario di un angolo θ nel piano (p, q). Questa rotazione produce un vettore risultante nel quale la componente q è stata azzerata. Infatti se $\mathbf{x} \in \mathbb{R}^n$ e

$$\mathbf{y} = \mathbf{J}(p, q, \theta)^T \cdot \mathbf{x} \tag{6}$$

allora

$$y_p = cx_p - sx_q \tag{7}$$
$$y_q = sx_p + cx_q \tag{8}$$
$$y_i = x_i, \ i \neq p, q \tag{9}$$

Da (8) si nota che y_q può essere azzerata ponendo:

$$c = \frac{x_p}{\sqrt{x_p^2 + x_q^2}}, \quad s = \frac{-x_q}{\sqrt{x_p^2 + x_q^2}} \tag{10}$$

Per una matrice $\mathbf{A}$ simmetrica, è possibile azzerare le componenti (p, q) e (q, p) applicando la (11).

$$\mathbf{B} = \mathbf{J}(p, q, \theta)^T \mathbf{A}\mathbf{J}(p, q, \theta) \tag{11}$$

1.2.2 Algoritmo One Sided Jacobi

L'idea base per questo algoritmo è quella di ruotare le colonne i e j di $\mathbf{A}$ per renderle ortogonali. In questo modo, le colonne i e j di $\mathbf{A}^T\mathbf{A}$ saranno implicitamente ortogonali.

Sia $\mathbf{J}(p, q, \theta)$ la matrice di rotazione che, applicata ad $\mathbf{A}$, produce:

$$\mathbf{B} = \mathbf{AJ} \tag{12}$$

dove le colonne i e j di $\mathbf{B}$ sono date da

$$b_{:i} = ca_{:i} - sa_{:j} \tag{13}$$
$$b_{:j} = sa_{:i} + ca_{:j} \tag{14}$$

l'elemento $(B^TB)_{ij}$ sarà

$$(B^TB)_{ij} = b_{:i}^T b_{:j} = (ca_{:i} - sa_{:j})^T(sa_{:i} + ca_{:j})$$

assumendo $(B^TB)_{ij} = 0, i \neq j$ si ottiene

$$cs(||a_{:i}||^2 - ||a_{:j}||^2) + (c^2 - s^2)(sa_{:i} + ca_{:j})$$

Dividendo per c^2 e considerando

$$\begin{aligned} t &= s/c \\ \alpha &= ||a_{:i}||^2 \\ \beta &= ||a_{:j}||^2 \\ \gamma &= a_{:i}^T a_{:j} \\ \tau &= (\beta - \alpha)/2\gamma \end{aligned}$$

si ottiene la seguente espressione:

$$t^2 + 2\tau t - 1 = 0 \tag{15}$$

Risolvendo e scegliendo come soluzione la radice con il minore valore assoluto,

$$t = min|-\tau \pm \sqrt{1 + \tau^2}| \tag{16}$$

ottieniamo i valori di c e s che rappresentano coseno e seno nella matrice di rotazione:

$$c = \frac{1}{\sqrt{1 + t^2}}, \quad s = ct \tag{17}$$

Con c e s così calcolati, gli elementi $(B^TB)_{ij}$ e $(B^TB)_{ji}$ saranno uguali a zero.[7]

2 CUDA

2.1 Calcolo parallelo

Negli ultimi decenni, c'è stato un crescente interesse per il calcolo parallelo. L'obiettivo primario del calcolo parallelo è migliorare la velocità di computazione e può essere definito come una forma di calcolo in cui molte operazioni vengono eseguite simultaneamente, in base al principio che spesso i problemi di grandi dimensioni possono essere suddivisi in problemi più piccoli, che vengono poi risolti contemporaneamente.

Dal punto di vista del programmatore, una domanda naturale è come mappare i calcoli simultanei sui computer. Supponendo di avere più risorse informatiche, il calcolo parallelo può quindi essere definito come l'uso simultaneo di più risorse di calcolo (core o computer) per eseguire i calcoli simultanei. Un grosso problema viene suddiviso in problemi più piccoli, ciascuno dei quali viene quindi risolto contemporaneamente su diverse risorse di elaborazione. Gli aspetti software e hardware del calcolo parallelo sono strettamente intrecciati insieme. In effetti, il calcolo parallelo di solito coinvolge due aree distinte delle tecnologie informatiche:

- Computer architecture (aspetto hardware)
- Parallel programming (aspetto software)

La computer architecture si concentra sul supporto del parallelismo a livello architettonico, mentre il parallel programming si concentra sulla risoluzione di un problema simultaneamente sfruttando appieno il potere computazionale dell'architettura del computer. Per ottenere l'esecuzione parallela nel software, l'hardware deve fornire una piattaforma che supporti l'esecuzione simultanea di più processi o thread multipli.

I processori più moderni implementano l'*architettura Harvard*, come mostrato in Figura 1, che comprende tre componenti principali:

- Memoria (instruction memory e data memory)
- Central processing unit (control unit e arithmetic logic unit)
- Interfacce di Input/Output

La componente chiave nell'elaborazione è la CPU, generalmente chiamata core. Al giorno d'oggi, la tendenza nella progettazione dei chip è quella di integrare più core in un singolo processore, generalmente definito multicore,

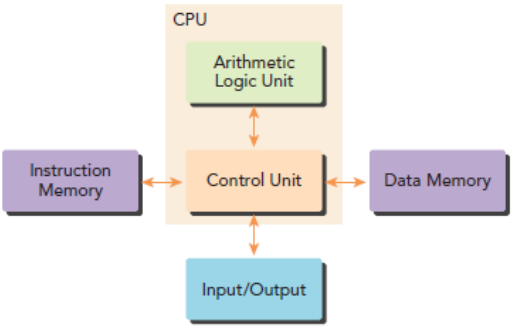


Figura 1: Architettura Harvard [3]

per supportare il parallelismo a livello di architettura. Pertanto, la programmazione può essere vista come il processo di mappatura del calcolo di un problema ai core disponibili in modo tale da ottenere l'esecuzione parallela.

Quando si implementa un algoritmo sequenziale, potrebbe non essere necessario comprendere i dettagli dell'architettura del computer per scrivere un programma corretto. Tuttavia, quando si implementano algoritmi per macchine multicore, è molto più importante che i programmatori siano consapevoli delle caratteristiche dell'architettura del computer sottostante. La scrittura di programmi paralleli sia corretti che efficienti richiede una conoscenza fondamentale delle architetture multicore.

2.2 Calcolo sequenziale e parallelo

Quando si scrive un programma, è naturale dividere il problema in una serie discreta di calcoli; ogni calcolo esegue un'attività specifica, come mostrato in Figura 2. Questo tipo di programma è chiamato *sequenziale*. Esistono due

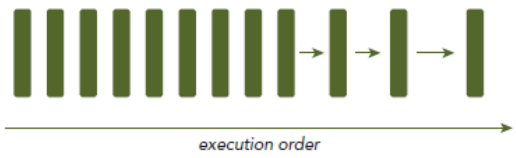


Figura 2: Ordine di esecuzione sequenziale [3]

modi per classificare la relazione tra due pezzi di codice: alcuni sono correlati da un vincolo di precedenza e pertanto devono essere calcolati in sequenza; altri non hanno tali restrizioni e possono essere calcolati contemporaneamente. Qualsiasi programma contenente attività eseguite contemporaneamente è chiamato *parallelo*. Come mostrato in Figura 3, un programma parallelo può avere alcune parti sequenziali.

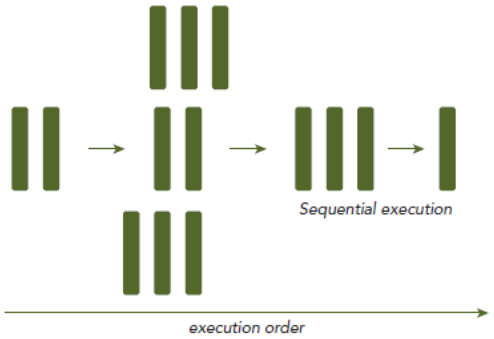


Figura 3: Ordine di esecuzione parallelo [3]

2.3 Struttura di programmazione CUDA

Il modello di programmazione CUDA consente di eseguire applicazioni su sistemi di elaborazione eterogenei semplicemente annotando il codice con una piccola serie di estensioni al linguaggio di programmazione C. Un ambiente eterogeneo è costituito da CPU integrate da GPU, ognuna con la propria memoria separata da un bus PCI-Express. Pertanto, è necessario notare la seguente distinzione:

- **Host**: la CPU e la sua memoria (host memory)
- **Device**: la GPU e la sua memoria (device memory)

Un componente chiave del modello di programmazione CUDA è il kernel, cioè il codice che viene eseguito sul device GPU. CUDA gestisce i kernel scritti dai programmatori su thread GPU. Dall'host, si definisce il modo in cui l'algoritmo viene mappato sul device in base ai dati dell'applicazione e alla capacità del device.

L'host può funzionare indipendentemente dal dispositivo per la maggior parte delle operazioni. Quando si chiama un kernel, il controllo viene immediatamente restituito all'host, liberando la CPU per eseguire task aggiuntivi. Un tipico programma CUDA è costituito da un codice seriale integrato da un codice parallelo. Come mostrato in Figura 4, il codice seriale viene eseguito sull'host, mentre quello parallelo viene eseguito sul device. Il codice del device è scritto usando CUDA C. È possibile inserire tutto il codice in un singolo file sorgente oppure utilizzare più file sorgente per creare l'applicazione o le librerie. Il compilatore NVIDIA C (nvcc) genera il codice eseguibile sia per l'host che per il device.

