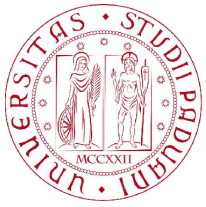
UNIVERSITÀ DI PADOVA  FACOLTÀ DI INGEGNERIA

Laurea Magistrale in Ingegneria Informatica

Sviluppo e implementazione di algoritmi paralleli in ambiente CUDA per la ricostruzione tridimensionale densa dell'ambiente

Relatore: Prof. Emanuele Menegatti

Correlatore: Ing. Alberto Pretto

Studente: Andrea Marchese

Sommario

Una delle capacità fondamentali che un robot autonomo deve possedere consiste nel saper creare autonomamente una mappa dell'ambiente che lo circonda, localizzando nello stesso momento se stesso all'interno della mappa costruita. Un robot con questa caratteristica è in grado di operare in ambienti diversi, non precostruiti, e svolgere compiti avanzati partendo dalle informazioni di base che possiede. Il problema della ricostruzione tridimensionale dell'ambiente e della localizzazione all'interno dello stesso prende il nome di Visual SLAM (Simultaneous Localization and Mapping) e negli ultimi anni ha ricevuto una forte attenzione dalla comunità scientifica.

L'obbiettivo di questa tesi è quello di presentare algoritmi paralleli in grado di migliorare l'efficienza di un sistema di visual SLAM basato su un approccio diretto [19], sviluppato all'interno del laboratorio IAS-LAB dell'Università di Padova dall'ing. Alberto Pretto. Tale sistema si basa su una strategia di minimizzazione della discrepanza tra pixel, sfruttando una triangolarizzazione dell'immagine omnidirezionale per ricostruire l'ambiente circostante e per localozzare il robot. Tale procedura di ottimizzazione ha il suo cuore nel calcolo di una matrice pseudoinversa dello Jacobiano del sistema, molto sparso e di dimensioni considerevoli. Il calcolo della matrice pseudoinversa viene ripetuto molte volte richiedendo un tempo considerevole che non permette al software di operare in tempo reale.

In questa tesi ci si prefigge quindi di sfruttare le nuove tecnologie nella parallelizzazione massiva messe a disposizione dall'NVIDIA [1], ovvero i nuovi chip grafici che includono l'architettura CUDA (Compute Unified Device Architecture), per velocizzare tale procedimento sfruttando in maniera massiva il parallelismo che tali dispositivi mettono a disposizione.

Nel capitolo 1 viene introdotto il concetto di programmazione della GPU, detto GPGPU (General-Purpose computing on Graphics Processing Units), e viene spiegato nel dettaglio il funzionamento del sistema di Visual SLAM su cui si è andati ad operate. Successivamente nel capitolo 2 viene fatta una panoramica completa sull'architettura CUDA e su come utilizzarla, focalizzando l'attenzione sulle tecniche che permettono di ottimizzare il codice per renderlo più efficiente

sfruttando a pieno le risorse disponibili.
Il capitolo 3 presenta l'approccio preliminare utilizzato nello sviluppo di questo lavoro di tesi mentre la sezione 4 specifica nel dettaglio il problema affrontato e le tecniche risolutive utilizzate.
I risultati del lavoro svolto sono raccolti nel capitolo 5 in cui vengono analizzate le performance ottenute.

Indice

Contenuti **II**

Indice delle figure **VI**

Indice delle tabelle **X**

1 Introduzione **1**

1.1 Il sistema di ricostruzione tridimensionale densa dell'ambiente . . 1

1.1.1 Il modello del movimento 3

1.1.2 Struttura dell'immagine e profondità 4

1.1.3 Procedura di ottimizzazione 5

1.2 GPGPU: General-Purpose computing on Graphics Processing Units 6

2 CUDA: Compute Unified Device Architecture **9**

2.1 Architettura 9

2.2 Modello di programmazione 11

2.2.1 Kernel e gerarchia dei thread 11

2.2.2 Gestione interna dei thread 12

2.3 Memoria 14

2.3.1 Global memory 15

2.3.2 Constant memory 16

2.3.3 Texture memory 16

2.3.4 Shared memory 18

2.3.5 Local memory 18

2.3.6 Registri 19

2.3.7 Calcolo delle risorse 20

2.4 Processo di compilazione 21
2.5 Ottimizzazioni 22
2.5.1 Flussi del codice 23
2.5.2 Coalescenza 23
2.5.3 Utilizzo della memoria 28
2.5.4 Granularità dei thread 31
2.6 Sincronizzazione 31

3 Approccio preliminare **33**
3.1 Modello di programmazione CUBLAS 33
3.2 Benchmark 34
3.3 Utilizzo cublas 36

4 Analisi e implementazione **39**
4.1 Matrice Pseudoinversa 39
4.1.1 Single Value Decomposition 40
4.1.2 Decomposizione QR 40
4.2 Matrice di input 41
4.3 Scelte implementative 42
4.3.1 Preambolo sulla struttura dei kernel 43
4.3.2 Prodotto della matrice di input trasposta per se stessa . . 44
4.3.3 Calcolo della matrice inversa 47
4.3.4 Prodotto della matrice inversa per la trasposta della matrice di input 53

5 Test e risultati **59**
5.1 Calcolo complessivo 59
5.2 Prodotto della matrice di input trasposta per se stessa 64
5.3 Calcolo della matrice inversa 66
5.4 Prodotto della matrice inversa per la trasposta della matrice di input 67

6 Conclusioni **73**
6.1 Sviluppi futuri 75

A Dispositivi grafici NVIDIA **77**
A.1 Compute capability 77
A.2 Scheda grafica utilizzata 79

B Codice **81**

Bibliografia **95**

Elenco delle figure

1.1 Ackermann steering geometry. 3
1.2 Esempio di triangolazione di un'immagine e sample points. 5
1.3 Confronto tra il trend di sviluppo delle CPU e GPU. 7

2.1 Architettura di una GPU con supporto a CUDA. 10
2.2 Distribuzione dei thread nella gerarchia a due livelli di CUDA. . . 12
2.3 Processo di compilazione di un file .cu contenente codice host e codice device. 22
2.4 Accessi non coalescenti per device con compute capability 1.0/1.1. 25
2.5 Accessi coalescenti per device con compute capability 1.0/1.1. . . 26
2.6 Accessi coalescenti per device con compute capability 1.2/1.3. . . 27
2.7 Prodotto tra matrici standard. 29
2.8 Prodotto tra matrici sfruttando la memoria condivisa. 30

3.1 Tempi di esecuzione del prodotto tra matrici rettangolari. 35
3.2 Tempi di esecuzione del prodotto tra matrici quadrate. 35

4.1 Struttura della matrice di input. 42
4.2 Prodotto tra la trasposta della matrice di input e se stessa. 44
4.3 Tecnica di calcolo per il prodotto tra la matrice di input e la sua trasposta. 45
4.4 Confronto tempi di esecuzione delle varie versioni dell'algoritmo per il calcolo della matrice inversa al variare di m. 51
4.5 Tecnica di calcolo per il prodotto tra la matrice inversa e la matrice di input trasposta. 55

4.6 Confronto tempi di esecuzione delle varie versioni dell'algoritmo per l'ultima fase del calcolo della matrice pseudoinversa al variare di m con n=20000. 57
4.7 Confronto tempi di esecuzione delle varie versioni dell'algoritmo per l'ultima fase del calcolo della matrice pseudoinversa al variare di m con n=80000. 57

5.1 Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=20000. 60
5.2 Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=40000. 60
5.3 Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=60000. 61
5.4 Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=80000. 61
5.5 Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=100000. 62
5.6 Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=120000. 62
5.7 Confronto tra i fattori di miglioramento delle prove effettuate. . . 63
5.8 Fattore di miglioramento medio delle prove effettuate al variare di n. 64
5.9 Tempi di esecuzione del primo prodotto in CPU e GPU. 65
5.10 Fattore di miglioramento del primo prodotto. 65
5.11 Tempi di esecuzione del calcolo dell'inversa in CPU e GPU. 66
5.12 Fattore di miglioramento del calcolo della matrice inversa. 67
5.13 Tempi di esecuzione dell'ultimo prodotto al variare di m con n=20000. 68
5.14 Tempi di esecuzione dell'ultimo prodotto al variare di m con n=40000. 68
5.15 Tempi di esecuzione dell'ultimo prodotto al variare di m con n=60000. 69
5.16 Tempi di esecuzione dell'ultimo prodotto al variare di m con n=80000. 69
5.17 Tempi di esecuzione dell'ultimo prodotto al variare di m con n=100000. 70
5.18 Tempi di esecuzione dell'ultimo prodotto al variare di m con n=120000. 70
5.19 Fattori di miglioramento delle prove effettuate. 71
5.20 Fattore di miglioramento medio delle prove effettuate. 71

5.21 Confronto tra i fattori di miglioramento del calcolo totale e dell'ultimo prodotto. 72

A.1 Scheda video NVIDIA msi N9800GT. 79

Elenco delle tabelle

A.1 Specifiche tecniche associate alle varie compute capability. 78
A.2 Caratteristiche supportate dalle varie compute capability. 79
A.3 Specifiche tecniche della scheda video NVIDIA 9800 GT. 80

Capitolo 1

Introduzione

L'obbiettivo di questo lavoro di tesi consiste nell'ottenere un consistente vantaggio prestazionale nei tempi di esecuzione del software di Visual SLAM [19], attraverso la reingegnerizzazione del software (o parte di esso) per sfruttare l'iperparallelismo delle moderne GPU.
Principalmente ci si propone di rendere il software eseguibile in tempo reale, eliminando l'attuale latenza che lo porta ad essere inutilizzabile per gli scopi per cui è stato progettato. Nella tesi verranno inoltre approfonditi aspetti sulla corretta ed efficiente programmazione delle GPU, focalizzando l'attenzione anche su alcuni aspetti che nella programmazione standard delle CPU non vengono analizzati, ma che diventano fondamentali nell'ambito delle schede grafiche.

Per raggiungere i risultati desiderati sarà inoltre necessario rivisitare alcuni algoritmi operanti su matrici al fine di adattarli ad un'esecuzione parallela secondo il paradigma di CUDA, effettuandone uno sviluppo ad hoc anche per quanto riguarda la tipologia delle matrici da elaborare.

1.1 Il sistema di ricostruzione tridimensionale densa dell'ambiente

Attualmente la maggior parte degli approcci al problema del Visual SLAM sono basati su points features: le features vengono rilevate e associate tra fotogrammi

consecutivi. Il movimento della fotocamera e la localizzazione 3D di tali punti viene stimata attraverso l'utilizzo di tecniche probabilistiche. Anche se queste sono tecniche potenti per la stima del movimento della fotocamera, considerando il solo ausilio della visione, una mappa di punti 3D non fornisce molte informazioni utili riguardo all'ambiente circostante: solitamente i punti che vengono mappati sono infatti sparsi e forniscono al robot una informazione incompleta riguardo la struttura e l'aspetto dell'ambiente.
Gli attuali sistemi di Visual SLAM usano tecniche che sfruttano la prospettiva [10, 3, 4, 5, 20, 7], la visione stereo [11, 6] o le telecamere panoramiche [2, 14], per ricercare punti [10, 3], linee [5, 7] o features planari [20]. La maggior parte di essi si basano su approcci che utilizzano filtri probabilistici come l'Extended Kalman Filters, Rao-Blackwellized Particle Filters [15] o framework di SLAM basati sui grafi [9].
L'approccio utilizzato in questo sistema stima la posizione della fotocamera sfruttando i vincoli della geometria epipolare, mentre per la costruzione di una mappa 3D densa dell'ambiente viene utilizzato un metodo stereo multi-vista basato sui superpixel. L'assunzione alla base del metodo utilizzato consiste nel considerare ogni oggetto che compone la scena rappresentabile con un insieme di superfici 3D proiettate sul piano dell'immagine. Un'approssimazione naturale della suddivisione di una scena in superfici è data dalla mesh triangolare, ovvero la suddivisione dell'immagine in triangoli che condividono a due a due un lato.
Tenendo in considerazione la topologia della suddivisione, viene calcolata contemporaneamente la posizione 3D e la struttura densa dell'ambiente (quindi la profondità dei punti che nella suddivisione costituiscono i vertici dei triangoli) come un problema di ottimizzazione dove il costo da minimizzare è fotometrico e non geometrico, come invece avviene nella maggioranza dei sistemi di ricostruzione e navigazione basati sulle features. La profondità di tutti i punti che giacciono in un triangolo può essere calcolata in forma chiusa dalla profondità dei tre vertici. Questo permette l'utilizzo nell'ottimizzazione di un gran numeri di punti dell'immagine (possibilmente tutti), dove il numero dei parametri dipende solo dalla suddivisione scelta.

1.1.1 Il modello del movimento

Per simulare il movimento del robot in un ambiente generico, la fotocamera omnidirezionale è stata posizionata sul tettuccio di una macchina che circola in un ambiente urbano. Il software assume che l'autovettura si muova su una superficie piana. In questo caso la cinematica può essere descritta con la Ackermann steering geometry.
Questo modello assume che in ogni istante tutte le linee perpendicolari al raggio delle ruote si incontrino in un unico punto detto *Instantaneous Center of Rotation* (ICR) (Fig. 1.1). Assumendo quindi che il veicolo ruoti attorno ad un solo punto per un certo periodo di tempo possiamo quindi modellare il movimento attraverso un numero discreto di rotazioni attorno ad un diverso r_{ICR}.

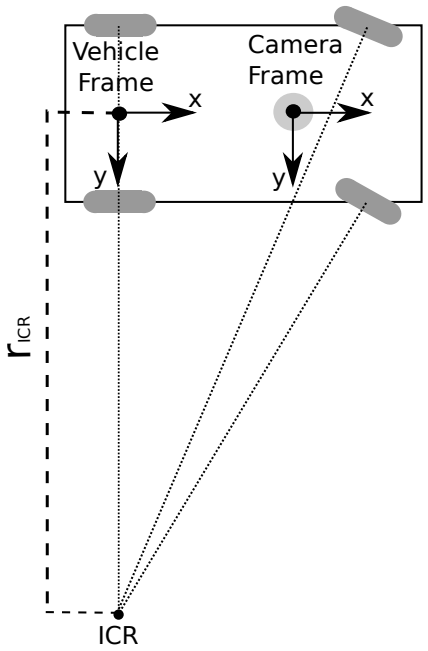


Figura 1.1: Ackermann steering geometry.

In questo modo è possibile parametrizzare la rigid-body motion (vedi [13]) con un

solo parametro, il raggio r_{ICR}. Possiamo quindi definire $\mathbf{G}\langle r_{ICR}\rangle \in \mathbb{SE}(3)$ come:

$$\mathbf{G}\langle r_{ICR}\rangle = \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 & cos\left(\frac{\alpha}{2}\right) \\ sin(\alpha) & cos(\alpha) & 0 & sin\left(\frac{\alpha}{2}\right) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

dove $\alpha = 2 \cdot asin\left(\frac{1}{2 \cdot r_{ICR}}\right)$.

1.1.2 Struttura dell'immagine e profondità

Per ogni immagine viene definita una struttura costituita da una sua suddivisione in triangoli. Questa suddivisione viene effettuata estraendo prima un certo numero di punti significativi, detti feature point, che andranno poi a costituire i vertici dei triangoli in cui verrà suddivisa l'immagine attraverso la triangolazione di Delaunay.
Oltre a questo set di caratteristiche ne verrà estratto un altro, costituito da un insieme di punti detti sample points scelti uniformemente all'interno dell'immagine in esame a cui verrà associato il triangolo all'interno del quale si vengono a trovare.

Le profondità che sarà necessario individuare per la ricostruzione 3D dell'immagine sono tutte e sole quelle dei vertici individuati come feature point, che andranno quindi a costituire le variabili del sistema assieme al raggio r_{ICR}. Le profondità dei sample point potranno essere invece calcolate in forma chiusa partendo da quelle dei vertici del triangolo di appartenenza seguendo il seguente schema:

sia $\mathbf{s}_i \in \mathbb{S}^2$ un sample point e siano $\mathbf{v}_{k_{i,1}}, \mathbf{v}_{k_{i,2}}, \mathbf{v}_{k_{i,3}} \in \mathbb{S}^2$ i tre vertici che ne definiscono il triangolo di appartenenza. Se le profondità dei vertici, rispettivamente $\lambda_{k_{i,1}}, \lambda_{k_{i,2}}, \lambda_{k_{i,3}}$ sono conosciute, allora la profondità di $\mathbf{s}_i$ può essere calcolata osservando che il determinante della seguente matrice deve essere pari a zero:

$$\begin{vmatrix} \mathbf{s}_i(x) & \mathbf{s}_i(y) & \mathbf{s}_i(z) & 1/\lambda(\mathbf{s}_i) \\ \mathbf{v}_{k_{i,1}}(x) & \mathbf{v}_{k_{i,1}}(y) & \mathbf{v}_{k_{i,1}}(z) & 1/\lambda_{k_{i,1}} \\ \mathbf{v}_{k_{i,2}}(x) & \mathbf{v}_{k_{i,3}}(y) & \mathbf{v}_{k_{i,2}}(z) & 1/\lambda_{k_{i,2}} \\ \mathbf{v}_{k_{i,3}}(x) & \mathbf{v}_{k_{i,3}}(y) & \mathbf{v}_{k_{i,3}}(z) & 1/\lambda_{k_{i,3}} \end{vmatrix} = 0 \tag{1.2}$$

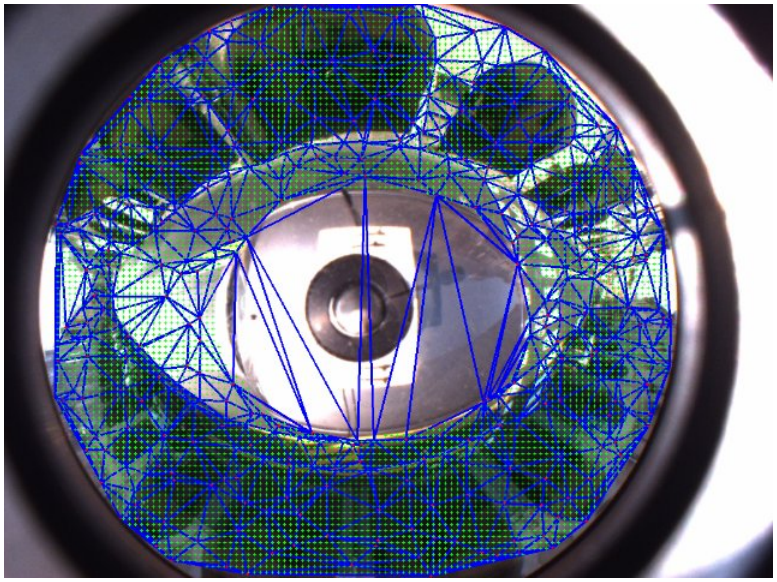


Figura 1.2: Esempio di triangolazione di un'immagine e sample points.

La profondità $\lambda(\mathbf{s}_i)$ è quindi data dalla seguente formula:

$$\lambda(\mathbf{s}_i) = \frac{|\mathbf{M}_4|}{\mathbf{s}_i(x)\ |\mathbf{M}_1|\ -\ \mathbf{s}_i(y)\ |\mathbf{M}_2|\ +\ \mathbf{s}_i(z)|\mathbf{M}_3|} \tag{1.3}$$

dove $|\mathbf{M}_i|$, $i = 1, \ldots, 4$ sono i minori della matrice definita nell' Eq. 1.2 ottenuti rimuovendo la prima riga e l' $i - esima$ colonna.

1.1.3 Procedura di ottimizzazione

Nella procedura di ottimizzazione utilizzata dal software viene utilizzato come parametro di movimento $\rho \in \mathbb{R}$ l'inverso del raggio r_{ICR}, $\rho \triangleq 1/r_{ICR}$. Questa parametrizzazione permette di rappresentare esplicitamente il moto di pura traslazione ($\rho = 0$ corrisponde a $r_{ICR} \to \infty$). La regola di aggiornamento per il parametro di movimento è quindi data da:

$$\widehat{\rho} \leftarrow \widehat{\rho} + \widetilde{\rho} \tag{1.4}$$

dove $\widehat{\rho}$ rappresenta il valore stimato e $\widetilde{\rho}$ l'incremento da trovare.
I parametri della struttura dell'immagine vengono rappresentati con l'inverso della profondità dei vertici $\zeta_i \triangleq 1/\lambda_i$, $i = \{1, \ldots, n\}$. Per rafforzare i vincoli di

cheirality, ζ_i è stata parametrizzata come $\zeta_i = \zeta_i(\xi) = e^{\xi_i}$, con $\xi_i \in \mathbb{R}$. Da queste considerazioni la regola di aggiornamento diventa:

$$\widehat{\zeta_i} \leftarrow \widehat{\zeta_i} \cdot \zeta_i(\widetilde{\xi}) = \widehat{\zeta_i} \cdot e^{\widetilde{\xi_i}} \tag{1.5}$$

dove come nel caso precedente $\widehat{\zeta_i}$ rappresenta il valore stimato e $\widetilde{\xi}$ l'incremento da trovare. Con questa parametrizzazione lo scopo della procedura di ottimizzazione è di trovare i parametri $\rho \in \mathbb{R}$ e $\xi \in \mathbb{R}^n$, $\xi = [\xi_1, \ldots, \xi_n]'$ che minimizzano la seguente formula:

$$\phi(\rho, \xi) = \frac{1}{2} \sum_{i=1}^{n} \left[\mathcal{I}\left\{\mathbf{v}_i\right\} - \mathcal{I}^*\left\{\mathbf{v}_i^*\right\}\right]^2 + \frac{1}{2} \sum_{i=1}^{m} \left[\mathcal{I}\left\{\mathbf{s}_i\right\} - \mathcal{I}^*\left\{\mathbf{s}_i^*\right\}\right]^2 \tag{1.6}$$

dove I, v_i, s_i afferiscono all'immagine di riferimento, mentre I^*, v_i^*, s_i^* fanno riferimento all'immagine corrente. Impostando $\theta = (\rho, \xi)$ e definendo $\mathbf{d}(\theta)$ come la funzione costo da minimizzare, durante ogni passo di aggiornamento è necessario trovare il valore ottimo tale che:

$$\theta^0 \;=\; \text{argmin}_\theta \; \frac{1}{2} \parallel \mathbf{d}(\theta) \parallel \tag{1.7}$$

Un'efficiente approssimazione del secondo ordine di $\mathbf{d}(\theta)$ è:

$$\mathbf{d}(\theta) \backsimeq \mathbf{d}(\mathbf{0}) + \frac{1}{2} \left(\mathbf{J}(\theta) + \mathbf{J}(\mathbf{0})\right) \theta \tag{1.8}$$

dove $\mathbf{J}(\theta)$ è lo Jacobiano di $\mathbf{d}(\theta)$ calcolato in θ. Sotto certe condizioni, $\mathbf{J}(\theta)\theta$ può essere calcolata senza conoscere il valore di θ. Impostando $\mathbf{d}(\theta) = 0$, possiamo calcolare θ^0 come:

$$\theta^0 = -\left(\left(\frac{\mathbf{J}_{\mathcal{I}^*} + \mathbf{J}_{\mathcal{I}}}{2}\right) \breve{\mathbf{J}}(\mathbf{0})\right)^+ = -\mathring{\mathbf{J}}^+ \tag{1.9}$$

dove + definisce la pseudoinversa.

1.2 GPGPU: General-Purpose computing on Graphics Processing Units

La GPGPU (General-Purpose computing on Graphics Processing Units) è un settore della ricerca informatica che ha come scopo l'utilizzo del processore della

scheda grafica, vale a dire la GPU, per scopi diversi dalla tradizionale creazione di immagini tridimensionali; in tali ambiti la GPU viene impiegata per elaborazioni estremamente esigenti in termini di potenza di elaborazione, e per le quali le tradizionali architetture di CPU non hanno una capacità di elaborazione sufficiente.

Tale tipo di elaborazioni sono, per loro natura, di tipo altamente parallelo, e in grado quindi di beneficiare ampiamente dell'architettura tipica delle GPU; a tale caratteristica intrinseca, a partire dal 2007 si è aggiunta l'estrema programmabilità offerta dalle ultime soluzioni commerciali fornite da NVIDIA, che al succedersi delle generazioni aumentano non solo la propria potenza elaborativa ma anche la propria versatilità.

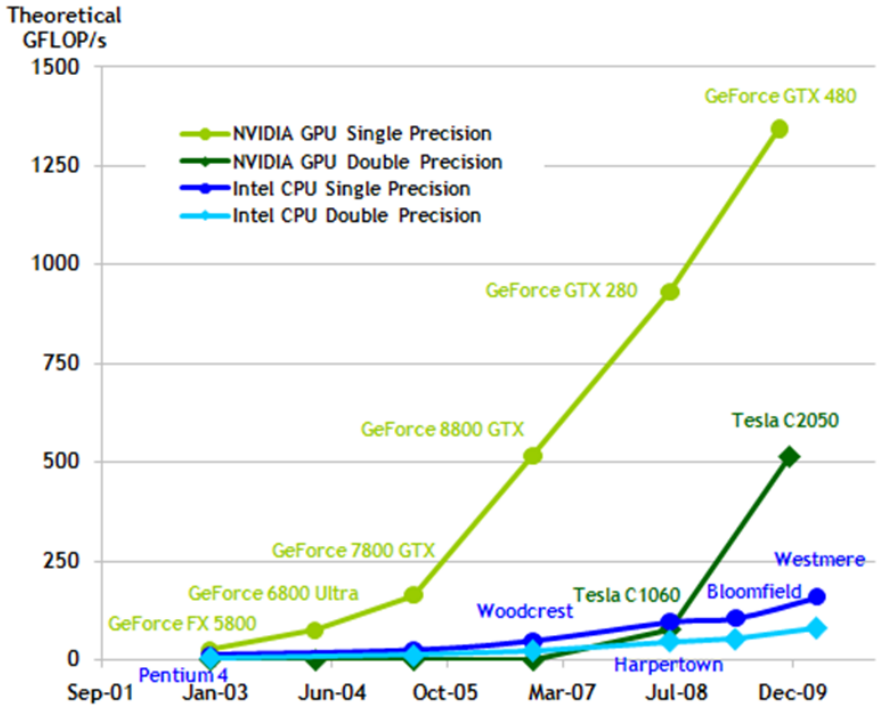


Figura 1.3: Confronto tra il trend di sviluppo delle CPU e GPU.

