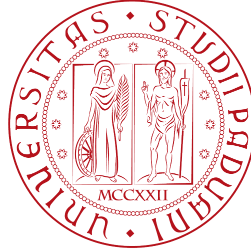
Corso di Laurea Magistrale in Ingegneria Informatica



Studio e analisi su GPU del Gradiente Coniugato con Precondizionatore FSAI

Relatori:
Dott. Carlo Fantozzi e
Dott. Carlo Janna

Laureando: **Andrea Castronuovo**
Matr. 626614

A. A. 2012 - 2013

Sommario

Nel campo dell'ingegneria civile, dell'ingegneria meccanica e di molte altre scienze applicate esistono numerosi problemi che sono riconducibili alla risoluzione di sistemi lineari. Presso il dipartimento ICEA dell'Università di Padova è stato sviluppato un solutore in linguaggio FORTRAN che mediante le direttive OpenMP è in grado di sfruttare il parallelismo della piattoforma multiprocessore a memoria condivisa.

Il solutore adottato è noto in letteratura come metodo del gradiente coniugato modificato. Il motivo della scelta di tale solutore, oltre a fatto che è facilmente parallelizzabile, è la sua efficienza nella risoluzione di sistemi lineari sparsi, simmetrici e definiti positivi. La sua efficienza cresce ulteriormente se viene utilizzata un'opportuna matrici di precondizionamento, o precondizionatore.

L'obiettivo delle tesi è modificare il metodo del gradiente coniugato modificato in modo da sfruttare l'iperparallelismo delle GPU (*Graphics Processors Unit*), riducendo il tempo di calcolo per determinare la soluzione del sistema lineare. Il calcolo viene spostato sulla scheda grafica utilizzando la libreria CUDA (*Compute Unified Device Architecture*).

L'impiego della GPU nell'esecuzione del gradiente coniugato modificato ha portato a importanti benefici. In particolare nei test condotti i tempi di esecuzione del solutore su GPU sono diminuiti rispetto ai tempi di esecuzione del solutore utilizzando 4 processori sino a ottenere uno speed up assoluto superiore a 2.

Indice

1 Introduzione **1**

1.1 Tipi di solutori di sistemi lineari . . . 2

1.2 GP-GPU: General-Purpose computing on GPU . . . 3

2 Metodo del Gradiente Coniugato Modificato **5**

2.1 Memorizzazione di una matrice sparsa . . . 6

2.1.1 COO (COOrdinate) . . . 7

2.1.2 ELLPACK . . . 8

2.1.3 Hybrid . . . 8

2.1.4 CSR (Compressed Sparse Row) . . . 10

2.2 Rivisitazione del prodotto matrice-vettore per il formato CSR . . . 11

2.3 Test . . . 12

3 Precondizionatore FSAI **17**

3.1 Calcolo del precondizionatore FSAI . . . 20

4 CUDA: Compute Unified Device Architecture **25**

4.1 Architettura . . . 26

4.2 Modello di programmazione . . . 26

4.3 Memoria . . . 33

4.3.1 Global memory . . . 33

4.3.2 Constant memory . . . 34

4.3.3 Texture Memory . . . 34

4.3.4 Shared memory . . . 35

4.3.5 Local memory . . . 36
4.3.6 Registri . . . 36
4.4 Coalescenza . . . 37

5 Gradiente Coniugato Modificato con le librerie CUBLAS e CUSPARSE 39
5.1 Modello di programmazione . . . 41
5.2 Test . . . 43

6 Calcolo del Precondizionatore FSAI con la libreria CUDA 49
6.1 Riduzione dei sistemi da risolvere . . . 52
6.2 Risoluzione di una sequenza di sistemi . . . 53

7 Conclusioni 63

A Codice OpenMP 67

B Codice con CUSPARSE e CUBLAS 107

C Codice del calcolo del precondizionatore FSAI 125

Elenco degli algoritmi

2.1 Metodo del Gradiente Coniugato 6
2.2 Prodotto matrice-vettore per il formato *CSR* 12
2.3 Metodo del Gradiente Coniugato Modificato 15
3.1 Calcolo della matrice G 23
6.1 Fase di triangolarizzazione nel metodo di eliminazione di Gauss 57
6.2 Fase di sostituzione all'indietro nel metodo di eliminazione di Gauss 57

Elenco delle figure

1.1 Confronto di sviluppo tra il trend delle GPU e CPU 4

2.1 Rappresentazione della matrice A in formato *COO* 9
2.2 Rappresentazione di una matrice A in formato *ELLPACK* 9
2.3 Rappresentazione della matrice A in formato *CSR* 11
2.4 Speed up del metodo del Gradiente Coniugato Modificato 16

3.1 Raccolta dei sistemi $A[\mathcal{G},\mathcal{G}]$ 23

4.1 Schema di uno Streaming Multiprocessor (SM) con architettura Fermi [17]. . . . 27
4.2 Sequenza di esecuzione di un programma C che utilizza la libreria CUDA [17] . . 29
4.3 Trasparent Scalability [11] 30
4.4 Struttura della memoria per un SM 32

5.1 Speed up del metodo del Gradiente Coniugato Modificato 45
5.2 Speed up del metodo del Gradiente Coniugato Modificato per la matrice *Cubo_4*. 47

6.1 Tempi di esecuzione della raccolta dei sistemi e del loro trasferimento 51
6.2 Confronto tra i tempi di esecuzione della raccolta e del traferimento dei sistemi ottenuti mediante l'utilizzo dell'algoritmo euristico 54
6.3 Speed up risoluzione di una sequenza di 20000 sistemi 60

Elenco delle tabelle

2.1 Caratteristiche delle matrici utilizzate per i test 13

2.2 Statistiche dei test 15

5.1 Caratteristiche della scheda grafica *GeForce GTS 450* 44

5.2 Statistiche dei test condotti sul Gradiente Coniugato Modificato (T_e $[s]$ tempo di esecuzione, n_i numero di iterazioni e S_a speed up assoluto) 45

5.3 Statistiche dei test condotti sul metodo del Gradiente Coniugato Modificato con la matrice *Cubo_4* (T_e $[s]$ tempo di esecuzione, n_i numero di iterazioni e S_a speed up assoluto). 47

6.1 Statistiche della raccolta dei sistemi su CPU e del loro trasferimento su GPU . . 51

6.2 Numero di sistemi da raccogliere al variare di α 53

6.3 Statistiche della raccolta dei sistemi su CPU e loro trasferimento su GPU 54

6.4 Tempi di esecuzione per la risoluzione di 20000 sistemi 61

Capitolo 1

Introduzione

Nel campo dell'ingegneria civile, meccanica e di molte altre scienze applicate esistono svariati problemi che sono riconducibili alla risoluzione di sistemi lineari di n equazioni in n incognite

$$Ax = b \tag{1.1}$$

dove $A \in \mathbb{R}^{n \times n}$ è una matrice simmetrica e definita positiva[1] (SPD), mentre $x, b \in \mathbb{R}^n$ sono due vettori chiamati, rispettivamente, vettore soluzione e termine noto. Il punto di partenza della presente tesi è un solutore sviluppato presso il dipartimento *ICEA* dell'Università di Padova; scritto in linguaggio *FORTRAN* mediante le direttive *OpenMP* [4] è in grado di sfruttare il parallelismo delle piattaforme multiprocessore a memoria condivisa, ottenendo sensibili miglioramenti rispetto all'esecuzione sequenziale.

L'obiettivo della tesi è modificare il solutore per sfruttare l'iperparallelismo delle GPU (*Graphics Processing Unit*) riducendo, quindi, ulteriormente i tempi di calcolo. Il calcolo viene spostato sulla scheda grafica utilizzando la libreria CUDA (*Compute Unified Device Architecture*). CUDA è una libreria messa a disposizione da NVIDIA, che permette al programmatore di interfacciarsi e sfruttare le enormi potenzialità delle GPU. A causa dell'architettura SIMD (*Single Instruc-*

[1]Una matrice quadrata A si dice definita positiva, se ogni vettore non nullo x soddisfa la disuguaglianza $x^T Ax > 0$.

tion Multiple Data) delle schede grafiche molti degli algoritmi sviluppati su CPU per operare sulle matrici sono stati rivisitati e adattati in modo da poter funzionare efficientemente su GPU. Il modello SIMD prevede un'unica unità di controllo (CU) che gestisce più unità di calcolo (ALU) operanti in maniera sincrona. Ad ogni passo tutti gli elementi eseguono la stessa istruzione, ma ciascuno su un dato differente.

La tesi è strutturata in modo da seguire l'ordine cronologico dello studio effettuato. Per ogni capitolo, escluso il primo e il quarto, ci sarà una prima parte descrittiva, nella quale verrà spiegato l'algoritmo adottato o eventuali modifiche apportate, e una seconda parte nella quale verranno descritti e commentati i test effettuati. Dopo una rapida introduzione sui diversi tipi di solutori presenti in letteratura e sui motivi che hanno portato all'implementazione di un solutore per GPU (sez. (1.1) e (1.2) del Capitolo 1), si passa alla descrizione del gradiente coniugato modificato mostrando i benefici del parallelismo su CPU utilizzando la libreria OpenMP (Capitolo 2). Successivamente si prosegue con l'illustrazione del calcolo del precondizionatore FSAI (*Factorized Sparse Approximate Inverse*), fondamentale per ottenere la convergenza del gradiente coniugato modificato in un numero relativamente piccolo di passi (Capitolo 3). In seguito si introducono l'architettura delle GPU e la libreria CUDA (Capitolo 4), necessari per la comprensione della successiva implementazione del gradiente coniugato modificato su GPU (Capitolo 5). Infine si conclude con l'implementazione di alcune fasi del calcolo del precondizionatore su GPU (Capitolo 6).

1.1 Tipi di solutori di sistemi lineari

La soluzione del sistema descritto dall'equazione (1.1) è data dal vettore

$$x = A^{-1}b \tag{1.2}$$

dove la soluzione x è unica se la matrice A è invertibile. Il calcolo della soluzione di un sistema lineare è un operazione piuttosto onerosa computazionalmente.

I metodi numerici per la soluzione di sistemi lineari si suddividono in due classi: metodi *diretti* e metodi *iterativi*. Con i metodi diretti si ottiene la soluzione cercata in un numero finito di passi; di solito tali metodi sono efficienti per matrici *dense*, cioè con pochi elementi diversi da zero. Diversamente, i metodi iterativi sono utilizzati per la soluzione di sistemi lineari *sparsi*, in cui la matrice del sistema possiede un numero elevato di elementi uguali a zero. Tali metodi generano una successione infinita di vettori che convergono alla soluzione del sistema.

I metodi diretti compiono un numero finito di combinazioni lineari tra le righe della matrice sino a ottenere una matrice triangolare, la quale è semplice da risolvere. Queste operazioni, generalmente, portano alla crescita dei termini diversi da zero, con un conseguente aumento dello spazio necessario per la memorizzazione della matrice. Il fenomeno è noto in letteratura come *fill-in*, cioè la riduzione della sparsità della matrice stessa. Perciò è indispensabile l'uso dei metodi iterativi che lasciano inalterata la matrice del sistema e forniscono vettori convergenti alla soluzione del sistema, permettendo all'utente di impostare l'accuratezza desiderata.

I modelli matematici che descrivono i problemi di interesse per numerose applicazioni nell'ambito delle scienze applicate forniscono matrici sparse. Per questo motivo è stato implementato il Gradiente Coniugato Modificato (GCM), un metodo iterativo, per la risoluzione di sistemi lineari. Caratteristica importante di tale metodo è che esso è costituito da una sequenza di prodotti matrice-vettore o scalari, i quali sono facilmente parallelizzabili.

1.2 GP-GPU: General-Purpose computing on GPU

La GPGPU è un settore dell'informatica che sfrutta la GPU per motivi diversi dalla ricostruzione di immagini tridimensionali. La GPU viene, solitamente, impiegata per l'elaborazione di immagini e richiede un'enorme potenza di calcolo per lo più di tipo parallelo. A partire dal 2007 NVIDIA ha cominciato a produrre schede grafiche programmabili, cioè GPU in grado di supportare l'esecuzione di programmi. L'importanza di conoscere questo modello di programmazione, oltre ai vantaggi dettati dal poter sfruttare il parallelismo delle GPU, consiste nel fatto

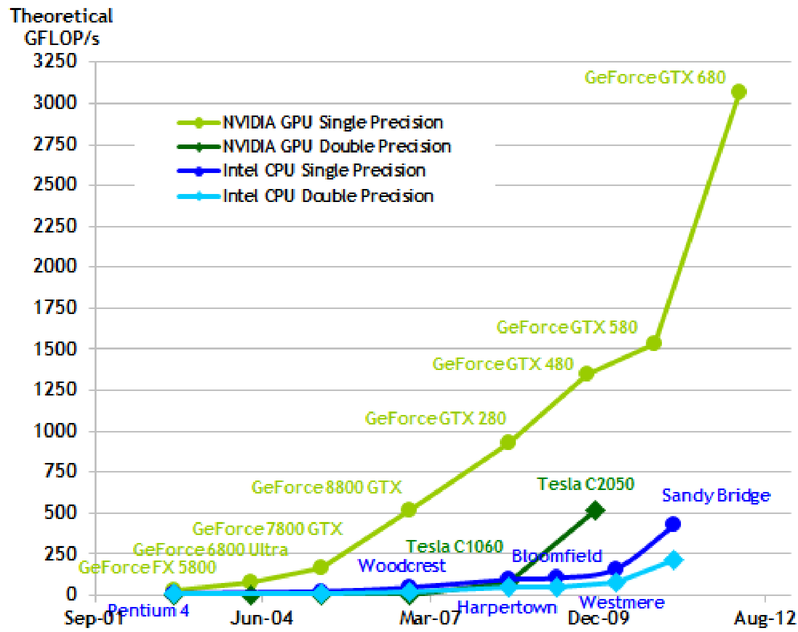


Figura 1.1: Confronto di sviluppo tra il trend delle GPU e CPU

che le GPU stesse sono tuttora in grande sviluppo. Le CPU sono sostanzialmente ad uno stato di sviluppo statico in termini di prestazioni del singolo core. Diversamente le schede grafiche stanno ancora seguendo la curva che ha caratterizzato la crescita dei PC fino agli ultimi anni. La figura 1.1 mostra il trend di sviluppo degli ultimi anni, in termini di potenza di calcolo, delle CPU e delle GPU. La differenza è netta tra i due modelli di programmazione e questo evidenzia l'importanza e le potenzialità delle schede grafiche. Tale crescita nelle prestazioni è dovuta all'aumento del numero di core impiegati: nelle GPU questo aumento è legato alla struttura massicciamente parallela richiesta dai calcoli grafici.

Capitolo 2

Metodo del Gradiente Coniugato Modificato

Il metodo del gradiente coniugato modificato, introdotto dai matematici M. Hestens (1906-1991), E. Stiefel (1909-1978) [1] e C. Lanczos [6], risolve sistemi lineari del tipo (1.1). Tale metodo può essere scomposto in due parti: calcolo del precondizionatore e metodo del gradiente coniugato. Il calcolo del precondizionatore sarà analizzato e spiegato nel capitolo 3, mentre il metodo del gradiente coniugato è riassumibile nei passi illustrati nell'algoritmo 2.3. L'algoritmo, mediante un insieme di prodotti matrice-vettore e di prodotti scalari, ottiene ad ogni ciclo il vettore x_k che ad ogni iterazione si avvicina sempre di più alla soluzione cercata. Il metodo del gradiente coniugato accetta in ingresso quattro parametri: la matrice A, il vettore b, il numero massimo di iterazioni e la tolleranza di uscita. In particolare l'algoritmo converge quando la norma del residuo $\|\tau_{k+1}\| = b - A \cdot x_k$ è inferiore alla tolleranza. Nel primo passo si inizializza il vettore x_0 a un valore arbitrario, successivamente l'algoritmo continua a iterare sino a ottenere un valore di τ_{k+1} inferiore al valore della soglia di convergenza. A causa degli errore di arrotondamento nei calcolatori il metodo del gradiente coniugato potrebbe anche non convergere alla soluzione cercata, perciò viene imposto il termine dell'algoritmo dopo un numero massimo di iterazioni. Diversamente in assenza di errori di arrotondamento l'algoritmo converge alla soluzione cercata in al più n passi, dove

Algoritmo 2.1 Metodo del Gradiente Coniugato

1. Choose x_0
2. $r_0 = b - Ax_0$
3. $p_0 = r_0$
4. **for** $k = 0, \ldots$,until convergence $(\tau_k < toll)$ **do**
5. $\quad t = Ap_k$
6. $\quad \alpha_k = r_k^T p_k / p_k^T t$
7. $\quad x_{k+1} = x_k + \alpha_k p_k$
8. $\quad r_{k+1} = r_k - \alpha_k t$
9. $\quad \beta_k = -r_{k+1}^T t / p_k^T t$
10. $\quad p_{k+1} = r_{k+1} + \beta_k p_k$
11. $\quad \tau_{k+1} = \|r_{k+1}\|$
12. **end for**

n è pari alla dimensione della matrice.
Uno degli aspetti positivi del gradiente coniugato sta nella sua semplicità e nel ridotto costo computazionale richiesto ad ogni iterazione, essendo l'operazione computazionalmente più dispendiosa il prodotto matrice-vettore (punto *5.* dell'algoritmo 2.1). L'uso del gradiente coniugato è indicato per problemi descritti da una matrice sparsa e per sfruttare tale sparsità è opportuno che la memorizzazione di A sia in forma compatta, per evitare un elevato e inutile dispendio di calcolo ad ogni iterazione dell'algoritmo. Pertanto, sarà indispensabile una rivisitazione del prodotto matrice-vettore (sez. 2.2), che avrà un costo inferiore a $O(n^2)$. In seguito verrano descritti i diversi tipi di formati per la memorizzazione di una matrice sparsa.

2.1 Memorizzazione di una matrice sparsa

Una matrice sparsa, come precedentemente accennato, è una matrice con un'alta percentuale di elementi nulli, superiore al 99% per alcuni problemi ingegneristici.

Risulta pertanto indispensabile memorizzare la matrice in un formato diverso dal formato tabellare esemplificato in (2.1). In letteratura esistono diversi tipi di formati per la memorizzazione di matrici di questo tipo e in seguito verrano esposti i formati utilizzati nell'implementazione del gradiente coniugato. Nelle successive sezioni per chiarire la struttura dei diversi formati, si utilizzerà la seguente matrice come esempio numerico

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \tag{2.1}$$

Inoltre i diversi tipi di formati verranno descritti per matrici quadrate $n \times n$ di tipo non simmetrico, aventi un numero di termini diversi da zero pari a n_t e un numero massimo di termini non nulli per riga pari a k.

2.1.1 COO (COOrdinate)

Il formato *COO* è un formato di memorizzazione estremamente semplice che prevede la generazione di tre vettori di lunghezza pari a n_t:

- *row*: vettore di numeri interi contenente gli indici di riga degli elementi non nulli della matrice A;
- *col*: vettore di numeri interi contenente gli indici di colonna degli elementi non nulli della matrice A;
- *data*: vettore di numeri reali contenente i valori dei termini non nulli della matrice A.

Questo tipo di formato richiede uno spazio di memoria proporzionale al numero di termini non nulli. In particolare, vengono salvati in memoria $2 \cdot n_t$ numeri interi (*row*, *col*) e n_t numeri reali (*data*). In figura 2.1 vengono riportati i tre vettori (*row*, *col* e *data*) per la matrice di esempio.

2.1.2 ELLPACK

Il formato *ELLPACK* [9] memorizza due matrici dense $n \times k$, dove k è il massimo numero di termini non nulli in una riga della matrice originale:

- *data*: matrice contenente i valori reali dei termini non nulli presi in ordine *row-major*. Per righe con un numero di termini diversi da zero inferiore a k vengono aggiunti dei termini di padding, ossia dei termini nulli.

- *indices*: matrice contente gli indici dei termini non nulli presi in ordine *row-major*. Analogamente alla matrice *data*, per righe con un numero di nonzeri inferiore a k vengono aggiunti dei termini di padding che, diversamente dal caso precedente, sono termini "*sentinella*" (i termini "*sentinella*" sono termini che non hanno significato per la matrici presa in considerazione, nel nostro caso potrebbero assumere valori negativi).

Si noti che questo tipo di memorizzazione per matrici sparse è efficiente se il numero massimo di termini diverso da zero è uguale, o differisce di poco, dal numero medio di termini non nulli. Inoltre per il formato *ELLPACK* gli indici di riga sono implicitamente definiti, mentre gli indici di colonna sono salvati esplicitamente nella matrice *indices*. Il termine a_{ij} della matrice A sarà posizionato nella i-esima riga delle due matrici e in posizione l, dove $indices(l) = j$ e $data(l) = a_{ij}$. Per maggior chiarezza si osservi la figura 2.2 dove si è utilizzato la rappresentazione della matrice di esempio in formato *ELLPACK* con un numero massimo di 3 termini non nulli per riga.

2.1.3 Hybrid

Il formato *Hybrid* combina insieme i formati *COO* e *ELLPACK*. Come specificato nell'articolo [9], mentre il formato *ELLPACK* è efficiente quando il numero medio e massimo di termini diversi da zero per riga non differiscono in maniera netta, la sua efficienza degrada rapidamente quando il numero di non zeri per riga varia. Diversamente l'efficienza del formato *COO* è invariante dalla distribuzione dei termini non nulli per riga. Perciò il formato *Hybrid* memorizza in formato

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$row = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix}$$
$$col = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$
$$data = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

Figura 2.1: Rappresentazione della matrice A in formato *COO*

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$$
$$indices = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

Figura 2.2: Rappresentazione di una matrice A in formato *ELLPACK*

ELLPACK un numero di colonne M in modo da avere le matrici *data* e *indices* piene, o con pochi elementi di padding, e il restante in formato *COO*. Il formato *Hybrid* cerca di sfruttare gli aspetti positivi di entrambi i formati.

2.1.4 CSR (Compressed Sparse Row)

Il formato *CSR* memorizza tre vettori, rispettivamente, di lunghezza pari a nt per i primi due vettori e $n+1$ per il terzo. Tali vettori sono descritti in seguito.

- ja: vettore di numeri interi contenente gli indici di colonna dei termini non nulli della matrice A memorizzati in ordine *row-major.*
- $coef$: vettore di numeri reali contenente il valore dei termini non nulli della matrice A memorizzati in ordine *row-major.*
- iat: vettore di numeri interi con una lunghezza pari a $n+1$ contenente le posizioni in cui si trova il primo elemento di ciascuna riga di A nel vettore $coef$.

L'uso combinato dei vettori ja e iat consente di individuare qualsiasi elemento non nullo a_{ij} memorizzato in $coef$. In particolare, l'elemento a_{ij} si troverà in una posizione l del vettore $coef$ compresa nell'intervallo $iat(i) \leq l < iat(i+1)$ e tale per cui $ja(l) = j$. Queste due condizioni permettono di determinare univocamente l'indice l per il quale $coef(l) = a_{ij}$. Lo spazio per la memorizzazione della matrice A è notevolmente ridotto rispetto alla memorizzazione tabellare, infatti si è passati da n^2 numeri reali a n_t numeri reali e $n_t + n + 1$ numeri interi, dove $n_t \ll n^2$.

Per chiarire come viene memorizzata una matrice A con questo formato, si osservi la figura 2.3. Si noti che l'ultima componente del vettore iat, pari a n_t, è indispensabile per individuare le componenti dell'ultima riga. Infatti l'elemento a_{ij} dell'ultima riga, avente indice p, appartiene all'intervallo $iat(n) \leq p < iat(n+1)$.

Questo tipo di formato è stato scelto nell'implementazione del gradiente coniugato su CPU, ottenendo così un considerevole risparmio di memoria e una conseguente diminuzione del tempo di accesso alla matrice che se memorizzata in forma tabellare non sarebbe stata contenuta in memoria centrale per i problemi di nostro interesse.

$$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

$$coef = \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix}$$

$$ja = \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix}$$

$$iat = \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix}$$

Figura 2.3: Rappresentazione della matrice A in formato *CSR*

2.2 Rivisitazione del prodotto matrice-vettore per il formato CSR

L'operazione prodotto matrice-vettore risulta essere piuttosto banale per matrici memorizzate in maniera tabellare. Utilizzando il formato CSR, tale operazione si complica leggermente.

Per descrivere il funzionamento di questa operazione con il formato *CSR*, consideriamo il prodotto matrice-vettore

$$Av = w$$

con A matrice quadrata $n \times n$ memorizzata in formato *CSR*, v e w vettori in $\mathbb{R}^n$. Poiché il prodotto matrice-vettore è composto da n prodotti scalari, si analizzi il prodotto scalare tra un vettore memorizzato in formato *CSR* e uno in maniera tabellare. Per questa analisi si utilizzi il prodotto dell'i-esima riga a_i della matrice A con il vettore v, il quale restituisce l'i-esima componente del vettore w. Questa operazione è riassunta nella seguente formula

$$w_i = \sum_{l=1}^{\#\, non-zeri\; riga\; i} a_{il} v_l$$

Algoritmo 2.2 Prodotto matrice-vettore per il formato *CSR*

1. **for** $i = 0, ..., n-1$ **do**
2. $\quad w_i = 0$
3. $\quad$ **for** $k = iat(i), ..., iat(i) - 1$ **do**
4. $\quad\quad j = ja(k)$
5. $\quad\quad w_i = w_i + coef(k) \cdot v_j$
6. $\quad$ **end for**
7. **end for**

L'elemento del vettore a_i di indice l è memorizzato nel vettore $coef$ ed è individuato attraverso il vettore iat. In particolare, l'indice l dell'elemento a_i è compreso nell'intervallo $iat(i) \leq l < iat(i+1)$. Invece l'indice colonna j è determinato con il vettore ja, in particolare $ja(l) = j$.

Lo schema del prodotto matrice-vettore è riassunto nei passi dell'algoritmo 2.2.

2.3 Test

In questa sezione sono esposti i test effettuati sul metodo del gradiente coniugato modificato, perché il solo metodo del gradiente coniugato, in genere, non riesce a convergere. Il metodo del gradiente coniugato modificato è riassunto nei passi illustrati nell'algoritmo 2.3. I due metodi differiscono in due punti, in particolare nei punti *3.* e *9.* del metodo del gradiente coniugato modificato è presente la moltiplicazione per la matrice K^{-1}, ossia il precondizionatore (il calcolo del precondizionatore FSAI sarà descritto nel successivo capitolo). Il precondizionatore riesce a ridurre il numero di iterazioni e di conseguenza il tempo di esecuzione del gradiente coniugato modificato stesso.