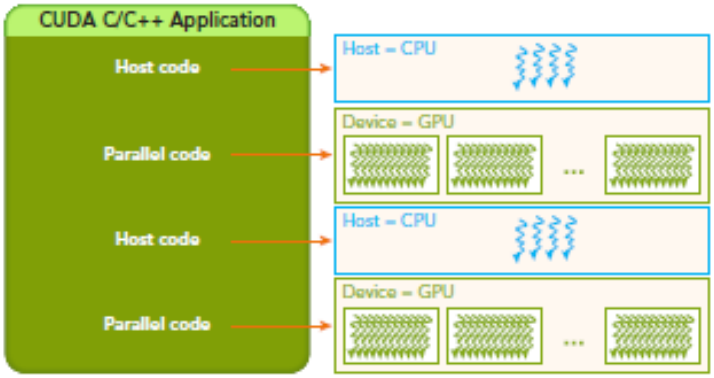


Figura 4: Struttura di programmazione Cuda [3]

Un flusso di elaborazione tipico di un programma CUDA segue questo modello:

1. Copia i dati dalla memoria della CPU alla memoria della GPU.
2. Chiama i kernel per operare sui dati memorizzati nella memoria della GPU.
3. Copia i dati dalla memoria della GPU alla memoria della CPU.

2.4 Organizzazione della memoria

Il modello di programmazione CUDA presuppone un sistema composto da un host e un device, ognuno con una propria memoria separata. I kernel funzionano con la memoria del device. Per consentire il pieno controllo e ottenere le migliori prestazioni, CUDA fornisce funzioni per allocare la memoria del dispositivo, liberare la memoria del dispositivo e trasferire i dati tra la memoria host e quella device. La Tabella 1 elenca le funzioni C standard e le corrispondenti funzioni CUDA C per le operazioni sulla memoria. La fun-

FUNZIONI C STANDARD	FUNZIONI CUDA C
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

Tabella 1: Funzioni sulla memoria host e device

zione utilizzata per eseguire l'allocazione della memoria GPU è cudaMalloc e la sua struttura è:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

Questa funzione alloca un intervallo lineare di memoria del dispositivo con la *size* specificata in byte. La memoria allocata viene restituita tramite *devPtr*. La somiglianza tra cudaMalloc e la libreria runtime standard malloc è intenzionale e ha lo scopo di mantenere l'interfaccia il più vicino possibile alle librerie di runtime C standard.

La funzione utilizzata per trasferire i dati tra l'host e il device è cudaMemcpy e la sua struttura è:

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t
    count, cudaMemcpyKind kind )
```

Questa funzione copia i byte specificati dall'area di memoria di origine, indicata da *src*, nell'area di memoria di destinazione, indicata da *dst*, con la direzione specificata dal tipo, dove *kind* assume uno dei seguenti tipi:

- cudaMemcpyHostToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Questa funzione mostra un comportamento sincrono perché l'applicazione host si blocca fino a quando cudaMemcpy ritorna e il trasferimento è completo. Ogni chiamata CUDA, ad eccezione del lancio del kernel, restituisce un codice di errore di tipo enumerato cudaError_t. Ad esempio, se la memoria GPU è allocata correttamente, restituisce:

cudaSuccess

Altrimenti, ritorna:

cudaErrorMemoryAllocation

È possibile convertire un codice di errore in un messaggio di errore human-readable con la seguente funzione di runtime CUDA:

```
char* cudaGetErrorString(cudaError_t error)
```

La funzione cudaGetErrorString è analoga alla funzione standard C strerror. Il modello di programmazione CUDA espone un'astrazione della gerarchia di memoria dall'architettura GPU. La Figura 5 illustra una struttura di memoria GPU semplificata, contenente due ingredienti principali: global memory e shared memory. Un approfondimento sulla gerarchia di memoria della GPU è presente nel capitolo 2.6. Una delle caratteristiche più notevoli

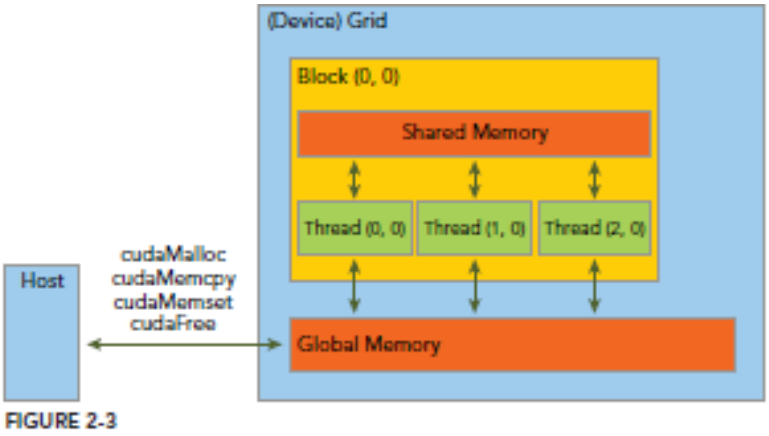


Figura 5: Struttura della memoria GPU [3]

del modello di programmazione CUDA è la gerarchia di memoria esposta. Ogni dispositivo GPU ha un set di diversi tipi di memoria utilizzati per scopi diversi. Un approfondimento su questa gerarchia è presente nel capitolo 2.6. La global memory è analoga alla memoria di sistema della CPU, mentre la shared memory è simile alla cache della CPU. Tuttavia, la shared memory può essere controllata direttamente da un kernel CUDA C.

2.5 Organizzazione dei thread

Quando un kernel viene avviata dal lato host, l'esecuzione viene spostata sul device in cui viene generato un numero elevato di thread e ogni thread esegue le istruzioni specificate dalla funzione kernel. Saper organizzare i thread è una parte fondamentale della programmazione CUDA che espone un'astrazione della gerarchia di thread per consentire di organizzare i thread. Questa è una gerarchia di thread a due livelli composta in blocchi di thread e griglie di blocchi, come mostrato in Figura 6. Tutti i thread generati da un singolo lancio del kernel sono collettivamente chiamati griglia. Tutti i thread in una griglia condividono lo stesso spazio di global memory. Una griglia è composta da molti blocchi di thread. Un blocco è un gruppo di thread che possono cooperare tra loro usando:

- sincronizzazione dei blocchi locali
- shared memory dei blocchi locali

Thread di blocchi diversi non possono cooperare. I thread si basano sulle seguenti due coordinate univoche per distinguersi l'una dall'altra:

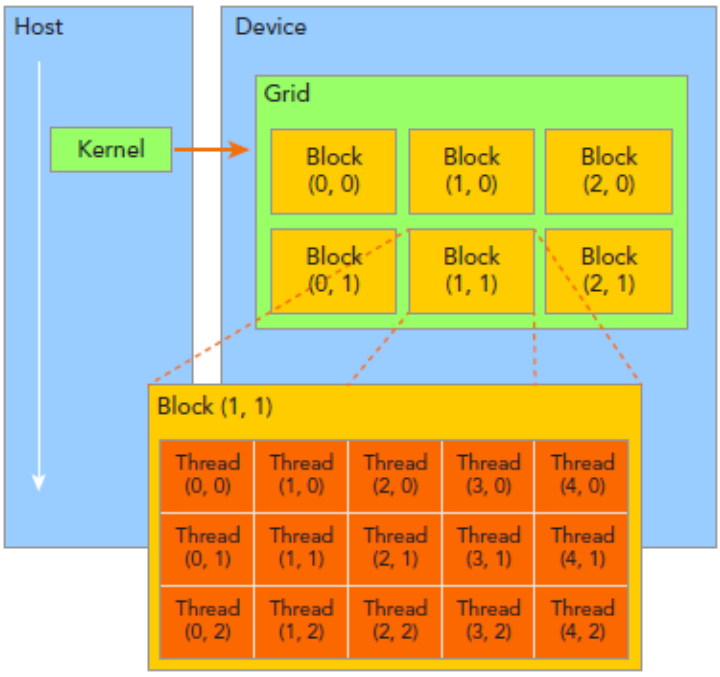


Figura 6: Gerarchia dei thread [3]

- blockIdx (indice del blocco all'interno di una griglia)
- threadIdx (indice del thread all'interno di un blocco)

Queste variabili appaiono come variabili predefinite pre-inizializzate a cui è possibile accedere nei kernel. Quando viene eseguita un kernel, le variabili di coordinate blockIdx e threadIdx vengono assegnate a ciascun thread dal CUDA runtime. Sulla base delle coordinate, è possibile assegnare porzioni di dati a thread diversi.

La variabile di coordinate è di tipo uint3. È una struttura contenente tre numeri interi senza segno e la prima, seconda e terza componente è accessibile attraverso i campi x, y e z.

blockIdx.x
blockIdx.y
blockIdx.z
threadIdx.x
threadIdx.y
threadIdx.z

CUDA organizza griglie e blocchi in tre dimensioni. La Figura 6 mostra un esempio di una struttura gerarchica di thread con una griglia 2D conte-

nente blocchi 2D. Le dimensioni di una griglia e di un blocco sono specificate dalle seguenti due variabili integrate:

- blockDim (dimensione del blocco, misurata in threads)
- gridDim (dimensione della griglia, misurata in blocchi)

Queste variabili sono di tipo dim3, un tipo di vettore intero basato su uint3 utilizzato per specificare le dimensioni. Quando si definisce una variabile di tipo dim3, qualsiasi componente lasciato non specificato viene inizializzato su 1. Ogni componente in una variabile di tipo dim3 è accessibile attraverso i suoi campi x, y e z, rispettivamente, come mostrato dai campi mostrati di seguito:

```
blockDim.x
blockDim.y
blockDim.z
```

2.6 Modello della memoria CUDA

Per i programmatori, ci sono generalmente due classificazioni di memoria:

- Programmabile: si controlla esplicitamente quali dati vengono inseriti nella memoria programmabile.
- Non programmabile: non si ha alcun controllo sul posizionamento dei dati e si fa affidamento su tecniche automatiche per ottenere buone prestazioni.

Nella gerarchia di memoria della CPU, la cache L1 e la cache L2 sono esempi di memoria non programmabile. CUDA espone molti tipi di memoria programmabile:

- Registers
- Shared memory
- Local memory
- Constant memory
- Texture memory
- Global memory

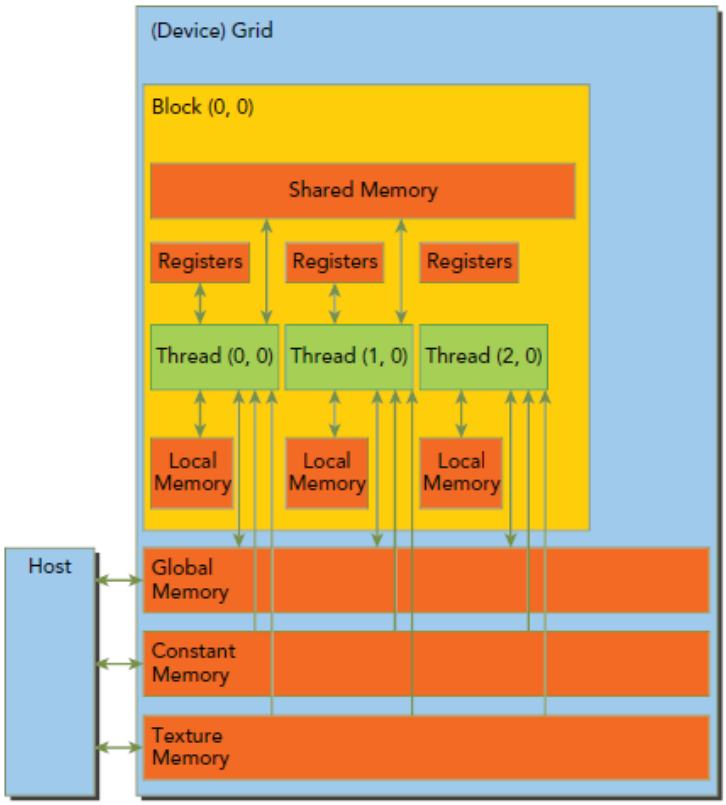


Figura 7: Gerarchia della memoria della GPU [3]

La Figura 7 illustra la gerarchia di questi spazi di memoria. Ognuno ha un diverso ambito, durata e comportamento di memorizzazione nella cache. Un thread in un kernel ha la sua local memory privata. Un blocco di thread ha una propria memoria shared, visibile a tutti i thread nello stesso blocco thread e il cui contenuto persiste per la durata del blocco thread. Tutti i thread possono accedere alla memoria globale. Esistono anche due spazi di memoria di sola lettura accessibili da tutti i thread: la memoria costante e texture. Gli spazi di memoria globale, costante e texture sono ottimizzati per usi diversi. La texture memory offre diverse modalità di indirizzo. I contenuti della memoria globale, costante e texture hanno la stessa durata dell'applicazione.

2.6.1 Registri

I registri sono lo spazio di memoria più veloce su una GPU. Una variabile dichiarata in un kernel senza altri qualificatori di tipo viene generalmente memorizzata in un registro. Le matrici dichiarate in un kernel possono anche

essere memorizzate in registri, ma solo se gli indici utilizzati per fare riferimento alla matrice sono costanti e possono essere determinati al momento della compilazione.

Le variabili nei registri sono private per ogni thread. Un kernel in genere usa i registri per contenere variabili thread private a cui si accede frequentemente. Le variabili nei registri condividono la loro durata con il kernel. Una volta che un kernel ha completato l'esecuzione, non è possibile accedere nuovamente a una variabile nel registro.

I registri sono risorse scarse che vengono suddivise tra gli warp attivi nella Shared Memory (SM).

2.6.2 Local Memory

Le variabili in un kernel idonee per i registri ma che non possono rientrare nello spazio di registro allocato per quel kernel si riverseranno nella memoria locale. Le variabili che è probabile che il compilatore inserisca nella memoria locale sono:

- Array locali referenziati con indici i cui valori non possono essere determinati in fase di compilazione.
- Grandi strutture locali o array che consumerebbero troppo spazio sui registri.
- Qualsiasi variabile che non rientra nel limite del registro del kernel.

Il nome "memoria locale" è fuorviante: i valori riversati nella memoria locale risiedono nella stessa posizione fisica della memoria globale, quindi gli accessi alla memoria locale sono caratterizzati da elevata latenza e bassa larghezza di banda e sono soggetti ai requisiti per un accesso efficiente alla memoria.

2.6.3 Shared Memory

Le variabili con il seguente attributo in un kernel sono immagazzinate nella memoria condivisa:
__shared__

Poiché la shared memory è su chip, ha una larghezza di banda molto più elevata e una latenza molto inferiore rispetto alla memoria locale o globale. È usato in modo simile alla cache L1 della CPU, ma è anche programmabile.

Ogni SM ha una quantità limitata di memoria condivisa che è partizionata tra i blocchi di thread. Pertanto, è necessario fare attenzione a non utilizzare

eccessivamente la memoria condivisa o si limiterà inavvertitamente il numero di warp attivi.

La memoria condivisa è dichiarata nell'ambito di una funzione del kernel ma condivide la sua durata con un blocco di thread. Quando viene eseguito il blocco, la sua allocazione di memoria condivisa verrà rilasciata e assegnata ad altri blocchi thread.

La shared memory serve come mezzo di base per la comunicazione tra thread. I thread all'interno di un blocco possono cooperare condividendo i dati memorizzati nella memoria condivisa. L'accesso alla shared memory deve essere sincronizzato utilizzando la seguente chiamata:

```
void __syncthreads();
```

Questa funzione crea una barriera che tutti i thread nello stesso blocco di thread devono raggiungere prima di poter procedere con qualsiasi altro thread. Creando una barriera per tutti i thread all'interno di un blocco, è possibile prevenire un potenziale rischio di errore nei dati. Si verificano quando esiste un ordinamento indefinito di accessi multipli nella stessa posizione di memoria da thread diversi, in cui almeno uno di questi accessi è una scrittura. _syncthreads può anche influire sulle prestazioni costringendo la SM a rimanere inattiva frequentemente.

2.6.4 Constant Memory

La memoria costante risiede nella memoria del device ed è memorizzata nella cache costante dedicata per la SM. Una variabile costante ha il seguente attributo:

__constant__

Le variabili costanti devono essere dichiarate globalmente, al di fuori di qualsiasi kernel. È possibile dichiarare una quantità limitata di memoria costante: 64 KB. La memoria costante viene dichiarata staticamente ed è visibile a tutti i kernel.

I kernel possono solo leggere dalla memoria costante che deve quindi essere inizializzata dall'host utilizzando:

```
cudaError_t cudaMemcpyToSymbol(const void* symbol, const
    void* src, size_t count);
```

Questa funzione copia i byte di conteggio dalla memoria a cui punta *src* alla memoria a cui punta il simbolo, che è una variabile che risiede sul dispositivo nella memoria globale o costante. Questa funzione è sincrona nella maggior parte dei casi.

La memoria costante funziona meglio quando tutti i thread in un warp leggono dallo stesso indirizzo di memoria. Ad esempio, è opportuno scrivere un coefficiente per una formula matematica nella memoria costante perché tutti i thread useranno lo stesso numero per condurre lo stesso calcolo su dati diversi. Se ogni thread in un warp legge da un indirizzo diverso e legge solo una volta, la memoria costante non è la scelta migliore perché una sola lettura dalla memoria costante si trasmette a tutti i thread in un warp.

2.6.5 Texture Memory

La memoria texture si trova nella memoria del device ed è memorizzata in una cache di sola lettura per SM. È un tipo di memoria globale a cui si accede tramite una cache di sola lettura dedicata che include il supporto per il filtro hardware, che può eseguire l'interpolazione in virgola mobile. La memoria texture è ottimizzata per spazi 2D, quindi i thread in un warp che la utilizzano per accedere ai dati 2D otterranno le migliori prestazioni. Per alcune applicazioni, questo è l'ideale e offre un vantaggio in termini di prestazioni grazie alla cache e all'hardware di filtro. Tuttavia, per altre applicazioni l'utilizzo della memoria texture può essere più lento della memoria globale.

2.6.6 Global Memory

La memoria globale è la memoria più grande, a più alta latenza e più comunemente usata su una GPU. Il nome *global* si riferisce alla sua portata e durata. È possibile accedere al suo stato sul device per tutta la durata dell'applicazione.

Una variabile nella memoria globale può essere dichiarata staticamente o dinamicamente. Si può dichiarare staticamente una variabile globale nel codice del dispositivo usando il seguente qualificatore: __device__

Nel Capitolo 2.4, si è definito come allocare dinamicamente la memoria globale dall'host utilizzando cudaMalloc e liberata utilizzando cudaFree. I puntatori alla memoria globale vengono quindi passati alle funzioni del kernel come parametri. Le allocazioni di memoria globale esistono per la durata dell'applicazione e sono accessibili a tutti i thread di tutti i kernel. È necessario prestare attenzione quando si accede alla memoria globale da più thread. Poiché non è possibile sincronizzare l'esecuzione di thread tra blocchi di thread, esiste il potenziale rischio che più thread in blocchi diversi modifichino contemporaneamente la stessa posizione nella memoria globale, il che porterà a un comportamento del programma indefinito.

La memoria globale risiede nella memoria del device ed è accessibile tramite transazioni di memoria a 32, 64 o 128 byte. Queste transazioni devono essere allineate, cioè il primo indirizzo deve essere un multiplo di 32 byte, 64 byte o 128 byte. L'ottimizzazione delle transazioni di memoria è fondamentale per ottenere prestazioni ottimali. Quando un warp esegue una operazione load/store, il numero di transazioni richieste per soddisfare quella richiesta dipende in genere dai seguenti due fattori:

- Distribuzione degli indirizzi di memoria tra i thread del warp.
- Allineamento degli indirizzi di memoria per transazione.

In generale, maggiore è il numero di transazioni necessarie per soddisfare una richiesta di memoria, maggiore è il potenziale di trasferimento di byte non utilizzati, con conseguente riduzione dell'efficienza del throughput.

3 Implementazione sequenziale

La rotazione di Jacobi è una rotazione di un sottospazio bidimensionale in uno spazio n-dimensionale, denotato da $\mathbf{J}$. Dopo l'applicazione di questa trasformazione, una coppia di elementi di una matrice simmetrica $\mathbf{B} \in \mathbb{R}^{n \times n}$ sono azzerati, come indicato in $\mathbf{B} \longmapsto \mathbf{J^T B J} = \mathbf{B'}$ dove $c = \cos(\theta)$, $s = \sin(\theta)$ e θ è l'angolo di rotazione nel piano (i, j). Solo le righe i-esima e j-esima di $\mathbf{B}$ sono interessate. In modo simile, solo le colonne j-esima e i-esima sono interessate. Gli elementi b'_{ij}, b'_{ji}, b'_{ii} e b'_{jj} in $\mathbf{B}$ vengono utilizzati per calcolare gli angoli nelle matrici di rotazione che eliminano gli elementi b'_{ij} e b'_{ji} come mostrato in Eq. (18).

$$\mathbf{J}(i,j,\theta) = \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & i & & j & \\ \end{array} \\ \begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \end{array} \tag{18}$$

L'algoritmo Jacobi esegue una sequenza di aggiornamenti della matrice $\mathbf{B}$ che viene ortogonalizzata, ogni nuova matrice $\mathbf{B}$ è più diagonale rispetto alla predecedente. Quando gli elementi fuori della diagonale sono abbastanza piccoli, possono essere considerati nulli. In particolare, ogni rotazione Jacobi comporta una pre-moltiplicazione e una post-moltiplicazione di $\mathbf{B}$ per matrici ortogonali. In generale, vengono eseguite $(n^2 - n)/2$ rotazioni (nel caso di una matrice simmetrica) cercando di rendere zero tutti gli elementi fuori diagonale. Queste $(n^2 - n)/2$ transformazioni costituiscono una scansione (sweep). Comunemente, le rotazioni di Jacobi vengono applicate utilizzando uno dei seguenti approcci: esecuzione di rotazioni cicliche per riga o per colonna. In questi approcci, la coppia (i, j) è selezionata riga per riga o colonna per colonna, rispettivamente. Ad esempio, se $n = 4$, la sequenza di rotazione è: $(i, j) = (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)$. [1]

Viene ora mostrata l'implementazione in CUDA dell'algoritmo sequenziale. Il kernel che esegue l'algoritmo One-Sided Jacobi sequential è compreso in un ciclo **while** mostrato di seguito.