L'importanza di conoscere questo modello di programmazione, oltre ai fini pratici del caso attuale e i vantaggi dettati dal poter sfruttare il parallelismo delle GPU, consiste anche nel fatto che quest'ultime, a differenza delle CPU, sono tuttora in grande sviluppo e, mentre le CPU sono sostanzialmente ad uno stato di sviluppo statico in termini di prestazioni singole (al giorno d'oggi si punta infatti ad affi-

ancare più core per aumentare le prestazioni), le schede grafiche stanno ancora seguendo la curva che ha caratterizzato la crescita dei PC fino agli ultimi anni. La figura 1.3 mostra il trend di sviluppo in termini di potenza di calcolo che hanno seguito le CPU e le GPU negli ultimi anni. La differenza è evidente, e si può ben comprendere l'importanza di esplorare il funzionamento di questa tecnica di programmazione per evidenziarne limiti e vantaggi.

Capitolo 2

CUDA: Compute Unified Device Architecture

Per sfruttare le caratteristiche che le moderne GPU mettono a disposizione (in particolare le GPU NVIDIA) è necessario comprendere tutte le caratteristiche che entrano in gioco nello sviluppo di un programma il cui obbiettivo è massimizzare l'efficienza messa a disposizione da questi strumenti.
In questo capitolo viene fornita una panoramica completa su ciò che bisogna conoscere prima di accingersi a sviluppare codice in CUDA fornendo una premessa sulle conoscenze acquisite nello studio di questo strumento e utilizzate per migliorare l'efficienza del software presentato nel capitolo precedente.

2.1 Architettura

CUDA lavora, concettualmente, sul modello architetturale riportato nella figura 2.1.
Il chip grafico, nel modello di CUDA, è costituito da una serie di multiprocessori, denominati Streaming MultiProcessor. Il numero di multiprocessori dipende dalle caratteristiche specifiche e dalla classe di prestazioni di ciascuna GPU. Ciascun multiprocessore è a sua volta formato da 8 Stream Processors il cui numero invece è fisso, indipendente dalla fascia di mercato del chip grafico. Ognuno di questi processori può eseguire una operazione matematica fondamentale (addizione, moltiplicazione, sottrazione, ecc) su interi o su numeri in virgola mobile

in singola precisione (32 bit). In ciascun multiprocessore ci sono anche due unità per funzioni speciali (che eseguono operazioni trascendenti come seno, coseno, inverso ecc.) e, solo per i chip basati su architettura GT200, una singola unità in virgola mobile a doppia precisione (64 bit). In un multiprocessore è anche presente una shared memory, accessibile da tutti gli streaming processor, delle cache per le istruzioni e per i dati e, infine, una unità di decodifica delle istruzioni. Gli altri tipi di memoria, così come evidenziato in figura, non sono privati di ogni SM ma sono accessibili da ognuno di essi e rappresentano i repository principali per le grandi moli di dati da salvare in GPU.
Un particolare importante da chiarire è che c'è una sola unità per 8 processori, per cui il funzionamento di ogni SP non può essere gestito singolarmente.

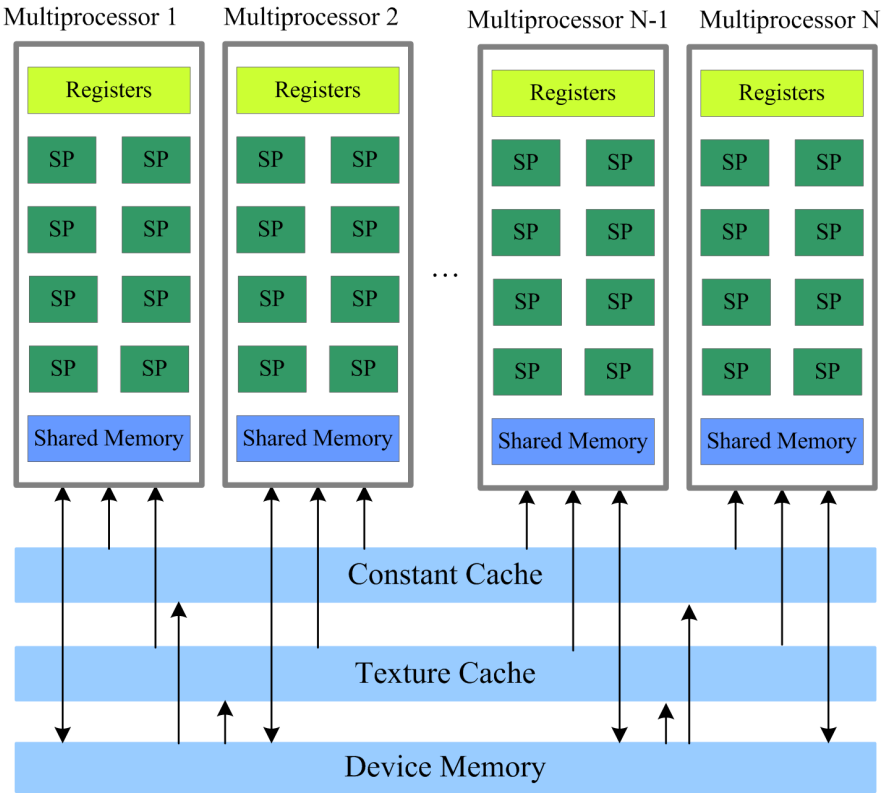


Figura 2.1: Architettura di una GPU con supporto a CUDA.

2.2 Modello di programmazione

Il modello di programmazione CUDA è la base di partenza da comprendere per poter iniziare a programmare correttamente la GPU.
Ci sono dei concetti che vanno compresi e assimilati per potersi approcciare correttamente a questo strumento e per poter capire gli argomenti più specifici trattati nei seguenti capitoli. Inoltre alcuni termini possono richiamare alla memoria conoscenze pregresse che però possono portare a conclusioni errate sul funzionamento di queste librerie.
In questo capitolo vengono stese le basi dello sviluppo di codice in GPU utilizzando le librerie messe a disposizione da NVIDIA.

2.2.1 Kernel e gerarchia dei thread

Una delle parole chiavi da conoscere nella programmazione CUDA e *kernel*. Questo termine, da non confondere con le conoscenze derivanti da sistemi operativi e algebra lineare, in questo ambito ha il significato di funzione. Una funzione assume il significato di *kernel* se alla sua firma viene apposta la specifica *__global__*. La caratteristica principale di un kernel è il fatto che il suo codice viene eseguito parallelamente N volte sulla base di specifiche che verranno chiarite successivamente.
La singola esecuzione del kernel viene effettuata da unità dette *thread*, il cui significato può essere accostato a quello normalmente associato ai thread CPU, nel senso che sono singoli flussi di codice che (nei limiti che vedremo) vivono di vita propria.
A differenza degli omonimi componenti in CPU, i *thread* in GPU possiedono un contesto molto più leggero (vedi [17]), tale per cui il cambio di contesto non rappresenta uno dei fattori da tenere in considerazione nell'analisi prestazionale del software e può essere considerato istantaneo.
Per stabilire il numero di thread che devono eseguire un singolo kernel e la loro organizzazione logica CUDA definisce una gerarchia a due livelli. Nel livello più alto si definisce la dimensione di una griglia o *grid* di blocchi o *block*. La griglia rappresenta un livello bidimensionale in cui vengono distribuiti i blocchi che a loro volta rappresentano una struttura tridimensionale in cui viene specificato il

numero di thread per dimensione.

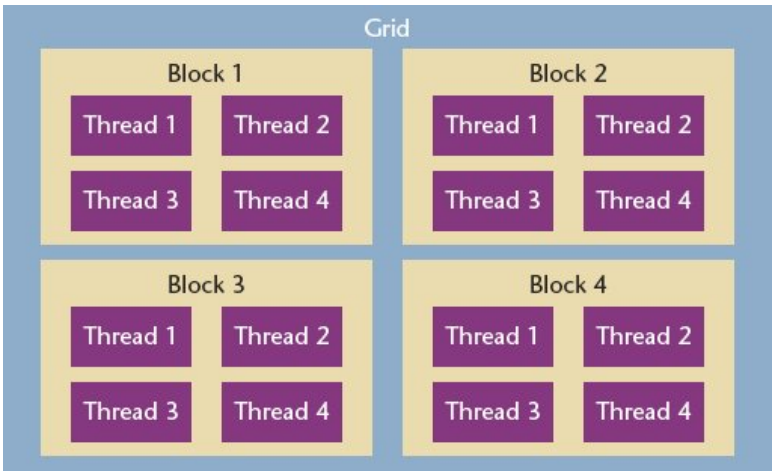


Figura 2.2: Distribuzione dei thread nella gerarchia a due livelli di CUDA.

Sulla base di questa struttura un kernel per essere lanciato deve ricevere come parametri supplementari due strutture che specifichino appunto le dimensioni di grid e block. Questo listato mostra la sintassi per lanciare una funzione kernel nominata $kernel\_function$.

```
dim3 block_size(x,y,z);
dim3 grid_size(x,y);
kernel_function<<<grid_size, block_size>>>(parametri della funzione);
```

La divisione logica di un kernel in grid e block è un aspetto cruciale nella progettazione di codice in CUDA.
Oltre ai limiti imposti dall'hardware, evidenziati nella tabella A.1, esistono altre accortezze che vanno tenute in considerazione per rendere il proprio codice più efficiente che verranno presentati in seguito.

2.2.2 Gestione interna dei thread

Un programma operante in GPU, per sfruttare al meglio le risorse disponibili, deve rispettare dei canoni dettati dalla struttura e dall'organizzazione interna degli SM che impongono dei vincoli sulle performance dei thread.

Gli Streaming Multiprocessor sono l'unità fondamentale dell'architettura CUDA, disegnati per eseguire concorrentemente un numero molto elevato di thread. Per fare questo implementano un'architettura detta SIMT (Single-Instruction, Multiple-Thread), la quale crea, gestisce, schedula ed esegue thread a gruppi di 32, detti *warp*. I singoli thread di un warp iniziano l'esecuzione con lo stesso program counter ma nel proseguo avranno il loro instruction address counter e registri, e saranno liberi di seguire rami dell'esecuzione indipendenti.
Un warp esegue un'istruzione comune alla volta, perciò per sfruttare al massimo l'efficienza della struttura è necessario che tutti i 32 thread concordino con lo stesso path di esecuzione. Maggiore è la divergenza, minore sarà l'efficienza del programma.

Quando uno o più thread block vengono assegnati ad un multiprocessore per essere eseguiti, vengono partizionati in warp che vengono schedulati da un *warp scheduler*. Ogni SM infatti, nonostante possa ospitare contemporaneamente un massimo di 24 (32 o 48 per le schede più recenti) warp, è stato progettato perchè in ogni istante di tempo ne possa eseguire uno ed uno solo. Una spiegazione per questa scelta risiede nel fatto che alcune istruzioni eseguite dai thread, come per esempio gli accessi in memoria, sono caratterizzate da una grande latenza perciò, mentre un warp aspetta che i dati richiesti siano pronti, viene messo in una coda di attesa e viene schedulato un altro degli warp assegnati al multiprocessore per essere eseguito.
Lo scopo di questa struttura è permettere all'hardware di essere sempre occupato nell'esecuzione di un warp nonostante i grandi tempi di latenza di alcune istruzioni. Nell'analisi delle prestazioni non bisogna tenere in considerazione overhead generato dal meccanismo di selezione dello warp da eseguire, in quanto non introduce tempi di attesa.

Per capire i ragionamenti che vanno fatti nella progettazione di un kernel per quanto riguarda la suddivisione dei blocchi, prendiamo in considerazione una scheda con compute capability 1.1 (come quella utilizzata per la realizzazione di questo lavoro di tesi). I limiti imposti dall'hardware sono:

- Numero massimo di thread per SM=768

- Numero massimo di thread per block=512
- Numero massimo di block per SM=8

Da questi dati e con i ragionamenti effettuati precedentemente avremo che ad ogni SM potranno essere assegnati un massimo di $768/32 = 24$ warp. Quindi per riempire la capacità elaborativa dello Streaming Multiprocessor si ha che innanzi tutto il numero di thread per block deve essere divisibile per 32, altrimenti avremmo l'assegnazione di warp parzialmente riempiti (verranno aggiunti dei thread fittizi per raggiungere i 32 necessari), inoltre il numero di warp per block deve essere un divisore di 24, altrimenti non potrà essere raggiunto il numero massimo di warp assegnabili ad ogni SM. Sempre per riempire lo SM, considerando il vincolo del numero massimo di blocchi assegnabili, ogni blocco non dovrebbe avere meno di $24/8 = 3$ warp, quindi 96 thread.
Riassumendo i ragionamenti fin qui effettuati sulla sola organizzazione interna dei thread quindi i numeri consigliati di thread per block (non importa distribuiti come nelle varie dimensioni) sono 64, 96, 128, 192, 256, 384.

2.3 Memoria

In CUDA la conoscenza e il corretto utilizzo dei vari tipi di memoria che la GPU mette a disposizione è fondamentale al fine di ottenere la massima efficienza nei programmi che si andranno a realizzare.
Nella GPU esistono i seguenti tipi di memoria:

- Global memory
- Constant memory
- Texture memory
- Shared memory
- Local memory
- Registri

2.3.1 Global memory

La memoria globale è la principale memoria che si ha a disposizione in GPU. La dimensione di questa memoria varia da dispositivo a dispositivo, ma ha comunque valori che variano dai 256MB ai 1024MB, che quindi assicurano una buona base per poter caricare i dati. Questa memoria, come è facilmente immaginabile, oltre a essere la più capiente è anche la più lenta e può rappresentare il vero collo di bottiglia dei software che verranno realizzati in CUDA in quanto necessita di 400-600 cicli per l'accesso ai dati.
E' buona prassi utilizzare inizialmente questa memoria, per poi cercare di smistare i dati o comunque gestirli quando possibile utilizzandone altri tipi. Il vantaggio di questa memoria oltre alle dimensioni è anche la visibilità in quanto questa è la sola memoria che sia in lettura che in scrittura sia accessibile da tutti i thread della griglia.

Per utilizzare la memoria globale è necessario innanzi tutto allocare lo spazio in memoria per ospitare i dati residenti in CPU. Questa operazione viene effettuata attraverso le seguenti istruzioni:

```
int N = ...;
size_t size = N * sizeof(float);
float* device_A;
cudaMalloc(&device_A, size);
```

Una volta allocato lo spazio è necessario copiare in GPU i dati presenti in CPU. Questa operazione può essere effettuata in diversi modi in quanto cuda presenta diverse primitive per la copia di dati, dipendenti anche dal tipo di dati che si vogliono copiare e dalla loro struttura.
La più semplice e la più utilizzata è la seguente:

```
float* host_A = (float*)malloc(size);
cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
```

Una volta terminata l'elaborazione i dati possono di nuovo essere scaricati dalla memoria globale della GPU alla CPU attraverso la medesima istruzione:

```
cudaMemcpy(host_A, device_A, size, cudaMemcpyDeviceToHost);
```

2.3.2 Constant memory

La memoria costante è, come si può intuire dal nome, una memoria utilizzabile per caricare i dati in sola lettura.
Per tutti i dati che nei kernel verranno solo letti questa memoria è preferibile rispetto alla memoria globale in quanto permette di ottenere prestazioni superiori grazie ai meccanismi di caching che mette a disposizione.
I limiti di questa memoria, oltre al fatto di essere in sola lettura, stanno nelle dimensioni della stessa, in quanto per tutte le versioni dei device finora realizzati la memoria costante ha a disposizione 64KB di spazio. Nel caso quindi in cui sia necessario allocare più dati in memoria risulta necessario suddividere l'esecuzione dell'algoritmo per poter caricare di volta in volta la porzione di dati su cui operare.

Per utilizzare la Constant memory è necessario creare una variabile con scope globale, ovvero visibile sia dal codice host che dal codice kernel, di tipo *__constant__*.
Di queste variabile verrà creata una copia sola il cui ciclo di vita sarà la durata dell'applicazione stessa e sarà visibile da tutti i thread del grid.
Questo codice di esempio mostra come utilizzare questo tipo di variabili:

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));
```

Dove il metodo *cudaMemcpyToSymbol* permette di caricare i dati in constant memory mentre *cudaMemcpyFromSymbol* permette di scaricarli in CPU.

2.3.3 Texture memory

La memoria di tipo Texture è un altro tipo di memoria read only che può essere utilizzata in alternativa alla memoria costante.
La Texture memory offre come la memoria di tipo Constant dei meccanismi di

caching ed è possibile utilizzarla con matrici a una, due o tre dimensioni. Evidenzia i maggiori vantaggi nei casi in cui il programma sia dotato di una località spaziale 2D per cui le operazioni di $fetch$ dei dati hanno le prestazioni migliori. Inoltre è possibile effettuare automaticamente e in maniera molto veloce l'interpolazione lineare dei dati tra elementi consecutivi della Texture sia 1D, 2D che 3D. Un'altra caratteristica che può risultare utile è la modalità di accesso ai dati, che può essere effettuata sia attraverso un indice intero, così come avviene normalmente l'accesso ai dati di una matrice, che attraverso un valore normalizzato compreso tra 0 e 1. In quest'ultimo caso viene effettuata un'interpolazione automatica tra i dati adiacenti che corrispondono alla posizione indicata dall'indice normalizzato.
Per quanto riguarda i limiti in termini di spazio di questa memoria, essi dipendono dal tipo di allocazione che viene fatta (ovvero lineare, 1D, 2D o 3D). In ogni caso viene messa a disposizione una grande quantità di memoria su cui fare il *binding* ad un riferimento texture. I limiti si possono trovare nella tabella A.1.

Una variabile di tipo *texture* è visibile sia dai kernel che dal codice host, quindi deve essere dichiarata con scope globale. Come per le variabili di tipo *__constant__* ha durata pari al ciclo di vita dell'applicazione, o fino a che non viene fatto l' *unbind* del riferimento alla memoria.
Per poter utilizzare variabili che risiedono in Texture memory quindi è necessario prima dichiarare un riferimento alla texture:

```
texture<float,1,cudaReadModeElementType> tex_ref;
```

e poi effettuare il binding dei dati che già risiedono in memoria globale:

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
cudaBindTexture(NULL, &device_A, tex_ref, &channelDesc, size);
```

Ora all'interno dei kernel i dati potranno essere letti attraverso le operazioni di $tex1Dfetch$ nel caso si abbia fatto il binding di una zona di memoria lineare, altrimenti attraverso le primitive $tex1D$, $tex2D$ e $tex3D$ nel caso si tratti di array multidimensionali.

Una volta terminata l'utilità di queste variabili è necessario effettuare l' *unbinding* per liberare le risorse:

```
cudaUnbindTexture(tex_ref);
```

2.3.4 Shared memory

La Shared memory è la memoria che viene condivisa da tutti i thread appartenenti ad un blocco, e può essere utilizzata per effettuare comunicazioni e sincronizzazioni tra i thread stessi. E' la memoria il cui uso, se possibile, va massimizzato in quanto offre le migliori caratteristiche prestazionali.
La Shared memory è suddivisa in 16KB (48KB nelle ultime versioni) di memoria per ogni Streaming Multiprocessor, e ricordando che ogni SM può ospitare fino a 8 blocchi di thread per avere il massimo delle performance ad ogni blocco non dovrebbero essere assegnati più di 2KB di memoria per occupare al massimo le risorse di elaborazione.

Le variabili che vengono memorizzate in memoria condivisa hanno un ciclo di vita limitato alla durata del kernel, per cui nascono e muoiono con la sua esecuzione. Devono essere create all'interno del kernel stesso apponendo al tipo di variabile la dicitura *__shared__*.
Bisogna tenere presente che un variabile di questo tipo è visibile da tutti i thread di un block, ma ne esisteranno copie diverse per thread appartenenti a blocchi distinti.

2.3.5 Local memory

La Local memory deve il suo nome non alla sua posizione fisica nella scheda (infatti è una memoria off-chip), ma bensì alla sua limitata visibilità. Infatti questa memoria è assegnata e visibile solo al singolo thread. Questa è la memoria che generalmente viene utilizzata meno sia perchè ha dei tempi di accesso paragonabili alla Global memory (quindi molto lenti), sia perchè essendo locale al thread non ha lo scopo di contenere i risultati globali delle operazioni eseguite dal *kernel*.
Ha dimensioni non trascurabili in quanto ogni thread ha a disposizione 16KB di

memoria (256KB nelle ultime versioni).

Le variabili che risiedono in memoria locale sono variabili che hanno visibilità ristretta al singolo thread e che quindi hanno come ciclo di vita il kernel.
La memorizzazione di variabili in questo tipo di memoria non è specificato dall'utente, ma solitamente è automatico da parte del compilatore per i seguenti tipi di variabile:

- array per cui non riesce a determinare se hanno o meno una dimensione costante;
- stuct o array di grandi dimensioni che occuperebbero eccessivamente lo spazio dei registri;
- qualsiasi variabile se il kernel utilizza più registri di quelli disponibili (fenomeno conosciuto anche come register spilling).

2.3.6 Registri

I registri sono locazioni di memoria da 32bit che presentano i migliori tempi di accesso tra i vari tipi di memoria, ma che al tempo stesso sono uno dei fattori da tenere in considerazione per non abbattere le performance del programma. Infatti ogni Streaming Multiprocessor ha a disposizione, dipendentemente dalle versioni della scheda video, 8K, 16K o 32K di registri. Questa può sembrare una cifra ragguardevole, ma ricordando che ogni SM può ospitare fino a 768, 1024 o 1536 thread, per non sprecare risorse ogni thread non dovrebbe utilizzare più di:

- 8000/768 = 10 registri per compute capability 1.0 e 1.1;
- 16000/1024 = 15 registri per compute capability 1.2 e 1.3;
- 32000/1536 = 20 registri per compute capability 2.0 e 2.1;

Quando questo non avviene il numero di thread ospitati contemporaneamente nello Streaming Multiprocessor diminuirà limitando le performance del kernel.

Per copiare i dati nei registri non c'è bisogno di nessuna sintassi particolare, basta tenere presente che i parametri del kernel locali al thread vengono copiati automaticamente all'interno di appositi registri, e ogni variabile dichiarata all'interno del kernel non di tipo *array* viene anch'essa salvata in un apposito registro. Potrebbe quindi essere conveniente copiare una posizione di memoria (globale, costante o texture) che viene letta spesso all'interno del kernel in un'apposita variabile locale che, essendo memorizzata in un registro, garantirà le migliori prestazioni.

2.3.7 Calcolo delle risorse

Nei paragrafi precedenti è stata spiegata la gestione dei thread, delle risorse disponibili e di come vengono allocate ai vari blocchi/thread. Risulta chiaro quindi che le risorse devono essere partizionate in modo adeguato tra le varie unità di esecuzione e in particolare bisogna ricordarsi del fatto che se non ci sono sufficienti registri o memoria condivisa disponibile per ogni multiprocessore per processare almeno un blocco di thread, il kernel non potrà essere eseguito.
Per assicurarsi dell'adeguatezza delle risorse rispetto al kernel che si vuole eseguire il manuale di programmazione [18] fornisce le seguenti formule per calcolare l'utilizzo di registri e memoria condivisa.

Il numero complessivo si warp W_{block} in un blocco si ottiene attraverso:

$$W_{block} = ceil(\frac{T}{W_{size}}, 1) \tag{2.1}$$

dove T rappresenta il numero di thread presenti in un blocco, W_{size} è la dimensione di un warp, ossia 32, mentre $ceil(x, y)$ è uguale ad x arrotondato per eccesso al più vicino multiplo di y.
Da questo è possibile calcolare il numero totale di registri R_{block} allocati per ogni blocco:

- per dispositivi con compute capability 1.x

$$R_{block} = ceil(ceil(W_{block}, G_W) \cdot W_{size} \cdot R_k, G_T) \tag{2.2}$$

– per dispositivi con compute capability 2.x

$$R_{block} = ceil(R_k \cdot W_{size} G_T) \cdot W_{block} \tag{2.3}$$

dove G_W è la granularità di allocazione del warp, uguale a 2, R_k è il numero di registri utilizzati dal kernel e G_T è la granularità di allocazione dei thread, uguale a 256 per dispositivi con compute capability 1.0 e 1.1, 512 per 1.2 e 1.3 e 64 per 2.x

La quantità di memoria shared S_{block} in byte allocata per ogni blocco, si ottiene attraverso:

$$S_{block} = ceil(S_k, G_S) \tag{2.4}$$

dove S_k rappresenta la quantità di memoria condivisa in byte utilizzata dal kernel e G_S rappresenta la granularità di allocazione di memoria condivisa, uguale a 512 per dispositivi con compute capability 1.x e 128 per dispositivi 2.x.

Per quanto riguarda i dati riguardanti l'occupazione di registri e memoria condivisa associata ad ogni kernel, sarà chiaro come reperirli nella sezione 2.4.

2.4 Processo di compilazione

Una volta compreso come sviluppare codice CUDA è necessario passare al passo successivo, ovvero la compilazione.
I file che contengono i kernel, di solito nominati con estensione .cu, non possono essere compilati con i compilatori standard, per esempio gcc, ma devono essere compilati dal compilatore fornito con il pacchetto cuda *nvcc*. Come si nota in figura 2.3 nvcc accetta in ingresso un file $nome\_file.cu$, ne estrae la parte host e passa il resto in formato .gpu a *nvopencc* che prosegue il processo di compilazione.