Per i test di questo e dei successivi capitoli si sono utilizzati un processore *Intel Core i7 950@3.07GHz* e una scheda grafica *GeForce GTS 450*. Per questa sezione verrà utilizzato il solo processore, poiché questi test sono stati condotti per verificare il parallelismo su CPU mediante la libreria OpenMP.

Per la misura delle performance, in termini di tempi di esecuzione, sono state condotte 10 prove per ogni configurazione e in seguito verranno rappresentati in forma tabellare i seguenti dati:

- numero di processori impiegati (n_p);
- media dei tempi di esecuzione delle 10 prove (T_e);
- numero di iterazioni (n_i);
- speed up assoluto ($S_a = \frac{T_1}{T_i}$), cioè il rapporto tra il tempo di esecuzione su singolo processore e il tempo di esecuzione con i processori.

Prima di ogni test sarà chiarito cosa si intende per configurazione, in particolare nella sezione corrente una configurazione è composta da due termini variabili: matrice di elementi in virgola mobile in doppia precisione (*double*) e numero di processori. Le matrici, utilizzate per i test, hanno caratteristiche che sono riassunte nella tabella 2.1 e il numero di processori può assumere i valori: 1, 2 e 4.

Nome della matrice	**Dimensione**	**# non-zeri**
Cubo_4	190581	7531389
Emilia_grande	923136	41005206
Stoc-F_1465	1465137	21005389

Tabella 2.1: Caratteristiche delle matrici utilizzate per i test

I risultati ottenuti sono riportati nella tabella 5.2. Si noti che al variare del numero di processori il numero di iterazioni non cambia, come, peraltro, ci si aspettava. Inoltre si osservi, come esposto dalla figura 5.1, che non si ottiene uno speed up assoluto perfetto, cioè raddoppiando il numero di processori lo speed up assoluto non raddoppia. Questo fenomeno è motivato dal costo eccessivo, legato al processore, nel portare i dati in memoria rispetto al tempo necessario per eseguire il calcolo effettivo. Si consideri il prodotto tra una matrice A memorizzata in formato *CSR* e un vettore. Per tale operazione è necessario effettuare tre accessi per individuare un solo elemento del vettore memorizzato in formato *CSR* e solo in seguito si può effettuare la moltiplicazione per il vettore.

Nel prossimo capito si introdurrà il concetto di precondizionatore e in particolare si concentrerà l'attenzione sul precondizionatore FSAI.

Algoritmo 2.3 Metodo del Gradiente Coniugato Modificato

1. Choose x_0
2. $r_0 = b - Ax_0$
3. $p_0 = K^{-1}r_0$
4. **for** $k = 0, \dots$, until convergence ($\tau_k < toll$) **do**
5. $t = Ap_k$
6. $\alpha_k = r_k^T p_k / p_k^T t$
7. $x_{k+1} = x_k + \alpha_k p_k$
8. $r_{k+1} = r_k - \alpha_k t$
9. $v = K^{-1} r_{k+1}$
10. $\beta_k = -r_{k+1}^T t / p_k^T t$
11. $p_{k+1} = r_{k+1} + \beta_k p_k$
12. $\tau_{k+1} = \|r_{k+1}\|$
13. **end for**

n_p	$T_e\,[s]$	n_i	S_a
1	18,9	583	1
2	12,9	583	1,46
4	9,8	583	1,93

(a) Matrice *Cubo_4*

n_p	$T_e\,[s]$	n_i	S_a
1	92,2	772	1
2	68,1	772	1,35
4	54,8	772	1,68

(b) Matrice *Emilia_grande*

n_p	$T_e\,[s]$	n_i	S_a
1	81,4	634	1
2	58,0	634	1,40
4	45,0	634	1,81

(c) Matrice *Stoc-F_1465*

Tabella 2.2: Statistiche dei test

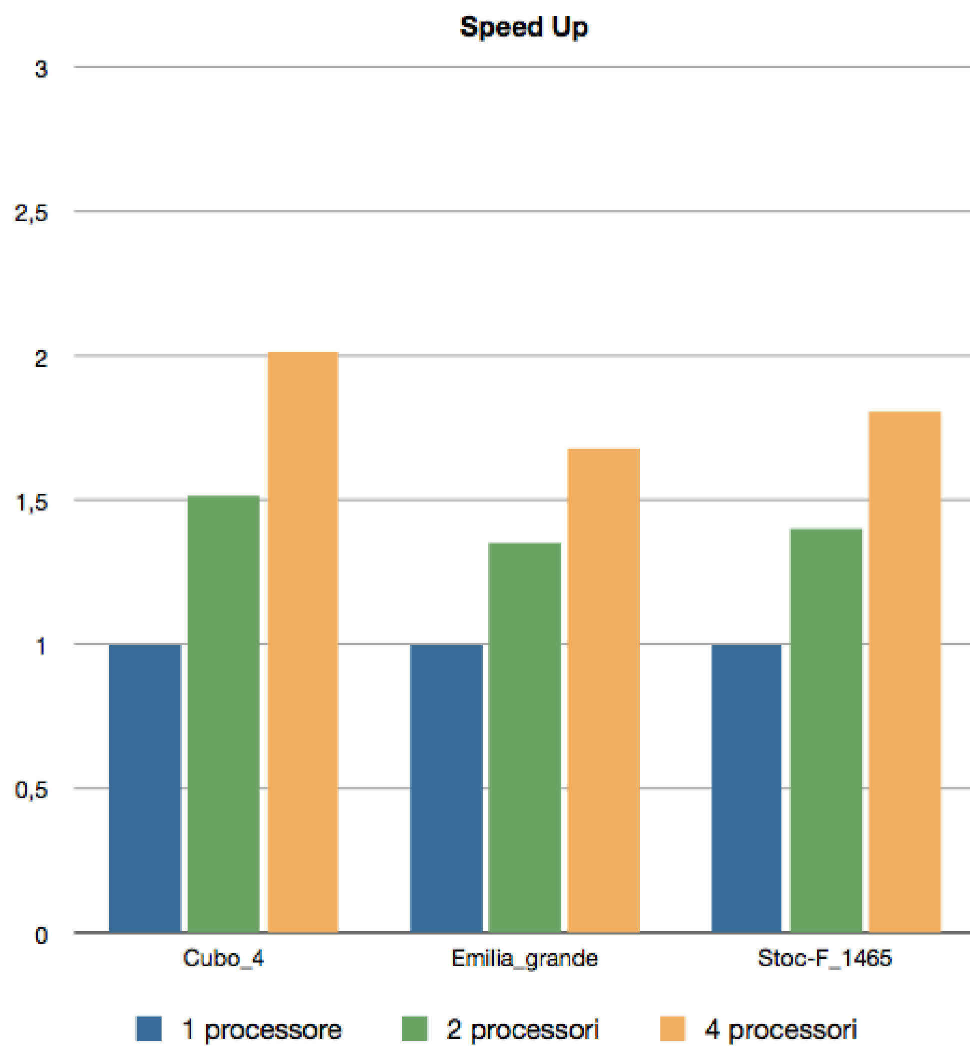


Figura 2.4: Speed up del metodo del Gradiente Coniugato Modificato

Capitolo 3

Precondizionatore FSAI

Uno dei motivi della scelta del metodo del gradiente coniugato modificato, oltre a essere facilmente parallelizzabile, è la sua efficienza nella risoluzione di sistemi lineari sparsi, simmetrici e definiti positivi. La sua efficienza cresce quando viene utilizzata un'opportuna matrice di precondizionamento, o precondizionatore. Questa matrice, indicata con il simbolo K^{-1} nell'algoritmo 2.3, riduce il numero di iterazioni del gradiente coniugato modificato, diminuendo il tempo di esecuzione del solutore. La matrice di precondizionamento per essere efficiente deve soddisfare le seguenti proprietà:

- il prodotto delle due matrici A e K^{-1} deve avere uno spettro, cioè l'insieme degli autovalori, raggruppato attorno all'unità, risultando simile alla matrice identità ($AK^{-1} \cong I$);
- il calcolo del precondizionatore deve essere veloce e poco costoso;
- l'occupazione di memoria dev'essere inferiore o al più paragonabile all'occupazione di memoria della matrice A. Questo significa che la matrice K^{-1} deve essere anch'essa una matrice sparsa.

Le proprietà, precedentemente elencate, sono in conflitto tra di loro, perciò la scelta della matrice K^{-1} è in genere il risultato di un compromesso. Ad esempio, la prima proprietà è soddisfatta dalla matrice A^{-1}, ma il calcolo di tale matrice ha un costo elevato e un'occupazione di memoria insostenibile per matrici di

grandi dimensioni, poiché l'inversa di una matrice sparsa è in generale densa. Perciò è necessario pesare accuratamente le precedenti proprietà, in modo da avere un precondizionatore poco costoso, ma al contempo dotato di buone proprietà spettrali.

In letteratura esistono due classi di precondizionatori: *algebrici* e *funzionali (fisici)*. I precondizionatori algebrici sono indipendenti dal problema che ha originato il sistema lineare, mentre i precondizionatori funzionali traggono vantaggio dalla conoscenza di tale problema. I precondizionatori algebrici, non essendo legati al tipo di problema, possono essere utilizzati come *black box* per la risoluzione di qualsiasi sistema lineare e hanno un campo di applicazione più vasto dei precondizionatori funzionali. I precondizionatori algebrici sono suddivisi a loro volta in varie categorie tra le più note vi sono: *fattorizzazione incompleta* e *approssimata inversa*. Nella prima categoria la matrice di precondizionamento K^{-1} viene calcolata implicitamente. Data la matrice A del sistema di equazione (1.1), si calcolano, con la fattorizzazione, due matrici triangolari L e U tali che:

$$A = L \cdot U$$

Poiché la fattorizzazione di una matrice sparsa fornisce due matrici con un grado di sparsità minore, le due matrici L e U sono in realtà approssimate con le due matrici $\widetilde{L}$ e $\widetilde{U}$, ottenute dalle prime trascurando alcuni termini secondo determinati criteri [15]. Il precondizionatore è applicato nei punti *3.* e *9.* dell'algoritmo 2.3, perciò per il suo utilizzo è necessario il calcolo del prodotto matrice-vettore, successivamente riportato:

$$v = K^{-1}r \cong A^{-1}r \tag{3.1}$$

secondo le precedenti affermazioni si ha quanto segue:

$$A^{-1} = \widetilde{L}^{-1} \cdot \widetilde{U}^{-1}$$

Moltiplicando per K entrambi i membri dell'equazione (3.1), si ottiene:

$$Kv = \widetilde{L}\left(\widetilde{U}v\right) = r$$

il vettore v si ricava con una banale sostituzione in avanti e all'indietro.

I precondizionatori algebrici appartenenti alla categoria delle approssimate inverse si ottengono attraverso il calcolo esplicito della matrice di precondizionamento. Il calcolo di questo tipo di precondizionatori può procedere principalmente in due modi. Nel primo modo si minimizza la norma di Frobenius[1] della seguente funzione:

$$\left\|I - AK^{-1}\right\|_F$$

Nel secondo modo si calcolano le due matrici K_1^{-1} e K_2^{-1} tali che:

$$K^{-1} = K_1^{-1}K_2^{-1}$$

K_1^{-1} e K_2^{-1} sono le approssimazioni delle inverse dei fattori triangolari della matrice A:

$$K_1^{-1} \simeq L^{-1}, \qquad K_2^{-1} \simeq U^{-1}$$

Il precondizionatore FSAI [3] combina questi due modi di procedere e verrà meglio descritto nella sezione seguente.

[1] La norma di Frobenius di una matrice A è data dalla seguente formula:

$$\|A\|_F = \sqrt{\sum_{j=1}^{n}\sum_{i=1}^{n} a_{ij}^2}$$

3.1 Calcolo del precondizionatore FSAI

Il precondizionatore FSAI tenta di sfruttare gli aspetti positivi di entrambi i modi di procedere precedentemente citati. In particolare cerca le due approssimazioni K_1^{-1} e K_2^{-1}, minimizzando le seguenti norme:

$$\left\|I - K_1^{-1}L\right\|_F \text{ e } \left\|I - K_2^{-1}U\right\|_F$$

Per le proprietà della matrice A, simmetrica e definita positiva, la matrice U coincide con la matrice L^T, perciò la matrice di precondizionamento K^{-1} può essere scomposta come segue:

$$K^{-1} = G^T G$$

dove G è un'approssimazione dell'inversa del fattore di Cholesky L della matrice A. In particolare G è la matrice che minimizza la norme:

$$\|I - GL\|_F \tag{3.2}$$

nel sottospazio $\mathcal{W}_{SL}$, cioè un sottospazio composto da tutte le matrici aventi il pattern $\mathcal{S}_L$, dove con il termine pattern si indica l'insieme formato dalle posizioni (i, j) dei termini non nulli di una matrice. Nel caso preso in considerazione il pattern $\mathcal{S}_L$ è "*compreso*" tra il pattern di una matrice diagonale e il pattern di una matrice triangolare inferiore completa, più precisamente vale:

$$\{(i, j) : 1 \leq i = j \leq n\} \subseteq \mathcal{S}_L \subseteq \{(i, j) : 1 \leq j \leq i \leq n\}$$

dove n è la dimensione della matrice A. Si noti che la ricerca della matrice G è effettuata nell'insieme $\mathcal{W}_{SL}$, poiché la minimizzazione della funzione (3.2) nel sottospazio di tutte le matrici $n \times n$ restituirebbe l'inversa della matrice L, il cui calcolo generalmente ha un costo computazionale troppo elevato. Dopo una serie di passaggi algebrici [3, 4, 5, 6] si dimostra che il calcolo dei coefficienti di G può essere ricondotto alla soluzione della seguente successione di sistemi densi:

$$A\left[\mathcal{G}_i,\mathcal{G}_i\right] g_i\left[\mathcal{G}_i\right] = i_m \tag{3.3}$$

con $A\left[\mathcal{G}_i,\mathcal{G}_i\right]$ è indicata la sottomatrice di A avente come coefficienti gli elementi nelle posizioni $(m,n) \in \mathcal{G}_i \times \mathcal{G}_i$ dove $\mathcal{G}_i = \{i : (i,j) \in \mathcal{S}_L\}$ è un insieme di cardinalità m, $i_m \in \mathbb{R}^m$ è il vettore nullo eccetto nella m-esima componente pari a 1 e $g_i \in \mathbb{R}^m$ è il vettore dell'i-esima riga di G. Quindi risolvendo n sistemi del tipo (3.3) si determina la matrice G che premoltiplicata per G^T restituisce la matrice di precondizionamento K^{-1}.

Algoritmicamente il calcolo del precondizionatore FSAI può essere riassunto nelle tre fasi: *ricerca di un pattern*, *risoluzione di una sequenza di sistemi che derivano da tale pattern* e *postifilter della matrice precedentemente ottenuta.* La prima fase è suddivisa a sua volta in due passi. Al primo passo si applica alla matrice A la funzione *prefilter*, la quale ne elimina i coefficienti più piccoli, e cioè i coefficienti a_{ij} tali che:

$$a_{ij}^2 \leq \delta \cdot a_{ii} \cdot a_{jj}$$

dove δ è un valore reale impostato dall'utente. Il secondo passo prevede il calcolo della potenza k della matrice $\widetilde{A}$, matrice risultante dal passo precedente. In particolare, per $k = 1$ si consideri $\widetilde{A}$ e successivamente si selezioni la parte triangolare inferiore indicata con il simbolo $\widetilde{S}_1 = Low(\widetilde{A})$. Per $k = 2$ si calcoli il prodotto tra $\widetilde{S}_1$ e $\widetilde{A}$, in seguito si selezioni la parte triangolare inferiore indicata con il simbolo $\widetilde{S}_2 = Low(\widetilde{S}_1 \cdot \widetilde{A})$. Ad un generico passo l di questa procedura otteniamo:

$$\widetilde{S_{l+1}} = Low(\widetilde{S}_l \cdot \widetilde{A}) \tag{3.4}$$

Questo processo ricorsivo continua per un numero di volte pari a k, fornendo la matrice F che sarà utilizzata come pattern nella fase successiva.

Nella seconda fase si calcola la matrice G ottenuta mediante la raccolta e la risoluzione di una sequenza di sistemi. Per la raccolta dei sistemi si osservi la figura 3.1. Nella parte sinistra è riportata la matrice pattern F, in particolare la riga i di tale matrice viene selezionata e individua gli elementi che comporranno

l'i-esimo sistema. Supponiamo che la riga i di F sia costituita dai quattro elementi nell'insieme $P = \{a, b, c, d\}$, allora il sistema i sarà composto da tutti gli elementi nelle posizioni (m, n) appartenenti al prodotto cartesiano tra l'insieme P e se stesso $(P \times P)$. Una volta raccolto si utilizzano due funzioni per la sua soluzione: il primo metodo fattorizza la matrice del sistema mediante la fattorizzazione di Cholesky e il secondo metodo compie una sostituzione in avanti e indietro determinando la soluzione del sistema. Il calcolo della matrice G è riassunto nei passi elencati nell'algoritmo 3.1.

La terza fase coincide con l'applicazione della funzione *postfilter* alla matrice G. Con questa funzione vengono eliminati tutti gli elementi di G che soddisfano la seguente disequazione:

$$g_{ij}^2 \leq \tau \left(\sum_{k=1}^{i} g_{ik}^2 \right) \qquad i = 1, ..., n \quad j = 1, ..., i-1$$

dove τ è un parametro impostato dall'utente.

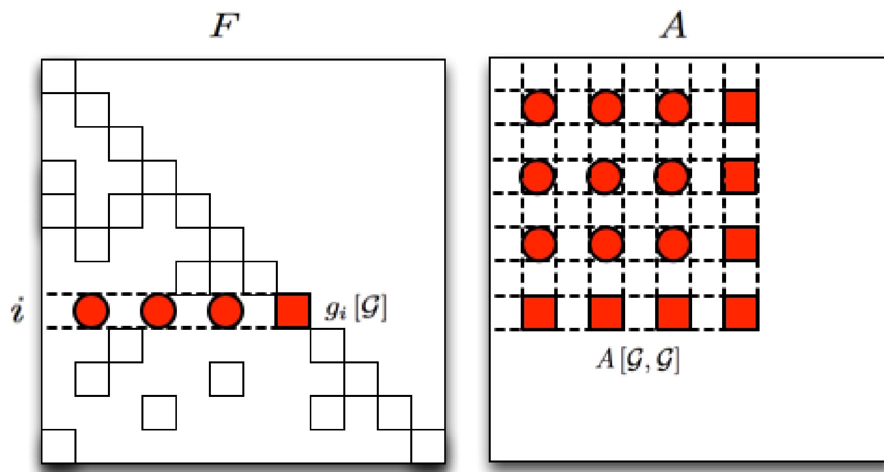


Figura 3.1: Raccolta dei sistemi $A[\mathcal{G},\mathcal{G}]$

Algoritmo 3.1 Calcolo della matrice G

Input: Matrix A and the nonzero pattern $\mathcal{S}_L$
Output: Matrix G

1. **for** $i = 1, ..., n$ **do**
2. Compute $\mathcal{G} = \{j : (i,j) \in \mathcal{S}_L\}$
3. $m = |\mathcal{G}|$
4. Gather $A[\mathcal{G},\mathcal{G}]$ from A
5. Solve $A[\mathcal{G},\mathcal{G}]\, g_i[\mathcal{G}] = i_m$
6. $g_i = g_i / \sqrt{g_{i,m}}$
7. **end for**

Capitolo 4

CUDA: Compute Unified Device Architecture

In questo capitolo si introdurranno le basi necessarie per comprendere e sviluppare un'applicazione su GPU al fine di massimizzare l'efficienza del suo utilizzo. Per illustrare il funzionamento di una GPU è indispensabile separare la spiegazione in due livelli: hardware e software. Di seguito faremo riferimento al livello hardware con il termine "*architettura*" e al livello software con il termine "*modello di programmazione*". Il modello di programmazione sfrutta l'architettura della GPU e fornisce agli sviluppatori un modo relativamente semplice per la realizzazione di software che valorizzi le potenzialità della GPU stessa. La descrizione di questi due livelli è rimandata alle successive sezioni, dove verranno sintetizzate le loro principali caratteristiche.

Con questo capitolo si vuole fornire una panoramica della libreria CUDA e, qualora si volessero approfondire questi argomenti, si rimanda alla lettura della guida per gli sviluppatori CUDA [11, 6].

4.1 Architettura

L'architettura delle schede grafiche NVIDIA si suddivide in tre diversi tipi: *Tesla*, *Fermi* e *Kepler*. In particolare con il termine *Compute Capability* si definisce la notazione ($x.x$) composta da due numeri: il primo numero determina l'architettura delle scheda grafica ($1 = Tesla$, $2 = Fermi$, $3 = Kepler$) e il secondo numero indica la versione di tale architettura. Di seguito analizzeremo le principali caratteristiche dell'architettura Fermi, poiché l'architettura Tesla è ormai datata e l'architettura Kepler, appena uscita, non è molto diffusa.

Un chip grafico con architettura Fermi è costituito da un numero fissato di multiprocessori indipendenti tra loro, denominati *Streaming Multiprocessor* (SM), il cui numero varia a seconda della GPU adoperata. Ogni SM è composto, in genere, da 32 *Scalar Processor* (SP) o *CUDA core*; in particolare un SP è formato da due unità che compiono le operazioni di matematica fondamentali (addizione, sottrazione, moltiplicazione, ecc.) su numeri interi e su numeri in virgola mobile. Inoltre in ciascun SM ci sono quattro unità speciali (SFU) che eseguono operazioni più complicate (seno, coseno, inverso, ecc). L'architettura di un singolo SM è mostrata in figura 4.1. Si noti che ogni SM ha una memoria condivisa tra tutti gli SP, una cache (L1) e un'unità di controllo (CU). Essendoci una sola CU, il funzionamento di ogni SP non può essere gestito singolarmente, poiché la CU seleziona un'istruzione che tutti gli SP devono eseguire in un dato istante, sebbene ogni SP utilizza dati differenti.

4.2 Modello di programmazione

Il modello di programmazione CUDA descrive un sistema ibrido che lavora sia su CPU che su GPU. In particolare le porzioni di codice sequenziali vengono elaborate su CPU (detto *host*), mentre le porzioni di codice parallele vengono eseguite su GPU (detto *device*). Con l'host si preparano e si caricano i dati nella memoria della scheda grafica e successivamente tali dati vengono utilizzati nell'elaborazione parallela sul device. La figura 4.2 mostra il comportamento appena descritto e introduce il concetto di *kernel* e cioè una funzione interamente eseguita su GPU.

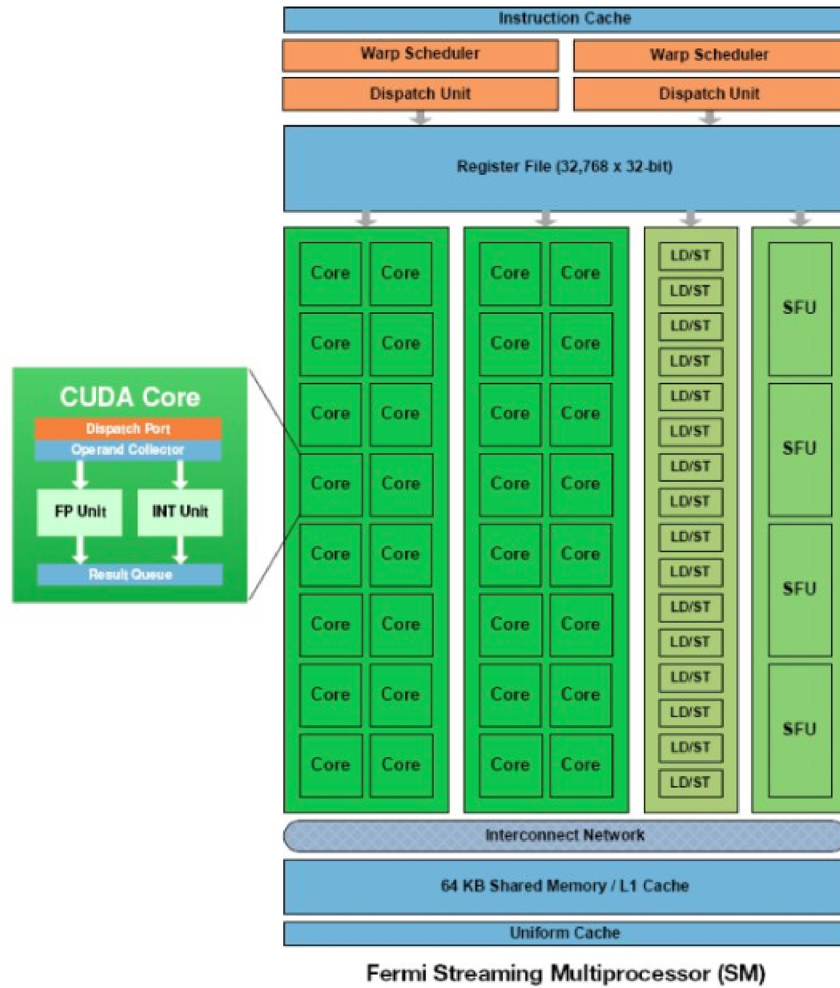


Figura 4.1: Schema di uno Streaming Multiprocessor (SM) con architettura Fermi [17].

Si ricordi che il linguaggio di programmazione utilizzato è C e, nello specifico, una funzione C assume il significato di kernel se all'inizio della sua firma viene posto il simbolo _ _*global*_ _. Oltre alla specifica _ _*global*_ _ esistono altre due specifiche: _ _*device*_ _ e _ _*host*_ _. Entrambe vengono collocate all'inizio della firma della funzione: la prima individua una funzione chiamata dal kernel principale e interamente elaborata dalla GPU, mentre la seconda determina una funzione che verrà eseguita su CPU.