```
while (!host_exit_flag) {
  ++iter;
  host_exit_flag = true;
  cudaMemcpy( exit_flag, &host_exit_flag, sizeof(bool),
      cudaMemcpyHostToDevice);
```

```
  for (int j = columns - 1; j >= 1; --j)
    for (int i = j - 1; i >= 0; --i) {
      rotate<<<1, rows>>> (B, i, j, rows, exit_flag);
  }
}
```

Codice 1: Loop algoritmo sequential

La variabile *iter* contiene le iterazioni necessarie per portare a termine con successo l'algoritmo di One-Sided Jacobi rotation.

host_exit_flag contiene il flag che permettere di uscire dal ciclo **while** quando le colonne della matrice sono sufficientemente ortogonali.

La coppia i e j rappresenta la coppia di colonne i-esima e j-esima che sta ruotando, B contiene la matrice su cui applicare la One-Sided Jacobi, è salvata in column-major order e single-precision floating point.

Il kernel *rotate* esegue la One-Sided Jacobi rotation descritta nel capitolo 1.2. Per chiamare questa funzione, viene usato un solo blocco e un numero di threads pari a *rows*, come si può vedere dalle parentesi angolari, in modo che tramite l'indice del thread si possa accedere al k-esimo elemento della coppia i-esima e j-esima di vettori colonna. Il kernel è definito come di seguito

```
1  __global__ void rotate (float * B, int i, int j, int  rows,
       bool * exit_flag){
2    int k = threadIdx.x;
3    __shared__ float alpha, beta, gamm, limit, tao, t, c, s;
4    float *pi, *pj;
5    if (k < rows) {
6      alpha = beta = gamm = 0;
7      __syncthreads();
8      pi = B + rows * i + k;
9      pj = B + rows * j + k;
10     atomicAdd(&alpha, *pi * *pi);
11     atomicAdd(&beta, *pj * *pj);
12     atomicAdd(&gamm, *pi * *pj);
13     __syncthreads();
14     if (* exit_flag) {
15       limit = fabsf(gamm) / sqrtf(alpha * beta);
16       if (limit > eps)
17         * exit_flag = false;
18     }
19     tao = (beta - alpha) / (2 * gamm);
20     t = sign (tao) / (fabsf(tao) + sqrtf(1 + tao * tao));
21     c = expf(-0.5f * log1pf(t * t));
```

```
22      s = c * t;
23      const float tmp = *pi;
24      *pi = c * tmp - s * *pj;
25      *pj = s * tmp + c * *pj;
26    }
27  }
```

Alla riga 3, le variabili *alpha*, *beta*, *gamm*, *limit*, *tao*, *t*, *c* e *s* con attributo *__shared* risiedono nella memoria shared in quanto sono comuni a tutti i thread del blocco.

Nella riga 6 si inizializzano le variabili *alpha*, *beta* e *gamm* a zero.

Alla linea 7, *__syncthreads*() attende che tutti i thread nel blocco abbiano raggiunto questo punto e tutti gli accessi alla memoria globale e shared effettuati da questi thread prima di *__syncthreads*() sono visibili a tutti i thread nel blocco.

La matrice B è formata da vettori colonna e i puntatori *pi* e *pj* puntano alla locazione di memoria in cui è contenuto l'elemento k-esimo, cioè alla riga k-esima, della coppia di vettori colonna i-esimo e j-esimo.

La funzione *atomicAdd*, presente nelle righe 10, 11 e 12, è definita come

```
float atomicAdd(float* address, float val);
```

legge la variabile floating point a 32 bit *old* memorizzata all'indirizzo di memoria *address* nella memoria globale o shared, calcola $(old + val)$, e salva il risultato allo stesso indirizzo di memoria. Queste tre operazioni sono eseguite in una sola operazione atomica. La funzione ritorna *old*. Nel kernel, viene usata per calcolare la norma del vettore colonna i-esimo e j-esimo e il loro prodotto scalare e scriverlo rispettivamente in *alpha*, *beta* e *gamm*.

Se la variabile *exit_flag* è true, si calcola *limit* e se è maggiore di *eps* dichiarata globalmente

```
static const float eps = 1e-4;
```

Codice 2: Dichiarazione della variabile *eps*

allora si porta *exit_flag* false.

Si procede con il calcolo di c e s che rappresentano rispettivamente la funzione $\sin(\theta)$ e $\cos(\theta)$ descritta nella formula (18).

Alle linee 24 e 25 si ruota la matrice moltiplicando opportunamente il contenuto dei puntatori *pi* e *pj* per c e s.

Successivamente, nel programma viene richiamato il kernel *computeSingVals* che effettua il calcolo dei valori singolari.

```
computeSingVals<<<columns, rows>>>(B, AUX1, rows, columns);
```

Vengono usati un numero di blocchi pari a *columns* e di threads pari a *rows*, in modo che tramite l'indice del blocco e dei thread si possa accedere rispettivamente ad un j-esimo vettore colonna e ad un suo k-esimo elemento. La variabile *AUX1* è un array che contiene i valori singolari.

Il kernel *computeSingVals* viene definito come

```
__global__ void computeSingVals (float * B, float * AUX1,
    int rows, int columns){
  int k = threadIdx.x;
  int j = blockIdx.x;
  __shared__ float t;
  if ((j < columns) && (k < rows)){
    float *pj = B + rows * j + k;
    t = 0;
    atomicAdd(&t, *pj * *pj);
    AUX1[j] = sqrtf(t);
  }
}
```

Codice 3: Kernel per il calcolo dei valori singolari

La variabile 32-bit floating point *t*, inizializzata a zero, contiene il quadrato della norma del vettore colonna j-esimo.

Per ottenere i valori singolari occorre fare la radice quadrata di *t* tramite la funzione *sqrtf*. L'array *AUX1* contiene i valori calcolati.

4 Implementazione parallela

Parallelizzare la One-Sided Jacobi implica il partizionamento di $n(n-1)/2$ coppie di colonne che devono essere ortogonali a ciascuna scansione (sweep) in gruppi di coppie di colonne indipendenti. Ogni sweep viene quindi elaborato un set alla volta, ortogonalizzando in parallelo le coppie di colonne all'interno del set corrente.

Le coppie di colonne per ciascun set vengono generate utilizzando un algoritmo di pianificazione round-robin. Concettualmente, ogni round rappresenta un set e gli abbinamenti all'interno di un round corrispondono agli abbinamenti di colonne all'interno di quel set. Ad esempio, di seguito sono riportati tutti i possibili set contenenti le rispettive coppie di colonne per $n = 6$:

$$\begin{aligned}
\text{Set } 1 &= \{(1,2),(3,4),(5,6)\} \\
\text{Set } 2 &= \{(1,4),(2,6),(3,5)\} \\
\text{Set } 3 &= \{(1,6),(2,3),(4,5)\} \\
\text{Set } 4 &= \{(1,5),(2,4),(3,6)\} \\
\text{Set } 5 &= \{(1,3),(2,5),(4,6)\}
\end{aligned}$$

In generale, ogni set contiene $\hat{n}/2$ coppie di colonne ortogonalizzate in parallelo, dove $\hat{n}$ è il prossimo numero intero pari maggiore o uguale a n. Se n è dispari, quindi ogni set avrà una colonna accoppiata con una colonna "fittizia"; le coppie che contengono la colonna fittizia non sono ortogonali. In base a questo schema, sono necessari $\hat{n}/2-1$ set per completare una sweep completa.

La coppia di colonne (p', q') ortogonalizzata dalla i-esima rotazione in un set viene calcolata direttamente dalla coppia di colonne (p, q) corrispondente ortogonalizzata dalla i-esima rotazione nel set precedente. In pratica, questo schema è più adatto per l'esecuzione su una GPU in cui la larghezza di banda di calcolo supera notevolmente la larghezza di banda di memoria. [5]

Nelle tradizionali implementazioni sequenziali dell'algoritmo Jacobi, il parallelismo non viene sfruttato, principalmente a causa della dipendenza dei dati di una rotazione con la sua precedente. L'algoritmo Jacobi parallelo sfrutta il massimo parallelismo per la decomposizione di una matrice simmetrica. Questo algoritmo richiede un numero di step pari a $n-1$, in cui in ogni step si compiono $n/2$ rotazioni. Tutte le rotazioni all'interno di uno step possono essere eseguite in parallelo. [1]

Si è usato CUDA per implementare una SVD parallela basata sul metodo One-Sided Jacobi descritto in 1.2. La matrice di input A $(m \times n)$ è una matrice reale in cui $m \geq n$. Se $m < n$, la SVD di A può essere calcolata

dalla SVD di $A^T = V\Sigma U^T$. Le matrici di input sono salvate in column-major order e single-precision floating point. [5]

Viene ora mostrata l'implementazione in CUDA dell'algoritmo parallel. Il codice che richiama il kernel che esegue l'algoritmo One-Sided Jacobi parallel è il seguente.