Questo compilatore accetta molte opzioni passabili in linea di comando, una molto interessante è la seguente:

```
--ptxas-options=-v
```

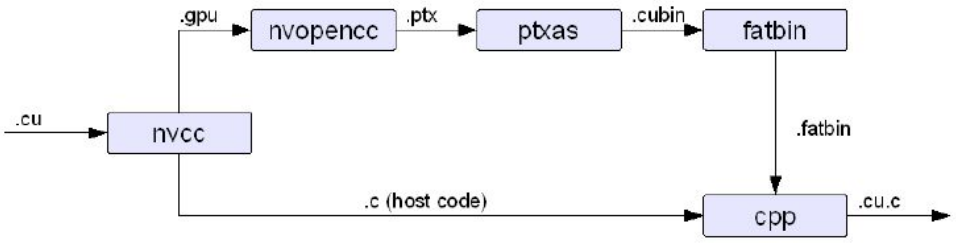


Figura 2.3: Processo di compilazione di un file .cu contenente codice host e codice device.

attraverso la quale è possibile vedere come risultato della compilazione l'analisi della memoria utilizzata dai vari kernel e in particolare il numero di registri utilizzati, cosa fondamentale per le considerazioni fatte in precedenza sulle limitazioni prestazionali che si possono avere utilizzando un numero elevato di registri. Riportiamo un esempio di risultato di compilazione con –ptxas attivato:

```
ptxas info : Compiling entry function '_Z26deviceJac2InvScaleFirstRowPfi' for 'sm_10'
ptxas info : Used 4 registers, 16+16 bytes smem, 4 bytes cmem[1], 4 bytes cmem[14]
ptxas info : Compiling entry function '_Z13deviceJac2InvPfS_i' for 'sm_10'
ptxas info : Used 7 registers, 32+16 bytes smem, 12 bytes cmem[1], 4 bytes cmem[14]
ptxas info : Compiling entry function '_Z21deviceJac2InvFirstRowPfS_i' for 'sm_10'
ptxas info : Used 10 registers, 1056+16 bytes smem, 20 bytes cmem[1], 4 bytes cmem[14]
```

dove nel corrispondente file .cu erano presenti i kernel deviceJac2InvScaleFirstRow, deviceJac2Inv e deviceJac2InvFirstRow.

2.5 Ottimizzazioni

Nel classico processo di apprendimento della tecnica di programmazione cuda un passo fondamentale è quello che viene fatto nel passaggio tra la prima versione del programma, in cui vengono rispettate tutte le specifiche ma che solitamente presenta delle performance deludenti, e la seconda, in cui si prende coscienza di concetti di programmazione più avanzati di cui tenere conto e sulla cui base riprogettare il software.
Questo è uno scoglio su cui è importante scontrarsi per toccare con mano quanto importanti siano gli accorgimenti (spiegati in [12]) che verranno presentati ai fini

della velocità di esecuzione del kernel.

2.5.1 Flussi del codice

Come già accennato nella sezione riguardante la gestione interna dei thread, lo Streaming Multiprocessor esegue i 32 thread afferenti ad uno stesso warp facendo eseguire a tutti la stessa istruzione contemporaneamente. La motivazione per questa tecnica sta nel fatto che in questo modo il costo per fare il fetch e processare un'istruzione viene ammortizzato dal grande numero di thread che la eseguono.
E' evidente che nel caso di punti del codice in cui il flusso di esecuzione differisce sulla base del thread che lo sta eseguendo, l'esecuzione del warp dovrà essere prolungata di tanti passi quanti sono necessari per completare tutti i rami di esecuzione presenti.
I punti del codice in cui può verificarsi questa perdita di efficienza sono:

- istruzioni condizionali (if e case), in questo caso è bene limitare al minimo il numero di rami differenti in cui si suddivide l'esecuzione del kernel;
- cicli for/while di lunghezze differenti, cioè quando il numero di iterazioni differisce tra i vari thread. In questo caso è bene cercare di uniformare i cicli se il programma lo consente.

Ovviamente questi ragionamenti sono legati al tipo di programma che si sta sviluppando. E' evidente che certi salti condizionali e certi cicli di lunghezze differenti non possono essere evitati, ma è bene prendere coscienza di questo fatto per evitare sprechi inutili.

2.5.2 Coalescenza

Uno dei concetti più importanti da apprendere per migliorare le performance del codice CUDA è la *coalescenza*. Questo termine indica una modalità di accesso ai dati che, se rispettata, permette di raggiungere picchi di performance molto elevati.
Gli accessi coalescenti vengono riferiti sempre ad accessi alla memoria globale,

molto lenta e spesso principale problema di inefficienza.
Gli accessi di un kernel vengono detti coalescenti se thread con id consecutivi accedono mediante la stessa istruzione a locazioni di memoria contigue. Questa tecnica si avvantaggia del fatto che thread in un warp eseguono in qualsiasi momento la stessa istruzione. In questo modo l'hardware combina tutti questi accessi in uno unico attraverso il quale richiede la zona di memoria interessata.
Attraverso gli accessi coalescenti si permette alla DRAM di fornire dati ad una velocità prossima alla massima banda consentita dalla memoria globale.
Solitamente gli accessi in memoria non vengono raggruppati per tutti i 32 thread del warp, ma si ragiona in termini di $half-warp$, ossia in gruppi di 16 thread. A seconda della compute capability del dispositivo, cambiano anche le linee guida da seguire nell'organizzare gli accessi di un half warp per ottenere la coalescenza.

Per i chip grafici con compute capability 1.0 o 1.1 gli accessi di un half-warp sono coalescenti se leggono un'area contigua di memoria di:

- 64 byte – ogni thread legge una word: int, float, ...
- 128 byte – ogni thread legge una double-word: int2, float2, ...
- 256 byte – ogni thread legge una quad-word: int4, float4, ...

In più, devono essere rispettate le seguenti restrizioni:

- L'indirizzo iniziale di una regione deve essere multiplo della grandezza della regione
- Il k-esimo thread di un half-warp deve accedere al k-esimo elemento di un blocco (letto o scritto), gli accessi devono cioè essere perfettamente allineati tra i thread

Un'eccezione alle rigide regole presentate sta nel fatto che gli accessi rimangono coalescenti anche se alcuni thread non partecipano (cioè non eseguono la lettura o la scrittura dei dati).

In figura 2.4 vediamo due accessi che si incrociano (a sinistra) e accessi che partono dall'indirizzo 132 invece che 128 (a destra). In entrambi i casi non sono

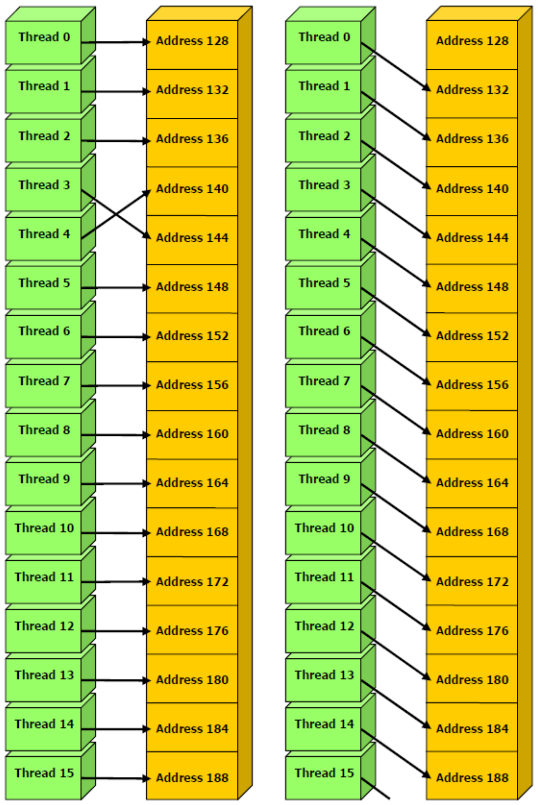


Figura 2.4: Accessi non coalescenti per device con compute capability 1.0/1.1.

rispettate le condizioni per la coalescenza degli accessi dal punto di vista dell'allineamento: nel primo caso, non c'è una corrispondenza sequenziale tra gli accessi, nel secondo non si parte da un multiplo della granularità della transazione (64 byte). L'esecuzione di accessi non coalescenti, con architetture che hanno compute capability 1.0 o 1.1, comporta l'esecuzione di ben 16 transazioni di memoria invece che 1.

La figura 2.5 presenta invece due casi di accessi coalescenti. A sinistra, vediamo 16 accessi a 32 bit, che partono dall'indirizzo 128, con corrispondenza uno a uno. A destra, la situazione è simile, ma alcuni thread non eseguono accessi.

Nel caso di compute capability 1.2 e 1.3, le condizioni per ottenere la coalescenza degli accessi sono fortunatamente un po' meno rigide. In particolare, una singola transazione di memoria è eseguita per un half warp se gli accessi di

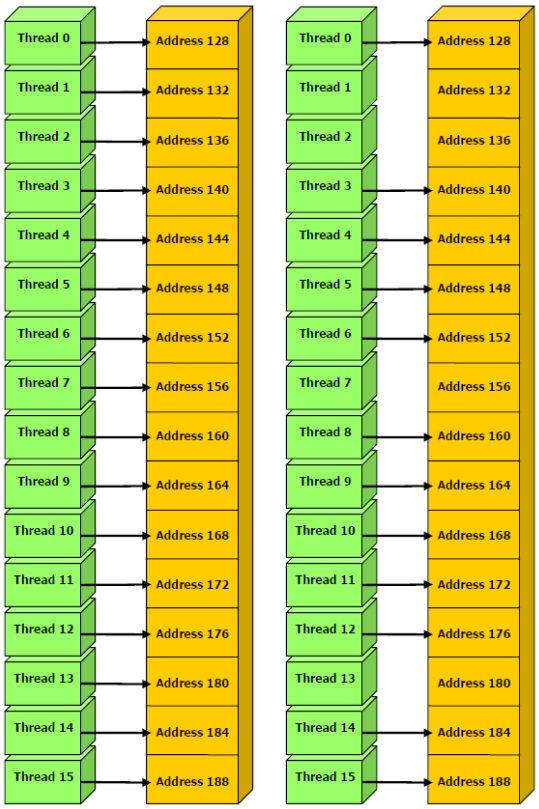


Figura 2.5: Accessi coalescenti per device con compute capability 1.0/1.1.

tutti i thread sono compresi all'interno dello stesso segmento di grandezza uguale a:

- 32 byte, se tutti i thread accedono a word di 8 bit;
- 64 byte, se tutti i thread accedono a word di 16 bit;
- 128 byte, se tutti i thread accedono word di 32 o 64 bit.

La coalescenza degli accessi in una singola transazione, se essi risiedono nello stesso segmento come specificato, è ottenuta per tutti gli schemi di indirizzi richiesti dall'half-warp, inclusi anche schemi dove più thread accedono allo stesso indirizzo. Se, invece, un half-warp indirizza parole in segmenti differenti, saranno eseguite tante transazioni quanti sono i segmenti indirizzati.

La figura 2.6 mostra la situazione nel caso degli accessi in memoria globale con schede dotate di chip con compute capability 1.2 e 1.3. Gli accessi sono a

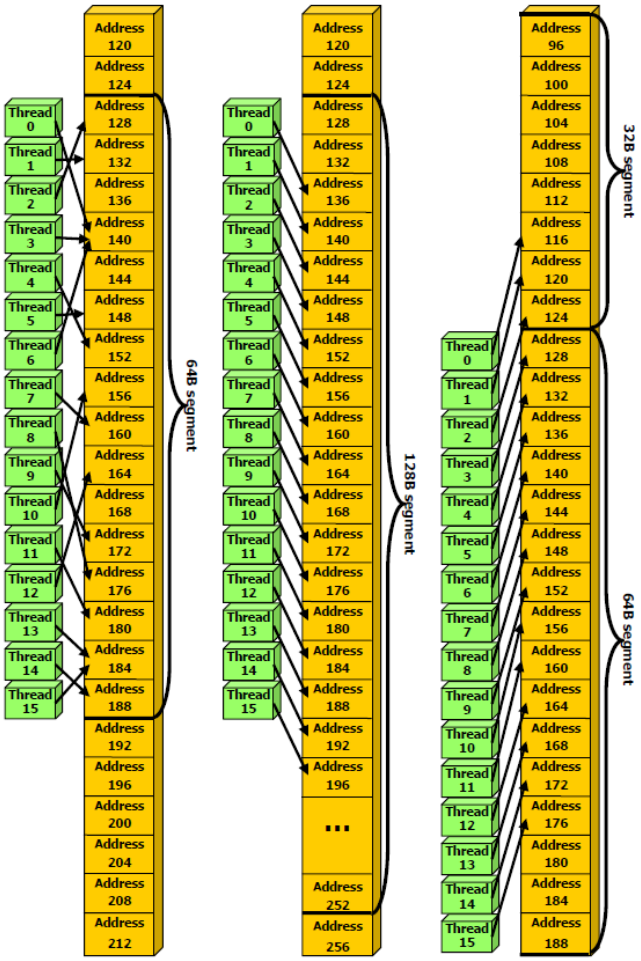


Figura 2.6: Accessi coalescenti per device con compute capability 1.2/1.3.

tutti su dati di 32 bit ciascuno. Nel primo caso, vediamo una serie di accessi, non necessariamente 1 a 1, che però rientrano all'interno dello stesso blocco a 64 byte. Nel secondo caso, vediamo degli accessi, non allineati rispetto all'inizio del blocco, che però risiedono tutti all'interno dello stesso blocco da 128 byte. Nel terzo caso, infine, la transazione è allineata partendo prima dell'indirizzo 128, e data la grandezza dati di 32 bit, sarebbero richieste 2 transazioni da 128 byte. Per la parte prima dell'indirizzo 128, però, i dati si trovano nei 64 byte bassi del segmento, e nei rispettivi 32 byte bassi di quest'ultimo (partono dopo l'indirizzo 96). Dunque, viene eseguita solo una transazione a 32 byte. Per la parte di dati collocati dopo l'indirizzo 128, abbiamo i dati tutti contenuti prima dell'indiriz-

zo 192, dunque compresi nei 64 byte alti. Di conseguenza, viene lanciata una transazione a 64 byte.

Purtroppo la scheda che è stata utilizzata per questo lavoro di tesi ha compute capability 1.1, quindi l'ultimo schema di accessi presentato non è stato possibile utilizzarlo. Si è quindi cercato di allineare il più possibile gli accessi anche se non sempre questo è stato possibile. Alcuni accessi però presentano dei pattern che con schede più avanzate possono essere accorpati in accessi coalescenti, quindi ci si aspetta che le performance aumentino con questo tipo di schede.

2.5.3 Utilizzo della memoria

Uno dei punti fondamentali da capire per realizzare dei programmi CUDA con performance soddisfacenti è che non tutte le memorie sono uguali (come spiegato nella sezione 2.3) ma bisogna cercare di utilizzare al meglio ogni tipo di memoria presente. Il punto cruciale consiste nel minimizzare gli accessi alla memoria globale sfruttando il più possibile la memoria shared.
La tecnica solitamente utilizzata per effettuare questa operazione consiste nel suddividere il dominio/codominio del problema in modo tale da permettere ad un blocco di thread di poter effettuare le sue elaborazioni in un sottoinsieme chiuso di dati. In questo modo i thread afferenti al blocco interessato collaboreranno per caricare in memoria shared la zona di memoria globale da elaborare, per poi procedere sfruttando la maggiore velocità di questa zona di memoria.
I passi fondamentali quindi per ogni thread saranno:

- caricare i dati dalla memoria globale alla memoria shared;
- sincronizzare tutti i thread del blocco in modo tale che ognuno possa leggere con sicurezza le posizioni della shared memory riempite da altri thread;
- elaborare i dati della memoria condivisa;
- effettuare una nuova sincronizzazione se necessario per assicurarsi che la memoria shared sia stata aggiornata con i risultati;
- scrivere i risultati nella memoria globale.

Per sfruttare questa tecnica spesso sono necessarie sostanziali modifiche al codice e una rielaborazione quasi completa degli algoritmi progettati precedentemente, ma tutto ciò viene ripagato da performance notevolmente migliori rispetto a quelle dell'algoritmo standard.
Per comprendere meglio questa tecnica riportiamo un esempio presente in [18] che può chiarire questo metodo, basato sul prodotto tra matrici.

La figura 2.7 mostra il prodotto tra matrici standard, dove per calcolare ogni

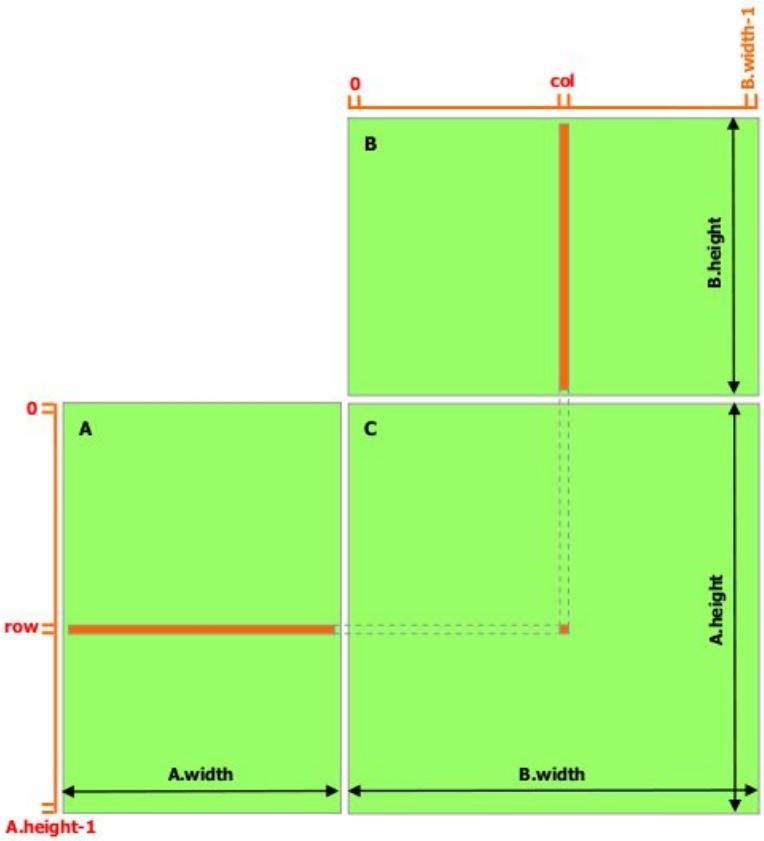


Figura 2.7: Prodotto tra matrici standard.

elemento è necessario caricare dalla memoria una riga e una colonna delle matrici di input. Se ad ogni thread venisse affidato il compito di calcolare un elemento della matrice e si organizzasse l'algoritmo in questo modo è evidente che gli accessi in memoria dominerebbero il tempo di esecuzione dell'algoritmo.
Quello che si può fare è affidare ad un blocco di thread il compito di calcolare una sottomatrice della matrice di output, così da poter riutilizzare i dati caricati

dalla memoria globale e far collaborare i thread per minimizzare gli accessi in memoria per ognuno di essi.

La figura 2.8 mostra questo procedimento. Dividendo le matrici di input in sot-

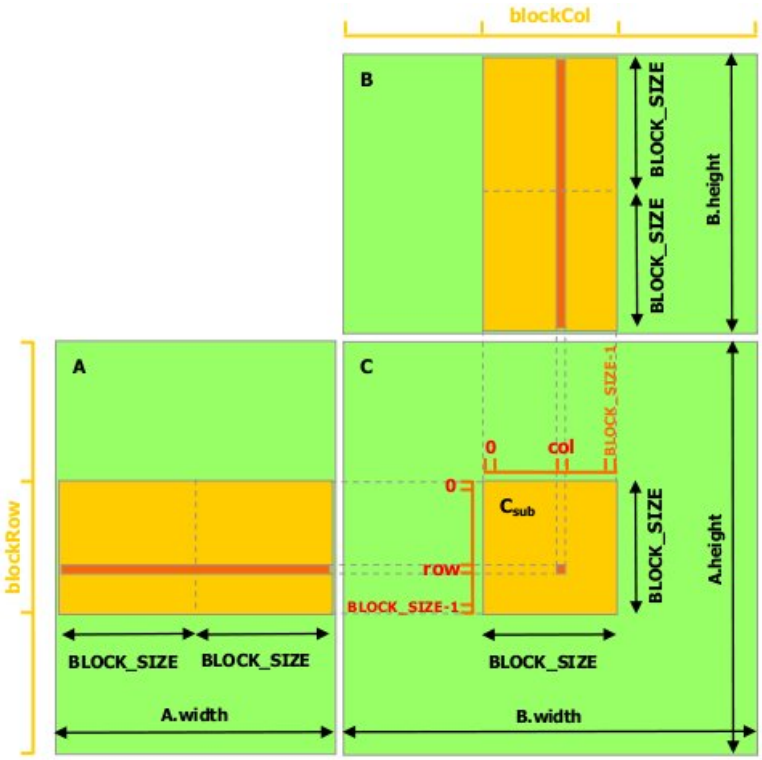


Figura 2.8: Prodotto tra matrici sfruttando la memoria condivisa.

tomatrici della dimensione *block_size* ogni thread del blocco dovrà caricare dalla memoria globale solo $\frac{A_{width}}{block\_size} + \frac{B_{height}}{block\_size}$ valori rispetto ai $A_{width} + B_{height}$ valori necessari nell'algoritmo standard. Oltre a questo i valori caricati verranno riutilizzati anche da altri thread riducendo così gli accessi in memoria dell'algoritmo complessivo da $(A_{width} \cdot B_{height}) \cdot (A_{height} \cdot B_{width})$ a $\frac{A_{height} \cdot B_{width}}{block\_size^2} \cdot \frac{A_{width}}{block\_size} \cdot \frac{B_{height}}{block\_size}$. Nell'esempio sono state utilizzate matrici di esempio con dimensioni suddivisibili per *block_size*, ma nella realtà questo dipende dall'input del problema. In ogni caso è sufficiente inserire dei controlli aggiuntivi all'interno del kernel per ovviare a questo problema.

2.5.4 Granularità dei thread

Un'importante decisione algoritmica che influenzerà le performance del programma è la scelta della granularità dei thread. A volte, infatti, risulta vantaggioso usare un numero inferiore di thread ma assegnare ad ognuno di essi più compiti da svolgere. Questo vantaggio deriva dal fatto che a volte esistono delle operazioni ridondanti tra i thread che possono essere riunificate.
Nell'algoritmo mostrato nel paragrafo precedente, per esempio, questa ridondanza è data dal fatto che due blocchi di thread adiacenti caricheranno le stesse righe di A. Questo è un chiaro esempio di ridondanza che può essere eliminata unificando i blocchi e assegnando ad ognuno di essi il compito di elaborare più elementi della matrice di input.
Risulta evidente che estendendo questo ragionamenti al limite si arriverebbe all'algoritmo seriale facendo svolgere tutto il lavoro ad un singolo thread, ma il punto da capire è proprio questo. Bisogna avere la sensibilità, anche attraverso prove pratiche, di capire il giusto trade off tra grado di parallelismo da assegnare all'algoritmo e quantità di lavoro da includere in ogni singolo thread. Inoltre alcuni limiti sono dovuti proprio dalle risorse che si hanno a disposizione, infatti la memoria shared ha una dimensione ridotta, perciò questo potrebbe essere un primo indicatore della granularità dei thread necessaria.

2.6 Sincronizazzione

Nella realizzazione di software paralleli, uno dei punti cruciali consiste nella sincronizzazione tra i vari processori. In CUDA i processori che concorrono all'esecuzione del programma possono essere visti come i vari thread che vengono lanciati.
Una delle limitazioni di CUDA consiste proprio nel fatto che i thread non possono comunicare tra loro, quindi nè scambiarsi messaggi nè sincronizzarsi, se non sfruttando attraverso qualche meccanismo (comunque poco efficiente e da evitare) la memoria globale.
In generale i thread appartenenti a blocchi diversi non hanno alcun sistema di sincronizzazione, mentre per i thread appartenenti allo stesso blocco esiste

un sistema per garantire che l'esecuzione del kernel si fermi in un punto preciso e continui quando tutti i thread hanno raggiunti quel punto. Consiste in una sorta di barriera di esecuzione che viene richiamata attraverso la primitiva *__syncthreads*(). Nella sezione 2.5.3 è stata descritta la situazione tipica in cui ci si avvale di questa tecnica. E' importante capire che non è possibile associare qualche tipo di controllo aggiuntivo a questa primitiva (per utilizzarla ad esempio come semaforo), in quanto non rappresenta niente di più che un punto di controllo del flusso del codice.

Questa caratteristica delle CUDA è dovuta proprio alla sua architettura e alla sua modalità di esecuzione, in quanto thread di blocchi differenti potrebbero venire eseguiti da Streaming Multiprocessor differenti con scheduler diversi che quindi non hanno a disposizione nessun meccanismo fisico condiviso (esclusa la memoria globale) con cui controllare e sincronizzare eventualmente più thread sulla base di variabili utilizzate dal programma.
Nella progettazione di software questa caratteristica va tenuta bene a mente, ed eventualmente vanno trovate tecniche per aggirare questa limitazione. Un'alternativa può essere suddividere l'esecuzione del programma in blocchi diversi che devono attendere il completamento del blocco precedente per poter avviare l'esecuzione. Ogni blocco in questo caso sarà rappresentato da un kernel a se stante che quindi potrà essere progettato indipendentemente garantendo la massima parallelizzazione possibile ed una gestione personalizzata della memoria necessaria.

Capitolo 3

Approccio preliminare

Nella fase iniziale di questo lavoro di tesi si è andati ad analizzare e studiare l'utilizzo delle librerie CUBLAS [16] per migliorare le performance del software in esame.
Le cuBLAS (CUDA Basic Linear Algebra Subprograms) sono librerie che si trovano in cima al driver NVIDIA CUDA e possono essere utilizzate senza un'interazione diretta con le CUDA vere e proprie.
Lo scopo di queste librerie è quello di fornire al programmatore delle funzioni di algebra lineare che, venendo eseguite nella GPU, sfruttano l'iperparallelismo che la scheda grafica mette a disposizione per eseguire le operazioni con un grande risparmio di tempo.