Al livello software un programma eseguito su GPU si suddivide in tre livelli innestati tra loro: *grid*, *block* e *thread*. Nel livello più alto si definisce la grid, una struttura tridimensionale composta da block. Ogni block è, a sua volta, una struttura tridimensionale formata da elementi chiamati thread. All'interno di un kernel è necessario identificare i thread in modo da farli operare su dati diversi. In particolare è possibile individuare univocamente un thread, utilizzando il *thread identifier*, unico all'interno del block, concatenandolo con il *block identifier*, unico all'interno del grid. La suddivisione del programma in block è stata pensata per garantire la scalabilità del programma. Infatti l'utente decide la ripartizione dei thread tramite i parametri *grid size* e *block size*, ma è l'hardware a decidere, in maniera totalmente autonoma, la distribuzione dei vari block agli SM disponibili come da figura 4.3. Si noti che, indipendentemente dall'architettura della GPU, ogni block viene assegnato e poi eseguito da uno e un solo SM.

La sintassi per una chiamata ad una funzione kernel è riportata di seguito:

```
dim3 block size(x,y,z);
dim3 grid size (x,y);
kernel function<<<grid size , block size>>>(parametri della funzione);
```

Nelle prime due righe si specificano le strutture tridimensionali block e grid, mentre nella terza riga viene effettuata la chiamata alla funzione kernel con i parametri necessari. La chiamata di una funzione kernel è asincrona, pertanto la CPU, una volta lanciato il kernel, continua la sua esecuzione indipendentemente dalle operazioni eseguite su GPU.

Un altro aspetto importante per sfruttare al meglio la GPU è capire come vengono

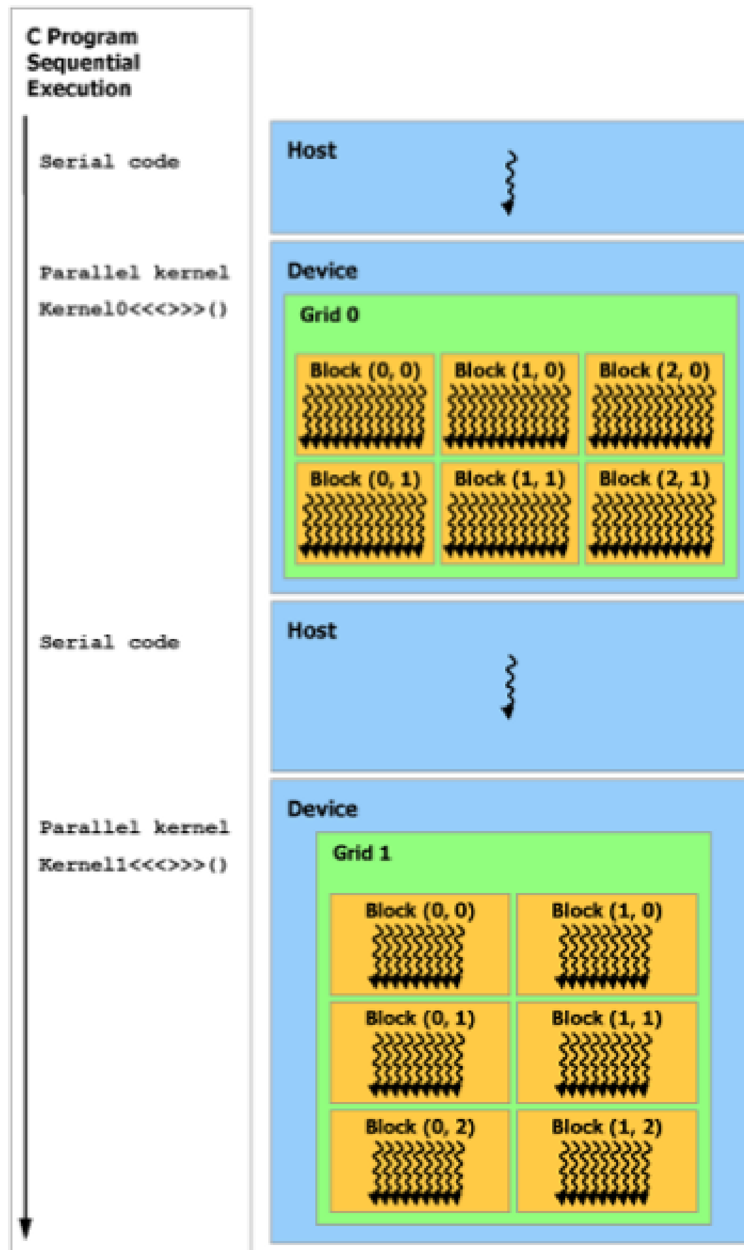


Figura 4.2: Sequenza di esecuzione di un programma C che utilizza la libreria CUDA [17]

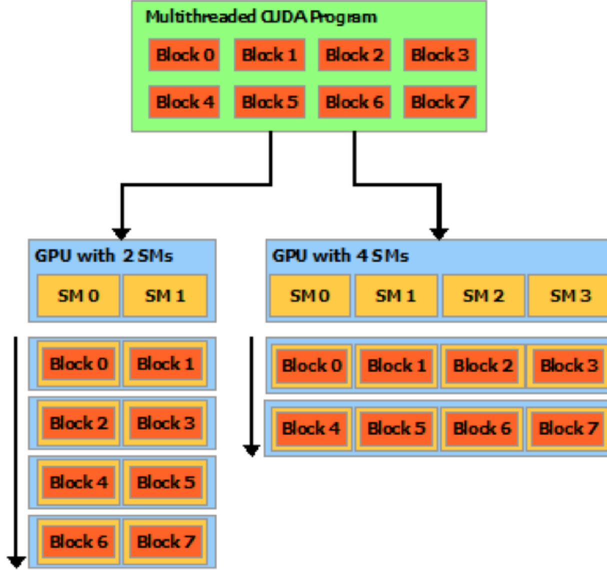


Figura 4.3: Trasparent Scalability [11]

schedulati ed eseguiti i thread. In particolare esiste una componente della GPU, denominata *warp scheduler*, che suddivide i thread in gruppi di 32 elementi, chiamati *warp*, e individua per ogni SM quali warp eseguire. I thread all'interno di un warp possono intraprendere rami di esecuzione diversi, ma, a causa dell'architettura SIMD di un SM, è necessario che la divergenza dei path di esecuzione sia minima. Un'architettura SIMD permette di eseguire un'unica istruzione ad ogni istante, perciò, ad esempio, se ogni thread di un determinato warp intraprende un ramo di esecuzione diverso il tempo necessario per terminare l'esecuzione aumenta proporzionalmente al numero di thread appartenenti al warp. Nelle GPU, con architettura Fermi, il numero di warp scheduler per SM è pari a due, perciò è possibile eseguire in parallelo due warp. A questo punto potrebbe sorgere spontanea la seguente domanda: "Perché vengono eseguiti più warp su un SM dato che è composto da 32 SP?" La risposta a questa domanda è legata al cambio di contesto dei thread eseguiti su GPU. Infatti essi hanno un contesto molto più leggero rispetto ai corrispettivi thread su CPU, perciò il cambio di contesto non rappresenta un fattore penalizzante nell'analisi di prestazioni del software. Anzi

da questo fattore si può trarre vantaggio, poiché alcune istruzioni, come gli accessi alla memoria, sono caratterizzati da una grande latenza. Quindi, un warp che effettua un accesso alla memoria viene messo in una coda di attesa, lasciando libero il SM per l'esecuzione di altri warp.

Ogni SM con architettura Fermi è soggetto a due vincoli da rispettare:

- numero massimo di block in un SM pari a 8,
- numero massimo di thread in un SM pari a 1536.

E' buona norma cercare di avvicinarsi il più possibile a tali vincoli per sfruttare appieno le potenzialità della GPU. I casi da analizzare sono dati da determinate combinazioni dei precedenti vincoli, riportati di seguito:

1. $\# \, block < 8$ e $\# \, thread < 1536$
2. $\# \, block < 8$ e $\# \, thread = 1536$
3. $\# \, block = 8$ e $\# \, thread < 1536$
4. $\# \, block > 8$ e $\# \, thread = 1536$
5. $\# \, block > 8$ e $\# \, thread < 1536$

Nei primi tre casi il motivo del sottoutilizzo delle potenzialità delle GPU è triviale. Nel quarto caso si ha una serializzazione dell'esecuzione dei block, cioè si raggruppano i block in insiemi di 8 elementi e si esegue un gruppo per volta. In questi caso non si ha nessuno spreco, in quanto, di volta in volta, lo SM è sfruttato al massimo. Infine nel quinto caso si ha la stessa serializzazione del caso precedente e, poiché il numero di thread è inferiore al massimo contenibile da un SM, si ha un sottoutilizzo delle risorse messe a disposizione dalla GPU.

L'architettura e il modello di programmazione di una GPU sono rappresentati insieme nella figura 4.4. In tale figura sono illustrati anche i diversi tipi di memoria che verranno descritti nella sezione successiva.

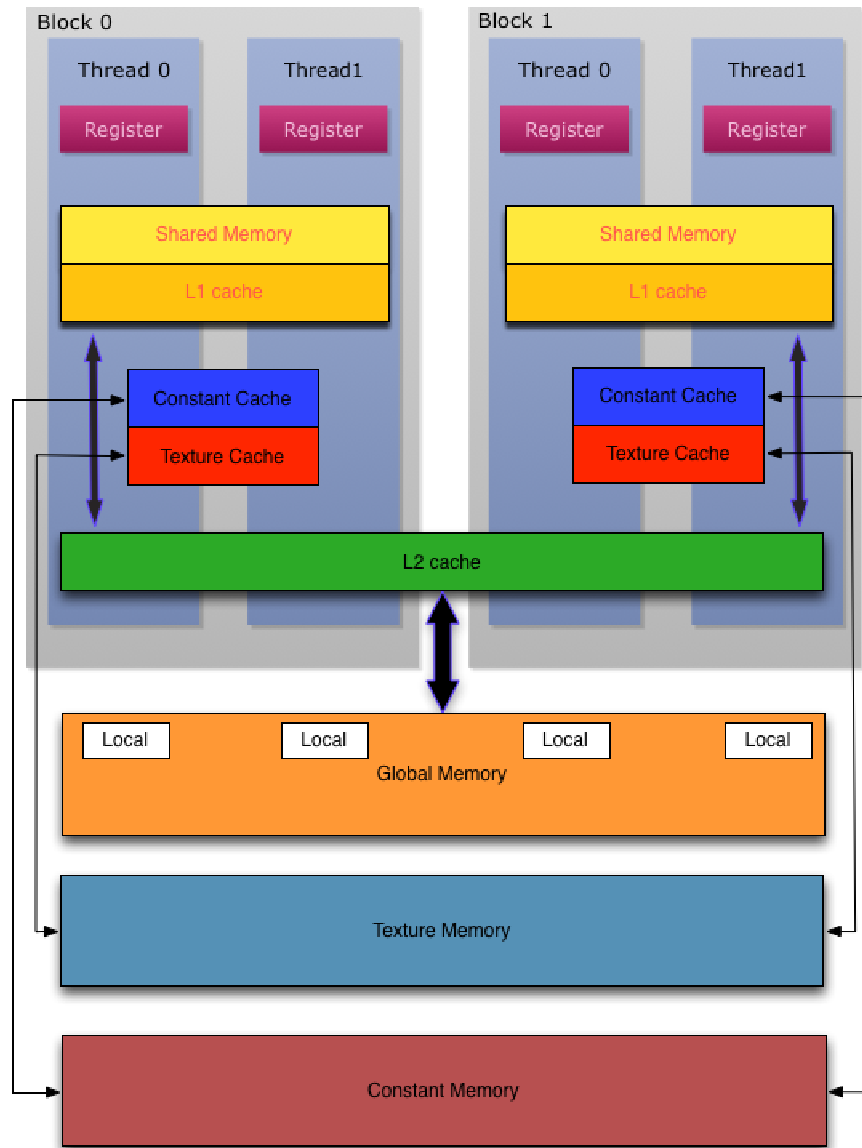


Figura 4.4: Struttura della memoria per un SM

4.3 Memoria

Un altro aspetto importante per migliorare l'efficienza del calcolo nelle GPU è l'accesso in memoria. La conoscenza e il corretto utilizzo dei vari tipi di memoria disponibili è fondamentale al fine di ottenere la massima efficienza nei programmi che si andranno a realizzare. In particolare nelle schede grafiche NVIDIA esistono i seguenti tipi di memoria:

- Global memory;
- Constant memory;
- Texture memory;
- Shared memory;
- Local memory;
- Register;

Successivamente verranno analizzati questi tipi di memoria, descrivendone le principali caratteristiche. In questa tesi non saranno definite le dichiarazioni delle variabili su tali memorie, poiché di scarso interesse ai fini della comprensione dei programmi sviluppati. Per eventuali approfondimenti si rimanda alla lettura della guida per sviluppatori CUDA [11].

4.3.1 Global memory

La global memory è la memoria principale delle scheda grafica e la sua dimensione varia a seconda del dispositivo; in genere l'oridne di grandezza è quello del GB. Questa memoria, oltre a essere la più capiente, è la più lenta perciò, se male utilizzata, può rappresentare il vero collo di bottiglia per il software sviluppato su GPU. Un thread, che accede alla memoria globale, deve attendere approssimativamente tra i 600 e gli 800 cicli di clock per ottenere il dato da utilizzare, pertanto è necessario ridurre il più possibile il suo utilizzo e sfruttare gli altri tipi di memoria aventi una latenza più bassa.

Un vantaggio della memoria globale è la sua visibilità: infatti, essa è accessibile, sia in lettura che scrittura, dalla CPU e dalla GPU. In genere, per applicazioni sviluppate su GPU l'host carica i dati nella memoria globale, poi tali dati vengono utilizzati dai thread presenti nel device.

4.3.2 Constant memory

Un altro tipo di memoria è la constant memory che, come si può intuire dal nome, è una memoria utilizzabile in sola lettura. In genere è preferibile caricare in questo tipo di memoria i dati che devono essere solo letti, poiché la memoria costante ha una latenza molto bassa. Mediante un meccanismo di caching, con questa memoria si raggiungono prestazioni assai superiori rispetto alle prestazioni ottenute con la memoria globale. Uno svantaggio della memoria costante è legato alla sua ridotta dimensione, limitata a 64 KB. Perciò nel caso, fosse necessario allocare più dati in questa memoria, è indispensabile suddividere l'esecuzione dell'algoritmo per poter caricare di volta in volta la porzione di dati su cui operare.

Una variabile per essere definita nella memoria constante deve utilizzare la specifica *__constant__*. Questa variabile sarà visibile da tutti i thread del grid e avrà un ciclo di vita pari alla durata del kernel.

4.3.3 Texture Memory

La memoria texture è un altro tipo di memoria a sola lettura, utilizzabile in alternativa alla memoria costante. Mediante un meccanismo di caching si ottiene una sensibile diminuzione del tempo di accesso alla memoria texture, e di conseguenza una ridotta latenza. In questa memoria è possibile allocare array di diverso rango: lineare o 1D, bidimensionale o 2D e tridimensionale o 3D. L'utilizzo di questa memoria evidenzia i maggiori vantaggi nel caso in cui il programma sia dotato di una località spaziale bidimensionale, poiché le operazioni di fetch dei dati hanno le prestazioni migliori. Un'altra importante caratteristica è la presenza di una doppia modalità di accesso ai dati: mediante un indice intero, come avviene normalmente nell'accesso ai dati in una matrice, oppure mediante un valore nor-

malizzato compreso tra 0 e 1. La scelta della modalità di accesso dipende dalle esigenze dell'applicazione.

La dimensione massima della memoria texture dipende dal tipo di allocazione effettuata ed è specificata nella tabella 9 della guida per sviluppatori CUDA [11]. Per qualsiasi tipo di allocazione utilizzata la dimensione della memoria texture è minore rispetto alla dimensione della memoria globale, perciò è necessario servirsi della memoria texture per la lettura di matrici, preferibilmente bidimensionali, piccole. Per l'impiego di una variabile allocata nella memoria texture è necessario effettuare inizialmente l'operazione denominata binding, e al termine l'operazione chiamata unbinding. Con la prima operazione si dichiara una variabile nella memoria texture, mentre con la seconda si libera la memoria precedentemente occupata. Inoltre questa variabile sarà visibile da tutti i thread del grid e avrà un ciclo di vita pari alla durata del kernel.

4.3.4 Shared memory

La shared memory, come precedentemente accennato, risiede *on-chip*; ovvero appartiene all'architettura del SM. Questa memoria viene condivisa tra tutti i thread appartenenti ad un determinato block ed è utilizzata dai thread stessi per interagire tra di loro. Un'altra importante caratteristica della memoria condivisa è la sua altissima velocità, o meglio una bassissima latenza, perciò è necessario sfruttare al massimo questo tipo di memoria. Lo svantaggio, come si può facilmente intuire, della memoria condivisa è la ridotta capacità, se confrontata con gli altri tipi di memorie; in particolare la sua dimensione, a seconda della configurazione scelta, è pari a 16KB o a 48KB. Con architettura Fermi è possibile scegliere tra due configurazioni per la dimensione della cache L1 e della memoria condivisa. A seconda del kernel sviluppato la scelta cade sulla configurazione 16KB L1 e 48KB shared memory, se l'applicazione ha un'elevato tasso di comunicazioni tra i thread, oppure sulla configurazione 48KB L1 e 16KB shared memory, se l'applicazione ha un elevato numero di thread che accede alla memoria globale.

La dichiarazione di una variabile nella memoria condivisa avviene all'interno di un kernel con la specifica __*shared*__. Questa variabile sarà visibile solo ai thread

appartenenti a un singolo block e ne esisteranno diverse copie, una per ogni block. Per sfruttare al massimo la GPU è necessario riempire la memoria condivisa in modo che ogni block possa coesistere con gli altri block. Ad esempio, scegliendo la configurazione (16KB L1 e 48KB shared memory) e sapendo che il numero di block massimo in un SM è pari a 8, la dimensione destinata ad ogni block è $48KB/8 = 6144B$.

4.3.5 Local memory

La local memory è una memoria estremamente lenta, il cui tempo di accesso è paragonabile alla global memory. Questa memoria, visibile dal solo thread, ha una dimensione non trascurabile, ma avendo un'alta latenza è poco utilizzata. Ogni thread dispone di al più 256 KB per la memorizzazione dei propri dati e il ciclo di vita di una variabile dichiarata in questa memoria è limitata all'esecuzione del kernel.

L'utilizzo della memoria locale non è gestito dal programmatore: infatti, è il compilatore a occuparsi della dichiarazione di variabili im questa memoria. In particolare vengono memorizzate nella memoria locale le seguenti variabili:

- array per i quali non si conosce la dimensione;
- strutture o array di grandi dimensioni che occuperebbero eccessivamente lo spazio di memoria on-chip assegnato a ogni thread;
- qualsiasi tipo di variabile che non riesce a essere contenuta all'interno dei registri (questo fenomeno è noto come register spilling).

4.3.6 Registri

I registri, come precedentemente accennato, sono locazioni di memoria che risiedono on-chip. Ogni registro ha una dimensione pari a 32 bit e il loro utilizzo migliora enormemente le performance dell'applicazione sviluppata, poiché ha una latenza estremamente bassa.

Ogni SM, con architettura Fermi, ha a disposizione 32K registri e, poiché ogni SM può ospitare fino a 1536 thread, per sfruttare al massimo la GPU ogni thread

può occupare 20 registri. Quando questo vincolo non viene rispettato il numero di thread in esecuzione contemporaneamente su un SM diminuisce, provocando un calo delle performance.

L'utilizzo da parte di un thread dei registri non prevede alcuna specifica e le variabili dichiarate nel kernel vengono automaticamente memorizzate nei registri. Un fattore che migliora le prestazioni del programma è la memorizzazione di un puntatore dell'array in un registro, quando questo array viene letto da molti thread.

4.4 Coalescenza

Un ulteriore grado di parallelismo viene fornito in fase di lettura/scrittura dei dati tramite il meccanismo di coalescenza. Gli accessi in memoria dei chip grafici compatibili con CUDA devono seguire delle regole ben precise per minimizzare il numero di transazioni verso la memoria globale, indispensabile considerato la velocità ridotta di quest'ultima.

Gli accessi di un kernel vengono definiti coalescenti se i thread con id consecutivi accedono a locazioni di memoria consecutive. Con questo tipo di accessi i thread appartenenti a un determinato warp accedono, con un'unica transazione, a una singola porzione di memoria sfruttando appieno la banda tra memoria globale e lo SM. Perciò, come riferito nella conferenza [12], è preferibile sfruttare il meccanismo di coalescenza, anche se ciò comporta un utilizzo solo parziale dei core della GPU.

Capitolo 5

Gradiente Coniugato Modificato con le librerie CUBLAS e CUSPARSE

In questa fase del lavoro di tesi è stato studiato e analizzato l'impiego delle librerie CUBLAS e CURSPARSE, al fine di migliorare le prestazioni del gradiente coniugato modificato sfruttando l'iperparallelismo delle GPU.

La libreria CUBLAS [13] (*CUDA Basic Linear Algebra Subprograms*) è una libreria sviluppata da NVIDIA che fornisce le principali funzioni di algebra lineare eseguite su GPU. Analogamente alla libreria BLAS (*Basic Linear Algebra Subprograms*) che implementa le funzioni di algebra lineare su CPU, la libreria CUBLAS classifica le sue funzioni in tre livelli.

- Livello 1: operazioni tra vettori.
- Livello 2: operazioni tra una matrice e un vettore.
- Livello 3: operazioni tra matrici.

La divisione di queste funzioni nei tre livelli è motivata dal numero di cicli innestati necessari per eseguire l'operazione selezionata, più precisamente per le operazioni del livello i sono essenziali i cicli innestati per portare a termine l'esecuzione della funzione scelta.

La libreria CUSPARSE [14] è un'altra libreria, messa a disposizione da NVIDIA, che fornisce un insieme di funzioni per la gestione di matrici sparse. Diversamente dal caso precedente le sue funzioni sono classificate in quattro livelli.

- Livello 1: operazioni tra un vettore memorizzato in formato sparso e un vettore memorizzato in formato denso.

- Livello 2: operazioni tra una matrice memorizzata in formato sparso e un vettore memorizzato in formato denso.

- Livello 3: operazioni tra una matrice memorizzata in formato sparso e un insieme di vettori memorizzati in formato denso (questo insieme può essere considerato come un'unica grande matrice densa).

- Conversione: operazioni che permettono la conversione tra i diversi formati di memorizzazione.

Entrambe le librerie possono lavorare con diversi tipi di dati: reali o complessi e in virgola mobile in singola o doppia precisione. Nell'implementazione del gradiente coniugato modificato si sono utilizzate esclusivamente le funzioni del livello 1 della libreria CUBLAS e le funzioni di conversione e di livello 2 della libreria CUSPARSE, poiché i vettori sono memorizzati in formato denso e le matrici in formato sparso.

Nella successiva sezione passiamo all'analisi del modello di programmazione per l'utilizzo delle due librerie.

5.1 Modello di programmazione

Il modello di programmazione di entrambe le librerie si basa su cinque passi.

1. Creazione della variabile *handle*, parametro necessario in tutte le chiamate a una funzione.

2. Allocazione delle matrici o dei vettori da utilizzare nella global memory della GPU. La dichiarazione avviene mediante la chiamata alla funzione *cudaMalloc*, la stessa funzione impiegata con la libreria CUDA.

3. Chiamata di una o più funzioni della libreria presa in considerazione. Ogni funzione restituisce un numero intero pari a 0, se l'operazione è stata conclusa correttamente, e diverso da 0, se l'esecuzione della funzione non è andata a buon fine.

4. Eliminazione mediante la funzione *cudaFree* delle matrici e dei vettori precedentemente dichiarati.

5. Eliminazione della variabile *handle.*

Per la libreria CUSPARSE, oltre ad allocare la matrice, è necessario creare una variabile, o meglio un descrittore, che identifica tale matrice. Esiste un tipo di descrittore diverso per ogni rappresentazione di una determinata matrice nei diversi formati e al termine dell'esecuzione del programma questa variabile identificativa della matrice deve essere eliminata mediante un apposito metodo.

Il metodo del gradiente coniugato modificato utilizza le librerie CUBLAS e CUSPARSE per sfruttare le potenzialità della GPU. Più precisamente il solutore lineare è composto da tre file: *driver.cu*, *CUDA_structures.cu* e *PCG_solv_FSAI.cu.* Il file *driver.cu* gestisce le chiamate alle funzioni che implementano i passi precedentemente elencati e durante la sua descrizione saranno specificati i ruoli dei file *CUDA_structures.cu* e *PCG_solv_FSAI.cu.* La funzione principale (main) nel file *driver.cu*, una volta letta la matrice A e determinata la matrice di precondizionamento K^{-1}, chiama la funzione nel file *CUDA_structures.cu* che crea le variabili handle per le librerie CUBLAS e CUSPARSE. Di seguito è riportato

un frammento di codice esemplificativo:

```
cublasHandle_t cublasHandle;
cusparseHandle_t cusparseHandle;
cublasStatus = cublasCreate(*cublasHandle);
//Check cublasStatus
cusparseStatus = cusparseCreate(*cusparseHandle);
//Check cusparseStatus
```

Nelle prime due righe vengono dichiarate le variabili handle e successivamente tali variabili vengono allocate.

Successivamente la funzione main alloca le matrici e i vettori necessari per l'esecuzione del metodo del gradiente coniugato modificato su GPU. Nel seguente frammento di codice viene illustrata la creazione di un vettore di lunghezza *size* di elementi *double* e il suo trasferimento nella global memory della GPU:

```
double *d_ptr;
cudaMalloc((void **) &d_ptr, size * sizeof(double));
cudaMemcpy(d_ptr, ptr, size * sizeof(double), cudaMemcpyHostToDevice);
```

Ora la funzione principale del file *driver.cu* effettua la chiamata alla funzione *PCG_solv_FSAI* che si trova nel file *PCG_solv_FSAI.cu* implementa il metodo del gradiente coniugato modificato mediante le librerie CUBLAS e CUSPARSE. Dopo l'inizializzazione delle variabili necessarie per l'esecuzione del solutore lineare si effettuano le chiamate ai metodo delle due librerie; in particolare, nel successivo frammento di codice sono riportate una chiamata a una funzione della libreria CUBLAS del Livello 1 e una chiamata a una funzione della libreria CUSPARSE del Livello 2.

```
cublasDnrm2(cublasHandle, n, d_x, incx, &result);
cusparseDcsrmv(cusparseHandle, cusparseOperation, m, n, nnz, &alpha, descrA, d_coef, d_iat, d_ja, d_x,
&betha, d_y);
```

dove nella prima chiamata si determina la norma quadra del vettore x, mentre nella seconda chiamata si calcola il prodotto tra la matrice A in formato CSR

e il vettore x in formato denso. Si noti la presenza della variabile `descrA`, il descrittore della matrice A, e del parametro `cusparseOperation` che, a seconda del valore assunto, permette il calcolo del prodotto tra A e x oppure tra A^T e x. Sperimentalmente si è osservato che il calcolo del prodotto matrice-vettore è dispendioso in termini di tempo di esecuzione soprattutto nel secondo caso. Per ovviare a questo problema si è convertita la matrice A nella sua trasposta attraverso la chiamata alla funzione `cusparseDcsr2csc` che trasforma la matrice A dal formato CSR al formato CSC, ottenendo così la matrice trasposta.