```
while(!host_exit_flag) {
  ++iter;
  host_exit_flag = true;
  cudaMemcpy(dev_exit_flag, &host_exit_flag, sizeof(bool),
      cudaMemcpyHostToDevice);
  for(int set = 0; set < cols; set++) {
    scheduling<<<1, 1>>> (dev_v1, dev_v2, cols);
    round <<<cols/2, rows>>> (dev_B, dev_v1, dev_v2, cols,
        rows, dev_exit_flag);
  }
  cudaMemcpy( &host_exit_flag, dev_exit_flag, sizeof(bool),
      cudaMemcpyDeviceToHost);
}
```

Codice 4: Loop algoritmo parallelo

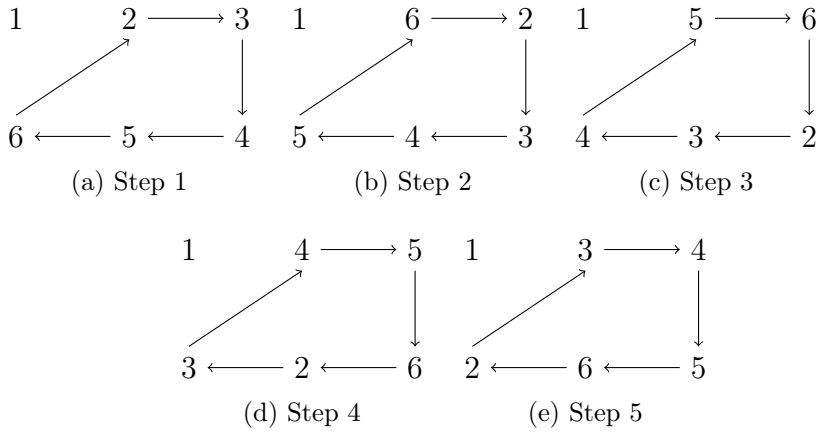
Le variabili *host_exit_flag*, *iter* e *B* svolgono le stesse funzioni descritte nel codice 4.

La variabile *set* rappresenta il round attuale dello scheduling round robin. Come spiegato in precedenza, il numero di set necessari per formare tutte le possibili combinazioni sono $n-1$, dove n è il numero di colonne della matrice originale.

I vettori *v1* e *v2* servono a rappresentare l'indice delle coppie di colonne della matrice da ruotare. Per ogni set gli elementi dei vettori con lo stesso indice rappresentano una coppia di colonne da ortogonalizzare mutualmente. L'aggiornamento dei vettori *v1* e *v2* viene effettuato dal kernel *scheduling*.

Il kernel *scheduling* ha lo scopo di modificare i vettori *v1* e *v2* ad ogni iterazione del ciclo in cui viene incrementato il valore di *set*. Infatti, per effettuare l'algoritmo di round robin è sufficiente tenere fisso il primo elemento del primo vettore e shiftare gli altri elementi come se fossero un unico vettore. Quindi, l'elemento 0 che rappresenta la prima colonna sarà sempre alla prima posizione del primo vettore. Gli altri elementi dello stesso vettore scorreranno verso destra, mentre il secondo vettore esegue uno shift verso sinistra. L'elemento uscente dal secondo vettore viene spostato nel posto vacante lasciato dall'elemento 1 di *v1* mentre l'ultimo elemento del primo vettore diventa l'ultimo del secondo. Supponendo di avere una matrice con 6 colonne, gli array *v1* e *v2* contengono gli interi da 1 a 3 e da 4 a 6 ri-

spettivamente. In questo caso, l'algoritmo di round robin è presentato nella figura 8.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5

Figura 8: Algoritmo Round Robin

Ogni iterazione di questo algoritmo viene effettuata sul device. In questo modo, non ci sarà la necessità di copiare i vettori da host a device e viceversa. Il costo computazionale per lanciare un kernel compensa di gran lunga quello che sarebbe necessario per la *cudaMemcpy*.

```
__global__ void scheduling (int *v1, int *v2, int cols){
  int tmp = v2[0];
  for (int i = 0; i < (cols/2) - 1; i++)
    v2[i] = v2[i+1];
  v2[cols/2 - 1] = v1[cols/2 - 1];
  for (int i = (cols/2) -1; i > 1; i--)
    v1[i] = v1[i-1];
  v1[1] = tmp;
}
```

Il kernel *round* ortogonalizza contemporaneamente ogni coppia di colonne in un set, usando una grid di $n/2$ blocchi, dove n è il numero di colonne della matrice. Per ogni set, l'm-esimo blocco è responsabile dell'ortogonalizzazione delle colonne b_{p_m} e b_{q_m}, dove (p_m, q_m) è l'm-esima coppia di colonne del set corrente.

Ogni blocco è composto da un numero di thread pari alle righe della matrice. Con un thread allocato per ogni elemento della coppia di vettori colonna, è possibile aggiornare ogni componente del vettore in parallelo, incrementando il parallelismo.[5]

Nei capitoli 4.1, 4.2 e 4.3 sono presentate tre varianti del kernel *round* a seconda della memoria in cui viene immagazzinata la matrice *B*.

Il calcolo dei valori singolari è eseguito dal kernel *computeSingVals*, codice 3, precedentemente descritto.

4.1 Global Memory

In questo capitolo viene mostrato il kernel *round* che usa la global memory.

Nelle linee 2 e 3 del codice 5, una volta ricavati i valori identificativi del blocco e del thread attuale, salvati nelle variabili *blockId* e *threadId* rispettivamente, viene estratta da *v1* e *v2* la coppia di indici i e j che corrispondono all'i-esimo e j-esimo vettore colonna da ortogonalizzare. Gli indici vengono salvati nelle variabili *i* e *j* come si può vedere alle righe 6 e 7.

I puntatori *pi* e *pj* punteranno agli elementi su cui ogni thread deve lavorare. Perciò, i e j rappresentano le colonne ottenute da *v1* e *v2*, mentre l'id del thread indica la riga su cui lavorare (ovvero l'elemento della colonna corrispondente). Tenendo conto del fatto che la matrice B è salvata in memoria in column-major order, è possibile accedere agli elementi specifici come nelle linee 8 e 9 del codice 5.

Le variabili *alpha*, *beta* e *gamm* rappresentano le omonime descritte nella teoria dell'algoritmo di Jacobi nel capitolo 1.2. Queste variabili sono salvate nella memoria shared; in questo modo il loro valore è condiviso tra i thread dello stesso blocco. La funzione *atomicAdd* permette ad ogni thread di incrementare il valore delle variabili in maniera sequenziale, evitando collisioni dovute all'accesso in parallelo. Ogni thread calcola quindi il proprio valore di *alpha*, *beta* e *gamm* per la riga che rappresentano e la aggiungono al valore complessivo che tiene conto della singola coppia su cui il blocco sta lavorando.

Viene calcolato il valore *limit* con lo scopo di essere confrontato con il valore ammissibile che soddisfa la convergenza. Quando la soglia viene superata, viene modificata la variabile *exit_flag*, che permette di uscire dalle iterazioni una volta ritornata all'host.

Seguendo le istruzioni spiegate in precedenza, si ricavano i valori delle variabili *s* e *c*, che rappresentano seno e coseno della matrice di rotazione. Queste variabili dipendono da *alpha*, *beta* e *gamm*. Sono quindi comuni per ogni coppia di colonne.

Infine, la matrice di rotazione viene applicata ruotando le colonne con i valori adeguati di *c* e *s*.