3.1 Modello di programmazione CUBLAS

Il modello di programmazione cublas si basa fondamentalmente su tre step principali:

– creazione delle strutture dati in GPU (matrici o vettori). Per effettuare questa operazione è necessario prima di tutto allocare in GPU lo spazio necessario a contenere i dati attraverso l'istruzione cublasAlloc(), e poi sfruttare questo spazio per copiare i dati da CPU in una vettore o in una matrice attraverso le istruzioni cublasSetVector() e cublasSetMatrix();

- Modifica dei dati caricati sul device attraverso le funzioni messe a disposizione dalle CUBLAS. Queste si dividono in BLAS di livello 1,2,3, a loro volta suddivise in funzioni a singola precisione (operanti su float), doppia precisione (operanti su double) e funzioni su numeri complessi;

- Aggiornamento dei dati in CPU attraverso le primitive cublasGetVector() e cublasGetMatrix() e deallocazione dello spazio in GPU attraverso la primitiva cublasFree().

Inoltre è necessario avviare le cublas attraverso cublasInit() prima di utilizzare qualsiasi operazione CUBLAS e poi spegnerle alla fine con cublasShutdown().

Un esempio di semplice utilizzo delle funzioni che sono state precedentemente presentate si trova in questo listato:

```
float* devPtrA;
float* a = 0;
a = (float *)malloc (M * N * sizeof (*a));
cublasInit();
cublasAlloc (M*N, sizeof(*a), (void**)&devPtrA);
cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
/*Operazioni in GPU*/
cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
cublasFree (devPtrA);
cublasShutdown();
```

CUBLAS inoltre è dotato di un sistema per recuperare e comprendere gli errori che avvengono in GPU durante l'esecuzione delle operazioni. Ogni funzione CUBLAS restituisce un tipo di dato cublasStatus che descrive come è avvenuta l'operazione, ottenibile anche attraverso la funzione cublasGetError().

3.2 Benchmark

Per valutare l'effettivo vantaggio nell'utilizzo delle cublas, sulla macchina utilizzata per sperimentarle è stata effettuato un benchmark sul prodotto tra matrici quadrate e rettangolari confrontando i tempi di esecuzione del prodotto standard, sfruttando i tipi di dato opencv e il prodotto in cublas.

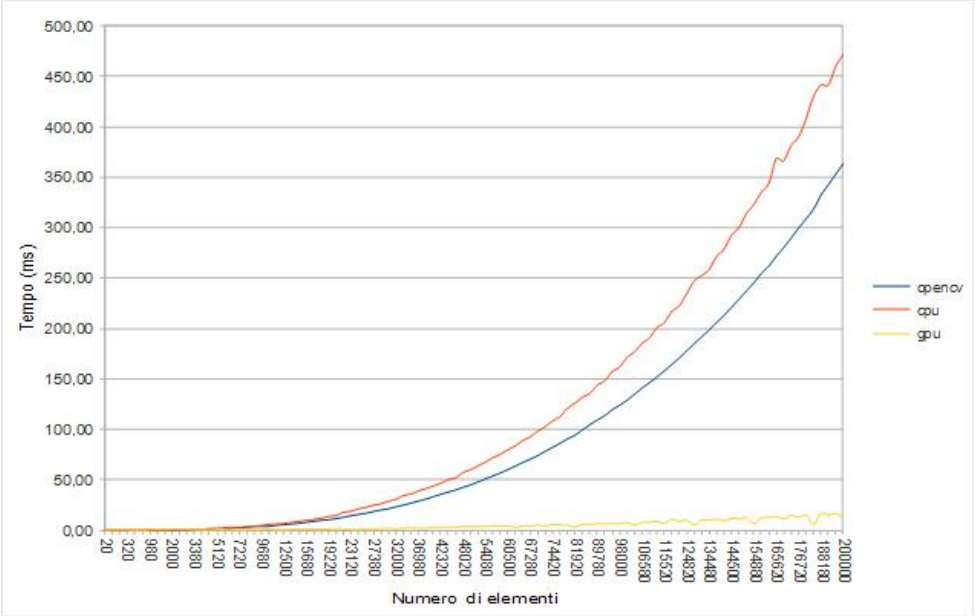


Figura 3.1: Tempi di esecuzione del prodotto tra matrici rettangolari.

Le figure 3.1 e 3.2 riassumono i risultati ottenuti.

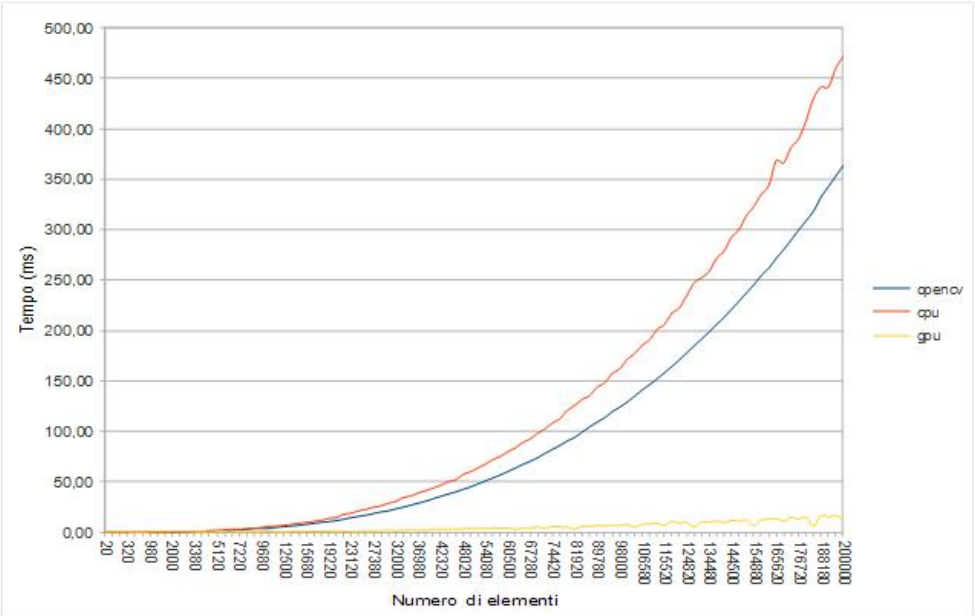


Figura 3.2: Tempi di esecuzione del prodotto tra matrici quadrate.

I grafici evidenziano che cublas ha prestazioni eccezionali su matrici di grandi dimensioni, mentre i suoi vantaggi non sono così evidenti per matrici di dimensioni ridotte. Comunque le prestazioni elevate su un gran numero di dati ne giustificano la sperimentazione per l'ottimizzazione di software che hanno necessità di effettuare operazioni di algebra lineare sui dati.

3.3 Utilizzo cublas

Una volta verificati i vantaggi ottenibili con CUBLAS si è passati ad analizzare il software per verificare i punti in cui è necessario intervenire, ovvero quelle operazioni più complesse e che comportano i tempi di calcolo maggiori. Queste saranno le operazioni in cui si effettuerà una riscrittura del codice per adattarlo alle cublas.
L'analisi temporale dei tempi di esecuzione ha riportato diversi punti in cui attraverso cublas dovrebbe essere possibile migliorarne le prestazioni. Si è quindi passati alla riscrittura del codice in cui sono stati incontrati i primi problemi.
Innanzi tutto cublas è stato realizzato per interfacciarsi perfettamente con il Foltran, che ha una memorizzazione dei dati in column major, e quindi adotta la stessa tecnica. Nel manuale viene fornita una macro per ovviare a questo problema nell'accesso ai dati ma per matrici già costruite (ovvero intervenendo a programma già avviato) non è possibile utilizzarla. Si è quindi deciso di effettuare il calcolo delle matrici coinvolte effettuandole la trasposta "al volo" su GPU (un parametro apposito delle funzioni di algebra lineare permette di farlo).
Inoltre CUBLAS non mette a disposizione istruzioni condizionali per operare sui dati in GPU, ma permette solo di eseguire operazioni atomiche restituendo subito il controllo alla CPU. Questo è uno svantaggio perchè il programma in esame non presenta situazioni in cui è necessario effettuare operazioni su un grande numero di dati contemporaneamente, ma effettua ciclicamente operazioni su porzioni di dati al termine delle quali effettua delle operazioni condizionate al verificarsi di determinate circostanze.
Questi vincoli rendono la programmazione complicata e non permettono di sfruttare appieno le potenzialità dello strumento che si ha a disposizione.
I primi test sui tempi di esecuzione di quelle zone di codice che per prime sono

state reimplementate hanno evidenziato infatti che sfruttando queste librerie le performance del software non sono aumentate, anzi sono diminuite rendendolo più lento.
Questi fatti hanno portato alla conclusione che le cublas non sono sufficienti come strumento per le operazioni richieste dal software, ma si ha la necessità di poter operare più liberamente sul codice in GPU gestendo direttamente il grado di parallelismo utilizzato. E' stato quindi deciso di passare all'utilizzo delle CUDA, scendendo di livello nell'architettura di queste librerie.

Capitolo 4

Analisi e implementazione

In questo capitolo verrà presentato il problema che ha costituito il core di questo lavoro di tesi, ovvero il calcolo della matrice pseudoinversa. Questo problema di carattere generale verrà affrontato tenendo ben presenti le caratteristiche strutturali della matrice di partenza, da sfruttare per realizzare algoritmi più specifici e con performance migliori.

4.1 Matrice Pseudoinversa

Una matrice pseudo-inversa è la generalizzazione della matrice inversa al caso in cui la matrice da invertire non sia quadrata.
In particolare, data la matrice rettangolare A di dimensioni $(n \times m)$ con $n \geq m$, la corrispondente matrice pseudoinversa è la matrice data da:

$$A^{+} = (A^{T} \cdot A)^{-1} \cdot A^{T} \tag{4.1}$$

di dimensioni $(m \times n)$, dove il + indica la matrice pseudoinversa.
L'equazione 4.1 mostra la tecnica standard per il calcolo della matrice pseudoinversa. Nell'analisi di partenza (utilizzando come riferimenti [8]) oltre a questo metodo sono state prese in considerazione altre due possibilità di calcolo, ovvero:

- utilizzando la SVD (*Singular Value Decomposition*);
- sfruttando la *decomposizione QR*.

4.1.1 Single Value Decomposition

In algebra lineare, la decomposizione ai valori singolari (o SVD, Singular Value Decomposition) è una particolare fattorizzazione basata sull'uso di autovalori e autovettori di una matrice a valori reali o complessi.
Data la matrice A di dimensioni $n \times m$ con valori nel campo reale, essa può essere decomposta attraverso questa tecnica nel seguente modo:

$$A = U \cdot S \cdot V^T \tag{4.2}$$

dove le matrici ottenute hanno le seguenti caratteristiche:

- U è una matrice unitaria di dimensioni $n \times n$;
- S è una matrice diagonale di dimensioni $n \times m$, con la caratteristica che i valori degli elementi nelle diagonali s_i sono tali che $s_1 \geq s_2 \geq ... \geq s_m \geq 0$;
- V^T è la trasposta di una matrice unitaria di dimensioni $m \times m$.

Per la costruzione delle matrici si fruttano gli autovalori e gli autovettori del prodotto $A \cdot A^T$ e $A^T \cdot A$.
In particolare gli autovalori del prodotto $A \cdot A^T$ costituiranno il quadrato dei valori s_i della matrice S mentre gli autovettori corrisponderanno alle colonne della matrice U. Invece del prodotto $A^T \cdot A$ verranno considerati solo gli autovettori che costituiranno le righe della matrice V^T. Una volta ottenute la decomposizione di A, la pseudoinversa A^+ è uguale a:

$$A^+ = V \cdot S^+ \cdot U^T \tag{4.3}$$

dove S^+ viene calcolata a partire dalla matrice S considerando i reciproci dei valori sulla diagonale.

4.1.2 Decomposizione QR

Un metodo alternativo per calcolare la matrice pseudoinversa sfrutta la decomposizione QR per facilitare il calcolo nella formula base del calcolo della matrice pseudoinversa.
Sia A una matrice $n \times m$, attraverso questa tecnica viene decomposta in:

$$A = Q \cdot R \tag{4.4}$$

dove:

- Q è una matrice unitaria di dimensioni $n \times n$.
- R è una matrice triangolare superiore di dimensioni $n \times m$, dove le $n - m$ righe inferiori sono nulle.

Ci sono diversi metodi per calcolare la decomposizione QR di una matrice come il Gram–Schmidt process, Householder transformations, o Givens rotations.
Ognuno di queste tecniche presenta un certo numero di vantaggi e svantaggi, che si possono osservare entrando nei dettagli implementativi di queste tecniche. In ogni caso ognuno dei metodi nominati agisce in modo iterativo sulla matrice di partenza apportando ognuno delle modifiche diverse. La costante è data dal fatto che in ognuno dei casi non è possibile tenere conto nel calcolo di pattern di sparsità particolari come quello della matrice in esame.

Una volta ottenuta la decomposizione QR della matrice A, la si può sfruttare in questo modo:

$$A^T \cdot A = (Q \cdot R)^T \cdot (Q \cdot R) = R^T \cdot Q^T \cdot Q \cdot R = R^T \cdot R \tag{4.5}$$

Una volta ottenuta la matrice $R^T \cdot R$ per calcolarne l'inversa può essere sfruttata la tecnica di forward and back substitution per poi effettuare il prodotto tra la matrice risultante con A^T.

4.2 Matrice di input

La matrice che costituisce l'input per il calcolo della matrice pseudoinversa è una matrice molto sparsa, con un pattern particolare da sfruttare per cercare di ottenere prestazioni superiori rispetto al normale calcolo per matrici generiche.
La nostra matrice, infatti, è composta da m colonne, di cui però solo la prima ha tutti gli elementi valorizzati mentre le altre possiedono valori solo per un sottoinsieme continuo di posizioni detto *blocco*, il cui totale corrisponde ad una colonna completamente riempita. La matrice di riferimento, quindi, è affiancata

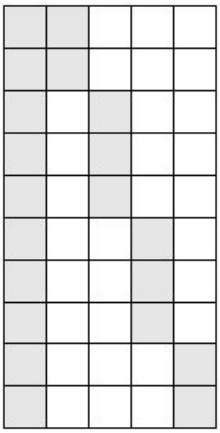


Figura 4.1: Struttura della matrice di input.

da due array di supporto dove vengono memorizzati gli indici di partenza e la dimensione di ogni blocco facente riferimento ad una colonna della matrice di input.

Osservando la struttura dell'input, si osserva quindi che in realtà la matrice di dimensione $n \times m$ possiede solo $2n$ valori, particolarità da tenere conto nei vari calcoli e nelle scelte progettuali.

4.3 Scelte implementative

Una volta comprese le varie alternative per il calcolo della matrice pseudoinversa e presa coscienza della struttura della matrice di input, si può passare a definire i passi matematici che verranno poi implementati in forma parallela in CUDA e che verranno presentati successivamente.

Innanzi tutto per il calcolo della pseudoinversa si è deciso di utilizzare la tecnica diretta standard data dalla definizione di pseudoinversa, escludendo le altre tecniche per i seguenti motivi:

– *Single Value Decomposition*: questo metodo presenta una complessità computazionale superiore rispetto agli altri presentati. In particolare la complessità prevalente è data dal calcolo degli autovalori e autovettori. In-

oltre tranne che per la prima fase, ovvero i prodotti $A \cdot A^T$ e $A^T \cdot A$, non è possibile sfruttare il pattern di sparsità della matrice.

- *Decomposizione QR*: questo metodo presenta il problema di fondo di essere scarsamente parallelizzabile, in quanto tutte e tre le tecniche di scomposizione presentano delle caratteristiche di iteratività, quindi difficilmente applicabili all'ambito parallelo. Sono state studiate delle tecniche parallele per implementare i corrispondenti algoritmi, ma con risultati non molto apprezzabili. Inoltre nel calcolo non è possibile tenere conto del pattern di sparsità della matrice di input.

I passi matematici da implementare saranno quindi quelli descritti dall'equazione 4.1 che ora verranno analizzati nel dettaglio.

4.3.1 Preambolo sulla struttura dei kernel

Prima di spiegare nel dettaglio come sono stati risolti e implementati gli algoritmi necessari al calcolo della matrice pseudoinversa è necessario spiegare come è stata gestita la taglia dell'input con riferimento alla struttura dei kernel. Infatti un problema che si è andati ad affrontare riguarda la scalabilità del software per taglie del problema elevate. Per esempio inserendo come parametro dimensionale del blocco di thread m, ovvero una delle dimensioni dell'input, quando questo supera i limiti imposti dall'architettura (per la scheda utilizzata 768) si verificherà un errore di tipo *invalid argument*.
Per risolvere questo problema nei kernel che presenteremo successivamente saranno utilizzati i seguenti parametri così valorizzati:

- $MAX\_BLOCK = 256$
- $MAX\_COLS = 256$
- $CUDA\_BLOCK\_SIZE\_1D = 256$

All'interno del kernel quindi ogni thread, nel caso di taglia superiore al valore del parametro, svolgerà la funzione di più thread eseguiti in maniera seriale. Questo può comportare un deterioramento delle performance ma, attraverso dei

test, è stato verificato che modificando il valore di questi parametri i tempi di esecuzione dei vari algoritmi non cambiano perciò sono stati tenuti questi valori come riferimento.
Per far svolgere ad un thread il compito di più thread eseguiti serialmente in tutti i kernel si trova il seguente codice:

```
int thx = threadIdx.x;
int iter=(TAGLIA + MAX_BLOCK - 1) / MAX_BLOCK;
int thx_mem;
for(int j=0; j<iter; j++)
{
  thx_mem = thx + j * MAX_BLOCK;
  if(thx_mem<TAGLIA)
  {
    /*codice da eseguire*/
  }
}
```

4.3.2 Prodotto della matrice di input trasposta per se stessa

Il primo passo del calcolo della matrice pseudoinversa consiste nell'effettuare il prodotto $A^T \cdot A$. Osservando nel dettaglio come viene sviluppato questo prodotto si osserva che la matrice risultante ha una struttura simmetrica e molto sparsa, come si può vedere in figura 4.2. Nel calcolo si vede facilmente che solo per

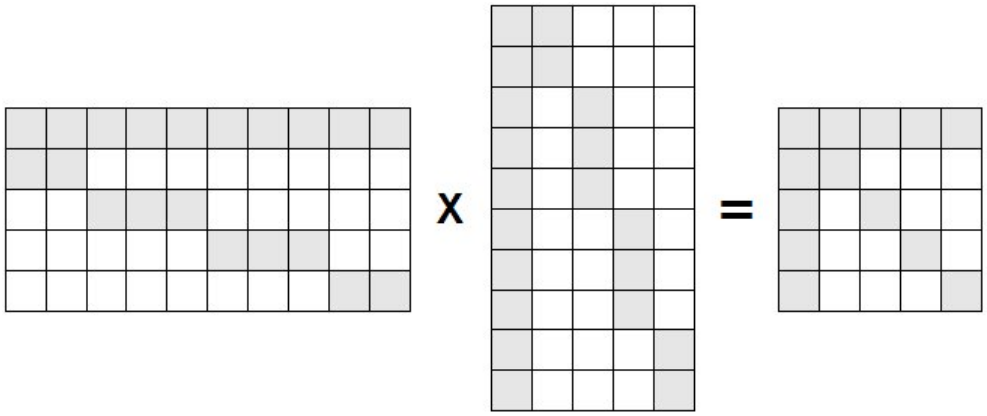


Figura 4.2: Prodotto tra la trasposta della matrice di input e se stessa.

l'elemento in posizione $(0, 0)$ sono necessari n prodotti e somme, mentre per gli

altri elementi sono sufficienti in media $\frac{n}{m}$ operazioni, per cui considerando che gli elementi della prima riga e della prima colonna della matrice risultante sono uguali, la complessità computazionale sarà pari a:

$$T = 2 \cdot n + 2 \cdot (m-1) \cdot \frac{n}{m} = O(n) \tag{4.6}$$

Un tempo molto inferiore a quello del calcolo normale che comporterebbe $O(n \cdot m^2)$ operazioni.

Analisi del parallelismo e algoritmo

Questo algoritmo presenta un grado di parallelismo dato dal numero di elementi da calcolare.
Teoricamente si potrebbe assegnare ad ogni thread il compito di calcolare un singolo valore della matrice, ottenendo così un grado di parallelismo pari a $2m-1$ ma, dallo studio delle tecniche di ottimizzazione, in particolare per minimizzare la quantità di dati caricati dalla memoria globale, si è deciso di assegnare ad ogni blocco di thread il compito di calcolare il valore di tre elementi (in realtà due grazie alla simmetria della matrice), così come evidenziato in figura 4.3.
In questo modo viene ridotta al minimo la quantità di dati caricati dalla memoria

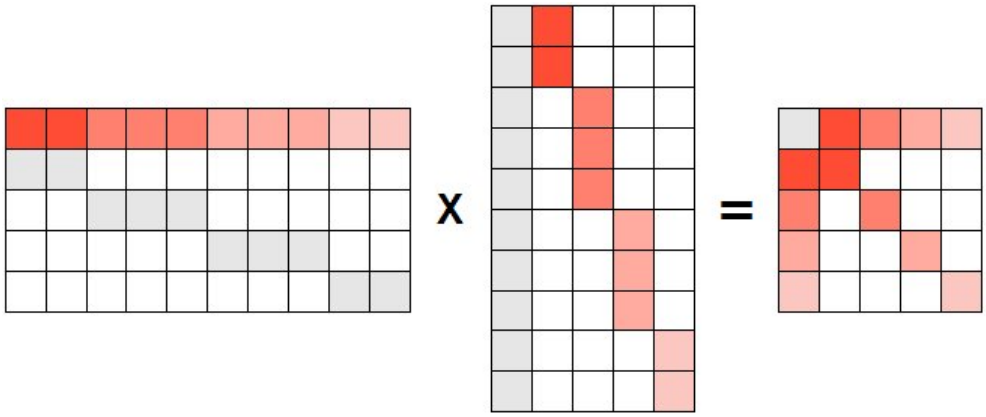


Figura 4.3: Tecnica di calcolo per il prodotto tra la matrice di input e la sua trasposta.

globale. Da come viene evidenziato in figura rimane fuori dal calcolo l'elemento in posizione (0,0). Grazie allo schema appena presentato, però, ogni blocco carica in memoria condivisa i dati necessari per calcolare anche una parte di questo

elemento.
Ciascun blocco, quindi, calcola la propria porzione del primo elemento, e ne salva il risultato parziale in memoria globale.
La struttura della griglia utilizzata per il kernel principale che si occupa del calcolo è la seguente:

- GRIGLIA: monodimensionale con $m - 1$ blocchi
- BLOCCO: monodimensionale con MAX_BLOCK thread

Ogni blocco si occupa del calcolo degli elementi analizzati precedentemente, in cui tutti i thread concorreranno nel caricamento dei dati dalla memoria per ottimizzare questa operazione (accessi coalescenti), mentre solo il thread con id=0 effettua il calcolo vero e proprio.

A questo punto dell'algoritmo rimane da effettuare il calcolo dell'elemento in posizione (0,0), i cui contributi dei blocchi eseguiti con il kernel precedente sono stati salvati in memoria globale. Per farlo si utilizza un kernel con la seguente struttura:

- GRIGLIA: monodimensionale con 1 blocco
- BLOCCO: monodimensionale con MAX_COLS thread

Il funzionamento di questo kernel è molto semplice, ovvero tutti i thread del blocco concorrono per caricare dalla memoria globale con accessi coalescenti i contributi al calcolo del primo elemento ottenuti con il kernel precedente. Successivamente solo il thread con $id = 0$ si occuperà di sommare i vari contributi e salvare il risultato in memoria globale.

In questo modo la complessità computazionale dell'algoritmo è data dal calcolo del singolo elemento della matrice (escluso il primo) e dalla somma dei contributi per il calcolo del primo elemento, diventando quindi $O(\frac{n}{m} + m)$.

Risorse utilizzate

Con le formule presentate nella sezione 2.3.7 andiamo ora a calcolare le risorse utilizzate dai due kernel che compongono il primo algoritmo.

Analizzando il primo kernel, detto $deviceMulJacTJac$, l'output del compilatore specifica:

```
ptxas info : Compiling entry function '_Z16deviceMulJacTJacPfPiS0_iiS_' for 'sm_10'
ptxas info : Used 11 registers, 3120+16 bytes smem, 12 bytes cmem[1], 4 bytes cmem[14]
```

Dalla struttura del kernel si ha che si ha che il numero di warp in un blocco è pari a $W_{block} = \frac{256}{32} = 8$, quindi il numero di registri allocati ad ogni blocco è pari a $R_{block} = ceil(8 \cdot 32 \cdot 11, 256) = 3328$. Considerando che ad ogni multiprocessore verranno assegnati al massimo $B_{SM} = \frac{768}{256} = 3$ blocchi si può concludere che $B_{SM} \cdot R_{block} = 9984 > 8k$ quindi ci sarà un'inefficienza nell'esecuzione su schede di compute capability 1.0 e 1.1 in quanto solo due blocchi su tre potranno essere assegnati ad ogni Streaming Multiprocessor.
Per quanto riguarda i ragionamenti riguardanti la shared memory, ad ogni blocco ne vengono assegnati $S_{block} = ceil(3136, 512) = 3584$ byte. Avendo ogni SM una disponibilità di 16KB di memoria condivisa e potendoci assegnare un massimo di tre blocchi contemporaneamente, si ha che $S_{block} \cdot B_{SM} = 10752B < 16KB$ e quindi sono stati rispettati i vincoli legati a questo fattore.