Infine la funzione principale nel file *driver.cu* chiama le funzioni per l'eliminazione delle variabili handle e per la deallocazione delle matrici e dei vettori. Di seguito viene riportato un frammento di codice esemplificativo.

```
cublasDestroy(cublasHandle);
cusparseDestroy(cusparseHandle);
cudaMemcpy(d_ptr, ptr, size * sizeof(double), cudaMemcpyDevicetToHost);
cudaFree(d_ptr);
```

Con le prime due chiamate si eliminano le variabili handle e con le ultime due chiamate si trasferisce su CPU il vettore ptr.

Nella successiva sezione saranno confrontati i tempi di esecuzione del metodo del gradiente coniugato modificato su GPU e su CPU, in particolare saranno mostrati i vantaggi dell'impiego della GPU per tale solutore lineare.

5.2 Test

In questa sezione si vogliono evidenziare i benefici dell'impiego della GPU nell'esecuzione del gradiente coniugato modificato. Le performance, in termini di tempo di esecuzione, sono state misurate per diverse configurazioni con due termini variabili: matrice e unità di elaborazione. I tipi di matrici e i tipi di unità di elaborazione sono state specificate nella sezione 2.3. In seguito all'introduzione sulla libreria CUDA (Capitolo 4) si è in grado di apprendere al meglio il significato dei diversi parametri nella tabella 5.1. Più precisamente nella colonna di sinistra

Caratteristica	**Valore**
Compute Capability	2.1
Global memory size	1024 MB
Number of SM	4
Number of SP per SM	48
Total number of SP	192
GPU clock rate	1566 MHz
Memory clock rate	1804 MHz
Warp size	32
Maximum number of thread per SM	1536

Tabella 5.1: Caratteristiche della scheda grafica *GeForce GTS 450*

sono sintetizzate le principali caratteristiche della scheda grafica adoperata per i test e nella colonna di destra i rispettivi valori. Si noti che per schede grafiche con Compute Capability 2.1 il numero di SP per SM è pari a 48, diversamente dalle schede grafiche della precedente versione 2.0, dove il numero di SP per SM è pari a 32. E' possibile ottenere tutte le informazioni circa le caratteristiche della GPU utilizzata mediante l'esecuzione del programma *deviceQuery* presente nel *CudaToolkit*.

Alla luce dei risultati ottenuti dai test, riassunti nella tabella 5.2, si osserva che per tutte le matrici considerate i tempi di esecuzione del gradiente coniugato modificato, impiegando la GPU, diminuiscono rispetto ai tempi di esecuzione utilizzando 4 processori. Come mostrato in figura 5.1 lo speed up assoluto, per le matrici *Emilia_grande* e *Stoc-F_1465*, è superiore a 2 e, per la matrice *Cubo_4,* è pari a 1,55. Questo fenomeno è legato al pattern troppo sparso del precondizionatore per la matrice *Cubo_4*. A causa del ridotto numero di elementi per riga, non si è in grado di sfruttare pienamente la banda disponibile tra memoria globale e SM della GPU.

Come ulteriore esperimento per la matrice *Cubo_4* per verificare il precedente fenomeno si sono confrontati l'esecuzione del gradiente coniugato modificato memorizzando le matrici nei formati CSR e Hybrid, descritti nella sezione 2.1. I tempi di esecuzione ottenuti da questi test sono esposti nella tabella 5.3, mentre lo speed up è mostrato nella figura 5.2. Si noti che lo speed up ottenuto con il formato Hybrid è superiore a 2, poiché l'esecuzione del gradiente coniugato mo-

Tipo	$T_e\,[s]$	n_i	S_a
CPU	9,8	583	1
GPU	6,3	584	1,55

(a) Matrice *Cubo_4*

Tipo	$T_e\,[s]$	n_i	S_a
CPU	54,8	772	1
GPU	26,7	773	2,05

(b) Matrice *Emilia_grande*

Tipo	$T_e\,[s]$	n_i	S_a
CPU	45,1	634	1
GPU	20,9	635	2,16

(c) Matrice *Stoc-F_1465*

Tabella 5.2: Statistiche dei test condotti sul Gradiente Coniugato Modificato ($T_e\,[s]$ tempo di esecuzione, n_i numero di iterazioni e S_a speed up assoluto)

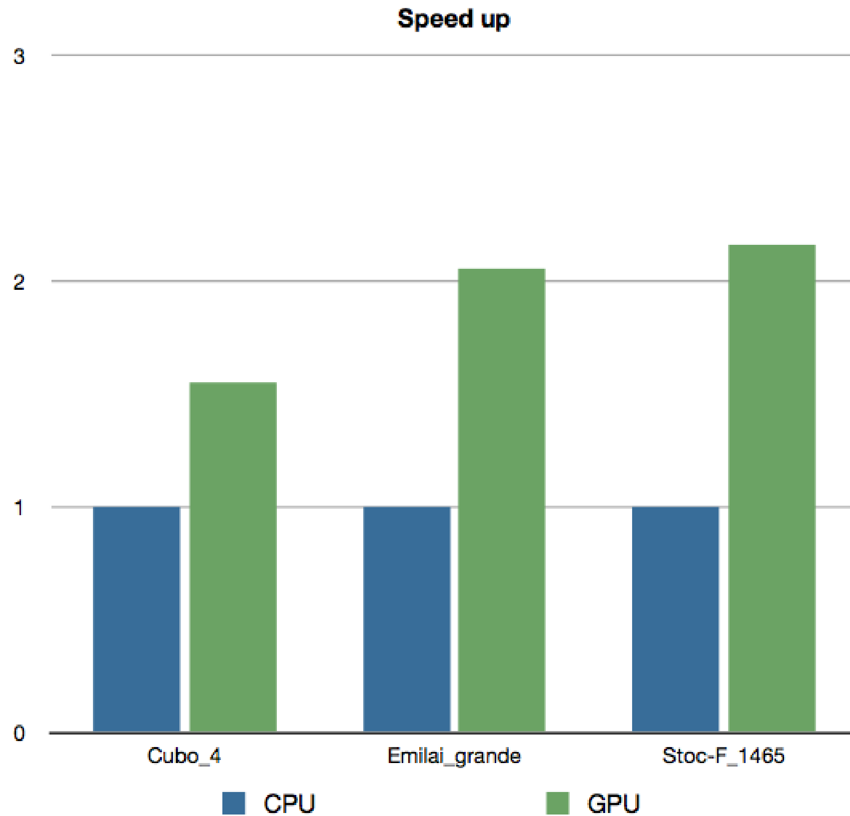


Figura 5.1: Speed up del metodo del Gradiente Coniugato Modificato

dificato su GPU con tale formato riesce a sfruttare maggiormente la banda della scheda grafica minimizzando il numero di transazioni necessarie per la lettura dei dati. Questo risultato è legato alla struttura del formato Hybrid. Esso memorizza le righe della matrice in due matrici della stessa dimensione, minimizzando così il numero di accessi alla memoria per individuare un determinato elemento.

Nel prossimo capitolo si analizzerà l'esecuzione su GPU di alcune parti del calcolo del precondizionatore FSAI, mostrando mediante opportuni test i sensibili miglioramenti ottenuti.

Tipo	$T_e\,[s]$	n_i	S_a
CPU	9,8	583	1
GPU CSR	6,3	584	1,55
GPU HYB	4,6	584	2,10

Tabella 5.3: Statistiche dei test condotti sul metodo del Gradiente Coniugato Modificato con la matrice *Cubo_4* ($T_e\,[s]$ tempo di esecuzione, n_i numero di iterazioni e S_a speed up assoluto).

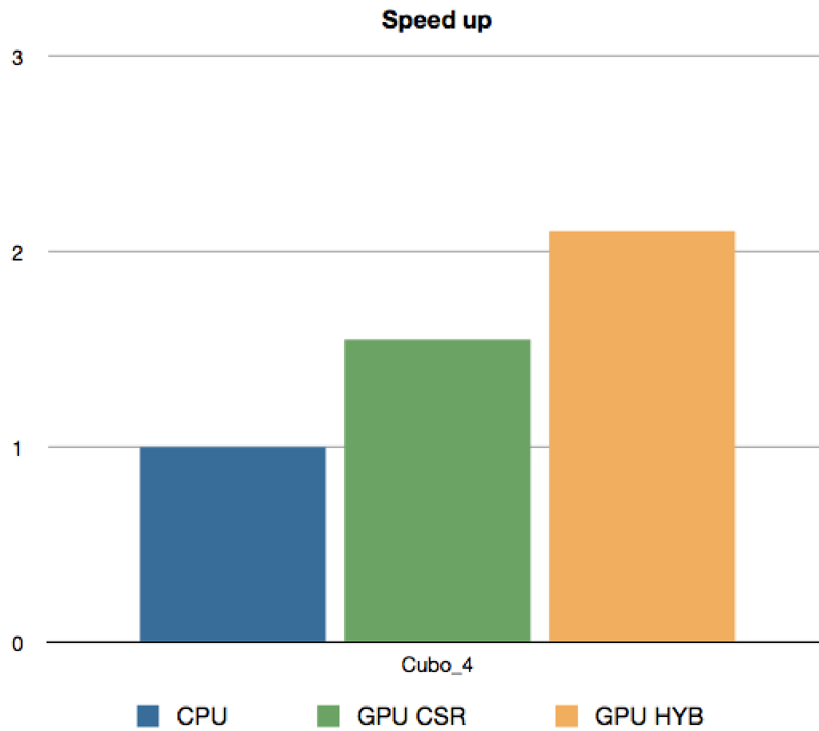


Figura 5.2: Speed up del metodo del Gradiente Coniugato Modificato per la matrice *Cubo_4*.

Capitolo 6

Calcolo del Precondizionatore FSAI con la libreria CUDA

Il calcolo del precondizionatore FSAI, descritto nel capitolo 3, prevede l'esecuzione delle tre fasi: ricerca del pattern, soluzione di una sequenza di sistemi e postfilter della matrice G. Nella prima e nella terza fase sono presenti molti accessi alla memoria e, poiché le GPU ottengono il miglior rendimento nel calcolo parallelo, tali fasi sono state lasciate implementate su CPU.

La seconda fase è la parte del calcolo del precondizionatore più onerosa dal punto di vista computazionale, perciò la componente da ottimizzare. Il primo passo della fase considerata prevede il calcolo della matrice G (algoritmo 3.1) che ha complessità computazionale dominante nel calcolo del precondizionatore. Per chiarezza consideriamo la matrice pattern F con un numero di righe pari a n, un numero di nonzeri pari a n_p e un numero medio di termini non nulli per riga pari a $m = n_p/n$. Si noti che ad ogni iterazione dell'algoritmo 3.1 nei punti *4.* e *5.* viene eseguita la raccolta e la soluzione dei sistemi, compiendo per ogni punto rispettivamente $O(m^2)$ e $O(m^3)$ operazioni. Di conseguenza il calcolo di G ha una complessità computazionale complessiva pari a $O(n \cdot m^3)$, poiché viene risolto un sistema per ogni iterazione dell'algoritmo precedentemente considerato. Le altre fasi del calcolo del precondizionatore hanno una complessità computazionale pari a $O(n)$.

Per una prima implementazione del calcolo del precondizionatore FSAI si è pen-

sato di raccogliere i sistemi su CPU, successivamente trasferirli sulla global memory della scheda grafica e infine risolvere la sequenza di sistemi mediante l'impiego della GPU. Inoltre si noti che la soluzione di un determinato sistema è totalmente indipendente dalla soluzione degli altri sistemi, perciò perfettamente parallelizzabile.

Ora esaminiamo i tempi di esecuzione della raccolta dei sistemi su CPU e del loro trasferimento sulla memoria della GPU per la sola matrice *Cubo_4*. Con l'analisi dei tempi di esecuzione di un'unica matrice si vuole focalizzare l'attenzione sul peso che ogni punto ha rispetto al tempo di esecuzione complessivo. Nella tabella 6.3 sono riportati i risultati ottenuti con la matrice *Cubo_4*: si noti che il tempo di trasferimento dei sistemi è circa un quarto del tempo necessario per la raccolta dei sistemi stessi. La differenza tra i due tempi esecuzione è mostrata in maniera più evidente in figura 6.2. Inoltre la matrice pattern F deve necessariamente essere trasferita su GPU poiché sarà utilizzata nel metodo del gradiente coniugato modificato, dato che la matrice di precondizionamento G appartiene all'insieme $\mathcal{W}_{SL}$ (insieme composto da tutte le matrici con il pattern F). In una successiva implementazione la matrice pattern F sarà trasferita nella memoria della scheda grafica e saranno raccolti i sistemi su GPU, eliminando il tempo di trasferimento di tali sistemi.

In precedenza si è calcolata la complessità computazionale del calcolo della matrice G e si è verificato che il fattore che incide maggiormente nel calcolo del precondizionatore è la risoluzione di una sequenza di sistemi. Per migliorare questa fase si è pensato, allora, ad un modo per ridurre il numero di sistemi da risolvere. Nella prossima sezione sarà illustrato un algoritmo euristico che cerca di fondere alcune righe della matrice pattern in un'unica riga diminuendo il numero di sistemi da raccogliere e successivamente da risolvere.

Tipo	T_e $[s]$
Raccolta dei sistemi	0,47
Trasferimento dei sistemi	0,13
Totale	0,60

Tabella 6.1: Statistiche della raccolta dei sistemi su CPU e del loro trasferimento su GPU

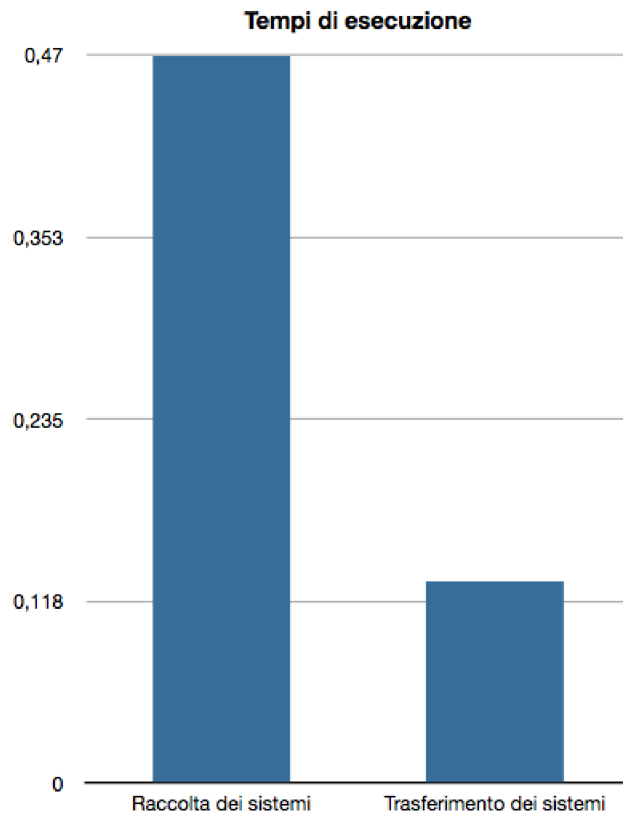


Figura 6.1: Tempi di esecuzione della raccolta dei sistemi e del loro trasferimento

6.1 Riduzione dei sistemi da risolvere

La matrice pattern F è caratterizzata da righe successive aventi un pattern simile. Più precisamente, pattern di righe successive differiscono per un numero ristretto di posizioni, perciò accorpando tali righe si raccolgono meno sistemi. Consideriamo due righe successive i e $i+1$ rispettivamente di dimensione d_i e d_{i+1} e indichiamo con la variabile *mismatch* il numero di posizioni del pattern appartenenti alla sola riga $i+1$. L'algoritmo euristico valuta se il costo di risolvere due sistemi di dimensione d_i e d_{i+1} è maggiore o minore del costo di risolvere un unico sistema di dimensione $d_i + mismatch$. Questa condizione è riassunta con la seguente disequazione:

$$(d_i + mismatch)^3 < \alpha \left(d_i^3 + d_{i+1}^3 \right) \tag{6.1}$$

dove $\alpha \geq 1$ è una constante fissata.

L'algoritmo euristico verifica per tutte le righe della matrice F se il vincolo (6.1) è rispettato. In particolare ad un passo generale di tale algoritmo si valuta se la riga presa in considerazione soddisfa il precedente vincolo con gli ultimi w pattern accorpati. Inoltre per valori di $\alpha > 1$ il vincolo (6.1) viene rilassato favorendo il "merging" di righe simili e di conseguenza l'ottenimento di una matrice di precondizionamento più densa che, in genere, agevola la convergenza del metodo del gradiente coniugato modificato. In alcuni casi l'operazione di merging degenera creando un'unica macroriga. Ad esempio, supponiamo che la matrice pattern F sia una matrice a banda[1] e che abbia in una generica riga esattamente quattro elementi nonnulli. Dopo quattro iterazioni dell'algoritmo euristico si ottiene il fenomeno precedentemente indicato, in particolare dalla quarta riga in poi il vincolo (6.1) è sempre verificato. A causa di tale fenomeno il numero di elementi di una riga frutto del merging di più righe dev'essere limitato a un valore fissato.

Di seguito riportiamo nella tabella 6.2 il numero di sistemi da raccogliere per la matrice *Cubo_4* al variare di α. Si noti che il numero di righe e di conseguenza il numero di sistemi per $\alpha \geq 1$ è circa un terzo del numero di righe della matrice F.

[1] Una matrice a banda è una matrice sparsa dove gli elementi nonnulli sono tutti posti in una banda diagonale che comprende la diagonale principale e, opzionalmente, una o più diagonali alla sua destra od alla sua sinistra.

Valore di α	Numero di righe
-	190581
1	58928
1,01	58711
1,05	55599
1,11	52812

Tabella 6.2: Numero di sistemi da raccogliere al variare di α

L'algoritmo euristico non porta un netto guadagno in questa fase del calcolo del precondizionatore, come si nota dalla tabella 6.3, poiché il tempo di esecuzione per compiere il merging tra righe simili ha un peso significativo rispetto alla raccolta dei sistemi e al loro trasferimento su GPU. Considerato che la risoluzione di una sequenza di sistemi ha un costo computazionale dominante nel calcolo del precondizionatore, l'operazione di merging, riducendo il numero di sistemi da risolvere, ha portato importanti benefici. All'inizio di questo capitolo abbiamo calcolato la complessità computazionale della risoluzione di una sequenza di sistemi dimostrando essere pari a $O(n \cdot m^3)$ dove $n \gg m$, perciò, mediante l'operazione di merging, si è ridotta la complessità del calcolo del precondizionatore a circa $O\left(\frac{n \cdot m^3}{3}\right)$. Per alcuni problemi di ingegneria meccanica si è riusciti a diminuire la complessità di questa fase di un fattore superiore a 3, ad esempio il problema meccanico descritto dalla matrice *Cubo_4* per $\alpha = 1,11$ si è ottenuto un fattore di diminuzione pari a 3,6. Inoltre nel campo delle scienze applicate il metodo del gradiente coniugato modificato viene utilizzato per effettuare migliaia simulazioni, dove la matrice pattern F è identica per le diverse simulazioni. Perciò il costo per effettuare l'operazione di merging è irrisorio se confrontato con il tempo necessario per la risoluzione del gradiente coniugato modificato.

Nella successiva sezione analizzeremo e studieremo la risoluzione di una sequenza di sistemi su GPU. In particolare verrà spiegato l'algoritmo per la soluzione dei sistemi su CPU e successivamente lo stesso algoritmo operante su GPU.

6.2 Risoluzione di una sequenza di sistemi

La risoluzione di una sequenza sistemi è la fase più onerosa del calcolo del pre-

Tipo	T_e senza euristico	T_e con euristico
Merging	0,0	0,29
Raccolta dei sistemi	0,47	0,16
Trasferimento dei sistemi	0,13	0,04
Totale	0,60	0,50

Tabella 6.3: Statistiche della raccolta dei sistemi su CPU e loro trasferimento su GPU

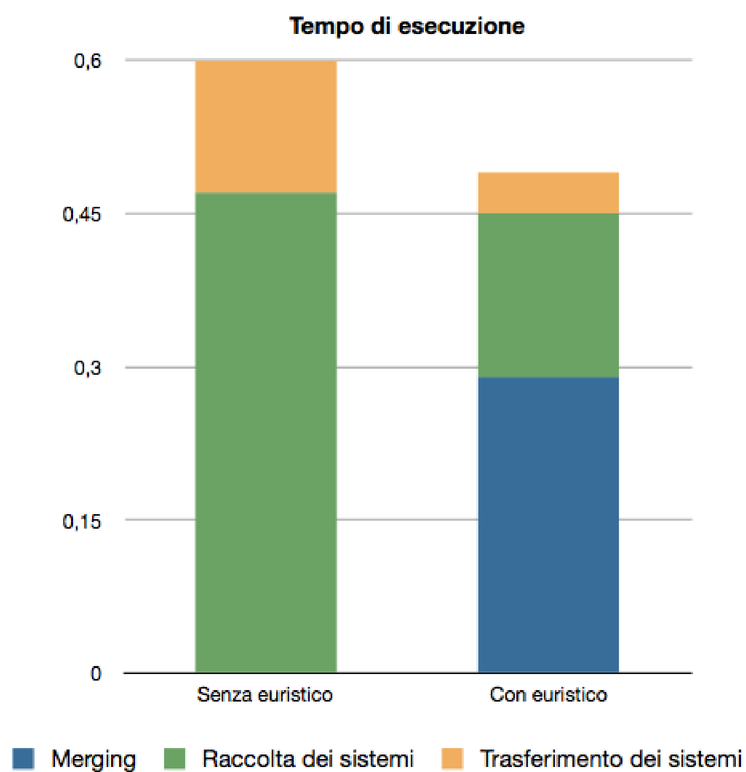


Figura 6.2: Confronto tra i tempi di esecuzione della raccolta e del traferimento dei sistemi ottenuti mediante l'utilizzo dell'algoritmo euristico

condizionatore, perciò a tale fase è stato dedicato molto tempo nello sviluppo della tesi. L'algoritmo impiegato per la risoluzione di una sequenza di sistemi è un metodo diretto noto in letteratura come metodo dell'eliminazione di Gauss. Questo algoritmo sfrutta la semplicità di soluzione di sistemi lineari descritti da una matrice in forma triangolare, inferiore o superiore. Consideriamo il seguente sistema:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{22}x_2 + \cdots + a_{2n}x_n & = & b_1 \\ \dots & \dots & \dots \\ a_{nn}x_n & = & b_n \end{cases}$$

dove $a_{ii} \neq 0$, $i = 1, 2, \dots, n$. La soluzione di tale sistema è descritta dal seguente vettore:

$$\begin{cases} x_n = & b_n / \, a_{nn} \\ x_k = & \left(b_k - \sum_{j=k+1}^{n} a_{kj} x_j \right) \Big/ \, a_{kk} \qquad k = n-1, \dots, 1 \end{cases}$$

Questa considerazione ci suggerisce di esaminare la possibilità di trasformare un generico sistema in un sistema equivalente di forma triangolare. Si noti che, quando a un'equazione del sistema sostituiamo una combinazione lineare dell'equazione con un'altra dello stesso sistema, il nuovo sistema risulta equivalente al precedente.

Per meglio illustrare il processo dell'eliminazione di Gauss scriviamo $Ax = b$ esplicitamente

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \dots\dots\dots\dots\dots\dots\dots\dots & \dots\dots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = b_n \end{cases}$$

Possiamo eliminare l'incognita x_1 dalle ultime $(n-1)$ righe, sommando all'i-esima equazione, $i = 2, \dots, n$ la prima moltiplicata per

$$m_{i1} = -\frac{a_{i1}}{a_{11}} \qquad i = 2, \ldots, n$$

Dopo queste operazioni si ottiene il seguente sistema:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ \ldots & \ldots & \ldots \\ a_{n2}^{(2)}x_2 + \cdots + a_{nn}^{(2)}x_n & = & b_n^{(2)} \end{cases}$$

dove

$$i = 2, \ldots, n : \begin{cases} a_{ij}^{(2)} = & a_{ij} + m_{i1}a_{1j} \\ b_i^{(2)} = & b_i + m_{i1}b_1 \end{cases}$$

Reiterando $n-1$ volte questo procedimento si ottiene:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ \ldots & \ldots & \ldots \\ a_{nn}^{(n)}x_n & = & b_n^{(n)} \end{cases}$$

dove

$$i = k+1, \ldots, n : \begin{cases} m_{ik} = & -a_{ik}^{(k)}/a_{kk}^{(k)} \\ a_{ij}^{(k+1)} = & a_{ij}^{(k)} + m_{ik}a_{kj}^{(k)} \qquad j = k+1, \ldots, n \\ b_i^{(k+1)} = & b_i^{(k)} + m_{ik}b_k^{(k)} \end{cases}$$

Questa sequenza di operazioni è conosciuta con il nome di triangolarizzazione ed è riassunta nell'algoritmo 6.1. Una volta compiuta questa prima fase bisogna procedere con la fase di sostituzione all'indietro ottenendo la soluzione cercata. La sostituzione all'indietro è sintetizzata nell'algoritmo 6.2; questa fase accetta in input la matrice A in forma triangolare superiore e il vettore b e restituisce la soluzione x.