```
1  __global__ void round (float *B, int *v1, int *v2, int cols
       , int rows, bool * exit_flag) {
```

```
2   int blockId = blockIdx.x; //max(blockId) = (cols/2) - 1
3   int threadId = threadIdx.x; //max(blockId) = rows - 1
4   __shared__ float alpha, beta, gamm;
5   if ((blockId < cols/2) && (threadId < rows)){
6     int i = *(v1 + blockId);
7     int j = *(v2 + blockId);
8     float * pi = B + rows * i + threadId;
9     float * pj = B + rows * j + threadId;
10    alpha = beta = gamm = 0;
11    __syncthreads();
12    atomicAdd(&alpha, *pi * *pi);
13    atomicAdd(&beta, *pj * *pj);
14    atomicAdd(&gamm, *pi * *pj);
15    __syncthreads();
16    if (*exit_flag) {
17      const float limit = fabsf(gamm) / sqrtf(alpha * beta)
          ;
18      if (limit > eps){
19        *exit_flag = false;
20      }
21    }
22    const float tao = (beta - alpha) / (2 * gamm);
23    const float t = sign (tao) / (fabsf(tao) + sqrtf(1 +
        tao * tao));
24    const float c = expf(-0.5f * log1pf(t * t));
25    const float s = c * t;
26    const float tmp = *pi;
27    *pi = c * tmp - s * *pj;
28    *pj = s * tmp + c * *pj;
29  }
30 }
```

Codice 5: Codice parallel global

4.2 Semi Shared Memory

In questa implementazione le variabili *alpha*, *beta*, *gamm*, *limit*, *tao*, *t*, *c*, *s*, *i* e *j* sono contenute nella shared memory e sono identificate dall'attributo *__shared*. I puntatori *pi* e *pj* che contengono l'indirizzo di memoria degli elementi della coppia di vettori colonna i-esimo e j-esimo non sono contenuti invece contenuti nella shared memory.

```
__global__ void round (float *B, int *v1, int *v2, int cols
    , int rows, bool * exit_flag) {
  int blockId = blockIdx.x;
  int threadId = threadIdx.x;
  float * pi, *pj;
  __shared__ float alpha, beta, gamm, limit, tao, t, c, s;
  __shared__ int i, j;
  if ((blockId < cols/2) && (threadId < rows)){
    i = *(v1 + blockId);
    j = *(v2 + blockId);
    pi = B + rows * i + threadId;
    pj = B + rows * j + threadId;
    alpha = beta = gamm = 0;
    __syncthreads();
    atomicAdd(&alpha, *pi * *pi);
    atomicAdd(&beta, *pj * *pj);
    atomicAdd(&gamm, *pi * *pj);
    __syncthreads();
    if ( *exit_flag) {
      limit = fabsf(gamm) / sqrtf(alpha * beta);
      if (limit > eps){
        *exit_flag = false;
      }
    }
    tao = (beta - alpha) / (2 * gamm);
    t = sign (tao) / (fabsf(tao) + sqrtf(1 + tao * tao));
    c = expf(-0.5f * log1pf(t * t));
    s = c * t;
    const float tmp = *pi;
    *pi = c * tmp - s * *pj;
    *pj = s * tmp + c * *pj;
  }
}
```

4.3 Shared Memory

Nell'implementazione in cui viene usata la shared memory per lavorare con la matrice B si può notare che tra le parentesi angolari che indicano i blocchi e i thread allocati per il kernel è presente una terza variabile come si può vedere nel codice seguente

```
round <<<cols/2, rows, 2*rows*sizeof(float)>>> (dev_B,
   dev_v1, dev_v2, cols, rows, dev_exit_flag);
```

Questo permette di allocare dinamicamente la memoria shared, che può essere utilizzata quando la quantità di memoria condivisa non è nota al momento della compilazione. In questo caso, la dimensione di allocazione della memoria shared per ogni blocco deve essere specificata (in byte) utilizzando un terzo parametro di configurazione come mostrato precedentemente.

La memoria allocata corrisponde a *2*rows*sizeof(float)* in quando ad ogni blocco viene assegnata una coppia di colonne. Ogni thread lavora su due elementi della coppia opportuna appartenenti alla riga corrispondente all'indice del thread. Mentre in precedenza le due colonne venivano modificate tramite puntatori, in questo caso vengono copiate nella memoria shared che richiede minor tempo di accesso per la lettura e la scrittura. La rotazione delle colonne viene quindi fatta internamente ad ogni blocco per poi trasferire di nuovo la variabile *arr* in global, aggiornando la matrice originale.

```
__global__ void round (float *B, int *v1, int *v2, int cols
    , int rows, bool * exit_flag) {
  int blockId = blockIdx.x;
  int threadId = threadIdx.x;
  __shared__ float alpha, beta, gamm, limit, tao, t, c, s,
     tmp;
  extern __shared__ float arr[];
  __shared__ int i, j;
  if ((blockId < cols/2) && (threadId < rows)){
    i = *(v1 + blockId);
    j = *(v2 + blockId);
    arr[threadId] = *(B + rows * i + threadId);
    arr[threadId+rows] = *(B + rows * j + threadId);
    alpha = beta = gamm = 0;
    __syncthreads();
    atomicAdd(&alpha, arr[threadId] * arr[threadId]);
    atomicAdd(&beta, arr[threadId+rows] * arr[threadId+rows
       ]);
    atomicAdd(&gamm, arr[threadId] * arr[threadId+rows]);
    __syncthreads();
    if (*exit_flag) {
      limit = fabsf(gamm) / sqrtf(alpha * beta);
      if (limit > eps){
        *exit_flag = false;
      }
    }
    tao = (beta - alpha) / (2 * gamm);
    t = sign (tao) / (fabsf(tao) + sqrtf(1 + tao * tao));
    c = expf(-0.5f * log1pf(t * t));
```

```
    s = c * t;
    tmp = arr[threadId];
    arr[threadId] = c * tmp - s * arr[threadId+rows];
    arr[threadId+rows] = s * tmp + c * arr[threadId+rows];
    *(B + rows * i + threadId) = arr[threadId];
    *(B + rows * j + threadId) = arr[threadId+rows];
  }
}
```

Il kernel con memoria shared dinamica, *round*, dichiara l'array nella shared memory utilizzando un array extern senza dimensione(si noti le parentesi vuote e l'uso dell'identificatore *extern*). La dimensione è implicitamente determinata dal terzo parametro di configurazione all'avvio del kernel.