Analizzando il secondo kernel, detto $deviceSumFirstEl$, l'output del compilatore specifica:

```
ptxas info : Compiling entry function '_Z16deviceSumFirstElPfi' for 'sm_10'
ptxas info : Used 8 registers, 1040+16 bytes smem, 12 bytes cmem[1], 4 bytes cmem[14]
```

Dalla struttura del kernel si ha che il numero di warp in un blocco è pari a $W_{block} = \frac{256}{32} = 8$, quindi il numero di registri allocati ad ogni blocco è pari a $R_{block} = ceil(8 \cdot 32 \cdot 8, 256) = 2048 < 8K$. Considerando che questo kernel è costituito da solo un blocco di thread questo vincolo è rispettato.
Per quanto riguarda i ragionamenti riguardanti la shared memory, ad ogni blocco ne vengono assegnati $S_{block} = ceil(1056, 512) = 1536B < 16KB$ byte. Essendoci un solo blocco di thread anche questo vincolo è rispettato.

4.3.3 Calcolo della matrice inversa

Una volta effettuato il calcolo presentato al passo precedente, è necessario calcolare la matrice inversa della matrice $m \times m$ risultante.

Il metodo utilizzato per il calcolo è quello della $Gauss - Jordan\ elimination$. Questa tecnica si basa nell'affiancare alla matrice di input una matrice identità di dimensioni $m \times m$, ed effettuare le operazioni necessarie (somme e prodotti tra righe) per riportare la matrice di input stessa nella forma della matrice identità. Una volta ottenuto il risultato si considerano solo le ultime m colonne della matrice totale, che corrisponderà alla matrice inversa cercata.

Questo metodo, per una matrice completamente definita di dimensioni $m \times m$, ha una complessità pari a:

$$\begin{aligned} T &= \sum_{i=0}^{m-1}(2(m-i) + 2(m-i)\cdot m) = 2(m+1)\cdot\sum_{i=0}^{m-1}(m-i) = \\ &= 2(m+1)\cdot\frac{m\cdot(m-1)}{2} = O(m^3) \end{aligned} \tag{4.7}$$

Ad una prima analisi questo metodo non sembra particolarmente vantaggioso, a causa della complessità computazionale, e parallelizzabile, a causa della natura iterativa del procedimento (a cui è dovuta la sommatoria da 0 a m-1).
In realtà osservando bene il procedimento e la matrice di input si osserva che non sono necessari tutti i passi dell'iterazione per riportare la matrice da elaborare in forma di identità ma solo i seguenti:

1. annullare ogni elemento della prima riga tranne il primo con la seguente operazione: $a_{0,i} = a_{0,i} - \frac{a_{0,i}}{a_{i,i}}$, aggiungendo all'elemento $a_{0,0}$ il valore $-\frac{a_{0,i}^2}{a_{i,i}}$;
2. scalare la prima riga per il valore dell'elemento $a_{0,0}$;
3. sottrarre ad ogni riga la prima scalata del valore $a_{i,0} = a_{0,i}$;
4. scalare ogni riga tranne la prima per il coefficiente della diagonale.

Come si può notare questo algoritmo è notevolmente più semplice del precedente in termini di calcoli da effettuare, con una complessità pari a:

$$T = 2m + m + 2m\cdot(m-1) + m\cdot(m-1) = O(m^2) \tag{4.8}$$

dove non sono presenti fasi iterative.

Analisi del parallelismo e algoritmo

Il metodo presentato si presta ad un'ottima implementazione parallela Per farlo suddividiamo il calcolo tra i primi due punti dell'algoritmo presentato in precedenza e gli ultimi due.
Il calcolo della prima riga, costituito dai primi due punti dell'algoritmo, verrà effettuato da due kernel distinti il cui compito sarà di costruire effettivamente la prima riga per poi scalarla per il valore dell'elemento in posizione (0,0) detto $FIRST\_EL$. Il primo calcolo avrà complessità O(m) in quanto ogni elemento andrà a calcolare una posizione della prima riga per poi, con una procedura iterativa che quindi comporterà la complessità citata, andare a calcolare l'effettivo valore dell'elemento (0,0). L'utilizzo dell'iterazione è giustificato dal fatto che ogni thread che eseguirà il kernel contribuisce a caricare dalla memoria dati che serviranno al calcolo del $FIRST\_EL$, ma mancando meccanismi pre la sincronizzazione più raffinati, bisogna salvare tutti i contributi in un array per poi assegnare ad un solo thread (quello con id=0 in questo caso) il compito di sommare tutti i valori. Il secondo kernel, che permette di scalare la prima riga, avrà una complessità O(1) assegnando ad ogni thread il compito di effettuare il calcolo di un elemento distinto della riga. Entrambi i kernel hanno la stessa struttura di thread, per altro molto semplice, costituita da:

- GRIGLIA: monodimensionale con un blocco
- BLOCCO: monodimensionale con MAX_COLS thread

E' importante osservare che avendo salvato la matrice di input su due righe gli accessi in memoria saranno coalescenti e permetteranno un vantaggio prestazionale.

Per quanto riguarda la seconda parte dell'algoritmo si può estendere il ragionamento fatto al punto precedente all'intera matrice.
Infatti ogni riga e ogni suo elemento necessitano di elaborazioni che non devono essere sincronizzate con nessuna fase precedente dell'algoritmo (se non quella sopra indicata). Quindi possono procedere indipendentemente garantendo un grado di parallelismo pari a $m \cdot (m-1)$.
Per ottenere questo si necessita di una struttura di thread di questo tipo:

- GRIGLIA: monodimensionale con m-1 blocchi (non serve l'elaborazione della prima riga)
- BLOCCO: monodimensionale con MAX_COLS thread

Ogni blocco quindi si occupa di elaborare una riga della matrice di output e ogni suo thread si prenderà carico di un elemento della riga corrispondente. Ogni riga viene quindi elaborata come in precedenza in $O(1)$ e, estendendo il ragionamento a tutte le righe, l'intera elaborazione comporta un tempo costante di elaborazione. E' evidente, per la struttura presentata in precedenza, che questo è un risultato teorico considerando completamente contemporanea l'esecuzione di tutti i thread lanciati nell'esecuzione del kernel. Questo non può essere vero e dipende anche dal tipo di dispositivo che si ha a disposizione per il calcolo. In linea di massima questo ragionamento ci fa capire come CUDA permetta di ottimizzare l'esecuzione di questo algoritmo.

Al fine di ottimizzare l'algoritmo presentato nel paragrafo successivo, il più costoso in termini di tempo di esecuzione, la matrice inversa viene costruita considerandola direttamente trasposta. Questa accortezza, nonostante non comporti variazioni nella progettazione dell'algoritmo, comporta una conseguenza fondamentale: gli accessi che prima venivano effettuati per riga adesso corrisponderanno ad accessi per colonna e quindi, come spiegato nella sezione 2.5.2, non possono essere coalescenti.
Testando l'algoritmo su matrici di input di dimensioni considerevoli è stata osservata una differenza elevata tra i tempi di esecuzione effettuando gli accessi per riga o per colonna. Si è quindi deciso di riprogettare l'algoritmo, aumentando leggermente gli accessi in memoria, ma facendo in modo di operare per riga anche sulla matrice trasposta (cioè per colonna nella matrice normale). Questo ha portato a modificare il terzo kernel presentato in precedenza impostando:

- GRIGLIA: monodimensionale con m blocchi (è necessario elaborare ogni colonna)
- BLOCCO: monodimensionale con MAX_COLS thread

I risultati di questa rielaborazione sono molto buoni e, confrontando i tempi di esecuzione delle varie versioni dell'algoritmo, è stato costruito il grafico in figura

4.4.

E' importante osservare specialmente la differenza tra i tempi di esecuzione dell'algoritmo non coalescente con quello coalescente. Questo è un tipico esempio di come la struttura degli accessi in memoria influenzi in maniera sostanziale i tempi di esecuzione del kernel.

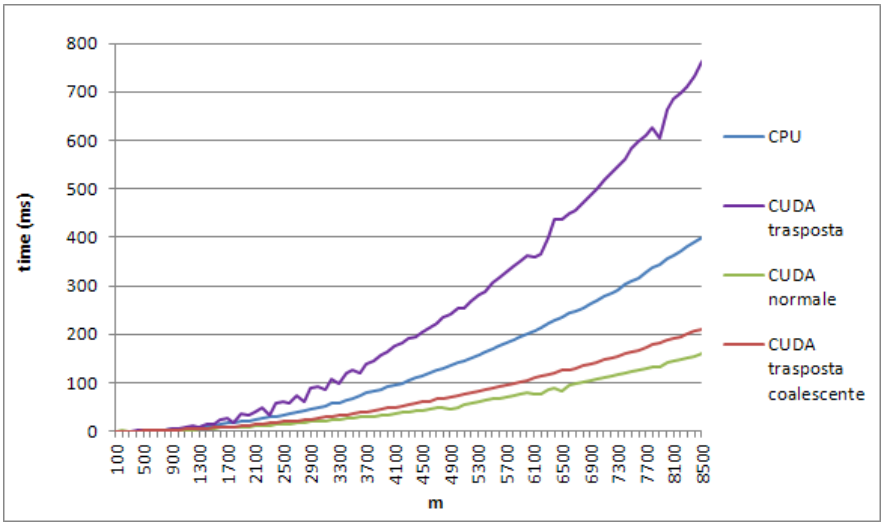


Figura 4.4: Confronto tempi di esecuzione delle varie versioni dell'algoritmo per il calcolo della matrice inversa al variare di m.

Risorse utilizzate

Analizzando il primo kernel, detto $deviceJac2InvFirstRow$, l'output del compilatore specifica:

```
ptxas info : Compiling entry function '_Z25deviceJac2InvFirstRowPfS_i' for 'sm_10'
ptxas info : Used 11 registers, 1056+16 bytes smem, 20 bytes cmem[1], 4 bytes cmem[14]
```

Dalla struttura del kernel si ha che si ha che il numero di warp in un blocco è pari a $W_{block} = \frac{256}{32} = 8$, quindi il numero di registri allocati ad ogni blocco è pari a $R_{block} = ceil(8 \cdot 32 \cdot 11, 256) = 2816 < 8K$. Considerando che questo kernel è costituito da solo un blocco di thread questo vincolo è rispettato.
Per quanto riguarda i ragionamenti riguardanti la shared memory, ad ogni blocco ne vengono assegnati $S_{block} = ceil(1072, 512) = 1536B < 16KB$ byte. Essendoci

un solo blocco di thread anche questo vincolo è rispettato.

Analizzando il secondo kernel, detto $deviceJac2InvScaleFirstRow$, l'output del compilatore specifica:

```
ptxas info : Compiling entry function '_Z26deviceJac2InvScaleFirstRowPfi' for 'sm_10'
ptxas info : Used 4 registers, 16+16 bytes smem, 4 bytes cmem[1], 4 bytes cmem[14]
```

Dalla struttura del kernel si ha che si ha che il numero di warp in un blocco è pari a $W_{block} = \frac{256}{32} = 8$, quindi il numero di registri allocati ad ogni blocco è pari a $R_{block} = ceil(8 \cdot 32 \cdot 4, 256) = 1024 < 8K$. Considerando che questo kernel è costituito da solo un blocco di thread questo vincolo è rispettato.
Per quanto riguarda i ragionamenti riguardanti la shared memory, ad ogni blocco ne vengono assegnati $S_{block} = ceil(32, 512) = 512B < 16KB$ byte. Essendoci un solo blocco di thread anche questo vincolo è rispettato.

Analizzando il terzo kernel, detto $deviceJac2Inv$, l'output del compilatore specifica:

```
ptxas info : Compiling entry function '_Z17deviceJac2InvPfS_i' for 'sm_10'
ptxas info : Used 9 registers, 32+16 bytes smem, 12 bytes cmem[1], 4 bytes cmem[14]
```

Dalla struttura del kernel si ha che si ha che il numero di warp in un blocco è pari a $W_{block} = \frac{256}{32} = 8$, quindi il numero di registri allocati ad ogni blocco è pari a $R_{block} = ceil(8 \cdot 32 \cdot 9, 256) = 2304$. Considerando che ad ogni multiprocessore verranno assegnati al massimo $B_{SM} = \frac{768}{256} = 3$ blocchi si può concludere che $B_{SM} \cdot R_{block} = 6912 < 8k$ quindi questo vincolo è rispettato.
Per quanto riguarda i ragionamenti riguardanti la shared memory, ad ogni blocco ne vengono assegnati $S_{block} = ceil(48, 512) = 512$ byte. Avendo ogni SM una disponibilità di 16KB di memoria condivisa e potendoci assegnare un massimo di tre blocchi contemporaneamente, si ha che $S_{block} \cdot B_{SM} = 1536B < 16KB$ e quindi sono stati rispettati i vincoli legati a questo fattore.

4.3.4 Prodotto della matrice inversa per la trasposta della matrice di input

L'ultima operazione da effettuare per ottenere la matrice pseudoinversa consiste nel moltiplicare la matrice ottenuta al passo precedente, che chiameremo E^{-1} per la matrice di input trasposta, ovvero A^T.
Nel valutare come effettuare questo prodotto anche in questo caso analizziamo la struttura delle matrici coinvolte. Si ha che la matrice E^{-1} è completamente definita e non si possono fare ipotesi particolari sulla struttura di quest'ultima, mentre A^T ha la ben nota struttura in cui solo due elementi per colonna sono definiti.
Questo aiuta molto nel calcolo in quanto il prodotto $riga \times colonna$ per il calcolo di ogni elemento ha un costo costante pari a 2 e non dipendente dalla taglia dell'input. Considerando le operazioni da effettuare si ha quindi che la complessità computazionale per quest'ultimo passo del calcolo della matrice pseudoinversa è pari a:

$$T = 2n \cdot m = O(n \cdot m) \tag{4.9}$$

Studiando meglio il procedimento per ottenere la matrice risultante si può osservare che, separando i due elementi di ogni colonna di A^T che concorrono nel calcolo di ogni elemento della matrice pseudoinversa, si può suddividere il calcolo nelle seguenti fasi:

- copiare la prima riga di A^T in ogni riga di A_{pinv} moltiplicata per il primo valore della corrispondente riga di E^{-1}

- effettuare il prodotto $colonna \times riga$ tra la i-esima colonna di E^{-1} e l'i-esimo blocco di A^T sommando la matrice risultante alla zona di A_{pinv} corrispondente

La giustificazione a questa suddivisione del calcolo sta nel fatto che in questo modo in CPU è possibile sfruttare le primitive messe a disposizione da OPENCV per il calcolo matriciale che permettono di ottenere prestazioni migliori rispetto al calcolo standard.

Analisi del parallelismo e algoritmo

L'ultima fase dell'algoritmo rappresenta anche la fase più onerosa in termini di tempo di calcolo ed è quindi stata la prima ad essere studiata ed implementata in CUDA.
Con riferimento al grado di parallelismo che presenta quest'algoritmo, è evidente che ogni elemento della matrice di output A_{pinv} può essere calcolato indipendentemente, quindi il grado di parallelismo è pari a $n \cdot m$.

Nella prima implementazione dell'algoritmo, è stata sviluppata la tecnica parallela equivalente a quella presentata precedentemente per il calcolo in CPU, utilizzando due kernel differenti per ognuna delle due fasi.
La prima fase è stata realizzata attraverso una struttura di thread costituita da:

- GRIGLIA: monodimensionale con n blocchi
- BLOCCO: monodimensionale con m thread

In cui ogni thread effettua due letture in memoria per effettuare il calcolo dell'elemento di posizione $(thread_{id}, blocco_{id})$ e una in scrittura per salvare il risultato. In questo modo si è massimizzato il grado di parallelismo, senza badare alla struttura degli accessi in memoria e alla ridondanza di molti accessi effettuati da thread differenti.
La seconda fase dell'algoritmo invece presenta una struttura di thread costituita da:

- GRIGLIA: monodimensionale con m blocchi
- BLOCCO: monodimensionale con m thread

Ogni thread in questo caso ha il compito di effettuare il calcolo di una riga della sottomatrice identificata dal blocco di appartenenza e di sommarne il risultato all'output del passo precedente. Anche in questo caso non è stata prestata particolare attenzione alla struttura degli accessi in memoria e a come alcune parti del codice presentino ridondanza nelle letture dalla memoria globale ma ci si è concentrati sulla correttezza dell'output e sull'utilizzo del parallelismo.

Una volta compresa la struttura dell'algoritmo parallelo si può andare ad indagarne la complessità computazionale. Sempre prestando attenzione solo alle operazioni effettuate e considerando tutti i thread come operanti contemporaneamente si ha che la durata dell'algoritmo è data dal blocco della matrice di input con dimensione maggiore. Considerando tutti i blocchi di dimensione uguale si può considerare l'algoritmo come operante in $O(\frac{n}{m})$.
In linea teorica quindi i vantaggi dovrebbero essere enormi, ed era ciò che ci si aspettava dall'esecuzione dell'algoritmo e dal confronto dei tempi di esecuzione. In realtà i risultati ottenuti non corrispondono minimamente alle attese e da come si può vedere in figura 4.6 e 4.7 i tempi di esecuzione sono per taglie del problema elevate molto peggiori della soluzione seriale.

Studiando le varie tecniche di ottimizzazione si è quindi deciso di riprogettare l'algoritmo dimenticando completamente la struttura del metodo utilizzato in CPU.
La tecnica che si è deciso di utilizzare riprende in parte l'esempio di prodotto matriciale presentato nella sezione 2.5.3. Suddividendo la matrice di output in fasce in corrispondenza dei blocchi della matrice di input si può assegnare ad un blocco di thread il compito di elaborare il risultato di quest'ultima.
Ogni thread avrà il compito di calcolare una riga della sottomatrice di output e gli input saranno caricati seguendo la logica del risparmio degli accessi in memoria e cercando di ottenere il più possibile accessi coalescenti. Ogni blocco di thread caricherà quindi due colonne di E^{-1} e una parte di due righe di A^T così come evidenziato in figura 4.5. Il prodotto tra le due matrici così ottenute corrisponderà

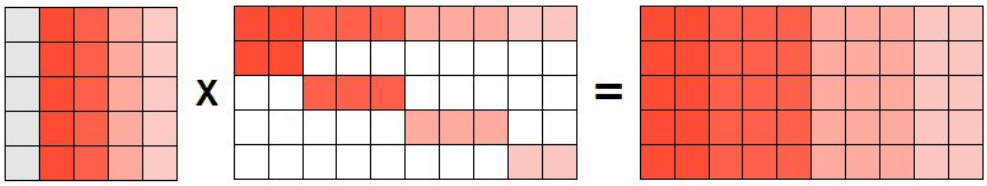


Figura 4.5: Tecnica di calcolo per il prodotto tra la matrice inversa e la matrice di input trasposta.

all'output della matrice A^{pinv} per quanto riguarda la fascia interessata.

Al fine di ottenere degli accessi coalescenti, e quindi risparmiando notevolmente tempo negli accessi in memoria, è stato deciso di utilizzare come matrici di input la trasposta di E^{-1}, e A memorizzata su due righe. In questo modo gli accessi in memoria dei singoli thread rispettano le regole enunciate precedentemente e si avranno delle prestazioni migliori.
Il kernel che esegue il calcolo ha quindi la seguente struttura:

- GRIGLIA: bidimensionale con $x = \frac{m+CUDA\_BLOCK\_SIZE\_1D-1}{CUDA\_BLOCK\_SIZE\_1D}$ e $y = m - 1$ blocchi
- BLOCCO: monodimensionale con $CUDA\_BLOCK\_SIZE\_1D$ thread

Sempre considerando tutti i thread come operanti in parallelo il grado di parallelismo dell'algoritmo sarà quindi pari a m^2 in quanto la matrice di output è suddivisa in m-1 fasce calcolate indipendentemente, e ogni fascia è costituita da m righe anch'esse calcolate indipendentemente.
Grazie a questa tecnica di calcolo la complessità dell'algoritmo dipende dal blocco di A con lunghezza massima, ma sempre considerando i blocchi di lunghezza uguale si può considerare l'algoritmo come operante in $O(\frac{n}{m})$. Come sempre questa è una stima basata solo sulle operazioni da effettuare per calcolare l'output e si può notare come sia esattamente la stessa complessità dell'algoritmo precedente, ma dai grafici si osserva come i tempi di esecuzione siano decisamente diminuiti.

Per capire le differenze esistenti tra le varie versioni dell'algoritmo sono stati effettuati dei test su matrici di dimensione diversa per confrontarne i tempi di esecuzione. I risultati sono riassunti nei grafici in figura 4.6 e 4.7. Si osserva subito come il nuovo algoritmo migliori notevolmente le prestazioni del precedente permettendo di raggiungere un fattore di miglioramento compreso tra 10 e 14. In particolare è interessante osservare come il vecchio algoritmo, oltre a non avere le prestazioni previste, peggiora con l'aumentare della taglie del problema, infatti dai grafici si osserva come per matrici con $n = 20000$ migliori i tempi della CPU di un fattore circa pari a 2, mentre per matrici con $n = 80000$ rallenti in maniera evidente l'esecuzione.

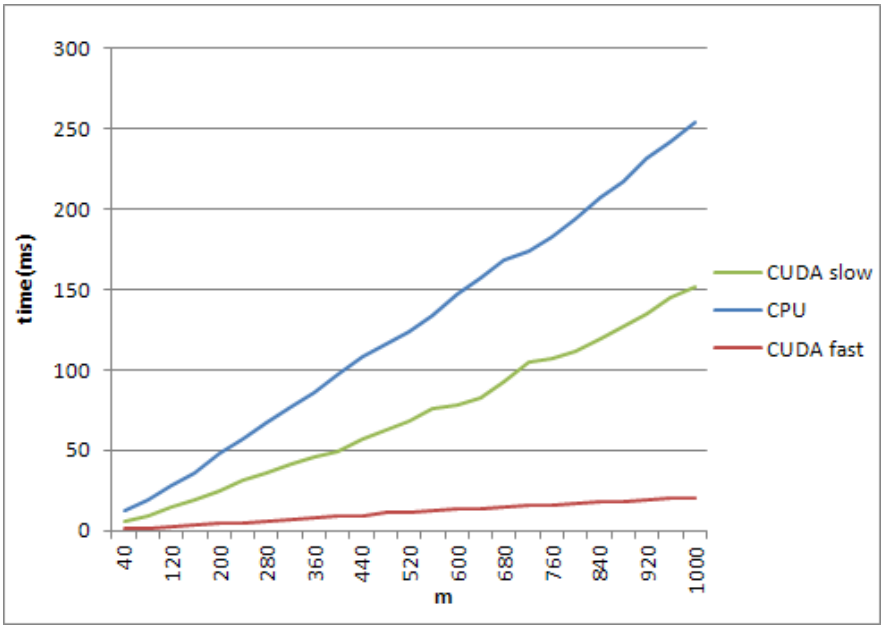


Figura 4.6: Confronto tempi di esecuzione delle varie versioni dell'algoritmo per l'ultima fase del calcolo della matrice pseudoinversa al variare di m con n=20000.

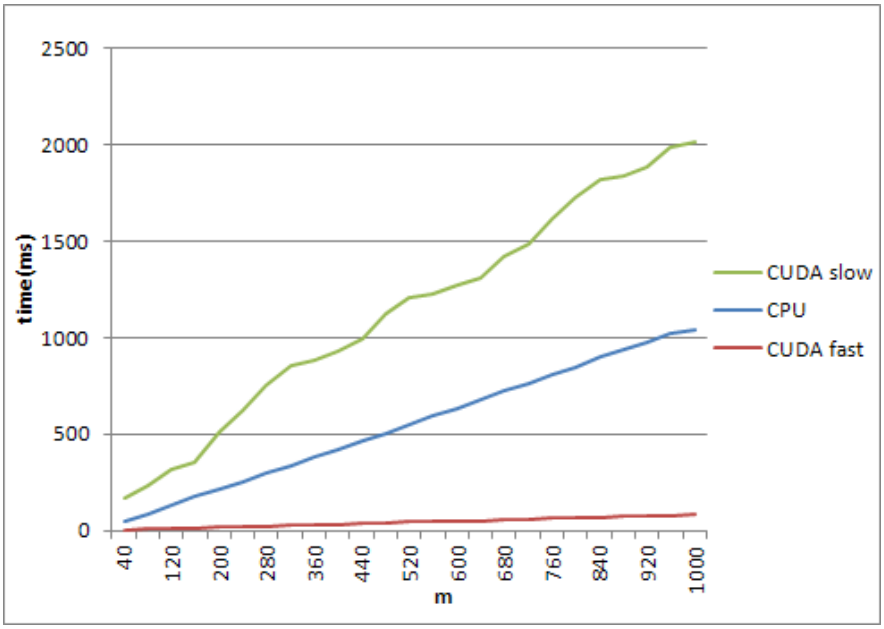


Figura 4.7: Confronto tempi di esecuzione delle varie versioni dell'algoritmo per l'ultima fase del calcolo della matrice pseudoinversa al variare di m con n=80000.

Risorse utilizzate

Per quanto riguarda il kernel che costituisce l'algoritmo, detto $deviceMulJ2invJac$, l'output del compilatore specifica:

```
ptxas info : Compiling entry function '_Z17deviceMulJ2invJacPfS_PiS0_iiS_' for 'sm_10'
ptxas info : Used 14 registers, 4160+16 bytes smem, 16 bytes cmem[1], 4 bytes cmem[14]
```

Dalla struttura del kernel si ha che si ha che il numero di warp in un blocco è pari a $W_{block} = \frac{256}{32} = 8$, quindi il numero di registri allocati ad ogni blocco è pari a $R_{block} = ceil(8 \cdot 32 \cdot 14, 256) = 3584$. Considerando che ad ogni multiprocessore verranno assegnati al massimo $B_{SM} = \frac{768}{256} = 3$ blocchi si può concludere che $B_{SM} \cdot R_{block} = 10752 > 8k$ quindi ci sarà un'inefficienza nell'esecuzione su schede di compute capability 1.0 e 1.1 in quanto solo due blocchi su tre potranno essere assegnati ad ogni Streaming Multiprocessor.
Per quanto riguarda i ragionamenti riguardanti la shared memory, ad ogni blocco ne vengono assegnati $S_{block} = ceil(4172, 512) = 4608$ byte. Avendo ogni SM una disponibilità di 16KB di memoria condivisa e potendoci assegnare un massimo di tre blocchi contemporaneamente, si ha che $S_{block} \cdot B_{SM} = 13824B < 16KB$ e quindi sono stati rispettati i vincoli legati a questo fattore.