Algoritmo 6.1 Fase di triangolarizzazione nel metodo di eliminazione di Gauss

Input: Matrix A and vector b

1. **for** $k = 1, \ldots, n-1$ **do**
2. **for** $i = k+1, \ldots, n$ **do**
3. $m_{ik} = -a_{ik}^{(k)}/a_{kk}^{(k)}$
4. $b_i^{(k+1)} = b_i^{(k)} + m_{ik} b_k^{(k)}$
5. **for** $j = k+1, ..., n$ **do**
6. $a_{ij}^{(k+1)} = a_{ij}^{(k)} + m_{ik} a_{kj}^{(k)}$
7. **end do**
8. **end do**
9. **end do**

Algoritmo 6.2 Fase di sostituzione all'indietro nel metodo di eliminazione di Gauss

Input: Matrix A and vector b

Output: Vector x

1. $x_n = b_n^{(n)}/a_{nn}^{(n)}$
2. **for** $k = n-1, ..., 1$ **do**
3. $sum = 0$
4. **for** $j = k+1, ..., n$ **do**
5. $sum = sum + a_{kj}^{(k)} x_j$
6. **end do**
7. $x_k = (b_k^{(k)} - sum)/a_{kk}^{(k)}$
8. **end do**

La fase di risoluzione di una sequenza di sistemi è stata portata su GPU ricorrendo ad un'unica funzione. In particolare tale funzione assegna ad un block la risoluzione di una sistema che fornirà la soluzione cercata mediante il metodo dell'eliminazione di Gauss. Ora focalizziamo l'attenzione sull'esecuzione di un singolo block, in modo da comprendere il funzionamento del metodo dell'eliminazione di Gauss su GPU. Innanzitutto bisogna specificare che la dimensione della grid è pari al numero di sistemi da risolvere e che la dimensione dei block è pari alla numero di righe del sistema moltiplicato per un determinato numero, il quale è ottimizzato per ogni tipo di dimensione del sistema. Ora analizzeremo i passi compiuti da ogni block riportando frammenti di codice esemplificativi.

Nell primo passo è eseguito il trasferimento del sistema lineare (matrice A e termine noto b) da risolvere nella shared memory del SM tramite il seguente ciclo iterativo:

```
for (int i = tx; i < n; i += blockDim.x) {
    if (ty < n) As(i,ty) = A[ty * n + i];
    if (ty == n) As(i,ty) = b[i];
}
```

dove con le variabili `tx` e `ty` si identifica il thread. In particolare si suddivide il sistema lineare in fasce di dimensione `blockDim.x` e a ogni iterazione ciascun thread individua attraverso il proprio id un elemento del sistema da copiare.

Nel secondo passo si esegue la triangolarizzazione della matrice del sistema. Per compiere questa operazione sono necessarie $n - 1$ iterazioni e in una generica iterazione j si compiono le seguenti operazioni: normalizzazione della riga j (punto 3. dell'algoritmo 6.1) ed eliminazione dei termini nella colonna j dalle righe $i = j + 1, \ldots, n$ (punto 6. dell'algoritmo 6.1). Di seguito riportiamo il frammento di codice che esegue queste due operazioni:

```
if ((tx == 0) && (ty > j)) As(j,ty) = As(j,ty) / As(j,j);
for (int i = j+1+tx; i < n; i += blockDim.x) {
        if (ty > j) As(i,ty) = -(As(i,j) * As(j,ty)) + As(i,ty);
```

```
}
```

nella prima riga si effettua la normalizzazione della riga j e si vuole precisare che non viene calcolata la divisione per `ty = j` poiché il suo valore è già noto.

Nel terzo passo si applica la sostituzione all'indietro. Per compiere questa operazione sono necessarie n iterazioni: in una generica iterazione i si aggiornano le componenti del vettore soluzione x_j con $j < i$ mediante la sottrazione del termine $a_{ji}^{(j)} x_i$. Nel seguente frammento di codice è implementato l'aggiornamento della i-esima componente del vettore soluzione.

```
As(ty,n) = -(As(ty,j) * As(j,n)) + As(ty,n);
```

I test condotti su questa funzione sono stati confrontati con i tempi di esecuzione per la risoluzione di una sequenza di sistemi mediante l'utilizzo della libreria LAPACK per varie configurazioni. I termini variabili per una configurazione sono il numero di sistemi (10000, 20000 e 40000) e la grandezza di un sistema (8, 12, 16, ... , 76). Nella tabella 6.4 sono esposti i tempi di esecuzione dei test risolvendo una sequenza di 20000 sistemi con entrambe le unità di elaborazione, CPU e GPU. Si noti che al variare della dimensione dei sistemi si è ottenenuto sempre un miglioramento impiegando la GPU. In particolare si osservi dalla figura 6.3 che lo speed up assume un andamento crescente fino a sistemi di dimensione 28 e poi decresce. Il motivo di questo comportamento è legato ai limiti imposti dall'architettura della GPU: più precisamente, per dimensione dei sistemi inferiore a 28 si ha una limitazione dal numero di block per SM, invece per dimensioni superiori a 28 si ha una limitazione data dalla dimensione della shared memory. Nella sezione 4.3 abbiamo visto che la dimensione della memoria condivisa per block per sfruttare al massimo la GPU è pari a $6144B$ e calcolando la dimensione ottimale dei sistemi con questa occupazione di memoria si ottiene che la dimensione ideale per un sistema è pari a $\sqrt{6144B/8} = 27,71 \simeq 28$. Infatti per tale valore della dimensione dei sistemi si ha lo speed up massimo.

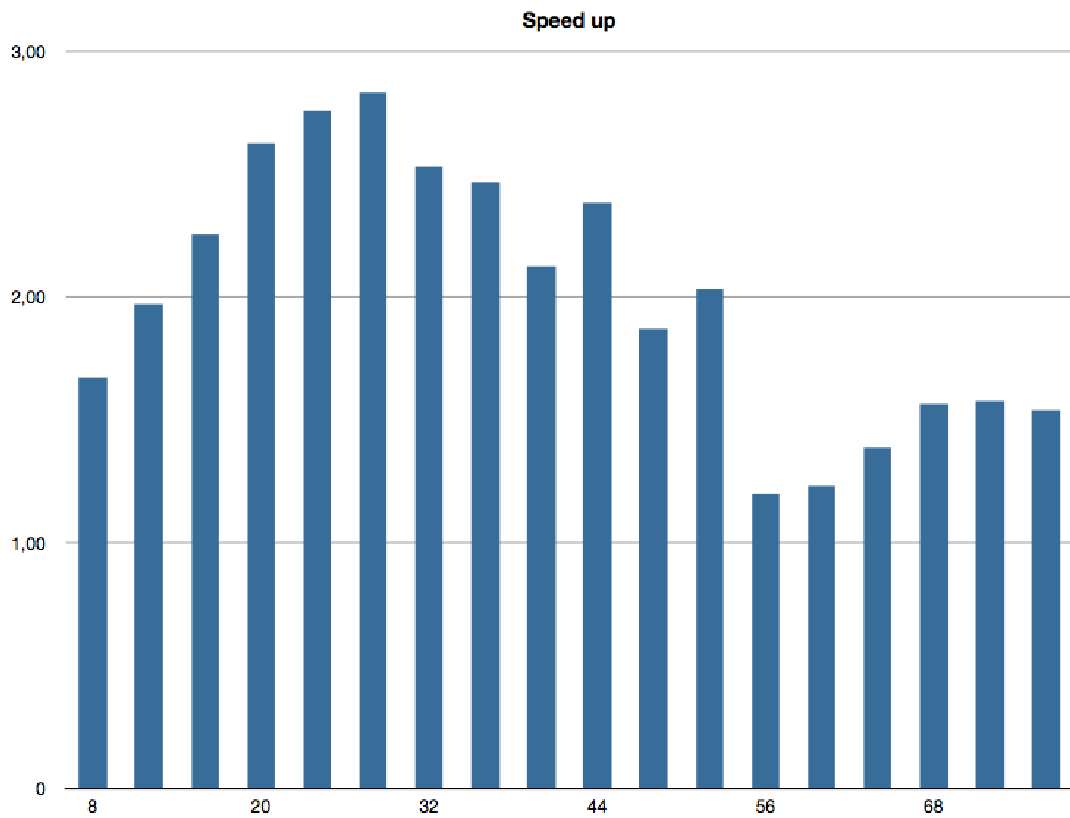


Figura 6.3: Speed up risoluzione di una sequenza di 20000 sistemi

Dimensione	T_e $[ms]$ **CPU**	T_e $[ms]$ **GPU**	**Speed up**
8	17,1	10,2	1,67
12	34,8	17,7	1,97
16	60,1	26,7	2,25
20	96,8	36,9	2,63
24	143,0	51,9	2,80
28	203,0	71,7	2,83
32	275,4	108,8	2,53
36	361,1	146,4	2,50
40	469,9	221,2	2,12
44	594,9	249,6	2,38
48	741,0	396,3	1,90
52	907,0	446,2	2,03
56	1094,7	913,6	1,20
60	1309,6	1064,0	1,23
64	1546,7	1115,3	1,39
68	1931,0	1234,1	1,56
72	2279,5	1445,9	1,58
76	2699,4	1752,9	1,54

Tabella 6.4: Tempi di esecuzione per la risoluzione di 20000 sistemi

Capitolo 7

Conclusioni

Nell'ambito delle scienze applicate il metodo del gradiente coniugato modificato trova ampio utilizzo nella soluzione dei sistemi lineari che scaturiscono da un gran numero di problemi ingegneristici. Tali sistemi lineari devono essere risolti anche migliaia di volte e di conseguenza un'efficiente implementazione di questi metodi può portare enormi vantaggi in termini di riduzione del tempo di calcolo.

L'obiettivo della tesi è stato lo studio e l'analisi del metodo del gradiente coniugato modificato su GPU. In particolar modo, sfruttando l'iperparallelismo delle schede grafiche è possibile diminuire i tempi di esecuzione del solutore lineare ottenendo un notevole risparmio in termini di tempo di esecuzione nelle simulazioni necessarie nel campo delle scienze applicate.

Nello sviluppo del gradiente coniugato modificato su GPU si sono studiate le reali potenzialità di una scheda grafica e si è compreso quali fattori bisogna considerare maggiormente per una buona implementazione su GPU. In CUDA i principali elementi da considerare sono il grado di parallelismo e l'utilizzo della memoria che, se sfruttati correttamente, possono portare a un significativo miglioramento del software sviluppato. E' stato necessario acquisire un'adeguata sensibilità nella valutazione di questi fattori al fine di ottenere performance migliori rispetto allo stesso software implementato su CPU.

Nonostante le difficoltà incontrate, i risultati raggiunti sono piuttosto soddisfacenti. Infatti, come mostrato nel capitolo 5, si è ottenuto uno speed up di 2x rispetto

all'implementazione del gradiente coniugato mediante le direttive OpenMP eseguito su CPU con 4 core. Si noti che la scheda grafica impiegata nei test ha un ridotto numero di SM, perciò le prestazioni di tale solutore possono ulteriormente migliorare mediante l'utilizzo di GPU più performanti.

Data la complessità del calcolo del precondizionatore FSAI si è deciso di implementare su GPU solo la fase computazionalmente più onerosa. In particolare ci si è concentrati sulla soluzione di una sequenza di sistemi densi di piccole dimensione, ottenendo anche in questa fase sensibili miglioramenti. Per ridurre ulteriormente il tempo di esecuzione di tale fase di calcolo si è implementato un algoritmo euristico che diminuisce il numero di sistemi da risolvere effettuando il merging di righe simili nella matrice che definisce il pattern del precondizionatore.

In seguito a queste considerazioni possiamo concludere che il metodo del gradiente coniugato modificato su GPU ottiene un importante incremento nelle prestazioni, come d'altronde ci si aspettava. Nel calcolo del precondizionatore FSAI ci sono ampi margini di miglioramento: ad esempio si potrebbe portare la raccolta dei sistemi densi su GPU, eliminando il tempo necessario al trasferimento dei sistemi nella memoria della GPU.

Bibliografia

[1] M. R. Hestenes and E. L. Stiefel, "Methods of conjugate gradients for solving linear systems", *Journal of Reasearch of the National Bureauof Standards B*, vol. 49, pp. 409-436, 1952.

[2] C. Lanczos, "Solutions of systems of linear equations by minimized-iterations", *Journal of Reasearch of the National Bureauof Standards B*, vol. 49, pp. 33-53, 1952.

[3] L. Yu. Kolotilina and A. Yu. Yeremin, "Factorized sparse approximate inverse preconditionings. I. Theory", *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 45-58, 1993.

[4] I. E. Kaporin, "New convergence results and preconditioning strategies for the coniugate gradient method", *Numerical Linear Algebra with Applications*, vol. 6, no. 7, pp. 515-531, 1999.

[5] L. Yu. Kolotilina, A. A. Nikishin, and A. Yu. Yeremin, "Factorized sparse approximate inverse preconditionings. IV. Simple approches to rising efficienty", *Numerical Linear Algebra with Applications*, vol. 1, no. 2, pp-179-210, 1994.

[6] M. Ferronato, "Preconditioning for Sparse Linear Systems at Dawn of the 21st Century: History, Current Devolpments, and Future Perspectives", *International Scholarly Research Network, ISRN Applied Mathematics*, vol. 2012, 2012.

[7] R. Grimes, D. Kincaid, and D. Young. “ITPACK 2.0 User’s Guide”, *Technical Report CNA-150*, Center for Numerical Analysis, University of Texas, August 1979

[8] OpenMP API Specifications. “OpenMP Application Program Interface”, July 2011, Version 3.1

[9] N. Bell and M. Garland. “Efficient Sparse Matrix-Vector Multiplication on CUDA”, *NVIDIA Technical Report NVR-2008-004*, December 2008

[10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. “Lapack User’s Guide”, *Society for Industrial Applied Mathematics*, Philadelphia, PA, third edition, 1999

[11] NVIDIA Corporation. “NVIDIA CUDA Programming Guide”, Ottobre 2012, Version 4.2

[12] V. Volkov. “Better Performance at Lower Occupancy”, UC Berkeley, 2010

[13] NVIDIA Corporation. “CUBLAS Library”, Ottobre 2012, Versione 5.0

[14] NVIDIA Corporation. “CUSPARSE Library”, Ottobre 2012, Versione 5.0

[15] Y. Saad, “Iterative Methods for Sparse Linear Systems”, *Second Edition*, 2003

[16] D. B. Kirk and W. W. Hwu. “Programming Massively Parallel Processors: A Hands-on Approach”, Elsevier, 2012

[17] NVIDIA Corporation. “Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Fermi”, Ottobre 2012, Version 1.1

Appendice A

Codice OpenMP

compute.c

In questo file attraverso la funzione compute_FSAI viene calcolato il precondizionatore FSAI per una determinata matrice.

```
#include "./interfaces/compute_FSAI.h"

/*
 * Calcola il precondizionatore FSAI, prefiltrandolo (delta) e postfiltrandolo (
     tau).
 *
 */

int compute_FSAI(OMPDTSTR *parDTSTR_mat_A, CSRMAT *mat_A, FSAI *prec){

    int nequ;
    int nterm;
    int nzmax_F;
    int *iat;
    int *ja;
    double *coef;
    int np;
    int *procstart;
    int *nz_Fi;
    int *iat_F;
    int *ja_F;
    double *coef_F;
```

```
double tau;
double delta;
int kappa;
int *iat_SCR;
int *ja_SCR;
double *diag_A;
int myid;
int firstrow;
int nrows_L;
int i;
int j;
int nz_F;
int indl;
int *iat_FT;
int *ja_FT;
double *coef_FT;
int **WIG1;
int *WIG2;
int ind1;
int info;

info = 0;

nequ = (*mat_A).nrows;
nterm = (*mat_A).nterm;
iat = (*mat_A).iat;
ja = (*mat_A).ja;
coef = (*mat_A).coef;

np = (*parDTSTR_mat_A).np;
procstart = (*parDTSTR_mat_A).procstart;

tau = (*prec).tau;
delta = (*prec).delta;
kappa = (*prec).kappa;
nzmax_F = 4 * kappa * nterm + np * nequ;
(*prec).nzmax_F = nzmax_F;

alloc_FSAI1(np, nequ, nzmax_F, prec);

nz_Fi = (*prec).nz_Fi;
iat_F = (*prec).iat_F;
ja_F = (*prec).ja_F;
coef_F = (*prec).coef_F;

iat_SCR = (int *) malloc((nequ + np) * sizeof(int));
if(iat_SCR == NULL) return 3;
```

```c
ja_SCR = (int *) malloc(nterm * sizeof(int));
if(ja_SCR == NULL) return 3;

diag_A = (double *) malloc(nequ * sizeof(double));
if(diag_A == NULL) return 3;

omp_set_num_threads(np);

#pragma omp parallel private(myid, firstrow, nrows_L, i, nz_F, indl)
{
    myid = omp_get_thread_num();
    firstrow = procstart[myid];
    nrows_L = procstart[myid + 1] - firstrow;
    indl = procstart[myid] + myid;
    for(i = firstrow; i < (firstrow + nrows_L); i++){

        ind1 = iat[i];
        while(ja[ind1] < i) ind1++;
        diag_A[i] = coef[ind1];

    }/*for*/
    #pragma omp barrier

    prefilter(myid, np, firstrow, nrows_L, nequ, nterm, delta, iat, ja, coef,
         diag_A, iat_SCR, ja_SCR, nz_Fi);
    nz_Fi[myid] += iat[firstrow];
    getLowerMatrixA(myid, firstrow, nrows_L, np, nzmax_F, nz_Fi, iat_SCR,
        ja_SCR, iat_F, ja_F, diag_A);

   #pragma omp barrier

    cpt_F_FSAI(myid, firstrow, np, nequ, nterm, nrows_L, nzmax_F, nz_Fi[myid
        ], tau, iat, ja, iat_F + indl, ja_F, coef, coef_F);

    #pragma omp barrier


    #pragma omp single nowait
    {

        for(i = 0; i < np; i++){
            nz_F += nz_Fi[i];
        }/*for*/

        (*prec).nz_F = nz_F;
        (*prec).nzmax_F = nzmax_F;
```

```
            (*prec).ja_F = ja_F;
        }/*#pragma omp single nowait*/

    }/*#pragma omp parallel private*/

    free(diag_A);
    free(ja_SCR);
    free(iat_SCR);

    alloc_FSAI2(nequ, (*prec).nz_F, prec);

    iat_FT = (*prec).iat_FT;
    ja_FT = (*prec).ja_FT;
    coef_FT = (*prec).coef_FT;

    WIG1 = buildMatrixOfInt(np + 1, nequ);
    if(WIG1 == NULL) return 2;
    WIG2 = (int *) malloc(np * sizeof(int));
    if(WIG2 == NULL) return 2;

    omp_set_num_threads(np);
    #pragma omp parallel private(myid)
    {
        myid = omp_get_thread_num();
        transpMatF(myid, np, nequ, nz_Fi, nzmax_F,(*prec).nz_F, procstart, iat_F,
             ja_F, iat_FT, ja_FT, WIG1, WIG2, coef_F, coef_FT);

    }/*#pragma omp parallel private*/

    deleteMatrixOfInt(WIG1);
    free(WIG2);

    return info;

}/*compute_FSAI*/

int compute_FSAI_with_pattern(OMPDTSTR *parDTSTR_mat_A, CSRMAT *mat_A, FSAI *prec
    , char *namePatternFile){

    int nequ;
    int nterm;
    int nzmax_F;
    int *iat;
    int *ja;
    double *coef;
    int np;
    int *procstart;
```

```
int *nz_Fi;
int *iat_F;
int *ja_F;
double *coef_F;
double tau;
double delta;
int kappa;
int *iat_SCR;
int *ja_SCR;
double *diag_A;
int myid;
int firstrow;
int nrows_L;
int firstrowPattern;
int i;
int j;
int l;
int nz_F;
int indl;
int *iat_FT;
int *ja_FT;
double *coef_FT;
int **WIG1;
int *WIG2;
int ind1;
int info;
CSRMAT pattern;


info = 0;

nequ = (*mat_A).nrows;
nterm = (*mat_A).nterm;
iat = (*mat_A).iat;
ja = (*mat_A).ja;
coef = (*mat_A).coef;

np = (*parDTSTR_mat_A).np;
procstart = (*parDTSTR_mat_A).procstart;

tau = (*prec).tau;
delta = (*prec).delta;
kappa = (*prec).kappa;
nzmax_F = 4 * kappa * nterm + np * nequ;
(*prec).nzmax_F = nzmax_F;

alloc_FSAI1(np, nequ, nzmax_F, prec);
```

```
nz_Fi = (*prec).nz_Fi;
iat_F = (*prec).iat_F;
ja_F = (*prec).ja_F;
coef_F = (*prec).coef_F;

readmat(&pattern, namePatternFile);

for(i = 0; i < np; i++){

    firstrow = procstart[i];
    nrows_L = procstart[i + 1] - firstrow;
    indl = procstart[i] + i;

    resize(i, np, firstrow, nrows_L, pattern.iat, pattern.ja, pattern.coef,
        iat_F, ja_F, coef_F, nz_Fi);

}//for

(*prec).nz_F = pattern.nterm;
(*prec).nzmax_F = nzmax_F;
(*prec).ja_F = ja_F;


alloc_FSAI2(nequ, (*prec).nz_F, prec);

iat_FT = (*prec).iat_FT;
ja_FT = (*prec).ja_FT;
coef_FT = (*prec).coef_FT;

WIG1 = buildMatrixOfInt(np + 1, nequ);
if(WIG1 == NULL) return 2;
WIG2 = (int *) malloc(np * sizeof(int));
if(WIG2 == NULL) return 2;

omp_set_num_threads(np);
#pragma omp parallel private(myid)
{
    myid = omp_get_thread_num();
    transpMatF(myid, np, nequ, nz_Fi, nzmax_F,(*prec).nz_F, procstart, iat_F,
         ja_F, iat_FT, ja_FT, WIG1, WIG2, coef_F, coef_FT);

}/*#pragma omp parallel private*/


deleteMatrixOfInt(WIG1);
free(WIG2);
```

```
    return info;
}/*compute_FSAI*/
```

driver.c

In questo file è presente la funzione principale main nella quale vengono effettuate le chiamate necessarie per l'esecuzione del gradiente coniugato modificato.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <string.h>
#include <time.h>

#include "./interfaces/structures.h"
#include "./interfaces/utility.h"
#include "./interfaces/compute_FSAI.h"
#include "./interfaces/PCG_solv_FSAI.h"

#define READ_PATTERN 0
#define COMPUTE_PATTERN 1


int
main(int argc, const char* argv[]){

    CSRMAT mat_A;
    OMPDTSTR parDTSTR_mat_A;
    double *vec_b;
    double *rhs;
    PCGSOLVER_FSAI PCG;
    FSAI prec;
    double *sol;
    int i;
    int j;
    int value;
    int info;
    double t_sol;
    double t_prec;
    double timespent;
```

```
int np;
int kappa;
double delta;
double tau;
int iout;
int itmax;
int isol = 1;
double tol_CG;
int nequ;
int nterm;
int *iat;
int *ja;
double *coef;
int flag;

np = atoi(argv[1]);
flag = READ_PATTERN;

info = readmat(&mat_A, "Emilia_grande.cmk");

if(info == 0) printf("Matrice␣letta␣correttamente...\n");

nequ = mat_A.nrows;
nterm = mat_A.nterm;
iat = mat_A.iat;
ja = mat_A.ja;
coef = mat_A.coef;

rhs = (double *) malloc(nequ * sizeof(double));
sol = (double *) malloc(nequ * sizeof(double));

uxvsetr(nequ, 1.0, sol);
axbnsy(nequ, nequ, nterm, iat, ja, coef, sol, rhs);
uxvsetr(nequ, 0.0, sol);

mk_OMPDTSTR(nequ, np, np, &parDTSTR_mat_A);
mk_PCGSOLVER_FSAI(nequ, 100, 10000, isol, 1e-10, &PCG);
set_FSAI(0, 0, nequ, np, RDEF, 2, 0.0f, &prec);
if(flag == COMPUTE_PATTERN){
    t_prec = omp_get_wtime();
    info = compute_FSAI(&parDTSTR_mat_A, &mat_A, &prec);
    t_prec = omp_get_wtime() - t_prec;
}//if
if(flag == READ_PATTERN){

    t_prec = omp_get_wtime();
    info = compute_FSAI_with_pattern(&parDTSTR_mat_A, &mat_A, &prec, "
```

```
            FSAI_Emilia");
        t_prec = omp_get_wtime() - t_prec;

    }///if
    if(info == 0) printf("Precondizionatore␣calcolato␣correttamente!!!!\n");
    else return 1;


    PCG_solv_FSAI(&mat_A, &parDTSTR_mat_A, &prec, &PCG, rhs, sol);

    printf("%g␣%d\n", t_sol, PCG.n_iter);

    printf("bnorm␣%g\n", PCG.bnorm);
    printf("n_iter␣%d\n", PCG.n_iter);
    printf("resiter␣%g\n", PCG.resiter);
    printf("resreal␣%g␣\n", PCG.resreal);

    dlt_CSRMAT(&mat_A);
    dlt_OMPDTSTR(&parDTSTR_mat_A);
    free(rhs);
    dlt_PCGSOLVER_FSAI(&PCG);
    dlt_FSAI(&prec);
    free(sol);

    return 0;
}/*main*/
```

PCG_solv_FSAI.c

In questo file è implementato il gradiente coniugato modificato. In particolare è composto da un'unica funzione PCG_solv_FSAI che accetta in input il sistema lineare e fornisce la soluzione cercata.