```
extern __shared__ float arr[];
```

5 Performance

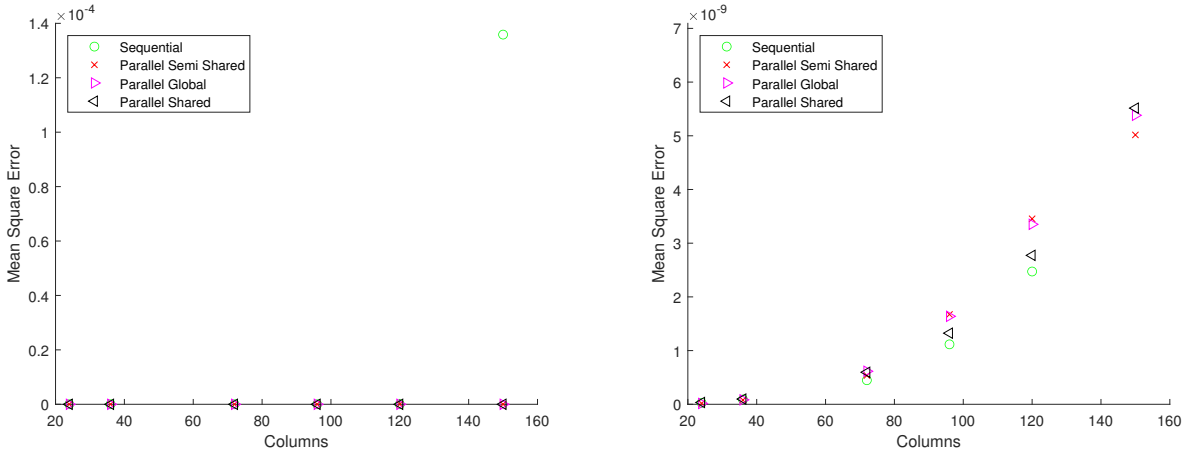
Per testare gli algoritmi presentati nei capitoli 3, 4.1, 4.2 e 4.3 sono state usate 6 matrici generate con Matlab tramite la funzione *rand* che restituisce un numero random distribuito uniformemente nell'intervallo $(0, 1)$. Queste matrici hanno un aspect ratio, cioè il rapporto tra la larghezza (colonne) e l'altezza (righe) di una matrice, di $4 : 3$. Le matrici generate hanno 32, 48, 96, 128, 160 e 200 righe.

I grafici presentati in questo capitolo sono stati generati usando un pc con CPU Intel i7-3630 QM @2.40 GHz, 8GB di RAM e una GPU NVIDIA Geforce GTX 610M poiché non era possibile usare la scheda JETSON per produrre i risultati.

I grafici 9a e 9b mostrano come cambia l'errore quadratico medio dei valori singolari calcolati al variare dell'algoritmo One-Sided Jacobi usato e del numero di colonne della matrice usata. Il MSE è calcolato come

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

dove n è il numero di colonne della matrice, Y è il vettore che contiene i valori singolari calcolati dalla funzione *svd* di Matlab e $\hat{Y}$ è l'array AUX1, mostrato nel codice 3, che contiene i valori singolari calcolati dalla GPU. Nel grafico 9a si può vedere che l'errore quadratico massimo è 10^{-4}, coerente con la variabile *eps* posta pari a questo valore nel codice 2. La figura 9b mostra che per tutte le matrici con meno di 150 colonne l'errore quadratico medio si mantiene nell'intorno di 10^{-9}.



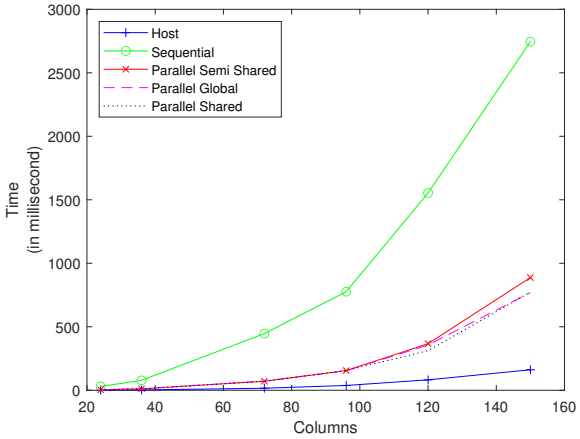
(a) Errore quadratico medio

(b) Dettaglio dell'errore quadratico medio

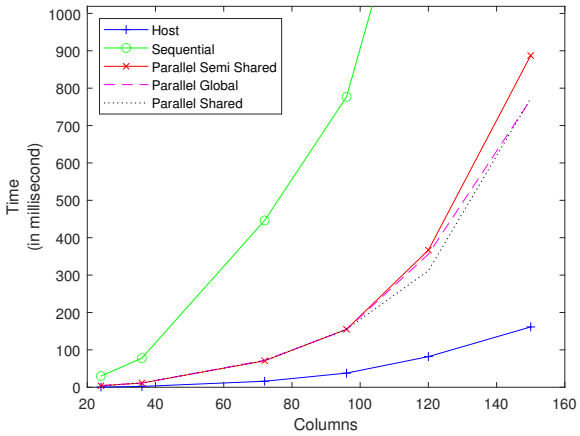
Figura 9: Confronto dell'errore quadratico medio

I grafici che seguono sono stati ottenuti usando la funzione *cudaEventRecord* che registra un evento. Questa funzione permette di gestire due *event*, chiamati *start* e *stop*, e di misurare il tempo intercorso tra i due. *Start* viene registrato prima del ciclo *while* presentato nel codice 1, mentre *stop* quando si esce da questo loop. Dato che nel ciclo si esegue la One Sided Jacobi rotation si misura il tempo trascorso tra l'entrata e l'uscita del ciclo.

Le figure 10a e 10b mostrano come cambia il tempo di esecuzione dell'algoritmo One-Sided Jacobi al variare della grandezza della matrice, in particolare del numero di colonne in quanto tutte le matrici di prova hanno lo stesso aspect ratio. L'algoritmo più veloce rimane quello che viene eseguito sull'host, mentre tra gli algoritmi testati sul device il migliore in termini di tempo è quello parallelo che usa la memoria shared. Questo è inoltre confermato dal fatto che questa è la memoria più veloce e può arrivare ad essere fino a 100 volte più veloce della memoria global. [3] [6]



(a) Relazione tra il numero di colonne e il tempo di esecuzione



(b) Dettaglio Columns vs. Time

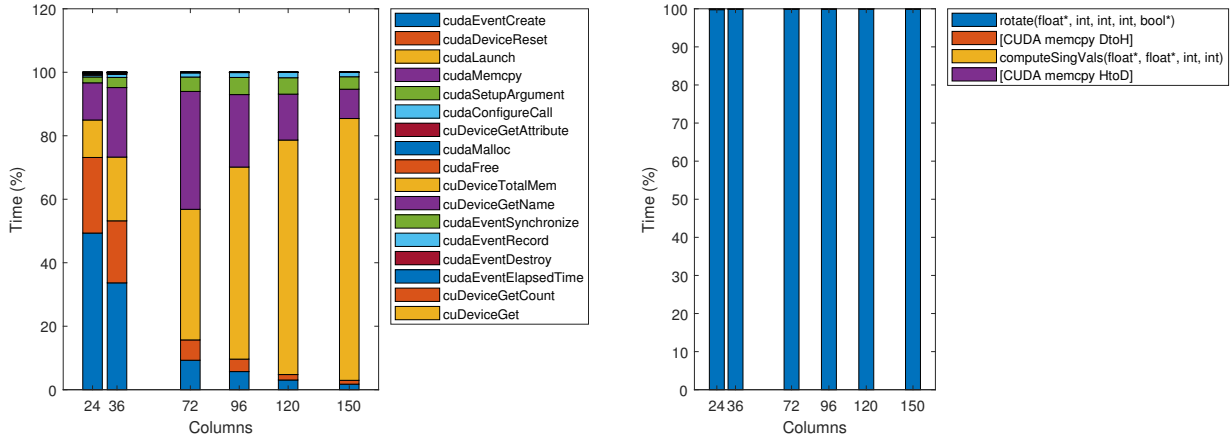
Figura 10: Confronto Columns vs. Time

I grafici che seguono sono stati ottenuti usando lo strumento *nvprof* che consente di raccogliere e visualizzare i dati di profiling dalla riga di comando. *nvprof* consente la raccolta di una sequenza temporale di attività correlate a CUDA su CPU e GPU, tra cui l'esecuzione del kernel, trasferimenti di memoria e chiamate ad API CUDA ed eventi. Le opzioni di profiling vengono fornite a nvprof tramite le opzioni della riga di comando. Nei grafici di sinistra si possono notare le chiamate alle API CUDA, mentre in quelli di destra le prestazioni dei kernel.

Sull'asse delle ordinate, la misura presa in considerazione è la percentuale di tempo sul totale del programma eseguito sul device. Sull'asse delle ascisse viene riportata la grandezza della matrice, in relazione al numero delle colonne. I grafici presentati sono divisi per tipo di algoritmo preso in considerazione.

Nei grafici 11a e 11b vengono mostrate le prestazioni dell'algoritmo sequenziale. Nella figura 11a si nota che per la matrice con 24 colonne le funzioni che occupano più tempo in percentuale sono *cudaEventCreate* (49%), *cudaDeviceReset* (23%), *cudaLaunch* (11%) e *cudaMemcpy* (11%), mentre per la matrice con 150 colonne sono *cudaLaunch* (82%) e *cudaMemcpy* (9%). La funzione *cudaDeviceReset* distrugge tutte le allocazioni fatte e ripristina lo stato sul device, *cudaLaunch* lancia una funzione del dispositivo. L'aumento della percentuale di tempo relativa a *cudaLaunch* con l'aumentare della dimensione della matrice è conseguenza del fatto che il kernel *rotate* viene richiamato $n(n-1)/2$ volte, dove n è il numero di colonne, come spiegato nel capitolo 3 e come si può vedere nel codice 1. Il numero di volte con cui viene chiamata la funzione sul device ha andamento quadratico con l'aumentare delle colonne.

Il kernel che impiega più tempo, come si può vedere nella figura 11b, è *rotate* per una percentuale di tempo maggiore del 99%.



(a) Profiling delle API CUDA dell'algoritmo sequenziale

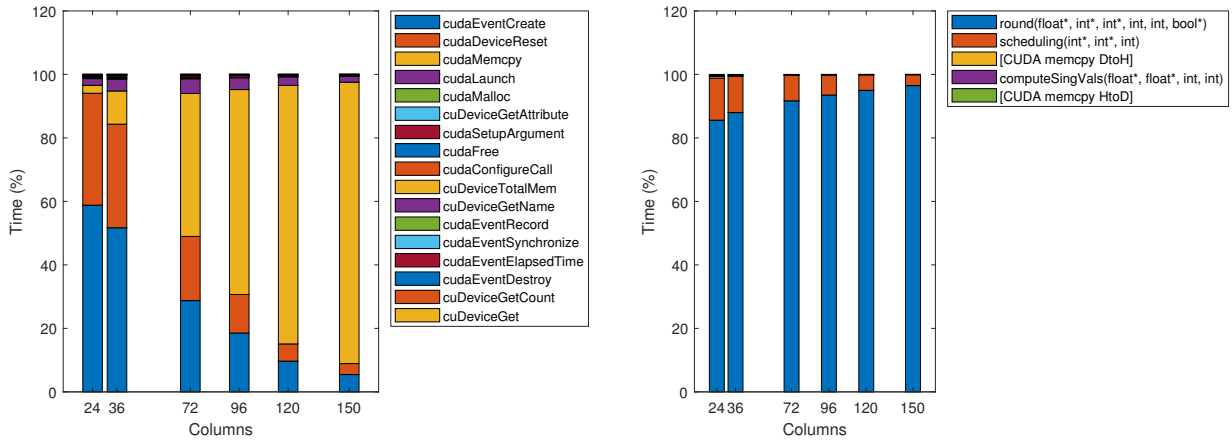
(b) Profiling dei kernel dell'algoritmo sequenziale

Figura 11: Prestazioni algoritmo sequenziale

Nei grafici 12a e 12b vengono mostrate le prestazioni dell'algoritmo parallel con la matrice B salvata nella memoria globale. Come mostrato in

figura 12a, le funzioni che richiedono più tempo per la matrice più piccola sono *cudaEventCreate* (58%), *cudaDeviceReset* (35%), *cudaMemcpy* (2%) e *cudaLaunch* (2%). Per la matrice più grande sono invece *cudaMemcpy* (88%), *cudaEventCreate* (5%), *cudaDeviceReset* (3%) e *cudaLaunch* (2%). Con l'aumentare della dimensione della matrice la funzione *cudaMemcpy* (88%) predomina dal punto di vista temporale perché occorre fare più copie dalla memoria dell'host a quella del device, e viceversa, della variabile *host_exit_flag* come mostrato nel codice 4.

Il kernel che impiega più tempo in percentuale è *round* con l'85% per la matrice più piccola e il 96% per quella più grande, mentre *scheduling* impiega rispettivamente il 13% e il 3%.



(a) Profiling delle API CUDA dell'algoritmo parallelo global

(b) Profiling dei kernel dell'algoritmo parallelo global

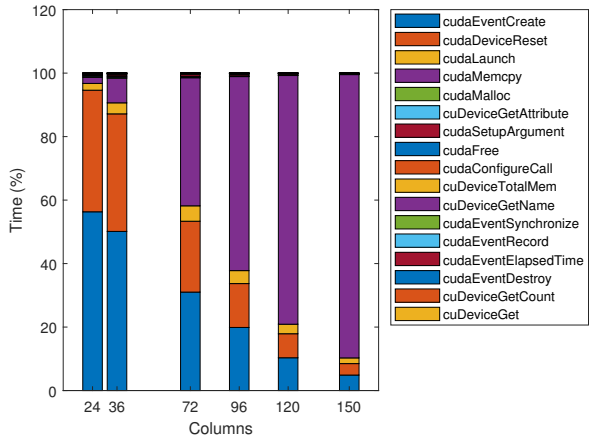
Figura 12: Prestazioni dell'algoritmo parallelo global

Nei grafici 13a e 13b vengono mostrate le prestazioni dell'algoritmo parallel con la matrice B salvata nella memoria globale e con l'uso delle variabili all'interno del kernel salvate nella shared.

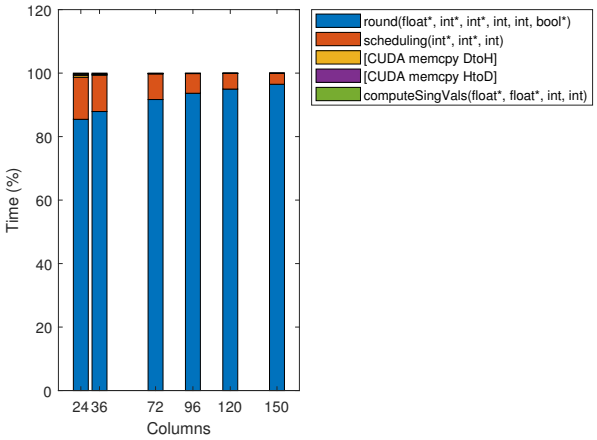