Capitolo 5

Test e risultati

In questo capitolo vengono raccolti e presentati i risultati del lavoro di tesi svolto. Il software è stato analizzato e testato nel suo complesso e nelle singole parti in cui è stato suddiviso e analizzato nel capitolo precedente. Sono stati rappresentati i tempi di esecuzione confrontandoli con la corrispondente parte eseguita in CPU per evidenziarne le differenze, inoltre per meglio comprendere lo speedup raggiunto sono stati rappresentati i fattori di miglioramento raggiunti e come questi evolvono sulla base della taglia del problema.

5.1 Calcolo complessivo

Per testare il calcolo complessivo della matrice pseudoinversa sono state effettuate delle prove con matrici di dimensione n=(20000,40000,60000,80000,10000,120000) al variare di m. In questo modo e stato possibile ottenere i grafici 5.1, 5.2, 5.3, 5.4, 5.5, 5.6.

Sono stati poi raccolti i risultati ottenuti dalle varie esecuzioni del software con taglie diverse del problema, dai quali abbiamo ricavato il fattore di miglioramento per ogni esecuzione dell'algoritmo.
In figura 5.7 si possono osservare come varino i fattori di miglioramento delle varie esecuzioni del software al variare di m. Si può notare come sostanzialmente con l'aumentare di m non si notino grosse differenze, ma il valore dello speedup fluttui attorno ad un valore medio che aumenta con l'aumentare di n.

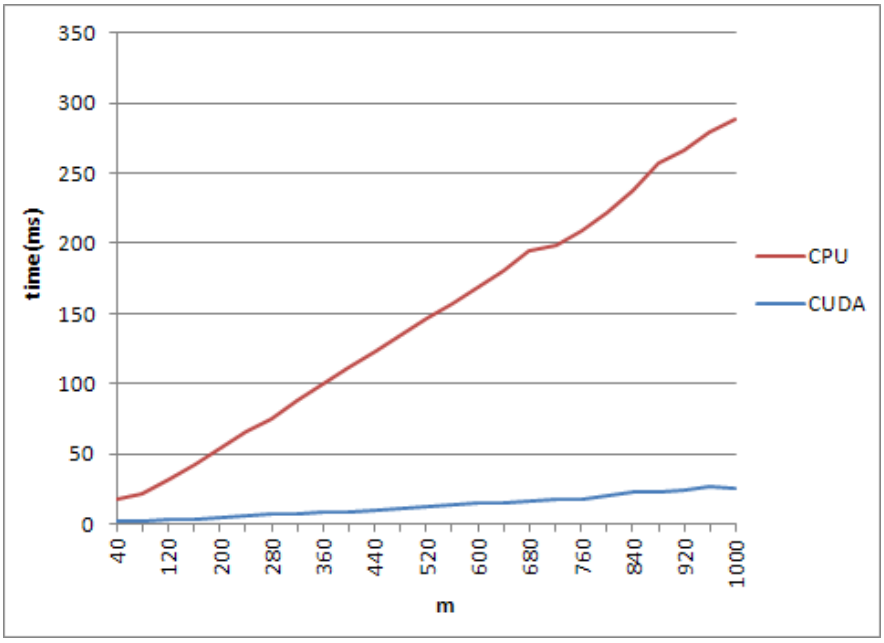


Figura 5.1: Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=20000.

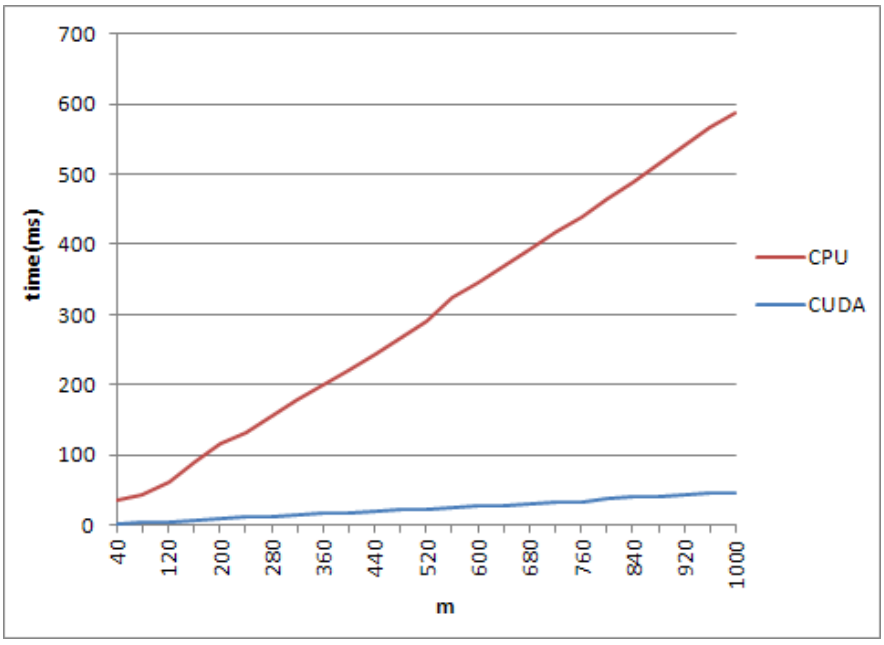


Figura 5.2: Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=40000.

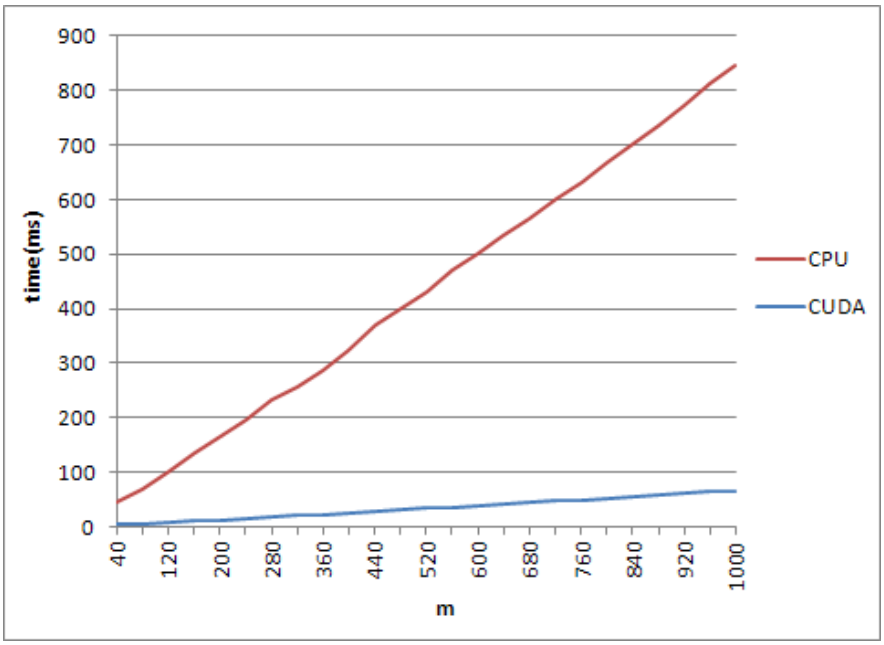


Figura 5.3: Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=60000.

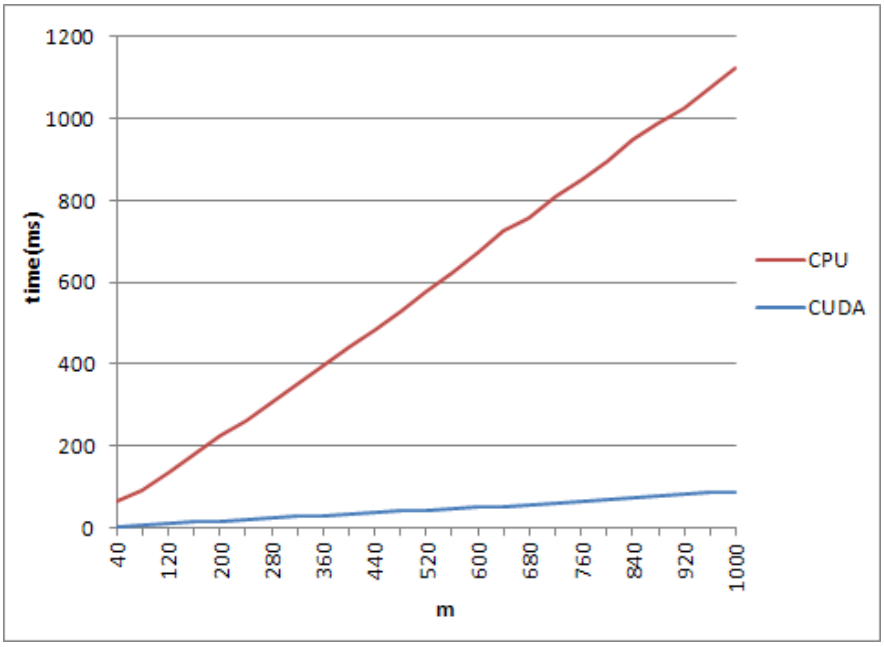


Figura 5.4: Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=80000.

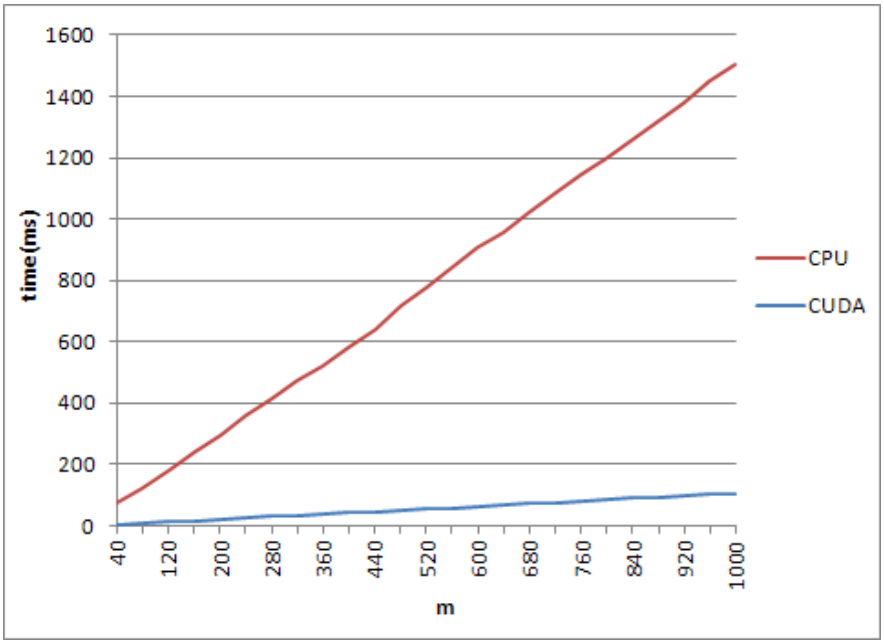


Figura 5.5: Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=100000.

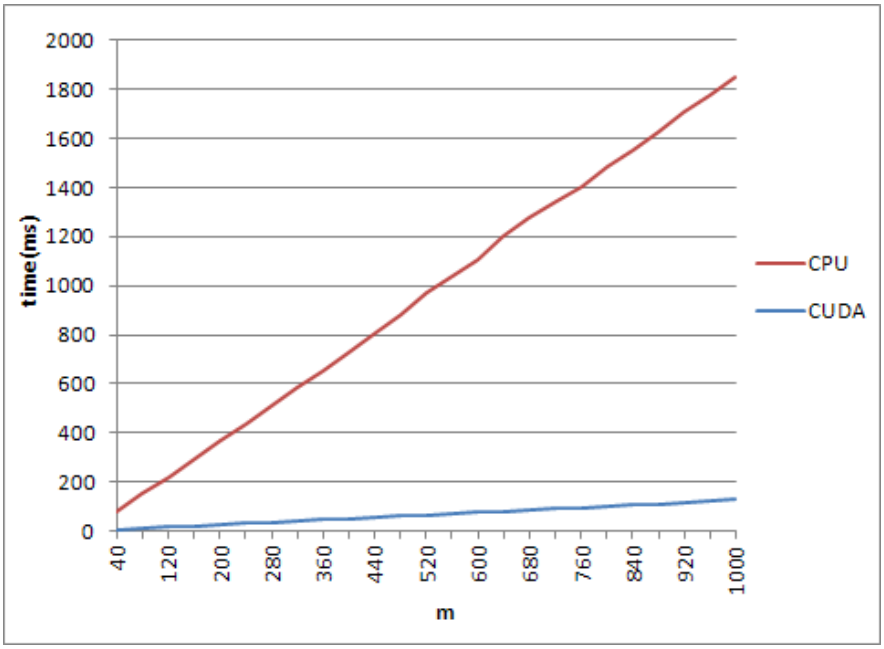


Figura 5.6: Tempi di esecuzione per il calcolo della matrice pseudoinversa al variare di m con n=120000.

Per osservare meglio questo fenomeno è stato realizzato il grafico in figura 5.8 dove è stato calcolato, per ogni esecuzione con n fissato, il fattore di miglioramento medio ottenuto. E' interessante come questo aumenti con l'aumento della taglia, passando da un valore di 10.5 per n=20000 fino ad arrivare a 13.6 con n=120000. Questo è un buon indicatore sull'efficienza dell'algoritmo ottenuto che, scalando il problema, non solo mantiene le performance desiderate ma le aumenta. Il motivo per questo comportamento può essere dato dal fatto che, aumentando la taglia del problema, vengono sempre più ammortizzati i costi per gli accessi in memoria nei confronti delle operazioni da effettuare sui dati.

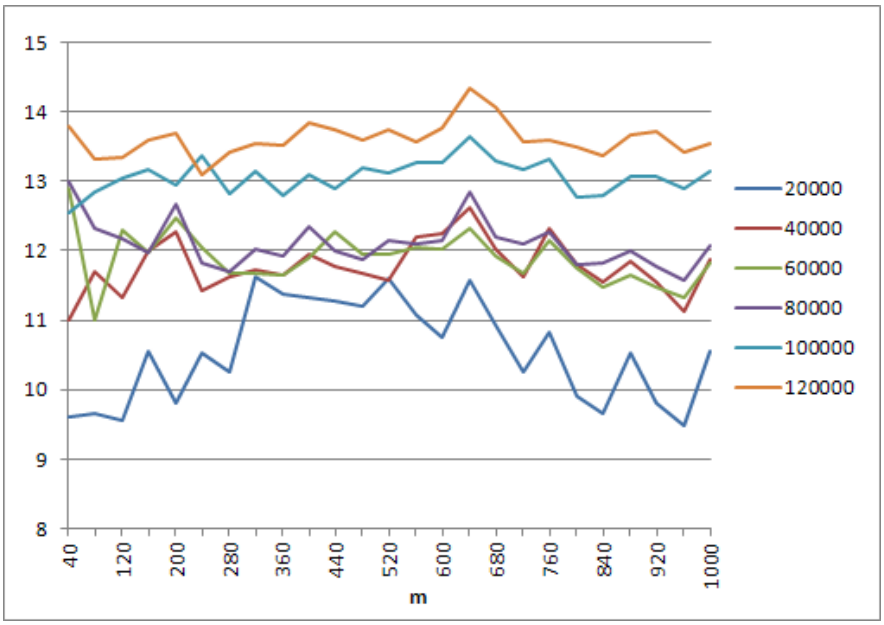


Figura 5.7: Confronto tra i fattori di miglioramento delle prove effettuate.

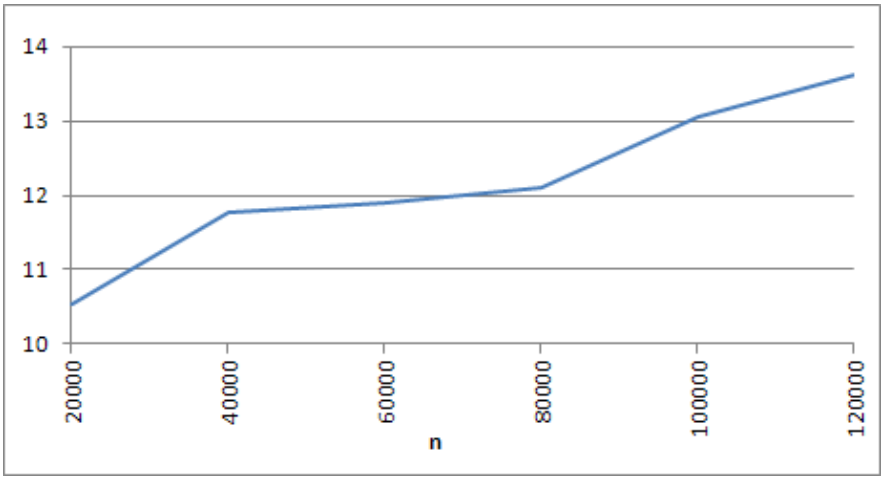


Figura 5.8: Fattore di miglioramento medio delle prove effettuate al variare di n.

5.2 Prodotto della matrice di input trasposta per se stessa

Vengono qui riportati i risultati relativi ai tempi di esecuzione del primo algoritmo che compone il calcolo della matrice pseudoinversa.
In figura 5.9 si osserva la differenza tra i tempi di esecuzione dell'algoritmo implementato in CUDA e quello operante in CPU al variare di n. Per confrontare l'algoritmo con taglie del problema diverse non è stato necessario modificare il valore di m perchè, come analizzato nella sezione 4.3.2, la complessità computazionale di questo algoritmo è O(n). In ogni caso sono state effettuate delle prove variando il valore di m, le quali hanno confermato le conclusioni a cui si era giunti nell'analisi matematica presentate in precedenza.

Per poter confrontare i tempi di esecuzione dell'algoritmo sono state create matrici di input di grandi dimensioni (fino a $n = 10^7$) per verificare la reale efficienza del programma, in quanto il problema non potrà mai raggiungere taglie di queste dimensioni (se non altro a causa della mancanza di sufficiente memoria nel dispositivo).
Dal grafico 5.10 si può osservare come lo speedup, dopo una fase iniziale altalenante (causata dai bassi tempi di esecuzione), si stabilizzi attorno ad un valore

circa pari a 3.

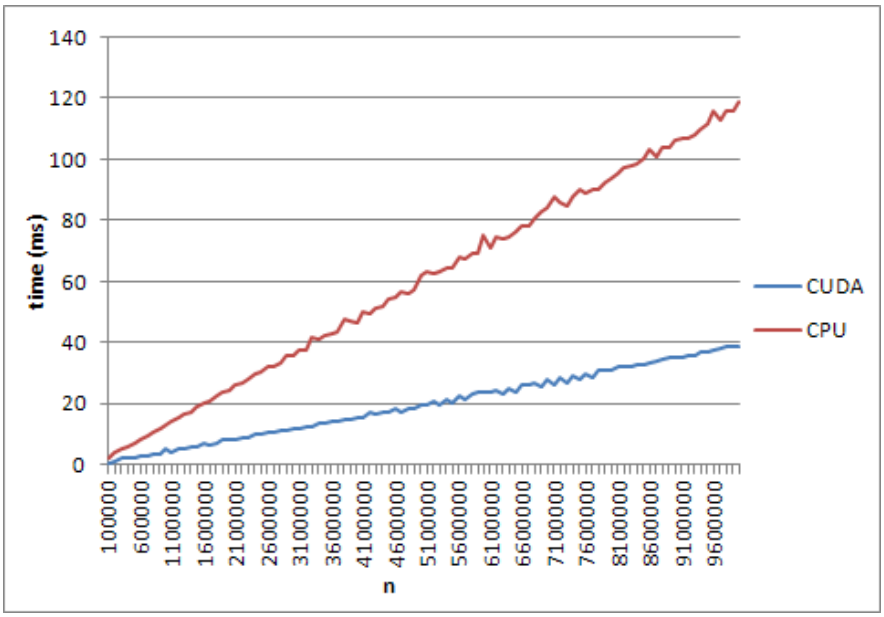


Figura 5.9: Tempi di esecuzione del primo prodotto in CPU e GPU.

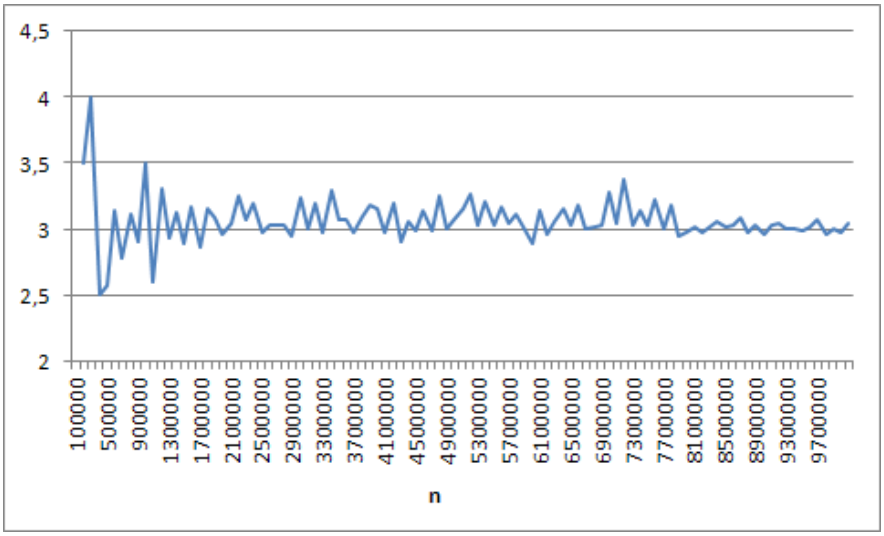


Figura 5.10: Fattore di miglioramento del primo prodotto.

5.3 Calcolo della matrice inversa

In questa sezione vengono riportati i risultati relativi ai tempi di esecuzione della parte dell'algoritmo che esegue il calcolo della matrice inversa utilizzando la tecnica della $Gauss - Jordan\ elimination$.
In figura 5.11 vengono riportati i tempi ottenuti dall'algoritmo operante in GPU e quello operante in CPU al variare di m (in questo caso è di scarso interesse variare n in quanto la matrice da invertire è di dimensioni $m \times m$).
Si può osservare come venga anche in questo caso apportato un miglioramento sostanziale alle performance dell'algoritmo, fattore evidenziato in figura 5.12. Dal grafico risulta che dopo una variazione iniziale (dovuta al fatto che i tempi di esecuzione sono praticamente pari a 0), il fattore di miglioramento si stabilizza attorno ad una media di circa 1.8-1.9.
Come risultato questo è nettamente inferiore allo speedup ottenuto nelle altre due parti dell'algoritmo, ma questo è dato anche dalla natura del problema, in cui alcune parti non possono essere parallelizzate ma devono invece essere eseguite in serie.

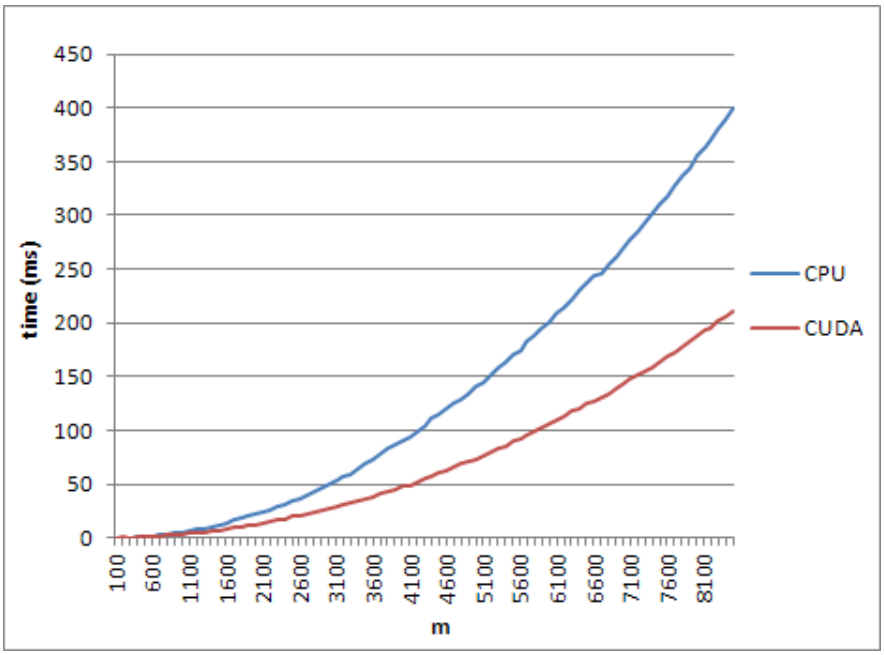


Figura 5.11: Tempi di esecuzione del calcolo dell'inversa in CPU e GPU.

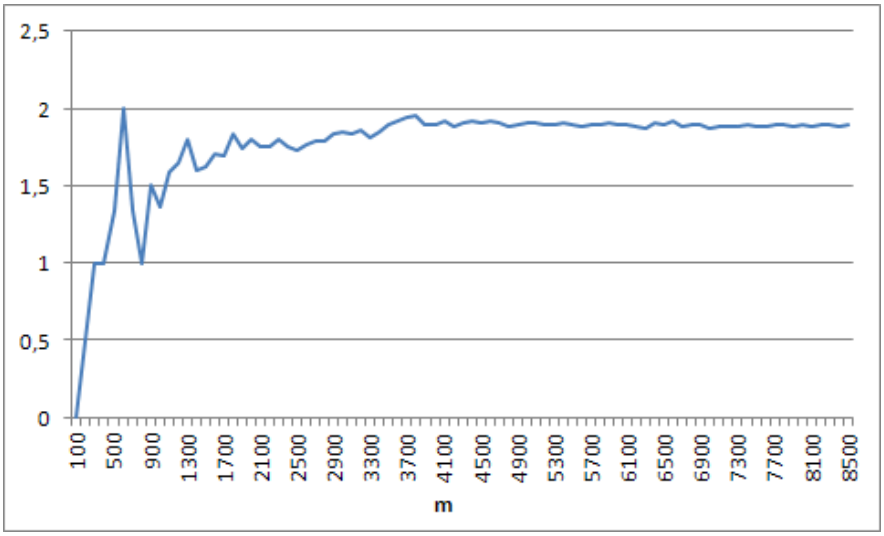


Figura 5.12: Fattore di miglioramento del calcolo della matrice inversa.