```
#include "./interfaces/PCG_solv_FSAI.h"

/*
 * Risolve il sistema con il PCG
 *
 */
```

```
int PCG_solv_FSAI(CSRMAT *mat_A, OMPDTSTR *parDTSTR_mat_A, FSAI *prec,
    PCGSOLVER_FSAI *PCG, double *rhs, double *sol){

    int exit_test;
    int myid;
    int firstrow;
    int nrows;
    int iter;
    int i;
    int prec_nequ;
    int prec_nnz;
    int prec_np;
    int prec_nb;
    double alpha;
    double beta;
    double ptap;
    double resini;
    double resiter;
    double resreal;
    double bnorm;
    double *sol_loc;
    double *res_loc;
    double *p_loc;
    double *Axp_loc;
    double *pres_loc;
    double *WR1_loc;
    double *ridv_1;
    double *ridv_2;
    int *vinfo;
    int np;
    int nb;
    int itmax;
    double *res;
    double *p;
    int *vecstart;
    int *procstart;
    int *procbloc;
    double *WR1;
    double tol_CG;
    int *iat;
    int *ja;
    double *coef;
    int nterm;
    int nequ;
    int info;

    np = (*parDTSTR_mat_A).np;
```

```
nb = (*parDTSTR_mat_A).nb;
vecstart = (*parDTSTR_mat_A).vecstart;
procstart = (*parDTSTR_mat_A).procstart;
procbloc = (*parDTSTR_mat_A).procbloc;

itmax = (*PCG).itmax;
res = (*PCG).res;
p = (*PCG).p;
WR1 = (*PCG).WR1;
tol_CG = (*PCG).tol_CG;
nequ = (*PCG).nequ;
exit_test = itmax;

iat = (*mat_A).iat;
ja = (*mat_A).ja;
coef = (*mat_A).coef;
nterm = (*mat_A).nterm;

prec_nequ = (*prec).nequ;
prec_nnz = (*prec).nz_F;
prec_np = (*prec).np;
prec_nb = (*prec).np;

if(((*mat_A).nrows != prec_nequ) || ((*mat_A).nrows != nequ)) return 1;

if((np != prec_np) || (nb != prec_nb)) return 2;

vinfo = (int *) malloc(sizeof(int) * np);
if(vinfo == NULL) return 3;
ridv_1 = (double *) malloc(sizeof(double) * np);
if(ridv_1 == NULL){
    free(vinfo);
    return 3;
}/*if*/
ridv_2 = (double *) malloc(sizeof(double) * np);
if(ridv_2 == NULL){
    free(vinfo);
    free(ridv_1);
    return 3;
}/*if*/

uxvseti(np, 0, vinfo);

omp_set_num_threads(np);

#pragma omp parallel private(myid, firstrow, nrows, iter, i, sol_loc, res_loc
    , p_loc, Axp_loc, pres_loc, WR1_loc, resini, resiter, alpha, beta, ptap,
```

```
    bnorm)

{
    myid = omp_get_thread_num();
    firstrow = procstart[myid];
    nrows = procstart[myid + 1] - firstrow;

    sol_loc = (double *) malloc(sizeof(double) * nrows);
    if(sol_loc == NULL) vinfo[myid] = 1;
    res_loc = (double *) malloc(sizeof(double) * nrows);
    if(res_loc == NULL) vinfo[myid] = 2;
    p_loc =  (double *) malloc(sizeof(double) * nrows);
    if(p_loc == NULL) vinfo[myid] = 3;
    Axp_loc =  (double *) malloc(sizeof(double) * nrows);
    if(Axp_loc == NULL) vinfo[myid] = 4;
    pres_loc =  (double *) malloc(sizeof(double) * nrows);
    if(pres_loc == NULL) vinfo[myid] = 5;
    WR1_loc =  (double *) malloc(sizeof(double) * nrows);
    if(WR1_loc == NULL) vinfo[myid] = 6;

    #pragma omp barrier

    for(i = 0; i < np; i++){
        if(vinfo[i] != 0){
            if(myid == i){
                if(vinfo[i] > 0) free(sol_loc);
                if(vinfo[i] > 1) free(res_loc);
                if(vinfo[i] > 2) free(p_loc);
                if(vinfo[i] > 3) free(Axp_loc);
                if(vinfo[i] > 4) free(pres_loc);
                if(vinfo[i] > 5) free(WR1_loc);
            }/*if*/
            else{
                free(sol_loc);
                free(res_loc);
                free(p_loc);
                free(Axp_loc);
                free(pres_loc);
                free(WR1_loc);
            }/*else*/
            info = 3;
        }/*if*/
    }/*for*/

    if((*PCG).isol != 0) apply_FSAI(myid, firstrow, nrows, nequ, procbloc,
        prec, rhs, WR1, sol_loc);
    else cblas_dcopy(nrows, sol + firstrow, 1, sol_loc, 1);
```

```
//CALCOLO DELLA NORMA DI b

bnorm = dnrm2_par(myid, np, nrows, rhs + firstrow, ridv_1, 1.0);

if(bnorm  == 0) vinfo[myid] = 1;
for(i = 0; i < np; i++){
    if(vinfo[i] != 0){

        free(sol_loc);
        free(res_loc);
        free(p_loc);
        free(Axp_loc);
        free(pres_loc);
        free(WR1_loc);

        uxvsetr(nrows, 0.0f, sol);

        info = 4;

    }/*if*/
}/*for*/

//CALCOLO DI r = b - A * x


resid_par(firstrow, nrows, nequ, nterm, iat, ja, coef, sol_loc, WR1,
    WR1_loc, rhs + firstrow, res_loc);

//CALCOLO DELLA NORMA DI r

resini = dnrm2_par(myid, np, nrows, res_loc, ridv_1, bnorm);

for(iter = 0; iter < itmax; iter++){


    cblas_dcopy(nrows, res_loc, 1, res + firstrow, 1);

    if((*PCG).isol != 0)apply_FSAI(myid, firstrow, nrows, nequ, procbloc,
         prec, res, WR1, pres_loc);
    else cblas_dcopy(nrows, res + firstrow, 1, pres_loc, 1);


    if(iter == 0){

        for(i = 0; i < nrows; i++){
            p_loc[i] = pres_loc[i];
```

```
        }/*for*/


    }else{

        beta = ddot_par(myid, np, nrows, pres_loc, Axp_loc, ridv_1, (-
            ptap));

        for(i = 0; i < nrows; i++){
            p_loc[i] = pres_loc[i] + beta * p_loc[i];
        }/*for*/


    }/*else*/


    cblas_dcopy(nrows, p_loc, 1, p + firstrow, 1);

    #pragma omp barrier

    axbnsy(nrows, nequ, nterm, iat + firstrow, ja, coef, p, Axp_loc);
    ptap = ddot_par(myid, np, nrows, p_loc, Axp_loc, ridv_1, 1.0);
    alpha = ddot_par(myid, np, nrows, res_loc, p_loc, ridv_2, ptap);

    for(i = 0; i < nrows; i++){
        sol_loc[i] += alpha * p_loc[i];
        res_loc[i] -= alpha * Axp_loc[i];

    }/*for*/

    resiter = dnrm2_par(myid, np, nrows, res_loc, ridv_1, bnorm);
    if(resiter <= tol_CG){
        exit_test = iter;
        iter = itmax + 1;
    }/*if*/

}/*for*/

if(iter >= itmax && exit_test != 0) info = 5;

cblas_dcopy(nrows, sol_loc, 1, sol + firstrow, 1);

#pragma omp barrier

axbnsy(nrows, nequ, nterm, iat + firstrow, ja, coef, sol, WR1_loc);
cblas_daxpy(nrows, (-1.0), rhs + firstrow, 1, WR1_loc, 1);
resreal = dnrm2_par(myid, np, nrows, WR1_loc, ridv_1, bnorm);
```

```
        #pragma omp single nowait
        {
            (*PCG).n_iter = exit_test;
            (*PCG).resini = resini;
            (*PCG).resiter = resiter;
            (*PCG).resreal = resreal;
            (*PCG).bnorm = bnorm;

        }/*#pragma omp single nowait*/

        free(sol_loc);
        free(res_loc);
        free(p_loc);
        free(Axp_loc);
        free(pres_loc);
        free(WR1_loc);

        #pragma omp barrier
    }

    free(vinfo);
    free(ridv_1);
    free(ridv_2);

    return info;

}/*PCG_solv_FSAI*/
```

structure.c

In questo file sono create le strutture necessarie per il calcolo del precondizionatore FSAI e per l'esecuzione del gradiente coniugato modificato.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#include "./interfaces/structures.h"

/*
```

```
 * Definisce la struttura dati per la matrice 'mat_A' salvata nel formato CSR.
 *
 */

int mk_CSRMAT(CSRMAT *mat_A,int nn, int nt){

    (*mat_A).nrows = nn;
    (*mat_A).nterm = nt;

    (*mat_A).iat = (int *) malloc(sizeof(int) * ((*mat_A).nrows + 1));
        if((*mat_A).iat == NULL){
                return 1;
        }/*if*/

        (*mat_A).ja = (int *) malloc(sizeof(int) * (*mat_A).nterm);
        if((*mat_A).ja == NULL){
                return 1;
        }/*if2*/

    (*mat_A).coef = (double *) malloc(sizeof(double) * (*mat_A).nterm);
        if((*mat_A).ja == NULL){
                return 1;
        }/*if*/

    return 0;

}/*mk_CSRMAT*/

/*
 * Eliminazione della struttura dati della matrice 'mat_A'.
 *
 */

int dlt_CSRMAT(CSRMAT *mat_A){

    free((*mat_A).iat);
    free((*mat_A).ja);
    free((*mat_A).coef);

    return 0;

}/*dlt_CSRMAT*/

/*
 * Creazione della struttura dati, 'DTSTR_var', per la gestione parallela dei
     vettori e delle matrici.
 *
```

```
 */

int mk_OMPDTSTR(int nequ, int nbloc, int nproc, OMPDTSTR *DTSTR_var){

    int nproc_available;
    int i, kk, bsize, nbproc, ind_bloc;

    if(nproc > nbloc) return 1;

    if(nbloc > nequ) return 4;

    nproc_available = omp_get_num_procs();

    if(nproc > nproc_available) printf("WARNING␣\n#␣thread␣>␣#␣processors␣
        available");

    (*DTSTR_var).np = nproc;
    (*DTSTR_var).nb = nbloc;

    (*DTSTR_var).vecstart = (int *)malloc(sizeof(int) * (nbloc + 1));
    if((*DTSTR_var).vecstart == NULL) return 2;

    (*DTSTR_var).procstart = (int *)malloc(sizeof(int) * (nproc + 1));
    if((*DTSTR_var).procstart == NULL) return 2;

    (*DTSTR_var).procbloc = (int *)malloc(sizeof(int) * (nproc + 1));
    if((*DTSTR_var).procbloc == NULL) return 2;

    bsize = nequ / nbloc;
    kk = nequ % nbloc;
    (*DTSTR_var).vecstart[0] = 0;
    (*DTSTR_var).vecstart[1] = bsize + kk;

    for (i = 2; i < (nbloc + 1); i++) {
        (*DTSTR_var).vecstart[i] = (*DTSTR_var).vecstart[i - 1] + bsize;
    }/*for*/

    nbproc = nbloc / nproc;
    kk = nbloc % nproc;
    (*DTSTR_var).procbloc[0] = 0;
    (*DTSTR_var).procstart[0] = 0;
    ind_bloc = nbproc + kk;
    (*DTSTR_var).procbloc[1] = ind_bloc;
    (*DTSTR_var).procstart[1] = (*DTSTR_var).vecstart[ind_bloc];

    for(i = 2; i < (nproc + 1); i++){
        ind_bloc += nbproc;
```

```
        (*DTSTR_var).procbloc[i] = ind_bloc;
        (*DTSTR_var).procstart[i] = (*DTSTR_var).vecstart[ind_bloc];
    }/*for*/

    return 0;

}/*mk_OMPDTSTR*/

/*
 * Eliminazione della struttra dati 'DTSTR_var'.
 *
 */

int dlt_OMPDTSTR(OMPDTSTR *DTSTR_var){

    free((*DTSTR_var).vecstart);
    free((*DTSTR_var).procbloc);
    free((*DTSTR_var).procstart);

    return 0;

}/*mk_OMPDTSTR*/

/*
 * Creazione della struttura dati, 'PCG_var', per l'utilizzo del PCG.
 *
 */

int mk_PCGSOLVER_FSAI(int nequ, int iout, int itmax, int isol, double tol_CG,
    PCGSOLVER_FSAI *PCG_var){

    if(nequ <= 0){

        printf("WARNING:␣neq␣<␣0");
        return 2;

    }/*if*/

    if(itmax <= 0){

        printf("WANING:␣itmax␣<␣0");
        return 2;

    }/*if*/

    if(tol_CG <= 0){
```

```
        printf("WARNING:␣tol_CG␣<␣0");
        return 2;

    }/*if*/

    (*PCG_var).nequ = nequ;
    (*PCG_var).iout = iout;
    (*PCG_var).itmax = itmax;
    (*PCG_var).isol = isol;
    (*PCG_var).tol_CG = tol_CG;

    (*PCG_var).res = (double *) malloc(sizeof(double) * nequ);
    if((*PCG_var).res == NULL){
        return 1;
    }/*if*/

    (*PCG_var).p =(double *) malloc(sizeof(double) * nequ);
    if((*PCG_var).p== NULL){
        return 1;
    }/*if*/

    (*PCG_var).WR1 = (double *) malloc(sizeof(double) * nequ);
    if((*PCG_var).WR1 == NULL){
        return 1;
    }/*if*/

    return 0;

}/*mk_PCGSOLVER_FSAI*/

/*
 * Eliminazione della struutra dati 'PCG_var'.
 *
 */

int dlt_PCGSOLVER_FSAI(PCGSOLVER_FSAI *PCG_var){

    free((*PCG_var).res);
    free((*PCG_var).p);
    free((*PCG_var).WR1);

    return 0;

}/*dlt_PCGSOLVER_FSAI*/

/*
 * Impostazione dei parametri per FSAI.
```

```
 *
 */

int set_FSAI(int DEBUG, int PATT0, int nequ, int np, double delta, int kappa,
    double tau, FSAI *prec){

    (*prec).DEBUG = DEBUG;
    (*prec).PATT0 = PATT0;
    if(nequ > 0){
        (*prec).nequ = nequ;
    }else{
        return 1;
    }/*else*/

    (*prec).np = np;

    if(kappa == IDEF){
      kappa = 2;
    }else{
        if (kappa > 1) {
            (*prec).kappa = kappa;
        }else return 1;
    }/*else*/

    if(delta == RDEF){
        delta = 0;
    }else{
        if (delta >= 0) {
            (*prec).delta = delta;
        }else return 1;
    }/*else*/

    if(tau == RDEF){
        tau = 0;
    }else{
        if (tau >= 0) {
            (*prec).tau = tau;
        }else return 1;
    }/*else*/

    return 0;

}/*set_FSAI*/

/*
 * Creazione della prima parte del precondizionatore 'prec'.
 *
```

```
 */

int alloc_FSAI1(int np, int nrows, int nzmax_F, FSAI *prec){

    (*prec).nz_Fi = (int *) malloc(np * sizeof(int));
    if((*prec).nz_Fi == NULL) return 2;

    (*prec).iat_F = (int *) malloc((nrows + np) * sizeof(int));
    if((*prec).iat_F == NULL) return 2;

    (*prec).ja_F = (int *) malloc(nzmax_F * sizeof(int));
    if((*prec).iat_F == NULL) return 2;

    (*prec).coef_F = (double *) malloc(nzmax_F * sizeof(double));
    if((*prec).coef_F == NULL) return 2;

    return 0;


}/**alloc_FSAI1*/

/*
 * Applica il precondizionatore.
 *
 */

void apply_FSAI(int myid, int firstrow, int nrows, int nequ, int *procbloc, FSAI
    *prec, double *vec, double *vscr, double *pvec_loc){

    FSAI_precond(myid, firstrow, (*prec).np, nrows, nequ, (*prec).nzmax_F, (*prec
        ).nz_F, procbloc, (*prec).iat_F, (*prec).ja_F, (*prec).iat_FT, (*prec).
        ja_FT,(*prec).coef_F, (*prec).coef_FT, vec, vscr, pvec_loc);


}/*apply_FSAI*/

/*
 * Creazione della seconda parte del precondizionatore 'prec'.
 *
 */

int alloc_FSAI2(int nrows, int nz_FT, FSAI *prec){

    (*prec).iat_FT = (int *) malloc((nrows + 1) * sizeof(int));
    if((*prec).iat_FT == NULL) return 2;

    (*prec).ja_FT = (int *) malloc(nz_FT * sizeof(int));
```

```
    if((*prec).ja_FT == NULL) return 2;

    (*prec).coef_FT = (double *) malloc(nz_FT * sizeof(double));
    if((*prec).coef_FT == NULL) return 2;

    return 0;

}/*alloc_FSAI2*/

/*
 * Eliminazione del precondizionatore 'prec'.
 *
 */

int dlt_FSAI(FSAI *prec){

    free((*prec).coef_FT);
    free((*prec).ja_FT);
    free((*prec).iat_FT);
    free((*prec).coef_F);
    free((*prec).ja_F);
    free((*prec).iat_F);
    free((*prec).nz_Fi);

    return 0;

}/*dlt_FSAI*/

/*
 * Creazione di una matrice di double height x width
 *
 */
double ** buildMatrixOfDouble(int height, int width) {

        int i;
        double *buffer;
        int bufferSize;
        double **matrix;

        bufferSize = height * width;
        buffer = (double *) malloc(bufferSize * sizeof(double));
        if (buffer == NULL) {
                printf("\n\nCan't␣allocate␣memory␣!!!\n");
                exit(1);
        }/*if*/

        matrix = (double **) malloc(height * sizeof(double *));
```

```
        if (matrix == NULL) {
                printf("\n\nCan't␣allocate␣memory␣!!!\n");
                exit(1);
        }/*if*/
        for (i = 0; i < height; i++) {

                matrix[i] = buffer + (i * width);

        }/*for*/

        return matrix;

}/*buildMatrixOfDouble*/


/*
 * Creazione di una matrice di int height x width
 *
 */

int ** buildMatrixOfInt(int height, int width){

        int i;
        int *buffer;
        int bufferSize;
        int **matrix;

        bufferSize = height * width;
        buffer = (int *) malloc(bufferSize * sizeof(int));
        if (buffer == NULL) {
                printf("\n\nCan't␣allocate␣memory␣!!!\n");
                exit(1);
        }/*if*/

        matrix = (int **) malloc(height * sizeof(int *));
        if (matrix == NULL) {
                printf("\n\nCan't␣allocate␣memory␣!!!\n");
                exit(1);
        }/*if*/
        for (i = 0; i < height; i++) {

                matrix[i] = buffer + (i * width);

        }/*for*/

        return matrix;
```

```
}/*buildMatrixOfInt*/


/*
 * Eliminazione della matrice di double 'matrix'
 *
 */

void deleteMatrixOfDouble(double **matrix){

    free(matrix[0]);
    free(matrix);

}/*deleteMatrixOfDouble*/

/*
 * Eliminazione della matrice di int 'matrix'
 *
 */

void deleteMatrixOfInt(int **matrix){

    free(matrix[0]);
    free(matrix);

}/*deleteMatrixOfInt*/
```

utility.c

In questo file sono implementate le principali funzioni di algebra lineare mediante le direttive openMP.

```
#include "./interfaces/utility.h"

/*
 * Calcola b = A * x .
 *
 */
void axbnsy(int n, int ntot, int nterm, int *iat, int *ja, double *coef, double *
    xvec, double *bvec){
```

```
    int i;
    int j;
    int m;
    int mm;

    for(i = 0; i < n; i++){

        m = iat[i];
        mm = iat[i + 1];

        bvec[i] = coef[m] * xvec[ja[m]];

        for(j = m + 1; j < mm; j++){

            bvec[i] = bvec[i] + coef[j] * xvec[ja[j]];

        }/*for*/

    }/*for*/
}/*axbnsy*/

/*
 * Imposta tutti i coefficienti dell'array di interi 'ix' al valore 'ia'.
 *
 */
void uxvseti(int n, int ia, int *ix){

    int i;

    for(i = 0; i < n; i++){

        ix[i] = ia;

    }/*for*/

}/*uxvseti*/

/*
 * Imposta tutti i coefficienti dell'array di double 'rx' al valore 'ra'.
 *
 */
void uxvsetr(int n, double ra, double *rx){

    int i;

    for(i = 0; i < n; i++){
```

```
        rx[i] = ra;

    }/*for*/

}/*uxvseti*/

/*
 * Calcolo in parallelo del residuo r = b - A * x .
 *
 */
void resid_par(int firstrow, int nrows, int nequ, int nterm, int *iat, int *ja,
    double *coef, double *x_loc, double *vscr, double *vscr_loc, double *b_loc,
    double *r_loc){

    int i;

    cblas_dcopy(nrows, x_loc, 1,  vscr + firstrow, 1);

    #pragma omp barrier

    axbnsy(nrows, nequ, nterm, iat + firstrow, ja, coef, vscr, vscr_loc);
    for(i = 0; i < nrows; i++){
        r_loc[i] = b_loc[i] - vscr_loc[i];
    }/*for*/

}/*resid_par*/

/*
 * Calcolo, in parallelo, del prodotto scalare tra il vettore 'v_loc' e 'w_loc'.
 *
 */

double ddot_par(int myid, int nproc, int nrows, double *v_loc, double *w_loc,
    double *ridv, double fac){

    int i;
    double a;
    double dotp;

    a = 0;

    for(i = 0; i < nrows; i++){

        a += v_loc[i] * w_loc[i];

    }/*for*/
```

```
    ridv[myid] = a;

    #pragma omp barrier

    dotp = 0;

    for(i = 0; i < nproc; i++){

        dotp += ridv[i];

    }/*for*/

    dotp = dotp / fac;

    return dotp;

}/*ddot_par*/

/*
 * Calcolo, in parallelo, della norma euclideana del vettore 'v_loc' divisa per
     il coefficiente 'fac'.
 * norm = sqrt(v^T * v) / fac
 *
 */

double dnrm2_par(int myid, int nproc, int nrows, double *v_loc, double *ridv,
    double fac){

    int i;
    double a;
    double norm;

    a = 0;

    for(i = 0; i < nrows; i++){

        a += pow(v_loc[i],2);

    }/*for*/

    ridv[myid] = a;

    #pragma omp barrier

    norm = 0;

    for(i = 0; i < nproc; i++){
```

```
        norm += ridv[i];

    }/*for*/

    norm = sqrt(norm) / fac;

    return norm;

}/*dnrm2_par*/

/*
 * Lettura della matrice A salvata in formato di coordinate e salvata nel formato
      CSR.
 *
 */

int readmat(CSRMAT *mat_A, char *filename){

    FILE *fileMat;
    int i;
    int j;
    int prerow;
    int row;
    int nrows;
    int nterm;
    int flag;

    fileMat = (FILE *)fopen(filename, "r");

    if(fileMat == NULL){

                printf("Il␣file␣non␣e'␣stato␣aperto␣correttamente");
        return 1;

        }/*if*/

        if(fscanf(fileMat, "%d␣%d", &nrows, &nterm) != 2){
        printf("Lettura␣dei␣parametri␣errata!!!\n");
        return 1;
    }/*if*/

        flag = mk_CSRMAT(mat_A,nrows,nterm);

    if(flag != 0){
        printf("Errore␣nella␣creazione␣della␣matrice␣A\n");
        return 1;
```

```
    }/*if*/

    prerow = 0;
    j = 0;

    for (i = 0; i < (*mat_A).nterm; i++) {


        if(fscanf(fileMat, "%d %d %lf", &row , &((*mat_A).ja[i]), &((*mat_A).coef
            [i])) != 3){

            printf("Errore nella lettura della matrice...");
            return 1;

        }/*if*/

        (*mat_A).ja[i]--;

        if(row > prerow){

            (*mat_A).iat[j] = i;
            j++;
            prerow = row;
        }/*if*/

    }/*for*/

    (*mat_A).iat[j] = (*mat_A).nterm;

        fclose(fileMat);

    return 0;

}/*readmat*/

/*
 * Calcolo, in parallelo, della matrice triangolare bassa compresa la diagonale.
 *
 */
void getLowerMatrixA(int myid, int firstrow, int nrows, int np, int nzmax_F, int
    *nz_Fi, int *iat, int *ja, int *iat_F, int *ja_F, double *diag_A){

    int i;
    int j;
    int k;
    int util;
    int value1;
```

```
    int value2;
    int nz;

    util = firstrow + myid;
    iat_F[util] = nzmax_F / np * myid;

    j = 0;
    nz = 0;

   for(i = 0; i < nrows; i++){

        value1 = iat[util + i];
        value2 = iat_F[util + j];
        k = 0;
        while(ja[value1 + k] <= (firstrow + i) && (value1 + k) < iat[util + i +
            1] ){

            ja_F[value2 + k] = ja[value1 + k];
            nz++;
            k++;

        }/*while*/

        j++;
        iat_F[util + j] = iat_F[util + j - 1] + k;

    }/*for*/

    nz_Fi[myid] = nz;

}/*getLowerMatrixA*/

/*
 * Conteggio del numero di termini non zero della matrice F assegnato al
     processore 'myid' per ogni riga di F^T.
 *
 */
void count_rowterms(int myid, int nrows, int nequ, int nterm_F, int *ja_F, int *
    WI){

    int i;
//    int zeri;

    uxvseti(nequ, 0, WI);

    for(i = 0; i < nterm_F; i++){
```

```
        WI[ja_F[i]]++;

    }/*for*/

}/*count_rowterms*/

/*
 * Calcolo, in parallelo, dei coefficienti della matrice F.
 *
 */

int cpt_F_FSAI(int myid, int firstrow, int nproc, int nequ, int nterm, int nrows,
     int nzmax_F, int nz_F, double tau, int *iat, int *ja, int *iat_F, int *ja_F,
     double *mat_A, double *mat_F){

    int *del_list;
    double **full_A;
    double **full_B;
    double *rhs;
    double *deleted;
    int mmax;
    int i;
    int j;
    int mm;
    int nn;
    int istart_F;
    int istart_F_old;
    int istop_F_old;
    int ind_del;
    int ind;
    int ind_F;
    int mrow;
    int irow;
    int irow_glo;
    int shift;
    double abstol;
    double fac;
    double fac1;
    double scal_1;
    double scal_2;
    int info;
    int val;
    double norma;

    val = 1;
    mmax = 0;
```

```
for(i = 0; i < nrows; i++){
    j = (iat_F[i + 1] - iat_F[i]);
    mmax = MAX(mmax, iat_F[i + 1] - iat_F[i]);

}/*for*/

full_A = buildMatrixOfDouble(mmax, mmax);
full_B = buildMatrixOfDouble(mmax, mmax);
del_list = (int *) malloc(mmax * sizeof(int));
if(del_list == NULL) return 1;
deleted = (double *) malloc(mmax * sizeof(double));
if(deleted == NULL) return 1;
rhs = (double *) malloc(mmax * sizeof(double));
if(rhs == NULL) return 1;

shift = firstrow;
ind_F = iat_F[0];
istop_F_old = ind_F;

for (irow = 0; irow < nrows; irow++){

    irow_glo = irow + shift;
    istart_F = ind_F;
    istart_F_old = istop_F_old;
    istop_F_old = iat_F[irow + 1];
    mrow = istop_F_old - istart_F_old;
    gather_fullsys_FSAI(mrow, irow_glo, nequ, nterm, mmax, iat, ja, ja_F +
        istart_F_old, mat_A, full_A, rhs);
    cblas_dcopy(mmax * mrow, *full_A, 1, *full_B, 1);

    dpotrf_("U", &mrow, *full_A, &mmax, &info);

    dpotrs_("U", &mrow, &val, *full_A, &mmax, rhs, &mrow, &info);
    scal_1 = 1.0f / rhs[mrow - 1];
    if(tau > 0){
        norma = cblas_dnrm2(mrow, rhs, 1);
        abstol = tau * norma;
        ind_del = 0;
        ind = istart_F_old;

        for(i = 0; i < mrow - 1; i++){
            if(fabs(rhs[i]) > abstol){
                mat_F[ind_F] = rhs[i];
                ja_F[ind_F] = ja_F[ind + i];
                ind_F++;
            }else{
                ind_del++;
```

```
                del_list[ind_del] = i;
                deleted[ind_del] = rhs[i];
            }/*else*/

        }/*for*/

        mat_F[ind_F] = rhs[mrow - 1];
        ja_F[ind_F] = ja_F[ind + mrow - 1];
        ind_F++;

        fac = 0;
        for(j = 0; j < ind_del; j++){
            nn = del_list[j];
            fac1 = deleted[j];
            fac += full_B[nn][nn] * pow(fac1, 2);
            for(i = j + 1; i < ind_del; i++){
                mm = del_list[i];
                fac += full_B[mm][nn] * fac1 * deleted[i];
            }/*for*/
        }/*for*/
        fac *= scal_1;
        scal_2 = sqrt(scal_1 / (1.0f + fac));
    }else{
        ind = istart_F_old;
        for(i = 0; i < mrow; i++){
            mat_F[ind_F] = rhs[i];
            ja_F[ind_F] = ja_F[ind + i];
            ind_F++;
        }/*for*/
        scal_2 = sqrt(scal_1);
    }/*else*/
    cblas_dscal((ind_F - istart_F), scal_2, mat_F + istart_F, 1);
    iat_F[irow + 1] = ind_F;
}/*for*/


nz_F = ind_F - iat_F[0];

deleteMatrixOfDouble(full_A);
deleteMatrixOfDouble(full_B);
free(del_list);
free(deleted);
free(rhs);

return 0;

}/*cpt_F_FSAI*/
```

```
/*
 * Raccoglie, il sistema da risolvere, per determinare i coefficienti di F dalla
     matrice A.
 *
 */

void gather_fullsys_FSAI(int mrow, int jendbloc, int nequ, int nterm, int mmax,
    int *iat, int *ja, int *vecinc, double *mat_A, double **full_A, double *rhs){

    int i;
    int k;
    int ii;
    int jj;
    int row;
    int endrow;

    for (i = 0; i < mrow; i++){

        ii = i;
        row = vecinc[i];
        jj = iat[row];
        endrow = iat[row + 1];

        while (ii < mrow){ /*row loop*/

            while (ja[jj] < vecinc[ii]){ /*col loop*/

                jj++;
                if(jj == endrow){
                    for(k = ii; k < mrow; k++){
                        full_A[k][i] = 0.0f;
                    }/*for*/
                }/*if*/

            }/*while*/

            if (vecinc[ii] == ja[jj]){

                full_A[ii][i] = mat_A[jj];
                ii++;

            }else{

                full_A[ii][i] = 0.0f;
                ii++;
```

```
            }/*else*/

        }/*while*/

    }/*for*/

    uxvsetr(mrow, 0.0f, rhs);

    rhs[mrow - 1] = 1;

}/*gather_fullsys_FSAI*/

/*
 * Esegue il prefiltraggio della matrice A.
 * a(i,j) < delta * a(i,i) * a(j,j)
 *
 */

void prefilter(int myid, int np, int firstrow, int nrows_L, int nrows_A, int
    nterm_A, double delta, int *iat, int *ja, double *coef, double *diag_A, int *
    iat_SCR, int *ja_SCR, int *nz_Fi){

    int i;
    int j;
    int jcol;
    int ind1;
    int ind2;
    int ind3;
    double toll;
    double pot;

    nz_Fi[myid] = 0;
    ind1 = firstrow;
    ind2 = iat[firstrow];
    ind3 = firstrow + myid;
    iat_SCR[ind3] = ind2;

    for(i = 0; i < nrows_L; i++){

        toll = diag_A[ind1 + i] * delta;
        for(j = iat[ind1 + i]; j < iat[ind1 + i + 1]; j++){

            jcol = ja[j];
            pot = pow(coef[j],2);
            if(pot > toll * diag_A[jcol]){
                nz_Fi[myid]++;
                ja_SCR[ind2] = jcol;
```