Le funzioni che impiegano più tempo per la matrice più piccola sono *cudaEventCreate* (56%), *cudaDeviceReset* (38%), *cudaMemcpy* (2%) e *cudaLaunch* (2%). Per la matrice più grande sono invece *cudaMemcpy* (89%), *cudaEventCreate* (4%), *cudaDeviceReset* (3%) e *cudaLaunch* (2%).

Il kernel che impiega più tempo in percentuale è *round* con l'85% per la matrice più piccola e il 96% per quella più grande, mentre *scheduling* impiega rispettivamente il 13% e il 3%.

I risultati sono simili a quelli precedentemente descritti: la variazione più importante è nel tempo di esecuzione, mostrata nel grafico 10b, e non nella percentuale di tempo impiegata.



(a) Profiling delle API CUDA dell'algoritmo parallelo semi-shared

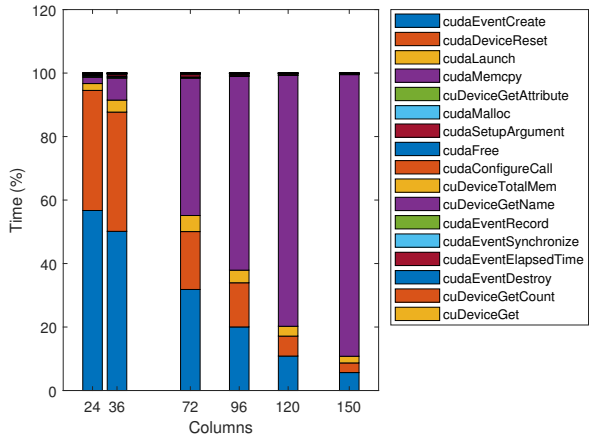


(b) Profiling dei kernel dell'algoritmo parallelo semi-shared

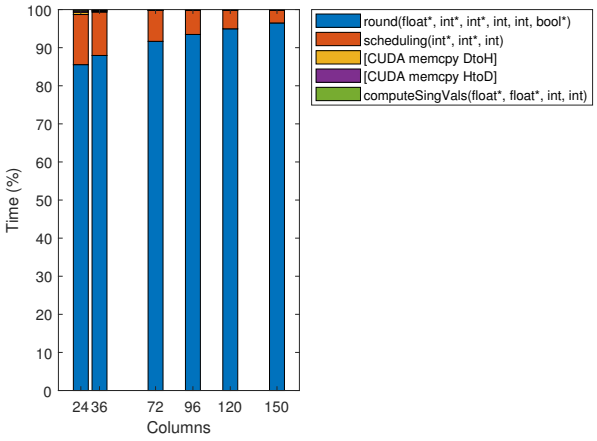
Figura 13: Prestazioni dell'algoritmo parallelo semi-shared

Nei grafici 14a e 14b vengono mostrate le prestazioni dell'algoritmo parallel con la matrice B e le variabili usate nel kernel salvate nella shared memory.

Anche in questo caso i risultati sono simili a quelli precedentemente illustrati. L'uso della memoria shared per allocare il vettore diminuisce il tempo per completare il kernel *round*, ma non la percentuale di tempo richiesta dalle singole funzioni.



(a) Profiling delle API CUDA dell'algoritmo parallelo shared



(b) Profiling dei kernel dell'algoritmo parallelo shared

Figura 14: Prestazioni dell'algoritmo parallelo semi-shared

6 Conclusioni

Il modello di programmazione CUDA ha uno stile progettuale intrinseco che complica l'implementazione di *round*. È impossibile sincronizzare l'esecuzione e le transazioni della memoria globale dei thread nei diversi blocchi all'interno di un kernel in esecuzione. Questo significa che alla fine di un set, quando il blocco b_1 scrive le colonne ortogonalizzate nella global memory, non c'è modo per garantire che il blocco b_2 vedrà e leggerà le colonne aggiornate all'inizio del set successivo.

L'unico modo per i thread contenuti in diversi blocchi di sincronizzare e condividere i dati è di richiamare il kernel dopo che ogni blocco ha scritto i suoi risultati nella memoria globale. Ciò limita lo scopo di *round* e forza l'host a richiamare il kernel una volta per set. Questo comporta le seguenti conseguenze indesiderabili:

- Per ogni esecuzione di *round*, ogni blocco di thread deve leggere le colonne della matrice dalla memoria globale nella shared memory o solo dalla memoria globale in base all'algoritmo implementato.

  Al contrario, se fosse possibile la sincronizzazione tra blocchi di thread, sarebbe possibile elaborare uno schema di partizionamento di coppie di colonne più efficiente. Un tale schema minimizzerebbe il numero di letture di memoria globale richieste da un blocco all'inizio di un nuovo set.

- Il test di convergenza deve essere eseguito sull'host. Occorre quindi copiare il valore *exit_flag* dal device all'host tramite cudaMemcpy alla fine di ogni set per determinare se è necessario continuare a chiamare il kernel *round*.

  Poiché l'host deve leggere la memoria del device dopo ogni set, ciò ha l'effetto di sincronizzare implicitamente l'host con il device dopo ogni chiamata di *round*. Ciò riduce di molto le possibilità di esecuzione simultanea del kernel, che a sua volta limita il throughput totale.

In linea di principio, una SVD basata sul metodo One-Sided Jacobi può essere adattata per sfruttare dispositivi con elevate capacità di parallelismo. Tuttavia, le caratteristiche fondamentali del modello di programmazione e dell'architettura hardware delle GPU NVIDIA rendono in pratica l'approccio di cui sopra difficile da applicare. In particolare, la mancanza di sincronizzazione sia tra i thread sia tra l'host e il device dopo ogni set comportano un sovraccarico aggiuntivo che alla fine limita la potenzialità di sfruttare appieno le capacità di calcolo della GPU.[5]

Un altro approccio per la SVD è presentato in [2]. È molto importante quando non possiamo più memorizzare l'intera matrice nella memoria shared a causa della dimensione elevata della matrice, dobbiamo quindi operare con la memoria globale che è più lenta. Invece di leggere e aggiornare ripetutamente le colonne una alla volta, gli algoritmi a blocchi possono operare con blocchi di colonne.

Sono illustrati due algoritmi per implementare l'algoritmo One-Sided Jacobi sulla memoria globale che differiscono solo nel modo in cui le colonne dei blocchi vengono ortogonalizzate. In queste implementazioni vengono utilizzate le routine della libreria cuSOLVER 8. La nostra GPU Jetson non può usare questa libreria perché lavora con la versione 6.5 di CUDA e ha una capacità di calcolo pari 3.2.

Le prestazioni mostrate nel capitolo 5 per l'implementazione dell'approccio sopra indicato sono state eseguite su un pc con CPU Intel i7-3630 QM @2.40 GHz, 8GB of RAM e una GPU NVIDIA Geforce GTX 610M. Per ulteriori lavori, in futuro, si consiglia di eseguire gli algoritmi descritti con una GPU ad alte prestazioni che può utilizzare le librerie cuSOLVER e con una compute capability ≥ 3.5: ciò consente al programmatore di aumentare il parallelismo usando kernel che hanno la possibilità di invocare altri kernel e di sfruttare librerie altamente ottimizzate per il calcolo parallelo.

Riferimenti bibliografici

[1] R. I. Acosta-Quiñonez, D. Torres-Roman, R. Rodrìguez-Avila e D. Robles-Valdez. *A parallel implementation of One-Sided Jacobi SVD for non-symmetric squared matrices on a high-performance GPU*. Jalisco: Instituto Politécnico Nacional, 2016.

[2] Wajih Halim Boukaram, George Turkiyyah, Hatem Ltaief e David E. Keyes. *Batched QR and SVD algorithms on GPUs with application in hierarchical matrix compression*. Thuwal: King Abdullah University of Science e Technology (KAUST), 2017.

[3] John Cheng, Max Grossman e Ty McKercher. *Professional Cuda C Programming*. A cura di Inc. John Wiley & Sons. Indianapolis, 2014.

[4] Sheetal Lahabar e P.J. Narayanan. *Singular Value Decomposition on GPU using CUDA*. Hyderabad: Center for Visual Information Technology International Institute of Information Technology, 2009.

[5] Michael V. Romer. *Computing the singular value decomposition in parallel on graphics processing units using a one-sided Jacobi method*. Austin: University of Texas, 2013.

[6] Jason Sanders e Kandrot Edward. *CUDA by example : an introduction to general-purpose GPU programming*. A cura di Addison-Wesley. Boston, 2010.

[7] Claudio Turchetti, Laura Falaschetti e Lorenzo Manoni. *Singular Value Decomposition Algorithms for Embedded Systems: A Comprehensive Treatment*. Università Politecnica delle Marche,