5.4 Prodotto della matrice inversa per la trasposta della matrice di input

L'ultimo prodotto che compone il calcolo della matrice pseudoinversa è il più interessante da analizzare, essendo quello che occupa il $90 - 95\%$ del tempo di esecuzione dell'algoritmo complessivo.
Per poterlo poi confrontare con il calcolo totale sono state effettuate le stesse prove effettuate per il calcolo complessivo della matrice pseudoinversa, ovvero con matrici di dimensione n=(20000,40000,60000,80000,10000,120000) e m variabile. In questo modo sono stati ottenuti i grafici 5.13, 5.14, 5.15, 5.16, 5.17, 5.18.

Il guadagno di tempo è sempre notevole per tutte le esecuzioni ma si è voluto riassumere anche in questo caso i fattori di miglioramento relativi a tutte le prove effettuate al variare di m ottenendo il grafico in figura 5.19 che evidenzia come lo speedup per ogni esecuzione fluttui attorno ad uno stesso valore medio. Per questo sie è andati a costruire il grafico 5.20 che mostra l'andamento del fattore di miglioramento medio delle prove eseguite a parità di n. E' evidente come questo migliori con l'aumentare della taglia del problema passando da circa 11.5 con n=20000 per arrivare a 13.9 con n=120000.

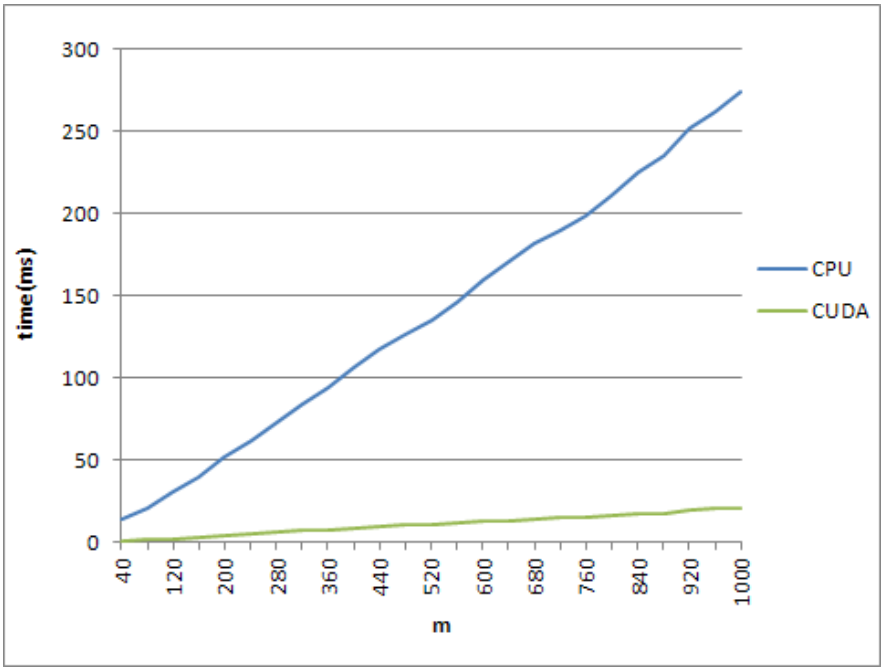


Figura 5.13: Tempi di esecuzione dell'ultimo prodotto al variare di m con n=20000.

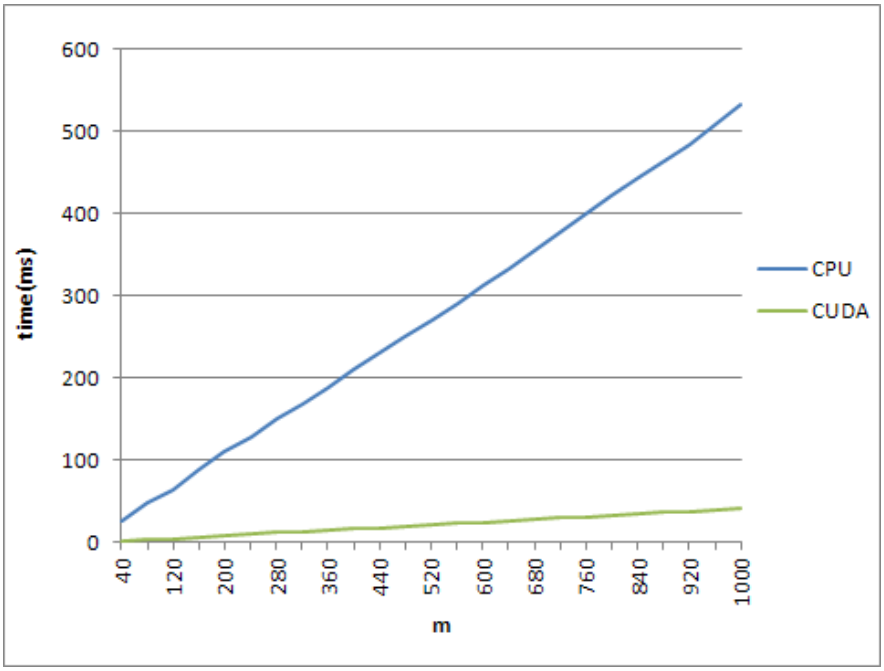


Figura 5.14: Tempi di esecuzione dell'ultimo prodotto al variare di m con n=40000.

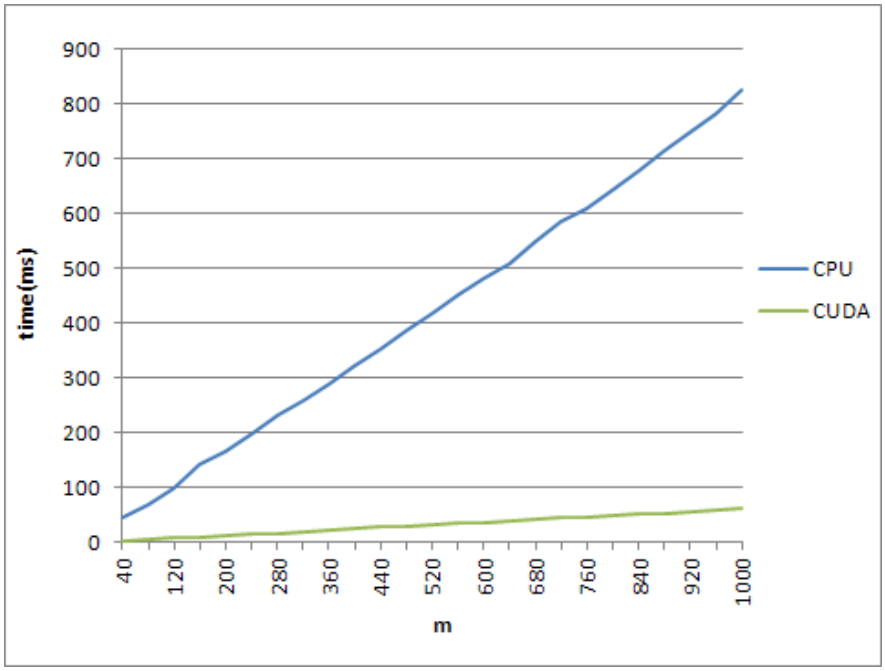


Figura 5.15: Tempi di esecuzione dell'ultimo prodotto al variare di m con n=60000.

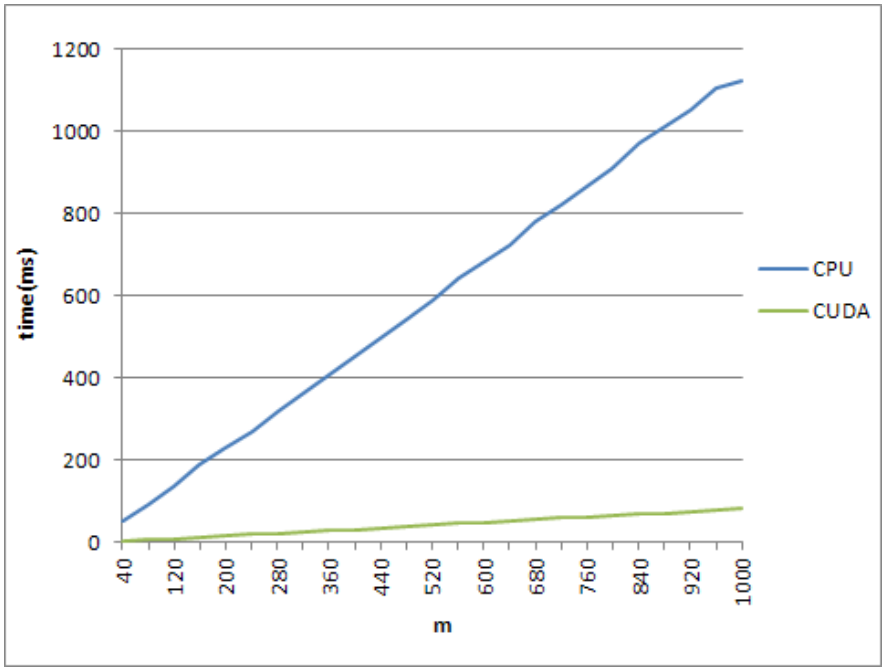


Figura 5.16: Tempi di esecuzione dell'ultimo prodotto al variare di m con n=80000.

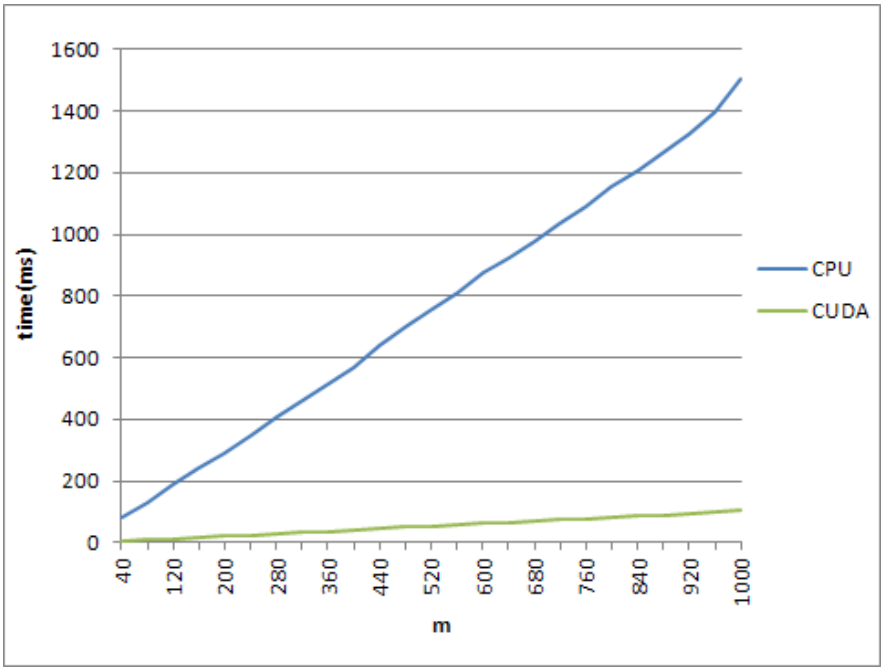


Figura 5.17: Tempi di esecuzione dell'ultimo prodotto al variare di m con n=100000.

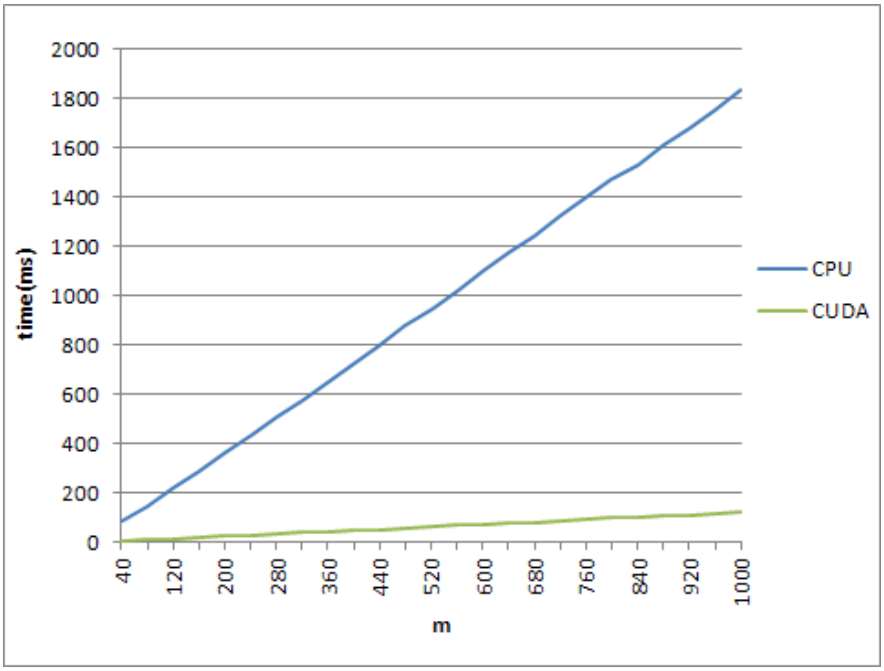


Figura 5.18: Tempi di esecuzione dell'ultimo prodotto al variare di m con n=120000.

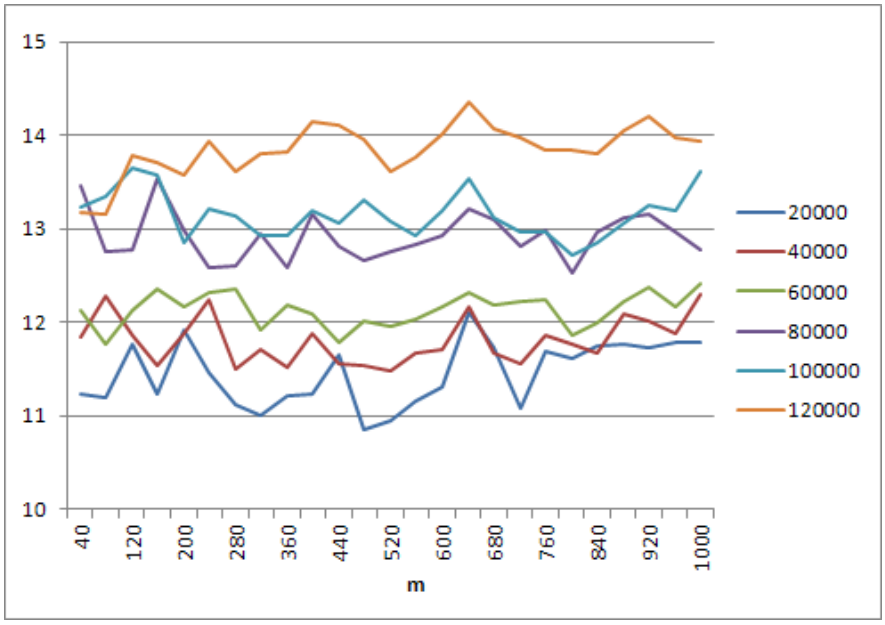


Figura 5.19: Fattori di miglioramento delle prove effettuate.

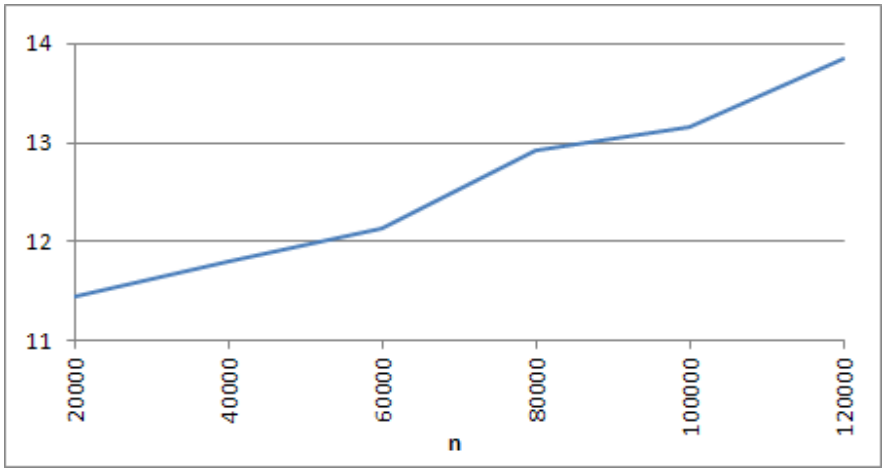


Figura 5.20: Fattore di miglioramento medio delle prove effettuate.

Dai risultati ottenuti si può osservare come ci sia una differenza tra lo speedup ottenuto nelle varie parti dell'algoritmo e l'algoritmo nel suo complesso. In particolare, essendo l'ultimo prodotto il più consistente in termini di tempo di esecuzione si è voluto confrontare l'andamento dei fattori di miglioramento relativi all'algoritmo totale e al calcolo dell'ultimo prodotto. Il grafico risultante è quello

in figura 5.21 in cui si osserva come lo speedup dell'algoritmo completo stia al di sotto di quello relativo all'ultimo prodotto. Questo risultato è corretto e mette in evidenza come i primi due passi dell'algoritmo, seppure in minima parte, influenzino le performance dell'algoritmo nel suo complesso a causa di valori di speedup ben inferiori a quello ottenuto nell'ultima parte.

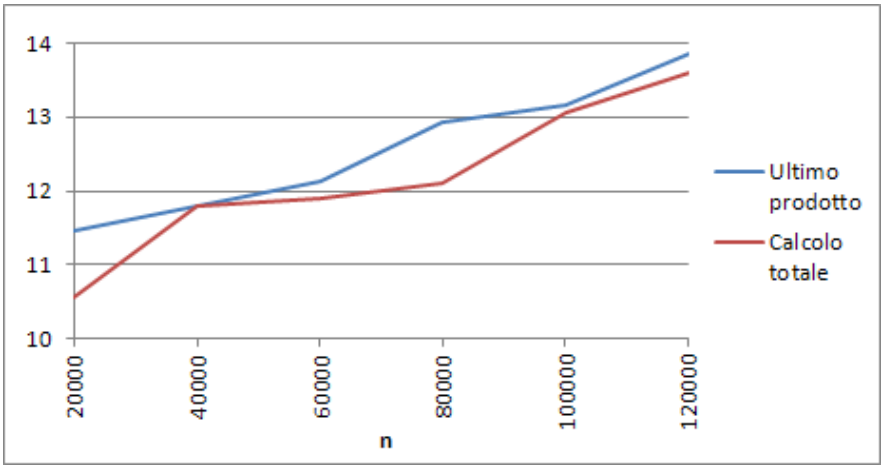


Figura 5.21: Confronto tra i fattori di miglioramento del calcolo totale e dell'ultimo prodotto.

Capitolo 6

Conclusioni

Il software di Visual SLAM su cui si è andati ad operare può ora avvalersi della funzione realizzata per il calcolo della matrice pseudoinversa. Questa permetterà di ridurre i tempi di elaborazione e, in base alla taglia del problema (ovvero della precisione che si vorrà nella ricostruzione dell'immagine) e con qualche sviluppo futuro, potrà operare in tempo reale permettendo ad un robot mobile dotato di fotocamera omnidirezionale di essere autonomo in qualsiasi ambiente esso venga posto ad operare.

Lo sviluppo di questo lavoro di tesi ha consentito di applicare le nozioni di calcolo parallelo acquisite durante il corso di studi. Inoltre, come illustrato nei capitoli precedenti, in CUDA il grado di parallelismo non è l'unico fattore di cui tenere conto per la realizzazione di algoritmi efficienti in quanto l'utilizzo della memoria è un fattore altrettanto importante. E' stato quindi necessario acquisire un'adeguata sensibilità nell'utilizzo dello strumento per stabilire il giusto trade off tra utilizzo di memoria e grado di parallelismo.

Lo studio delle tecniche di programmazione delle GPU, unito alle possibili ottimizzazioni del codice, ha permesso di ottenere i risultati che inizialmente ci si aspettava attraverso l'utilizzo di questo strumento. Anche se teoricamente le operazioni eseguibili da una GPU, rispetto a quelle di una CPU, potrebbero raggiungere fattori di speedup superiori a quelli ottenuti, ci si deve poi scontrare con la realtà specifica del problema da risolvere che talvolta, come nel caso in esame, presenta come collo di bottiglia non tanto la complessità temporale del problema (in parallelo questa come si è visto viene abbattuta) ma piuttosto la quantità di

dati necessari alle operazioni di calcolo dell'output.
Nonostante le difficoltà incontrate, i risultati ottenuti sono più che soddisfacenti. L'iniziale obbiettivo di uno speedup di 10x è stato raggiunto e superato: dai grafici conclusivi presentati nel capitolo 5 abbiamo osservato come, con l'aumentare della taglia del problema, il fattore di miglioramento arrivi a 14.
E' importante ricordare che per lo sviluppo di questo lavoro di tesi ci si è avvalsi di un dispositivo grafico relativamente obsoleto se confrontato con le schede grafiche più recenti (ovvero quelle con compute capability 2.x, vedi A). Lo speedup ottenuto con questo dispositivo grafico può quindi essere migliorato ulteriormente grazie all'utilizzo di schede video più avanzate. Il codice CUDA, infatti, può operare su qualsiasi scheda grafica e, se correttamente progettato, permette di sfruttare le migliori caratteristiche del dispositivo su cui va ad operare.

Nella realizzazione di questo lavoro di tesi sono stati molti i problemi affrontati. E' stato innanzi tutto necessario apprendere un nuovo stile di programmazione che ha richiesto molto tempo, rendendo necessaria la consultazione di diversi manuali e forum per risolvere problemi pratici che non si incontrano nella normale programmazione C/C++.
Inoltre le caratteristiche della scheda utilizzata hanno posto in diverse situazioni delle limitazioni sulle operazioni che si cercavano di svolgere. Esempi significativi sono la poca memoria shared disponibile (triplicata negli ultimi dispositivi), i registri in numero limitato (nelle schede 2.x sono stati quadruplicati), la mancanza della doppia precisione e la mancanza della memoria surface (sarebbe stato interessante testarla per verificarne l'utilità e l'efficienza).
Inoltre un altro problema che è stato affrontato riguarda la precisione con cui vengono salvati i risultati in CPU e GPU. Si credeva che fosse la stessa e perciò spesso ci si è trovati ad analizzare dati e ricercare eventuali problemi anche laddove questi non esistevano, ma erano generati dal modo in cui vengono gestiti i numeri in virgola mobile nelle GPU. Queste infatti presentano delle differenze nella gestione degli arrotondamenti in alcune operazioni che, sommate per un gran numero di dati, possono dare origine a differenze che ad una prima analisi possono essere associate a bug del sistema.

6.1 Sviluppi futuri

Nel futuro altre parti del codice potranno essere reingegnerizzate e reimplementate in CUDA, la maggior parte del codice può essere, infatti, efficientemente parallelizzata. Inoltre sarebbe utile testare il codice sviluppato in questo lavoro di tesi con schede di ultima generazione e valutare la possibilità di rendere tale codice più specifico per tali dispositivi.

Appendice A

Dispositivi grafici NVIDIA

A.1 Compute capability

Il termine compute capability indica un numero decimale composto da due cifre x.y che indicano rispettivamente un numero di revisione maggiore e minore dell'architettura del dispositivo indicandone la famiglia di appartenenza. Al giorno d'oggi può assumere valori 1.0, 1.1, 1.2, 1.3, 2.0, 2.1. Nel manuale di programmazione [18] vengono fornite le tabelle A.1 e A.2 che riassumono le caratteristiche tecniche associate alle varie versioni.

Le tabelle evidenziano come siano presenti molti vincoli tecnici da valutare attentamente prima della scelta del dispositivo da utilizzare. Sarà quindi necessario uno studio preliminare sul tipo di software da implementare per definirne le caratteristiche che comporteranno la scelta del device più adeguato.
Nello sviluppo di questo lavoro di tesi infatti ci si è scontrati con un vincolo dovuto appunto alla inadeguatezza del dispositivo rispetto al software da implementare: mentre l'intero software opera con numeri floating point a doppia precisione la scheda video non ha una compute capability sufficente a supportarli, comportando un cast dei dati per poterli utilizzare nella GPU.

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory	16 KB				48 KB
per multiprocessor					
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or to a CUDA array	65536 x 32768				65536 x 65535
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				
Maximum number of textures that can be bound to a kernel	128				
Maximum width for a 1D surface reference bound to a CUDA array	N/A				8192
Maximum width and height for a 2D surface reference bound to a CUDA array					8192 x 8192
Maximum number of surfaces that can be bound to a kernel					8
Maximum number of instructions per kernel	2 million				

Tabella A.1: Specifiche tecniche associate alle varie compute capability.

	Compute Capability				
Feature Support (Unlisted features are supported for all compute capabilities)	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Integer atomic functions operating on 32-bit words in global memory	No	yes			
Integer atomic functions operating on 64-bit words in global memory	No		Yes		
Integer atomic functions operating on 32-bit words in shared memory					
Warp vote functions					
Double-precision floating-point numbers	No			Yes	
Floating-point atomic addition operating on 32-bit words in global and shared memory	No				Yes
__ballot()					
__threadfence_system()					
__syncthreads_count(), __syncthreads_and(), __syncthreads_or()					
Surface functions					

Tabella A.2: Caratteristiche supportate dalle varie compute capability.

A.2 Scheda grafica utilizzata

Figura A.1: Scheda video NVIDIA msi N9800GT.

La scheda grafica utilizzata per la realizzazione di questo lavoro di tesi è la della serie NVIDIA 9800 GT, che si può vedere in figura A.1. Questo dispositivo, con compute capability 1.1, non rappresenta il top del mercato ma un buon compromesso qualità/prezzo per verificare l'utilità e il funzionamento dell'architettura CUDA.
Le caratteristiche tecniche specifiche della scheda sono riassunte nella tabella A.3.