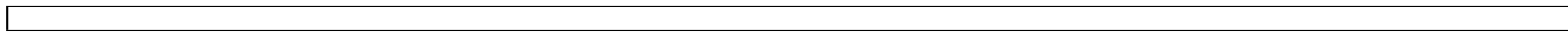
```
                ind2++;
            }/*if*/
        }/*for*/

        iat_SCR[ind3 + i + 1] = ind2;
    }/*for*/

}/*prefilter*/

/*
 * Calcolo, in parallelo, della trasposta della matrice F.
 *
 */

void transpMatF(int myid, int nproc, int nequ, int *nterm_F, int ntmax_F, int
    nterm_FT, int *procstart, int *iat_F, int *ja_F, int *iat_FT, int *ja_FT, int
     **WI1, int *WI2, double *mat_F, double *mat_FT){

    int firstrow;
    int nrows;
    int istart_F;
    int istart1_F;

    firstrow = procstart[myid];
    nrows = procstart[myid + 1] - firstrow;
    istart_F = firstrow + myid;
    istart1_F = iat_F[istart_F];

    count_rowterms(myid, nrows, nequ, nterm_F[myid], ja_F + istart1_F, WI1[myid +
         1]);
    #pragma omp barrier

    WI2[myid] = mkiat_FTloc(nrows, nequ, nproc, firstrow, WI1, iat_FT + firstrow)
        ;
    #pragma omp barrier

    mkiat_FTglo(myid, nrows, nequ, nproc, firstrow, WI1, WI2, iat_FT + firstrow);
    #pragma omp barrier

    mvcoef(myid, firstrow, nrows, nequ, ntmax_F, nterm_FT, iat_F + istart_F, ja_F
        , ja_FT, mat_F, mat_FT, WI1[myid]);

}/*transpMatF*/

/*
 * Ogni processore imposta i puntatori all'inzio di ogni riga della matrice FT.
 * Calcolo in locale del vettore iat_FT.
```

```
 *
 */

int mkiat_FTloc(int nrows, int nequ, int nproc, int firstrow, int **WI, int *
    iat_FT){

    int i;
    int j;
    int tmp1;
    int tmp2;
    int nnz;
    for(i = 0; i < nrows; i++){
        iat_FT[i] = WI[1][firstrow + i];
    }/*for*/

    for(j = 2; j < (nproc + 1); j++){
        for(i = 0; i < nrows; i++){
            iat_FT[i] += WI[j][firstrow + i];
        }/*for*/
    }/*for*/

    tmp1 = iat_FT[0];
    iat_FT[0] = 0;

    for(i = 1; i < nrows; i++){

        tmp2 = iat_FT[i];
        iat_FT[i] = iat_FT[i - 1] + tmp1;
        tmp1 = tmp2;
    }/*for*/

    nnz = iat_FT[nrows - 1] + tmp1;

    return nnz;

}/*mkiat_FTloc*/

/*
 * Calcolo del vettore iat_FT e aggiornamento della matrice 'WI1'.
 *
 */

void mkiat_FTglo(int myid, int nrows, int nequ, int nproc, int firstrow, int **
    WI1, int *nnz, int *iat_FT){

    int i;
    int j;
```

```
    int ntprec;

    ntprec = 0;

    /* Calcolo del vettore iat_FT. */

    for(i = 0; i < myid; i++){

        ntprec += nnz[i];

    }/*for*/

    for(i = 0; i < nrows; i++){

        iat_FT[i] += ntprec;

    }/*for*/

    if(myid == (nproc - 1)) iat_FT[i] = ntprec + nnz[myid];

    /* Aggiornamento della matrice 'WI1'. */

    for(i = firstrow; i < (firstrow + nrows); i++){
        WI1[0][i] = iat_FT[i - firstrow];
    }/*for*/

    for(j = 1; j < nproc; j++){
        for(i = firstrow; i < (firstrow + nrows); i++){
            WI1[j][i] += WI1[j - 1][i];
        }/*for*/
    }/*for*/

}/*mkiat_FTglo*/

/*
 * Calcolo, in parallelo, dei coefficienti della matrice F^T dalla matrice F.
 *
 */

void mvcoef(int myid, int firstrow, int nrows, int nequ, int nterm_F, int
    nterm_FT, int *iat_F, int *ja_F, int *ja_FT, double *mat_F, double *mat_FT,
    int *punt){

    int i;
    int j;
    int irow;
    int shift;
```

```
    shift = firstrow;

    for(i = 0; i < nrows; i++){
        for(j = iat_F[i]; j < iat_F[i + 1]; j++){

            irow = ja_F[j];
            ja_FT[punt[irow]] = i + shift;
            mat_FT[punt[irow]] = mat_F[j];
            punt[irow]++;

        }/*for*/
    }/*for*/

}/*mvcoef*/

/*
 * Calcolo, in parallelo, del vettore 'pvec_loc'.
 * p = F^T * F * x
 *
 */

void FSAI_precond(int myid, int firstrow, int np, int nrows, int nequ, int
    nzmax_F, int nz_FT, int *procbloc, int *iat_F, int *ja_F, int *iat_FT, int *
    ja_FT, double *mat_F, double *mat_FT, double *vec, double *vscr, double *
    pvec_loc){

    int istart_F;
    int i;

    istart_F = firstrow + myid;
    axbnsy(nrows, nequ, nzmax_F, iat_F + istart_F, ja_F, mat_F, vec, pvec_loc);

    cblas_dcopy(nrows, pvec_loc, 1, vscr + firstrow, 1);
    #pragma omp barrier

    axbnsy(nrows, nequ, nz_FT, iat_FT + firstrow, ja_FT, mat_FT, vscr, pvec_loc);

}/*FSAI_precond*/

void resize(int myid, int np, int firstrow, int nrows_L, int *iat, int *ja,
    double *coef, int *iat_SCR, int *ja_SCR, double *coef_SCR, int *nz_Fi){

    int i;
    int j;
    int jcol;
    int ind1;
```

```
    int ind2;
    int ind3;

    nz_Fi[myid] = 0;
    ind1 = firstrow;
    ind2 = iat[firstrow];
    ind3 = firstrow + myid;
    iat_SCR[ind3] = ind2;

    for(i = 0; i < nrows_L; i++){

        for(j = iat[ind1 + i]; j < iat[ind1 + i + 1]; j++){

            nz_Fi[myid]++;
            ja_SCR[ind2] = ja[j];
            coef_SCR[ind2] = coef[j];
            ind2++;
        }/*for*/

        iat_SCR[ind3 + i + 1] = ind2;
    }/*for*/

}/*resize*/
```



Appendice B

Codice con CUSPARSE e CUBLAS

CUDA_structures.c

In questo file sono create le strutture necessarie al gradiente coniugato modificato per eseguire su GPU.

```
#include <stdio.h>
#include <stdlib.h>

#include <helper_functions.h>
#include <helper_cuda.h>

#include <cuda_runtime_api.h>
#include <cusparse_v2.h>
#include <cublas_v2.h>

#include "./interfaces/CUDA_structures.h"

extern "C"{

        #include "./interfaces/structures.h"

}

int mk_CUDAMAT(int typeOfMatrix, CSRMAT *mat_A, CUDAMAT *CUDAmat_A, CUDADTSTR *
    CUDADTSTR_var){

        int nterm;
```

```
int nequ;
int *iat_A;
int *ja_A;
double *coef_A;
int *d_iat_A;
int *d_ja_A;
double *d_coef_A;
cusparseStatus_t cusparseStatus;

(*CUDAmat_A).typeOfMatrix_A = typeOfMatrix;
nequ = (*mat_A).nrows;
nterm = (*mat_A).nterm;
(*CUDAmat_A).nrows = nequ;
(*CUDAmat_A).nterm = nterm;

iat_A = (*mat_A).iat;
ja_A = (*mat_A).ja;
coef_A = (*mat_A).coef;

checkCudaErrors(cudaMalloc((void **)&d_ja_A, nterm * sizeof(int)));
checkCudaErrors(cudaMalloc((void **)&d_iat_A, (nequ + 1) * sizeof(int)));
checkCudaErrors(cudaMalloc((void **)&d_coef_A, nterm * sizeof(double)));

cudaMemcpy(d_ja_A, ja_A, nterm * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_iat_A, iat_A, (nequ + 1) * sizeof(int),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_coef_A, coef_A, nterm * sizeof(double),
    cudaMemcpyHostToDevice);

(*CUDAmat_A).d_ja_A = d_ja_A;
(*CUDAmat_A).d_iat_A = d_iat_A;
(*CUDAmat_A).d_coef_A = d_coef_A;

(*CUDAmat_A).descr_A = 0;
cusparseStatus = cusparseCreateMatDescr(&((*CUDAmat_A).descr_A));
if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
        printf("CUSPARSE cusparseCreateMatDescr error\n");
        return 1;
}//if

(*CUDAmat_A).hyb_A = 0;
if((*CUDAmat_A).typeOfMatrix_A == HYB_TYPE){

        cusparseStatus = cusparseCreateHybMat(&((*CUDAmat_A).hyb_A));
        if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                printf("CUSPARSE cusparseCreatHybMat error\n");
                return 1;
```

```
                }//if interno

        }//if esterno

        cusparseSetMatType((*CUDAmat_A).descr_A, CUSPARSE_MATRIX_TYPE_GENERAL);
        cusparseSetMatIndexBase((*CUDAmat_A).descr_A, CUSPARSE_INDEX_BASE_ZERO);

        if((*CUDAmat_A).typeOfMatrix_A == HYB_TYPE){

                cusparseStatus = cusparseDcsr2hyb((*CUDADTSTR_var).cusparseHandle
                    , nequ, nequ, (*CUDAmat_A).descr_A, d_coef_A, d_iat_A, d_ja_A
                    , (*CUDAmat_A).hyb_A, 55, CUSPARSE_HYB_PARTITION_AUTO);
                if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                        printf("CUSPARSE␣cusparseDcsr2hyb␣error␣%d␣\n",
                            cusparseStatus);
                        return 1;
                }//if interno

        }//if esterno

        return 0;

}//mk_CUDAMAT

int dlt_CUDAMAT(CUDAMAT *CUDAmat_A){

        cudaFree((*CUDAmat_A).d_ja_A);
        cudaFree((*CUDAmat_A).d_iat_A);
        cudaFree((*CUDAmat_A).d_coef_A);
        if((*CUDAmat_A).typeOfMatrix_A == HYB_TYPE) cusparseDestroyHybMat((*
            CUDAmat_A).hyb_A);
        cusparseDestroyMatDescr((*CUDAmat_A).descr_A);

        return 0;

}//dlt_CUDAMAT

int mk_CUDADTSTR(int devID, CUDADTSTR  *CUDADTSTR_var){

        cublasStatus_t cublasStatus;
        cusparseStatus_t cusparseStatus;

        (*CUDADTSTR_var).devID = devID;

        checkCudaErrors(cudaGetDeviceProperties((&(*CUDADTSTR_var).deviceProp),
            devID));
```

```
        (*CUDADTSTR_var).cublasHandle = 0;
        cublasStatus = cublasCreate(&((*CUDADTSTR_var).cublasHandle));
        if(cublasStatus != CUBLAS_STATUS_SUCCESS){

                printf("CUBLAS␣initialization␣error\n");
                return 1;

        }//if

        (*CUDADTSTR_var).cusparseHandle = 0;
        cusparseStatus = cusparseCreate(&((*CUDADTSTR_var).cusparseHandle));
        if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                printf("CUSPARSE␣initialization␣error\n");
                return 1;
        }//if

        return 0;

}//mk_CUDADTSTR



int dlt_CUDADTSTR(CUDADTSTR *CUDADTSTR_var){

        cusparseDestroy((*CUDADTSTR_var).cusparseHandle);
        cublasDestroy((*CUDADTSTR_var).cublasHandle);

        return 0;

}//dlt_CUDADTSTR

int mk_CUDAPCGSOLVER_FSAI(int nequ, int iout, int itmax, int isol, double tol_CG,
     double *sol, double *rhs, CUDAPCGSOLVER_FSAI *CUDAPCG_var){

        double *d_x;
        double *d_r;
        double *d_pr;
        double *d_p;
        double *d_Ax;
        double *d_dummy;
        double *d_rhs;
        double *d_a;
        double *d_b;
        double *d_a1;
        double a = 1.0;
        double b = 0.0;
```

```
        double a1 = -1.0;

        (*CUDAPCG_var).nequ = nequ;
        (*CUDAPCG_var).iout = iout;
        (*CUDAPCG_var).itmax = itmax;
        (*CUDAPCG_var).isol = isol;
        (*CUDAPCG_var).tol_CG = tol_CG;

        checkCudaErrors(cudaMalloc((void **)&d_x, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_r, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_pr, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_p, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_Ax, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_dummy, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_rhs, nequ * sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_a, sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_b, sizeof(double)));
        checkCudaErrors(cudaMalloc((void **)&d_a1, sizeof(double)));

        cudaMemcpy(d_x, sol, nequ * sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_r, rhs, nequ * sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_rhs, rhs, nequ * sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_a, &a, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_a1, &a1, sizeof(double), cudaMemcpyHostToDevice);


        (*CUDAPCG_var).d_x = d_x;
        (*CUDAPCG_var).d_r = d_r;
        (*CUDAPCG_var).d_pr = d_pr;
        (*CUDAPCG_var).d_p = d_p;
        (*CUDAPCG_var).d_Ax = d_Ax;
        (*CUDAPCG_var).d_dummy = d_dummy;
        (*CUDAPCG_var).d_rhs = d_rhs;
        (*CUDAPCG_var).d_a = d_a;
        (*CUDAPCG_var).d_b = d_b;
        (*CUDAPCG_var).d_a1 = d_a1;

        return 0;

}//mk_CUDAPCGSOLVER_FSAI


int dlt_CUDAPCGSOLVER_FSAI(CUDAPCGSOLVER_FSAI *CUDAPCG_var){

        cudaFree((*CUDAPCG_var).d_x);
        cudaFree((*CUDAPCG_var).d_r);
```

```
        cudaFree((*CUDAPCG_var).d_pr);
        cudaFree((*CUDAPCG_var).d_p);
        cudaFree((*CUDAPCG_var).d_Ax);
        cudaFree((*CUDAPCG_var).d_dummy);
        cudaFree((*CUDAPCG_var).d_rhs);
        cudaFree((*CUDAPCG_var).d_a);
        cudaFree((*CUDAPCG_var).d_b);
        cudaFree((*CUDAPCG_var).d_a1);

        return 0;

}//dlt_CUDAPCGSOLVER_FSAI

int mk_CUDAFSAI(int typeOfMatrix_F, int typeOfMatrix_FT, FSAI *prec, CUDAFSAI *
    CUDAprec, CUDADTSTR *CUDADTSTR_var){

        int nequ;
        int nz_F;
        int *iat_F;
        int *ja_F;
        double *coef_F;
        int *iat_FT;
        int *ja_FT;
        double *coef_FT;
        int *d_iat_F;
        int *d_iat_FT;
        int *d_ja_F;
        int *d_ja_FT;
        double *d_coef_F;
        double *d_coef_FT;
        cusparseStatus_t cusparseStatus;

        (*CUDAprec).typeOfMatrix_F = typeOfMatrix_F;
        (*CUDAprec).typeOfMatrix_FT = typeOfMatrix_FT;

        nequ = (*prec).nequ;
        nz_F = (*prec).nz_F;
        iat_F = (*prec).iat_F;
        ja_F = (*prec).ja_F;
        coef_F = (*prec).coef_F;
        iat_FT = (*prec).iat_FT;
        ja_FT = (*prec).ja_FT;
        coef_FT = (*prec).coef_FT;

        (*CUDAprec).nz_F = nz_F;

        checkCudaErrors(cudaMalloc((void **)&d_coef_F, nz_F * sizeof(double)));
```

```
checkCudaErrors(cudaMalloc((void **)&d_coef_FT, nz_F * sizeof(double)));
checkCudaErrors(cudaMalloc((void **)&d_ja_F, nz_F * sizeof(int)));
checkCudaErrors(cudaMalloc((void **)&d_ja_FT, nz_F * sizeof(int)));
checkCudaErrors(cudaMalloc((void **)&d_iat_F, (nequ + 1) * sizeof(int)));
checkCudaErrors(cudaMalloc((void **)&d_iat_FT, (nequ + 1) * sizeof(int)))
    ;

checkCudaErrors(cudaMemcpy(d_iat_F, iat_F, (nequ + 1) * sizeof(int),
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_ja_F, ja_F, nz_F * sizeof(int),
    cudaMemcpyHostToDevice));
checkCudaErrors(cudaMemcpy(d_coef_F, coef_F, nz_F * sizeof(double),
    cudaMemcpyHostToDevice));

(*CUDAprec).d_iat_F = d_iat_F;
(*CUDAprec).d_ja_F = d_ja_F;
(*CUDAprec).d_coef_F = d_coef_F;
(*CUDAprec).d_iat_FT = d_iat_FT;
(*CUDAprec).d_ja_FT = d_ja_FT;
(*CUDAprec).d_coef_FT = d_coef_FT;

(*CUDAprec).descr_F = 0;
cusparseStatus = cusparseCreateMatDescr(&((*CUDAprec).descr_F));
if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
        printf("CUSPARSE␣cusparseCreateMatDescr␣error\n");
        return 1;
}//if

(*CUDAprec).hyb_F = 0;
if((*CUDAprec).typeOfMatrix_F == HYB_TYPE){

        cusparseStatus = cusparseCreateHybMat(&((*CUDAprec).hyb_F));
        if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                printf("CUSPARSE␣cusparseCreatHybMat␣error\n");
                return 1;

        }//if interno

}//if esterno

cusparseSetMatType((*CUDAprec).descr_F, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase((*CUDAprec).descr_F, CUSPARSE_INDEX_BASE_ZERO);

if((*CUDAprec).typeOfMatrix_F == HYB_TYPE){

        cusparseStatus = cusparseDcsr2hyb((*CUDADTSTR_var).cusparseHandle
            , nequ, nequ, (*CUDAprec).descr_F, d_coef_F, d_iat_F, d_ja_F,
```

```
            (*CUDAprec).hyb_F, 52, CUSPARSE_HYB_PARTITION_AUTO);
        if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                printf("CUSPARSE␣cusparseDcsr2hyb␣error␣%d␣\n",
                    cusparseStatus);
                return 1;
        }//if interno

}//if esterno

(*CUDAprec).descr_FT = 0;
cusparseStatus = cusparseCreateMatDescr(&((*CUDAprec).descr_FT));
if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
        printf("CUSPARSE␣cusparseCreateMatDescr␣error\n");
        return 1;
}//if



cusparseStatus = cusparseDcsr2csc((*CUDADTSTR_var).cusparseHandle, nequ,
    nequ, nz_F, d_coef_F, d_iat_F, d_ja_F, d_coef_FT, d_ja_FT, d_iat_FT,
    CUSPARSE_ACTION_NUMERIC, CUSPARSE_INDEX_BASE_ZERO);

if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){

        printf("CUSPARSE␣cusparseDcsr2csc␣error␣%d\n", cusparseStatus);
        return 1;

}



(*CUDAprec).hyb_FT = 0;
if((*CUDAprec).typeOfMatrix_FT == HYB_TYPE){

        cusparseStatus = cusparseCreateHybMat(&((*CUDAprec).hyb_FT));
        if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                printf("CUSPARSE␣cusparseCreatHybMat␣error\n");
                return 1;

        }//if interno

}//if esterno

cusparseSetMatType((*CUDAprec).descr_FT, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase((*CUDAprec).descr_FT, CUSPARSE_INDEX_BASE_ZERO);

if((*CUDAprec).typeOfMatrix_FT == HYB_TYPE){
```

```
            cusparseStatus = cusparseDcsr2hyb((*CUDADTSTR_var).cusparseHandle
                , nequ, nequ, (*CUDAprec).descr_FT, d_coef_FT, d_iat_FT,
                d_ja_FT, (*CUDAprec).hyb_FT, 50, CUSPARSE_HYB_PARTITION_AUTO)
                ;
            if(cusparseStatus != CUSPARSE_STATUS_SUCCESS){
                    printf("CUSPARSE cusparseDcsr2hyb error %d \n",
                        cusparseStatus);
                    return 1;
            }//if interno

        }//if esterno

        return 0;

}//mk_CUDAFSAI

int dlt_CUDAFSAI(CUDAFSAI *CUDAprec){

        if((*CUDAprec).typeOfMatrix_F == HYB_TYPE) cusparseDestroyHybMat((*
            CUDAprec).hyb_F);
        cusparseDestroyMatDescr((*CUDAprec).descr_F);
        if((*CUDAprec).typeOfMatrix_FT == HYB_TYPE) cusparseDestroyHybMat((*
            CUDAprec).hyb_FT);
        cusparseDestroyMatDescr((*CUDAprec).descr_FT);

        cudaFree((*CUDAprec).d_iat_F);
        cudaFree((*CUDAprec).d_iat_FT);
        cudaFree((*CUDAprec).d_ja_F);
        cudaFree((*CUDAprec).d_ja_FT);
        cudaFree((*CUDAprec).d_coef_F);
        cudaFree((*CUDAprec).d_coef_FT);

        return 0;

}//dlt_CUDAFSAI
```

driver.cu

In questo file è presente la funzione principale main nella quale vengono effettuate le chiamate necessarie per l'esecuzione del gradiente coniugato modificato su GPU.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <omp.h>
#include <math.h>

#include <helper_functions.h>
#include <helper_cuda.h>

#include <cuda_runtime_api.h>
#include <cusparse_v2.h>
#include <cublas_v2.h>

#include "./interfaces/CUDA_structures.h"
#include "./interfaces/PCG_solv_FSAI.h"

#define READ_PATTERN 0
#define COMPUTE_PATTERN 1

extern "C"{

        #include "./interfaces/utility.h"
        #include "./interfaces/structures.h"
        #include "./interfaces/compute_FSAI.h"