Processori sequenziali	112
Clock core (MHz)	600 MHz
Clock shader (MHz)	1500 MHz
Clock memoria (MHz)	900 MHz
Quantità memoria	512MB
Interfaccia di memoria	256-bit GDDR3
Banda di memoria (GB/s)	57.6
Fill Rate texture (miliardi/s.)	33.6

Tabella A.3: Specifiche tecniche della scheda video NVIDIA 9800 GT.

La caratteristica principale che si può leggere nella tabella sono il numero di processori sequenziali, ovvero il numero di Streaming Processor presenti nel dispositivo. Ricordando quello detto nella sezione 2.1 si ha che, essendoci 8 SP per ogni Streaming Multiprocessor, la scheda possiede $N = 14$ SM.
Questo dato ci porta a poter calcolare il numero massimo di thread che possono essere allocati contemporaneamente ai vari processori. Essendo infatti ogni Streaming Multiprocessor in grado di schedulare un massimo di 768 thread alla volta, si ha che in ogni istante $N_{th} = 768 \cdot 14 = 10752$ thread possono essere assegnati ai vari processori per essere eseguiti. Un numero considerevole che da solo rende l'idea sulle potenzialità del dispositivo.

Appendice B

Codice

```
#define MAX_BLOCK 256
#define MAX_COLS 256

__global__ void deviceMulJacTJac( float *jac2, int* b_start_indices,
int* block_size, int cols, int esm_jac_t_cols, float *esm_jac_t)
{

  int thx = threadIdx.x;
  int blx = blockIdx.x;

  float diag_el = 0;
  float simm_el = 0;
  float first_el = 0;

  __shared__ int start_idx_s, bsize_s;

  if(thx == 0)
  {
    start_idx_s = b_start_indices[blx];
    bsize_s = block_size[blx];
  }
  __syncthreads();

  int iter=(bsize_s + MAX_BLOCK - 1) / MAX_BLOCK;

  __shared__ float diag_block[MAX_BLOCK];
  __shared__ float simm_block[MAX_BLOCK];
  __shared__ float first_el_block[MAX_BLOCK];

  int thx_mem;

  for(int j=0; j<iter; j++)
  {
```

```
    thx_mem = thx + j * MAX_BLOCK;

    if( thx_mem < bsize_s )
    {

      float first_col_el = esm_jac_t[start_idx_s + thx_mem];
      float second_col_el = esm_jac_t[esm_jac_t_cols + start_idx_s + thx_mem];

      diag_block[thx] = second_col_el * second_col_el;
      simm_block[thx] = second_col_el * first_col_el;
      first_el_block[thx] = first_col_el * first_col_el;

      __syncthreads();

      if(thx==0)
      {
         int limit;

         if(j==iter-1)
            limit=bsize_s % MAX_BLOCK;
         else
            limit=MAX_BLOCK;

         for(int i=0; i < limit; i++)
         {
            diag_el += diag_block[i];
            simm_el += simm_block[i];
            first_el += first_el_block[i];
         }

      }

      __syncthreads();
    }
  }

  if(thx==0)
  {
     jac2[cols + blx + 1] = diag_el;
     jac2[blx + 1] = simm_el;
     jac2[cols + cols + blx + 1] = first_el;
  }
}

__global__ void deviceSumFirstEl( float *jac2, int cols )
{

  int thx = threadIdx.x;
  int iter = (cols - 1 + MAX_COLS - 1) / MAX_COLS;
```

```
  float res = 0;

  __syncthreads();

  __shared__ float first_el_blocks[MAX_COLS];

  for(int j=0; j<iter; j++)
  {
    int thx_mem = thx + j * MAX_COLS;
    if(thx_mem < cols -1)
    {

        first_el_blocks[thx] = jac2[cols + cols + thx_mem + 1];

        __syncthreads();

        if(thx == 0)
        {
           int limit;

           if(j==iter-1)
              limit=(cols-1) % MAX_COLS;
           else
              limit=MAX_COLS;

           for(int i=0; i < limit; i++)
           {
              res+=first_el_blocks[i];
           }
        }

        __syncthreads();
    }
  }

  if(thx==0)
  {
    jac2[0]=res;
  }
}

cudaError_t mulJacTJac( float *jac2, float *esm_jac_t,
int* b_start_indices,int* block_size, int cols,
int esm_jac_t_cols, float *jac2_inv_t, float * esm_jac_pinv_t )
{

  dim3 grid,block;

  grid.x = cols-1;
  block.x = MAX_BLOCK;
```

```
  CREATE_CUDA_EVENT_WAIT

  deviceMulJacTJac <<< grid, block >>> (jac2, b_start_indices,block_size,
  cols, esm_jac_t_cols, esm_jac_t);

  grid.x=1;
  block.x=MAX_COLS;

  deviceSumFirstEl <<< grid, block >>> (jac2,cols);

  CUDA_KERNEL_WAIT_POINT

  DESTROY_CUDA_EVENT_WAIT

  return cudaGetLastError();
}

__device__ float jac2_inv_first_el;

__global__ void deviceJac2InvFirstRow(float *jac2_inv_t,float *jac2,int cols)
{

  int thx = threadIdx.x;
  int iter = (cols - 1 + MAX_COLS - 1) / MAX_COLS;
  float first_el;
  if(thx==0)
  {
    first_el=jac2[0];
  }
  __syncthreads();

  __shared__ float first_elem_parts[MAX_COLS];

  for(int j=0; j<iter; j++)
  {
    int thx_mem = thx + j * MAX_COLS;
    if(thx_mem < cols -1)
    {

       float diag_el=jac2[cols + thx_mem +1];
       float first_row_el=jac2[thx_mem + 1];

       if(diag_el!=0)
       {
          float coeff=-first_row_el/diag_el;
          jac2_inv_t[cols * (thx_mem + 1)] = coeff;
          first_elem_parts[thx] = coeff * first_row_el;
       }
       else
```

```
        {
            jac2_inv_t[cols * (thx_mem + 1) + thx_mem + 1] = 0;
            first_elem_parts[thx] = 0;
        }

        __syncthreads();

        if(thx == 0)
        {
            int limit;

            if(j==iter-1)
                limit=(cols - 1) % MAX_COLS;
            else
                limit=MAX_COLS;

            for(int i=0; i < limit; i++)
            {
                first_el+=first_elem_parts[i];
            }
        }

        __syncthreads();
    }
  }

  if(thx == 0)
  {
    jac2_inv_first_el=first_el;
  }
}

__global__ void deviceJac2InvScaleFirstRow(float *jac2_inv_t, int cols )
{

  int thx = threadIdx.x;
  int iter = (cols + MAX_COLS - 1) / MAX_COLS;

  __shared__ float coeff;

  if(thx==0)
  {
    coeff= 1 / jac2_inv_first_el;
  }
  __syncthreads();

  for(int j=0; j<iter; j++)
  {
    int thx_mem = thx + j * MAX_COLS;
    if(thx_mem < cols )
```

```
    {
        jac2_inv_t[cols * thx_mem] = jac2_inv_t[cols * thx_mem] * coeff;
    }
  }
}

__global__ void deviceJac2Inv_T_coalescente(float *jac2_inv_t,
float *jac2, int cols )
{

  int thx = threadIdx.x + 1;
  int index_row = blockIdx.x;
  int iter = (cols + MAX_COLS - 1) / MAX_COLS;

  __shared__ float coeff;
  float coeff_diag;
  if(thx==1)
     coeff=jac2_inv_t[cols * index_row];

  __syncthreads();

  for(int j=0; j<iter; j++)
  {
    int thx_mem = thx + j * MAX_COLS;
    if(thx_mem < cols)
    {
       coeff_diag = jac2[ cols + thx_mem ];
       if(coeff_diag!=0)
          jac2_inv_t[cols * index_row + thx_mem ] =
          (jac2_inv_t[cols * index_row + thx_mem ]
          + coeff * (- jac2[ thx_mem ])) / coeff_diag;
    }
  }
}

__global__ void deviceJac2Inv_T(float *jac2_inv_t,float *jac2, int cols )
{

  int thx = threadIdx.x;
  int index_row = blockIdx.x + 1;
  int iter = (cols + MAX_COLS - 1) / MAX_COLS;

  __shared__ float coeff_tot;
  __shared__ float coeff_diag;

  if(thx==0)
  {
    coeff_tot = - jac2[ index_row ];
    coeff_diag = jac2[ cols + index_row ];
  }
```

```
  __syncthreads();

  if(coeff_diag!=0)
  {
     for(int j=0; j<iter; j++)
     {
       int thx_mem = thx + j * MAX_COLS;
       if(thx_mem < cols )
       {
          jac2_inv_t[cols * thx_mem + index_row ] =
          (jac2_inv_t[cols * thx_mem + index_row]
          + jac2_inv_t[cols * thx_mem] * coeff_tot) / coeff_diag;
       }
     }
  }
}

__global__ void deviceJac2Inv_normale(float *jac2_inv_t,float *jac2, int cols )
{

  int thx = threadIdx.x;
  int index_row = blockIdx.x + 1;
  int iter = (cols + MAX_COLS - 1) / MAX_COLS;

  __shared__ float coeff_tot;
  __shared__ float coeff_diag;

  if(thx==0)
  {
    coeff_tot = - jac2[ index_row ];
    coeff_diag = jac2[ cols + index_row ];
  }
  __syncthreads();

  if(coeff_diag!=0)
  {
     for(int j=0; j<iter; j++)
     {
       int thx_mem = thx + j * MAX_COLS;
       if(thx_mem < cols )
       {
          jac2_inv_t[cols * index_row + thx_mem ] =
          (jac2_inv_t[cols * index_row + thx_mem ]
          + jac2_inv_t[cols*index_row + thx_mem] * coeff_tot) / coeff_diag;
       }
     }
  }
}

cudaError_t computeJac2Inv( float *jac2, float *jac2_inv_t, int cols)
```

```
{

  dim3 grid,block;

  grid.x = 1;
  block.x = MAX_COLS;

  CREATE_CUDA_EVENT_WAIT

  deviceJac2InvFirstRow <<< grid, block >>> (jac2_inv_t, jac2, cols);

  grid.x = 1;
  block.x=MAX_COLS;

  deviceJac2InvScaleFirstRow <<< grid, block >>> (jac2_inv_t,cols);

  grid.x = cols - 1;
  block.x=MAX_COLS;

  deviceJac2Inv <<< grid, block >>> (jac2_inv_t, jac2, cols);


  CUDA_KERNEL_WAIT_POINT
  DESTROY_CUDA_EVENT_WAIT

  return cudaGetLastError();
}

texture<float,1,cudaReadModeElementType> jac2_inv_tx;
texture<float,1,cudaReadModeElementType> esm_jac_tx;
texture<int,1,cudaReadModeElementType> b_start_indices_tx;
texture<int,1,cudaReadModeElementType> block_size_tx;

#define BLOCK 256
#define GRID 10000

__global__ void MulJ2invJac_First(float *esm_jac_pinv, int esm_jac_rows,
int esm_jac_pinv_cols, int jac2_inv_cols, int cols)
{

   int index_thread=threadIdx.x;
   int index_block=blockIdx.y*GRID + blockIdx.x;

   int iter=(cols + BLOCK - 1) / BLOCK;
   int thx_mem;

   for(int j=0; j<iter; j++)
   {
      thx_mem = index_thread + j * BLOCK;
      if(thx_mem<cols)
```

```
            esm_jac_pinv[thx_mem*esm_jac_pinv_cols+index_block] =
            tex1Dfetch(esm_jac_tx,2*index_block) *
            tex1Dfetch(jac2_inv_tx,thx_mem*jac2_inv_cols);
    }
}

__global__ void MulJ2invJac_second( float *esm_jac_pinv, int esm_jac_pinv_cols,
int jac2_inv_cols, int cols)
{

    int index_thread=threadIdx.x;
    int index_block=blockIdx.x;

    int iter=(cols + BLOCK - 1) / BLOCK;
    int thx_mem;

    for(int j=0; j<iter; j++)
    {
         thx_mem = index_thread + j * BLOCK;
         if(thx_mem<cols)
         {
            int block_size_loc=tex1Dfetch(block_size_tx,thx_mem-1);
            int index_pinv_base=esm_jac_pinv_cols*index_block
            + tex1Dfetch(b_start_indices_tx,thx_mem-1);
            float jac2_inv_value=tex1Dfetch(jac2_inv_tx,
            jac2_inv_cols*index_block+thx_mem);
            int index_esm_base=1+2*tex1Dfetch(b_start_indices_tx,thx_mem-1);
            if(index_thread!=0)
            {
               for( int j=0; j<block_size_loc; j++)
               {
                  esm_jac_pinv[index_pinv_base+j]+=jac2_inv_value
                  * tex1Dfetch(esm_jac_tx,index_esm_base+2*j);
               }
            }
         }
    }
}

cudaError_t callMulJ2invJac( float *esm_jac,float *jac2_inv,float *esm_jac_pinv,
int* b_start_indices,int* block_size, int esm_jac_rows, int esm_jac_pinv_cols,
int jac2_inv_cols,  int cols, int jac_inv_dim, int esm_jac_dim,
 int b_start_indices_dim, int block_size_dim)
{

    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
    cudaBindTexture(NULL, &jac2_inv_tx, jac2_inv, &channelDesc, jac_inv_dim );
    cudaBindTexture(NULL, &esm_jac_tx, esm_jac, &channelDesc, esm_jac_dim );

    channelDesc = cudaCreateChannelDesc<int>();
```

```
    cudaBindTexture(NULL, &b_start_indices_tx, b_start_indices,
    &channelDesc, b_start_indices_dim );
    cudaBindTexture(NULL, &block_size_tx, block_size,
    &channelDesc, block_size_dim );

    dim3 grid,block;

    block.x=BLOCK;
    grid.y=(esm_jac_rows + GRID - 1)/GRID;
    grid.x=GRID;

    CREATE_CUDA_EVENT_WAIT
    MulJ2invJac_first <<< grid,block >>> (esm_jac_pinv,esm_jac_rows,
    esm_jac_pinv_cols,jac2_inv_cols,cols);

    block.x=BLOCK;
    grid.x=cols;

    MulJ2invJac_second <<< grid,block >>> (esm_jac_pinv,esm_jac_pinv_cols,
    jac2_inv_cols,cols);

    CUDA_KERNEL_WAIT_POINT
    DESTROY_CUDA_EVENT_WAIT

    cudaUnbindTexture(jac2_inv_tx);
    cudaUnbindTexture(esm_jac_tx);
    cudaUnbindTexture(b_start_indices_tx);
    cudaUnbindTexture(block_size_tx);

    return cudaGetLastError();
}

#define CUDA_BLOCK_SIZE_1D 256

__global__ void deviceMulJ2invJac( float *jac2_inv_t, float *esm_jac_pinv_t,
int* b_start_indices, int* block_size, int w, int h, float * esm_jac_t )
{

  int thx = threadIdx.x;
  int x = blockIdx.x * CUDA_BLOCK_SIZE_1D + thx;
  int b_index = blockIdx.y;

  if( x < w )
  {
    __shared__ int start_idx_s, bsize_s;

    if(thx == 0)
    {
      start_idx_s = b_start_indices[b_index];
      bsize_s = block_size[b_index];
```

```
    }

    __syncthreads();

    int init_y = start_idx_s, end_y = start_idx_s + bsize_s;

    int block_iter = (bsize_s + CUDA_BLOCK_SIZE_1D - 1)/CUDA_BLOCK_SIZE_1D;

    __shared__ float as[2*CUDA_BLOCK_SIZE_1D];
    __shared__ float bs[2*CUDA_BLOCK_SIZE_1D];

    as[thx] = jac2_inv_t[x];
    as[CUDA_BLOCK_SIZE_1D + thx] = jac2_inv_t[w*(b_index + 1) + x];

    while(block_iter)
    {
      int limit_y = init_y + CUDA_BLOCK_SIZE_1D;
      if(limit_y > end_y)
        limit_y = end_y;

      if(init_y + thx < end_y)
      {
        bs[thx] = esm_jac_t[init_y + thx];
        bs[CUDA_BLOCK_SIZE_1D + thx] = esm_jac_t[h + init_y + thx];
      }

      if( thx == 0 && w - x  < limit_y - init_y )
        for (int y = init_y + w - x, i = w - x; y < limit_y ; y++,i++)
        {
          bs[i] = esm_jac_t[y];
          bs[CUDA_BLOCK_SIZE_1D + i] = esm_jac_t[h + y];
        }

      __syncthreads();

      for (int y = init_y , i = 0; y < limit_y ; y++,i++)
      {
        esm_jac_pinv_t[ w*y + x ] = as[thx]*bs[i] +
        as[CUDA_BLOCK_SIZE_1D + thx] * bs[CUDA_BLOCK_SIZE_1D + i];
      }

      init_y += CUDA_BLOCK_SIZE_1D;
      block_iter--;
    }
  }
}

cudaError_t mulJ2invJac( float *jac2_inv_t, float *esm_jac_t,
float *esm_jac_pinv_t, int* b_start_indices,int* block_size, int w, int h )
{
```

```
  dim3 grid,block;

  grid.x = (w + CUDA_BLOCK_SIZE_1D - 1)/CUDA_BLOCK_SIZE_1D;
  grid.y = w - 1; // Size of b_start_indices and block_size
  block.x = CUDA_BLOCK_SIZE_1D;

  CREATE_CUDA_EVENT_WAIT

  deviceMulJ2invJac <<< grid, block >>> (jac2_inv_t, esm_jac_pinv_t,
  b_start_indices,block_size, w,h, esm_jac_t);

  CUDA_KERNEL_WAIT_POINT
  DESTROY_CUDA_EVENT_WAIT

  return cudaGetLastError();
}

cudaError_t mulJacTJac( float *jac2, float *esm_jac_t,int* b_start_indices,
int* block_size, int cols, int esm_jac_t_cols,
 float *jac2_inv_t, float * esm_jac_pinv_t )
{

  dim3 grid,block;

  grid.x = cols-1;
  block.x = MAX_BLOCK;

  CREATE_CUDA_EVENT_WAIT

  deviceMulJacTJac <<< grid, block >>> (jac2, b_start_indices,
  block_size, cols, esm_jac_t_cols, esm_jac_t);

  grid.x=1;
  block.x=MAX_COLS;

  deviceSumFirstEl <<< grid, block >>> (jac2,cols);

  grid.x = 1;
  block.x = MAX_COLS;

  deviceJac2InvFirstRow_new <<< grid, block >>> (jac2_inv_t, jac2, cols);

  grid.x = 1;
  block.x=MAX_COLS;

  deviceJac2InvScaleFirstRow <<< grid, block >>> (jac2_inv_t,cols);

  grid.x = cols;
  block.x=MAX_COLS;
```

```
  deviceJac2Inv_new <<< grid, block >>> (jac2_inv_t, jac2, cols);

  grid.x = (cols + CUDA_BLOCK_SIZE_1D - 1)/CUDA_BLOCK_SIZE_1D;
  grid.y = cols - 1;
  block.x = CUDA_BLOCK_SIZE_1D;

  deviceMulJ2invJac <<< grid, block >>> (jac2_inv_t, esm_jac_pinv_t,
  b_start_indices,block_size, cols ,esm_jac_t_cols, esm_jac_t);

  CUDA_KERNEL_WAIT_POINT

  DESTROY_CUDA_EVENT_WAIT

  return cudaGetLastError();
}

void ESMTracker::efficientPseudoinverse2(CvMat * esm_jac,
CvMat * esm_jac_pinv, int esm_jac_cols, int *block_size, CvMat * jac2__ )
{
    cudaError_t err=cudaSuccess;
    QTime timer;
    timer.start();
    int i,j;
    int rows = esm_jac->rows, cols = esm_jac_cols;

    int b_start_indices[cols - 1];
    b_start_indices[0] = 0;
    int max_block_size=0;
    int index_max=0;
    for(i = 1; i < cols - 1; i++)
    {
      b_start_indices[i] = b_start_indices[i-1] + block_size[i-1];
      if(block_size[i-1]>max_block_size)
      {
          max_block_size=block_size[i-1];
          index_max=i-1;
      }
    }

    CvMat *jac2 = cvCreateMat(2, cols, CV_32FC1);
    CvMat *esm_jac_pinv_float_t = cvCreateMat(esm_jac_pinv->cols,
    esm_jac_pinv->rows,CV_32FC1);
    CvMat *jac2_inv_t=cvCreateMat(cols,cols,CV_32FC1);
    CvMat *esm_jac_float=cvCreateMat(esm_jac->rows,esm_jac->cols,CV_32FC1);
    CvMat *esm_jac_float_t = cvCreateMat(esm_jac->cols,esm_jac->rows,CV_32FC1);

    cvSetIdentity(jac2_inv_t);
    cvConvert(esm_jac,esm_jac_float);
    cvTranspose(esm_jac_float, esm_jac_float_t);
```

```
    float *jac2_cuda,*esm_jac_cuda, *jac2_inv_t_cuda, *esm_jac_pinv_cuda;
    int *b_start_indices_cuda,*block_size_cuda;

    cudaMalloc((void**)&esm_jac_pinv_cuda,
    esm_jac_pinv->cols*esm_jac_pinv->rows*sizeof(float));
    cudaMalloc((void**)&jac2_inv_t_cuda, cols*cols*sizeof(float));
    cudaMalloc((void**)&jac2_cuda, jac2->cols*3*sizeof(float));
    cudaMalloc((void**)&esm_jac_cuda,
    esm_jac_float_t->cols*esm_jac_float_t->rows*sizeof(float));
    cudaMalloc((void**)&b_start_indices_cuda,(cols-1)*sizeof(int));
    cudaMalloc((void**)&block_size_cuda,(cols-1)*sizeof(int));

    cudaMemcpy(jac2_inv_t_cuda,jac2_inv_t->data.fl,
    cols*cols*sizeof(float),cudaMemcpyHostToDevice);
    cudaMemcpy(esm_jac_cuda,esm_jac_float_t->data.fl,
    esm_jac_float_t->cols*esm_jac_float_t->rows*sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(b_start_indices_cuda,b_start_indices,
    (cols - 1)*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(block_size_cuda,block_size,
    (cols - 1)*sizeof(int),cudaMemcpyHostToDevice);

    err=mulJacTJac(jac2_cuda,esm_jac_cuda,b_start_indices_cuda,block_size_cuda,
    cols,esm_jac_float_t->cols,jac2_inv_t_cuda,esm_jac_pinv_cuda);
    if(err!=cudaSuccess)
      qDebug("CUDA ERROR in mulJacTJac=%s",cudaGetErrorString(err));

    cudaMemcpy(esm_jac_pinv_float_t->data.fl,esm_jac_pinv_cuda,
    esm_jac_pinv->cols*esm_jac_pinv->rows*sizeof(float),cudaMemcpyDeviceToHost);
    cvTranspose(esm_jac_pinv_float_t, esm_jac_pinv);

    cvReleaseMat(&jac2);
    cvReleaseMat(&jac2_inv_t);
    cvReleaseMat(&esm_jac_float);
    cvReleaseMat(&esm_jac_float_t);
    cvReleaseMat(&esm_jac_pinv_float_t);

    cudaFree(esm_jac_pinv_cuda);
    cudaFree(jac2_inv_t_cuda);
    cudaFree(jac2_cuda);
    cudaFree(esm_jac_cuda);
    cudaFree(b_start_indices_cuda);
    cudaFree(block_size_cuda);
}
```

Bibliografia

[1] Nvidia cuda home page: www.nvidia.com/object/cuda_home.html.

[2] Henrik Andreasson, Tom Duckett, and Achim Lilienthal. Mini-slam: Minimalistic visual slam in large-scale environments based on a new interpretation of image similarity. In *Proc. of the 2007 IEEE International Conference on Robotics and Automation (ICRA)*, 2007.

[3] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1–16, 2007.

[4] E. Eade and T. Drummond. Scalable monocular slam. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 469–476, June 2006.

[5] E. D. Eade and T. W. Drummond. Edge landmarks in monocular slam. In *British Machine Vision Conference (BMVC)*, volume 1, pages 469–476, 2006.

[6] P. Sim R. Elinas and J. Little. slam: Stereo vision slam using the rao-blackwellised particle filter and a novel mixture proposal distribution. In *Proc. of the 2006 IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[7] Klein G. and D. Murray. Improving the agility of keyframe-based slam. In *In Proc. European Conference on Computer Vision (ECCV, Marseille)*, 2008.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix computation*. The Johns Hopkins University Press, 1989.

[9] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Robotics: Science and Systems (RSS)*, Atlanta, GA, USA, 2007.

[10] H. Jin, P. Favaro, and S. Soatto. A semi-direct approach to structure from motion. *The Visual Computer*, 19:1–18, 2003.

[11] Il-Kyun Jung and Simon Lacroix. High resolution terrain mapping using low altitude aerial stereo imagery. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 946, Washington, DC, USA, 2003. IEEE Computer Society.

[12] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 2010.

[13] Yi Ma, Stefano Soatto, Jana Kosecka, and S. Shankar Sastry. *An Invitation to 3D Vision*. Springer, 2004.

[14] B. Micusik and J. Kosecka. Piecewise planar city modeling from street view panoramic sequences. In *IEEE conference on Computer Vision and Pattern Recognition*, 2009.

[15] Michael Montemerlo and Sebastian Thrun. *FastSLAM: A Scalable Method for the Simultaneous Localization and Mapping Problem in Robotics*. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, January 2007.

[16] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *CUBLAS Library*, February 2010.

[17] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *CUDA C Best Practices Guide*, August 2010.

[18] NVIDIA Corporation, 2701 San Tomas Expressway Santa Clara, CA 95050. *NVIDIA CUDA C Programming Guide*, October 2010.

[19] A. Pretto, Soatto S., and Menegatti E. Scalable dense large-scale mapping and navigation. In *Proceddings of: Workshop on Omnidirectional Robot Vision (ICRA 2010).*, 2010.

[20] G. Silveira, E. Malis, and P. Rives. An efficient direct method for improving visual SLAM. In *Proc. of the 2007 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4090–4095, Italy, 2007.