}

int main(int argc, char** argv){

        CSRMAT mat_A;
        CUDAMAT CUDAmat_A;
        OMPDTSTR parDTSTR_mat_A;
        CUDADTSTR parCUDADTSTR_var;
        double *rhs;
        PCGSOLVER_FSAI PCG;
        CUDAPCGSOLVER_FSAI CUDAPCG_var;
        FSAI prec;
        CUDAFSAI CUDAprec;
        double *sol;
        int info;
        double t_sol;
        double t_prec;
        double delta;
        double tau;
```

```
int iout = 1;
int isol = 1;
int itmax = 5000;
double tol_CG = 1e-10;
int nequ;
int nterm;
int *iat;
int *ja;
double *coef;
int np;
int nzmax;
int *vecstart;
int devID;
int typeOfMatrix_A = HYB_TYPE;
int typeOfMatrix_F = HYB_TYPE;
int typeOfMatrix_FT = HYB_TYPE;
int flag;
CSRMAT pattern;


devID = findCudaDevice(argc, (const char **) argv);
checkCudaErrors(cudaGetDevice(&devID));
devID = 0;
if(devID < 0){
        printf("exiting...\n");
        exit(0);
}//if


np = atoi(argv[1]);
flag = READ_PATTERN;

info = readmat(&mat_A, "Cubo_4.cmk");

if(info == 0) printf("Matrice␣letta␣correttamente...\n");

nequ = mat_A.nrows;
nterm = mat_A.nterm;
iat = mat_A.iat;
ja = mat_A.ja;
coef = mat_A.coef;

rhs = (double *) malloc(nequ * sizeof(double));
sol = (double *) malloc(nequ * sizeof(double));
uxvsetr(nequ, 1.0, sol);
axbnsy(nequ, nequ, nterm, iat, ja, coef, sol, rhs);
uxvsetr(nequ, 0.0, sol);
```

```
mk_OMPDTSTR(nequ, np, np, &parDTSTR_mat_A);
mk_PCGSOLVER_FSAI(nequ, 100, itmax, isol, 1e-10, &PCG);
set_FSAI(0, 0, nequ, np, RDEF, IDEF, 0.0f, &prec);

if(flag == COMPUTE_PATTERN){
        t_prec = omp_get_wtime();
        info = compute_FSAI(&parDTSTR_mat_A, &mat_A, &prec);
        t_prec = omp_get_wtime() - t_prec;
}//if
if(flag == READ_PATTERN){

        readmat(&pattern, "FSAI_Cubo");
        prec.iat_F = pattern.iat;
        prec.ja_F = pattern.ja;
        prec.coef_F = pattern.coef;
        prec.nz_F = pattern.nterm;

}///if



if(info == 0) printf("Precondizionatore␣calcolato␣correttamente!!!!\n");
else return 1;

mk_CUDADTSTR(devID, &parCUDADTSTR_var);
mk_CUDAMAT(typeOfMatrix_A, &mat_A, &CUDAmat_A, &parCUDADTSTR_var);
mk_CUDAFSAI(typeOfMatrix_F, typeOfMatrix_FT, &prec, &CUDAprec, &
    parCUDADTSTR_var);
mk_CUDAPCGSOLVER_FSAI(nequ, iout, itmax, isol, tol_CG, sol, rhs, &
    CUDAPCG_var);

PCG_solv_FSAI(&CUDAmat_A, &parCUDADTSTR_var, &CUDAprec, &CUDAPCG_var, rhs
    , sol);
printf("resiter␣%g\n", CUDAPCG_var.resiter);
printf("resreal␣%g\n", CUDAPCG_var.resreal);
printf("n_iter␣%d\n", CUDAPCG_var.n_iter);

free(rhs);
free(sol);
dlt_CSRMAT(&mat_A);
dlt_CUDAMAT(&CUDAmat_A);
dlt_OMPDTSTR(&parDTSTR_mat_A);
dlt_CUDADTSTR(&parCUDADTSTR_var);
dlt_PCGSOLVER_FSAI(&PCG);
dlt_CUDAPCGSOLVER_FSAI(&CUDAPCG_var);
dlt_CUDAFSAI(&CUDAprec);
```

```
        cudaDeviceReset();

        return 0;

}//main
```

PCG_solv_FSAI.cu

In questo file è implementato il gradiente coniugato modificato su GPU. In particolare è composto da un'unica funzione PCG_solv_FSAI che accetta in input il sistema lineare e fornisce la soluzione cercata.

```
#include "./interfaces/PCG_solv_FSAI.h"

/*
 * Risolve il sistema con il PCG
 *
 */

int checkCublasStatus ( cublasStatus_t status, const char *msg )
{
    if ( status != CUBLAS_STATUS_SUCCESS ) {
        fprintf (stderr, "!!!! CUBLAS %s ERROR \n", msg);
        return 1;
    }
    return 0;
}


int PCG_solv_FSAI(CUDAMAT *CUDAmat_A, CUDADTSTR *parCUDADTSTR_var, CUDAFSAI *
    CUDAprec, CUDAPCGSOLVER_FSAI *CUDAPCG_var, double *rhs, double *sol){

        int nequ;
        int nterm;
        int nz_F;
        int itmax;
        int *d_iat_A;
        int *d_ja_A;
        double *d_coef_A;
```

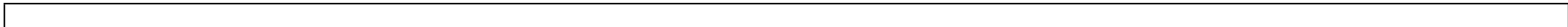
```
int *d_iat_F;
int *d_ja_F;
double *d_coef_F;
int *d_iat_FT;
int *d_ja_FT;
double *d_coef_FT;
double *d_x;
double *d_r;
double *d_pr;
double *d_p;
double *d_Ax;
double *d_dummy;
double *d_rhs;
double *d_a;
double *d_b;
double *d_a1;
double a = 1.0;
double b = 0.0;
double a1 = -1.0;
double alpha;
double nalpha;
double beta;
double bnorm;
double resini;
double resiter;
double resreal;
double ptap;
double tol_CG;
int typeOfMatrix_A;
int typeOfMatrix_F;
int typeOfMatrix_FT;
int i;

//CUBLAS variable
cublasStatus_t cublasStatus;
cublasHandle_t cublasHandle;

//CUSPARSE variable
cusparseStatus_t cusparseStatus;
cusparseHandle_t cusparseHandle;
cusparseMatDescr_t descr_A;
cusparseHybMat_t hyb_A;
cusparseMatDescr_t descr_F;
cusparseHybMat_t hyb_F;
cusparseMatDescr_t descr_FT;
cusparseHybMat_t hyb_FT;
```

```
cublasHandle = (*parCUDADTSTR_var).cublasHandle;
cusparseHandle = (*parCUDADTSTR_var).cusparseHandle;

typeOfMatrix_A = (*CUDAmat_A).typeOfMatrix_A;
nequ = (*CUDAmat_A).nrows;
nterm = (*CUDAmat_A).nterm;
descr_A = (*CUDAmat_A).descr_A;
hyb_A = (*CUDAmat_A).hyb_A;
d_iat_A = (*CUDAmat_A).d_iat_A;
d_ja_A = (*CUDAmat_A).d_ja_A;
d_coef_A = (*CUDAmat_A).d_coef_A;

typeOfMatrix_F = (*CUDAprec).typeOfMatrix_F;
nz_F = (*CUDAprec).nz_F;
descr_F = (*CUDAprec).descr_F;
hyb_F = (*CUDAprec).hyb_F;
d_iat_F = (*CUDAprec).d_iat_F;
d_ja_F = (*CUDAprec).d_ja_F;
d_coef_F = (*CUDAprec).d_coef_F;
typeOfMatrix_FT = (*CUDAprec).typeOfMatrix_FT;
descr_FT = (*CUDAprec).descr_FT;
hyb_FT = (*CUDAprec).hyb_FT;
d_iat_FT = (*CUDAprec).d_iat_FT;
d_ja_FT = (*CUDAprec).d_ja_FT;
d_coef_FT = (*CUDAprec).d_coef_FT;

tol_CG = (*CUDAPCG_var).tol_CG;
itmax = (*CUDAPCG_var).itmax;
d_x = (*CUDAPCG_var).d_x;
d_pr = (*CUDAPCG_var).d_pr;
d_r = (*CUDAPCG_var).d_r;
d_p = (*CUDAPCG_var).d_p;
d_Ax = (*CUDAPCG_var).d_Ax;
d_dummy = (*CUDAPCG_var).d_dummy;
d_rhs = (*CUDAPCG_var).d_rhs;
d_a = (*CUDAPCG_var).d_a;
d_b = (*CUDAPCG_var).d_b;
d_a1 = (*CUDAPCG_var).d_a1;

cublasStatus = cublasDnrm2(cublasHandle, nequ, d_rhs, 1, &bnorm);


if((*CUDAPCG_var).isol != 0){

        //F * x = t
        if(typeOfMatrix_F == HYB_TYPE) cusparseStatus = cusparseDhybmv(
            cusparseHandle, CUSPARSE_OPERATION_NON_TRANSPOSE, &a, descr_F
```

```
            , hyb_F, d_rhs, &b, d_dummy);
        else cusparseStatus = cusparseDcsrmv(cusparseHandle,
            CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nz_F, &a,
            descr_F, d_coef_F, d_iat_F, d_ja_F, d_rhs, &b, d_dummy);

        if(typeOfMatrix_FT == HYB_TYPE) cusparseStatus = cusparseDhybmv(
            cusparseHandle, CUSPARSE_OPERATION_NON_TRANSPOSE, &a,
            descr_FT, hyb_FT, d_dummy, &b, d_x);
        else cusparseStatus = cusparseDcsrmv(cusparseHandle,
            CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nz_F, &a,
            descr_FT, d_coef_FT, d_iat_FT, d_ja_FT, d_dummy, &b, d_x);

  }


           //A * x = Ax
if(typeOfMatrix_A == HYB_TYPE) cusparseStatus = cusparseDhybmv(
    cusparseHandle, CUSPARSE_OPERATION_NON_TRANSPOSE, &a, descr_A, hyb_A,
    d_x, &b, d_Ax);
  else cusparseStatus = cusparseDcsrmv(cusparseHandle,
      CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nterm, &a, descr_A,
      d_coef_A, d_iat_A, d_ja_A, d_x, &b, d_Ax);

          //Ax + r = r
  cublasDaxpy(cublasHandle, nequ, &a1, d_Ax, 1, d_r, 1);
  cublasDnrm2(cublasHandle, nequ, d_r, 1, &resini);
  resini = resini / bnorm;
  resiter = resini;

  i = 0;

  while((i < itmax) && (resiter > tol_CG)){

                            // F * r = t
                      if(typeOfMatrix_F == HYB_TYPE) cusparseStatus =
                           cusparseDhybmv(cusparseHandle,
                          CUSPARSE_OPERATION_NON_TRANSPOSE, &a,
                          descr_F, hyb_F, d_r, &b, d_dummy);
                else cusparseStatus = cusparseDcsrmv(cusparseHandle,
                    CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nz_F, &
                    a, descr_F, d_coef_F, d_iat_F, d_ja_F, d_r, &b,
                    d_dummy);

                           // F^T * t = pr
                if(typeOfMatrix_FT == HYB_TYPE) cusparseStatus =
                    cusparseDhybmv(cusparseHandle,
                    CUSPARSE_OPERATION_NON_TRANSPOSE, &a, descr_FT,
```

```
        hyb_FT, d_dummy, &b, d_pr);
      else cusparseStatus = cusparseDcsrmv(cusparseHandle,
          CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nz_F,
           &a, descr_FT, d_coef_FT, d_iat_FT, d_ja_FT,
          d_dummy, &b, d_pr);

    if(i == 0){

                            //p = pr
    cublasStatus = cublasDcopy(cublasHandle, nequ, d_pr, 1,
        d_p, 1);

    }else{

                            //pr * Ax = beta
    cublasStatus = cublasDdot(cublasHandle, nequ, d_pr, 1,
        d_Ax, 1, &beta);
    beta = beta / (-ptap);

                            //p = beta * p
    cublasStatus = cublasDscal(cublasHandle, nequ, &beta, d_p
        , 1);

                            //p = p + r
    cublasStatus = cublasDaxpy(cublasHandle, nequ, &a, d_pr,
        1, d_p, 1);


}//else


                        // Ax = A * p
if(typeOfMatrix_A == HYB_TYPE) cusparseStatus = cusparseDhybmv(
    cusparseHandle, CUSPARSE_OPERATION_NON_TRANSPOSE, &a, descr_A
    , hyb_A, d_p, &b, d_Ax);
else cusparseStatus = cusparseDcsrmv(cusparseHandle,
    CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nterm, &a,
    descr_A, d_coef_A, d_iat_A, d_ja_A, d_p, &b, d_Ax);


                // p * Ax = ptap
cublasStatus = cublasDdot(cublasHandle, nequ, d_p, 1, d_Ax, 1, &
    ptap);

// r * p = alpha
cublasStatus = cublasDdot(cublasHandle, nequ, d_r, 1, d_p, 1, &
    alpha);
```

```
            alpha = alpha / ptap;
            cublasStatus = cublasDaxpy(cublasHandle, nequ, &alpha, d_p, 1,
                d_x, 1);
            nalpha = -alpha;
            cublasStatus = cublasDaxpy(cublasHandle, nequ, &nalpha, d_Ax, 1,
                d_r, 1);

            cublasStatus = cublasDnrm2(cublasHandle, nequ, d_r, 1, &resiter);
            resiter = resiter / bnorm;
                            cudaDeviceSynchronize();

            i++;

    }//while

  if(typeOfMatrix_A == HYB_TYPE) cusparseStatus = cusparseDhybmv(
      cusparseHandle, CUSPARSE_OPERATION_NON_TRANSPOSE, &a, descr_A, hyb_A,
      d_x, &b, d_Ax);
    else cusparseStatus = cusparseDcsrmv(cusparseHandle,
        CUSPARSE_OPERATION_NON_TRANSPOSE, nequ, nequ, nterm, &a, descr_A,
        d_coef_A, d_iat_A, d_ja_A, d_x, &b, d_Ax);

            cublasDaxpy(cublasHandle, nequ, &a1, d_rhs, 1, d_Ax, 1);
    cublasDnrm2(cublasHandle, nequ, d_Ax, 1, &resreal);
    resreal = resreal / bnorm;

            (*CUDAPCG_var).resini = resini;
    (*CUDAPCG_var).resiter = resiter;
    (*CUDAPCG_var).n_iter = i;
    (*CUDAPCG_var).resreal = resreal;
    (*CUDAPCG_var).bnorm = bnorm;

            return 0;

}//PCG_solv_FSAI
```



Appendice C

Codice del calcolo del precondizionatore FSAI

CUDA_cpt_F_FSAI

In questo file è impementato l'algoritmo euristico utilizzato per ridurre il numero di sistemi da risolvere.

```
#include "./interfaces/CUDA_utility.h"

int CUDA_cpt_F_FSAI(int nequ, int nterm, int nz_F, double tau, int *iat, int *ja,
     int *iat_F, int *ja_F, double *mat_A, double *mat_F, double *
    d_block_of_system){

        int mmax;
        int i;
        int j;
        int k;
        int l;
        int irow;
        QueueOfLinkedList queue;
        NodeList *ptrQueue;
        NodeList *bestNodeList;
        Node *ptr;
        double *matrixBlock;
        int dimRow;
        int mismatch;
```

```
int numOfCompare;
LinkedList *list;
QueueOfLinkedList *queueList;
int dimList;
int maxSize;
int maxRhs;
double coef1;
double coef2;
int value;
double cost;
double maxRed;
double nativeCost;
double mergeCost;
double costRed;
int tot_rows;
int tot_nnz;
double tot_cost;
double tot_densita;
int numRhs;
double discr;
double constant;
int numeroMedioElem;
int maxValue = 200;
int window;
double *rhs;
int *iatOfMatrix;
int *iatOfRhs;
double *d_rhs_of_system;
double *d_sol_of_system;
int *d_iat_matrix;
int *d_iat_rhs;
int blockSize;
int sizeQueue;
int sizeOfListQueue;
int mm;
int nn;
dim3 dimBlock;
dim3 dimGrid;
int sharedMemorySize;
int sizeRhs;
int numberOfSystem;

float system_time;
float elapsed_time;
float merge_time;
float gather_time;
cudaEvent_t startCuda;
```

```
cudaEvent_t endCuda;

checkCudaErrors(cudaEventCreate(&startCuda));
checkCudaErrors(cudaEventCreate(&endCuda));
matrixBlock = (double *) malloc(MAX_NUMBER_SYSTEM * MAX_SYSTEM_SIZE *
    MAX_SYSTEM_SIZE * sizeof(double));
if(matrixBlock == NULL) return -1;
iatOfMatrix = (int *) malloc((MAX_NUMBER_SYSTEM + 1) * sizeof(int));
if(iatOfMatrix == NULL) return -1;
iatOfRhs = (int *) malloc((MAX_NUMBER_SYSTEM + 1) * sizeof(int));
if(iatOfRhs == NULL) return -1;
rhs = (double *) malloc(MAX_SYSTEM_SIZE * MAX_NUMBER_SYSTEM * maxRhs *
    sizeof(double));
if(rhs == NULL) return -1;
system_time = 0.0f;

constant = 1.01;
numeroMedioElem = 39;
window = 10;

mmax = 0;

for(i = 0; i < nequ; i++){

        mmax = MAX(mmax, iat_F[i + 1] - iat_F[i]);

}//for

//Calcolo dei coefficienti
discr = sqrt(3.0 * constant * (4.0 - constant));
printf("discr␣%g\n", discr);
coef1 = (double) numeroMedioElem * ((constant + 2.0) - discr ) / ( 2.0 *
    (constant - 1.0));
coef2 = (double) numeroMedioElem * ((constant + 2.0) + discr ) / ( 2.0 *
    (constant - 1.0));
if (constant > 1.0){
        if(coef2 < (double) maxValue) maxValue = coef2;
        printf("I␣due␣coefficienti␣sono␣pari␣a␣%g␣e␣%g\n", coef1, coef2);
}//if

if (constant <= 1.0){
        coef2 = (double) nterm;
}//if

createQueueListOfLinkedList(&queue, window);
maxSize = 0;
maxRhs = 0;
```

```
for(i = 0; i < nequ; i++){

        irow = iat_F[i];
        dimRow = (iat_F[i + 1] - irow);

        j = 0;
        numOfCompare = MIN(queue.window, queue.size);
        ptrQueue = queue.last;
        mismatch = mmax;
        dimList = 0;
        maxRed = DBL_MIN;

        while((j < numOfCompare) && (ptrQueue != queue.first)){

                list = ptrQueue->list;
                if(list->size < maxValue){

                        value = countNumberOfMismatch(list, &ja_F[irow],
                            dimRow);
                        nativeCost = pow((double)list->size, 3) + pow((
                            double)dimRow, 3);
                        mergeCost  = pow((double)(list->size + value), 3)
                            ;
                        costRed = nativeCost - mergeCost;

                        if(costRed > maxRed){
                                dimList = list->size;
                                mismatch = value;
                                bestNodeList = ptrQueue;
                                maxRed = costRed;

                        }//if interno
                        j++;
                }//if esterno

                ptrQueue = ptrQueue->prev;
        }//while

        if(pow((double)(dimList + mismatch), 3) >(constant * (pow((double
            )dimList, 3) + pow((double)dimRow, 3)))){

                //Non faccio il merge

                list = (LinkedList *) malloc(sizeof(LinkedList));

                if(list == NULL){
```

```
				printf("LinkedList␣creation␣failed!!!!");
				return -1;

			}//if

			createLinkedList(list);

			for(j = 0; j < dimRow; j++) addFromBack(ja_F[irow + j],
			    list);

			enqueue(list, &queue);
			addFromBack(i, queue.last->listOfRow);
			queue.last->numberOfRow = 1;
			maxSize = MAX(maxSize, queue.last->list->size);
			maxRhs = MAX(maxRhs, queue.last->numberOfRow);


		}else{

			//Faccio il merge

			mergeList(bestNodeList->list, ja_F + irow, dimRow);
			addFromBack(i, bestNodeList->listOfRow);
			bestNodeList->numberOfRow++;
			maxSize = MAX(maxSize, bestNodeList->list->size);
			maxRhs = MAX(maxRhs, bestNodeList->numberOfRow);

		}//else

	}//for
	queueList = (QueueOfLinkedList *) malloc(sizeof(QueueOfLinkedList) *
	    maxSize);
	for(i = 0; i < maxSize; i++) createQueueListOfLinkedList(&queueList[i],
	    queue.window);
	ptrQueue = queue.first->next;
	k = 0;
	i = 0;

	while(i < queue.size){

		k = ptrQueue->list->size;

		if(queueList[k - 1].size == 0){

			queueList[k - 1].first->next = ptrQueue;
			queueList[k - 1].last = ptrQueue;
```

```
                queueList[k - 1].last->prev = queueList[k - 1].first;
                ptrQueue = ptrQueue->next;
                queueList[k - 1].last->next = NULL;
                queueList[k - 1].size++;

        }
        else{

                queueList[k - 1].last->next = ptrQueue;
                ptrQueue->prev = queueList[k - 1].last;
                queueList[k - 1].last = ptrQueue;
                ptrQueue = ptrQueue->next;
                queueList[k - 1].last->next = NULL;
                queueList[k - 1].size++;

        }//else
        i++;

}//while

checkCudaErrors(cudaMalloc((void **)&d_rhs_of_system, MAX_SYSTEM_SIZE *
    MAX_NUMBER_SYSTEM * maxRhs * sizeof(double)));
checkCudaErrors(cudaMalloc((void **)&d_sol_of_system, MAX_SYSTEM_SIZE *
    MAX_NUMBER_SYSTEM * maxRhs * sizeof(double)));
sizeQueue = queueList[0].size;
sizeOfListQueue = queue.size;
k = 0;
i = 0;

//Ciclo che scorre tutta la coda

while(i < sizeOfListQueue){

        blockSize = 0;
        iatOfMatrix[blockSize] = 0;
        iatOfRhs[blockSize] = 0;
        numberOfSystem = MAX_NUMBER_SYSTEM;
        if(sizeOfListQueue - i < MAX_NUMBER_SYSTEM) numberOfSystem =
            sizeOfListQueue - i;

        //Ciclo per raccogliere MAX_NUMBER_SYSTEM sistemini


        while(blockSize < numberOfSystem){

                if(sizeQueue == 0){
```

```
                    k++;
                    while((k < maxSize) && queueList[k].size == 0) k
                        ++;
                    if(k >= maxSize) break;
                    ptrQueue = queueList[k].first->next;
                    sizeQueue = queueList[k].size;


                }//if

                gather_fullsys_FSAI_with_LinkedList(k + 1, k + 1, iat, ja
                    , ptrQueue->list, mat_A, matrixBlock + iatOfMatrix[
                    blockSize]);

                }//for*/

                iatOfRhs[blockSize] = iatOfRhs[blockSize - 1] + (k + 1);
                iatOfMatrix[blockSize] = iatOfMatrix[blockSize - 1] + (k
                    + 1) * (k + 1);
                ptrQueue = ptrQueue->next;

            }//while

            //Allocazione dei sistemini appena raccolti nella memoria della
                GPU

            //Contiene i sistemini
            cudaMemcpy(d_block_of_system, matrixBlock, iatOfMatrix[blockSize
                - 1]  * sizeof(double), cudaMemcpyHostToDevice);
            //Contiene i termini noti
            cudaMemcpy(d_rhs_of_system, rhs, iatOfRhs[blockSize - 1] * sizeof
                (double), cudaMemcpyHostToDevice);

            i += blockSize;
        }//while

        free(matrixBlock);
        free(rhs);
        for(i = 0; i < maxSize; i++) deleteQueueOfLinkedList(&(queueList[i]));
        free(queueList);

        return 0;

}//CUDA_cpt_F_FSAI
```

solve.cu

In questo file è implementata la fase del calcolo del precondizionatore FSAI che prevede la risoluzione di una sequenza di sistemi. Il metodo adottato è il metodo dell'eliminazione di Gauss.

```
void gauss_solve_gpu2 (double *A, double *b, double *x, int n, int batch)
{
    double *As = (double*)shmem;
    double *Val = (double*)(As + (n+pad) * (n+1));
    int *Loc = (int*)(Val + pivot_thrds);
    const int ofs = pad;
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int blkNum = blockIdx.y * gridDim.x + blockIdx.x;

    if (blkNum >= batch) return;

    A += blkNum * n * n;
    b += blkNum * n;
    x += blkNum * n;

    for (int i = tx; i < n; i += blockDim.x) {
        if (ty  < n) As(i,ty) = A[ty * n + i];
        if (ty == n) {
            As(i,ty) = b[i];
        }
    }

    int j = 0;
    do {
        __syncthreads();
        if ((tx == 0) && (ty > j)) {
            As(j,ty) = mulOp (As(j,ty), rcpOp (As(j,j)));
        }

        __syncthreads();
        for (int i = j+1+tx; i < n; i += blockDim.x) {
            if (ty > j) {
                As(i,ty) = fmnaOp (As(i,j), As(j,ty), As(i,ty));
            }
        }

        j++;
```

```
    } while (j < n);

    j = n-1;
    do {
        __syncthreads();
        if ((tx == 0) && (ty < j)) {
            As(ty,n) = fmnaOp (As(ty,j), As(j,n), As(ty,n));
        }
        j--;
    } while (j);

    __syncthreads();
    if ((tx == 0) && (ty < n)) x[ty] = As(ty,n);
}

int fast_solve (double *A_d, double *b_d, double *x_d, int n, int batch){

        dim3 dimBlock(dimX[n], n+1);
    dim3 dimGrid;
    if (batch <= GRID_DIM_LIMIT) {
        dimGrid.x = batch;
        dimGrid.y = 1;
        dimGrid.z = 1;
    } else {
        dimGrid.x = GRID_DIM_LIMIT;
        dimGrid.y = (batch + GRID_DIM_LIMIT-1) / GRID_DIM_LIMIT;
        dimGrid.z = 1;
    }

        int smem_size = (sizeof(A_d[0]) * (n + padding[n]) * (n+1));

    gauss_solve_gpu2<<<dimGrid,dimBlock,smem_size>>>(A_d,b_d,x_d,n,batch);
    cudaError_t err = cudaGetLastError();

    if (cudaSuccess != err) {
        return -2;
    }
    return 0;

}
```