UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

A.A. 2010/2011

Tesi di Laurea

Risoluzione di PDE tramite metodi MultiGrid in ambiente CUDA

RELATORE

Ing. Daniele Carnevale

CANDIDATO

Claudio Pupparo

CORRELATORE

Prof. Sergio Galeani

Alla mia famiglia,

ai miei amici

Indice

Introduzione

PDE é un acronimo stante per **P**artial **D**ifferential **E**quations, ossia Equazioni Differenziali alle Derivate Parziali. Tali equazioni, come il nome suggerisce, descrivono una relazione fra una o più funzioni in più variabili e le loro derivate parziali rispetto alle stesse variabili. Le PDE vengono utilizzate per descrivere matematicamente sistemi fisici, elettrodinamici, meccanici, biologi, finanziari e nell'ambito di computer graphics.

Per tale tipologia di equazioni non sempre é possibile trovare una soluzione esplicita; sono stati quindi sviluppati diversi metodi che forniscono il valore della soluzione in determinati punti del dominio su cui si stà effettuando il calcolo; tali metodi vengono definiti **Metodi Numerici**. I metodi numerici furono sviluppati ben prima della realizzazione dei calcolatori; fra gli algoritmi utilizzati in tale contesto figurano importanti nomi di studiosi del XVIII e XIX secolo: Newton, Lagrange, Gauss, Eulero. Il lavoro qui presentato trova il fulcro nei **Metodi MultiGrid**, miranti alla risoluzione numerica di PDE. Gli algoritmi utilizzati allo scopo risultano essere molto complessi, richiedendo grandi quantità di calcoli; diventa così importante il fattore tempo; in diversi domini applicativi non basta trovare la soluzione ad un problema, ma vi é la necessità di trovarlo in tempi utili, per ovvie conseguenze. Per soddisfare tale requisito prestazionale, sono state utilizzate le librerie CUDA. CUDA é l'acronimo di **C**ompute **U**nified **D**evice **A**rchitecture, ed identifica un paradigma di architettura parallela, in

cui il centro di calcolo é spostato dalla Cpu alla Gpu, il processore grafico, per sua natura strutturato in maniera fortemente parallelizzata.
L'utilizzo congiunto di metodi MultiGrid e librerie Cuda é stato qui sperimentato su equazioni importanti nella Teoria del Controllo; sono state quindi poste le basi per la risoluzione di equazioni più avanzate, quale ad esempio l'equazione di **Grad-Shafranov** (magnetofluidodinamica - dinamica dei fluidi elettricamente conduttori), regolante il movimento del plasma all'interno di una forma toroidale (come il Tokamak, utilizzato nella fusione termonucleare).
Il presente lavoro é così strutturato:

- Il capitolo 1 fornisce una visione d'insieme sui metodi numerici per la risoluzione di PDE, tramite la descrizione di algoritmi mano a mano più performanti, arrivando infine ai Metodi Multigrid.

- Nel capitolo 2 é fornita una descrizione dell'ambiente utilizzato per implementare i metodi risolutivi, le librerie CUDA

- Nel capitolo 3 vengono trattati i problemi risolti tramite l'utilizzo congiunto dei Metodi MultiGrid e dell'ambiente CUDA.

- Il capitolo 4 descrive uno studio prestazionale degli algoritmi risolutori, concentrandosi in particolare sul confronto fra Cpu e Gpu.

Capitolo 1

Metodi per la risoluzione di PDE

1.1 Fondamenti

In questo capitolo verranno descritti i metodi **MultiGrid** per la risoluzione di PDE nel contesto dei "" *Problemi di Valori al Contorno*"" (**Boundary Value Problem**), detti anche problemi di Cauchy; verranno affrontati principalmente problemi lineari, mentre al caso non lineare sarà dedicato il paragrafo conclusivo. Un problema al contorno é costituito da una equazione differenziale, e da diverse condizioni che specificano il valore della soluzione (**condizioni di Dirichlet**), o delle relative derivate (**condizioni di Neumann**), nei punti sul contorno del dominio in cui la soluzione viene cercata. Un esempio di quanto appena detto é il seguente:

$$-u(x)'' + \sigma u(x) = f(x), \qquad 0 < x < 1, \quad \sigma > 0 \tag{1.1.1}$$

$$u(0) = u(1) = 0. \tag{1.1.2}$$

Tale problema descrive la distribuzione della temperatura allo stato stazionario (ossia non dipendente dal tempo) in una barra metallica, in cui il raggio é trascurabile rispetto alla lunghezza per rendere il problema monodimensionale.

Come primo passo verso la risoluzione, deve essere effettuata la **discretizzazione**

dell'equazione; l'intervallo in cui si deve effettuare il calcolo viene diviso in una **griglia** di punti, definita Ω^h, in cui ogni punto é distanziato dal vicino di una quantità pari a $h = \frac{x_N - x_0}{N} = \frac{1}{N}$, dove N é il numero di punti escluso x_0. La figura Figura 1.1 riporta una griglia su un dominio monodimensionale, in cui per ogni punto x_i la soluzione approssimata vale v_i. É bene sottolineare che grazie alle condizioni di contorno, il

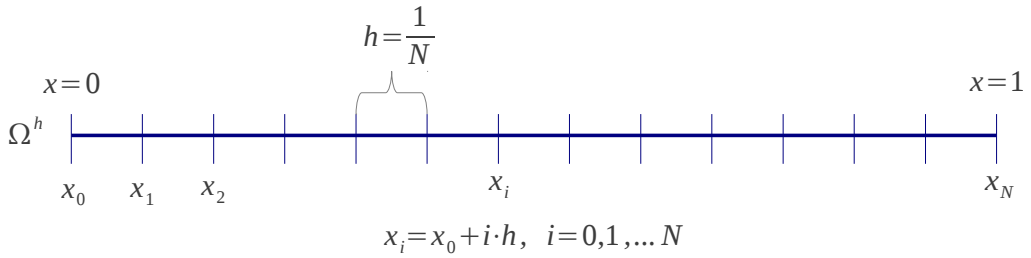

Figura 1.1: Griglia 1D.

valore della soluzione lungo gli estremi dell'intervallo, ossia x_0 e x_N, é già noto.

Prima di poter procedere nella risoluzione dell'equazione é necessario trovare una rappresentazione discreta delle derivate (di qualsiasi ordine) della funzione; allo scopo si utilizza **l'espansione di Taylor**.

Utilizzando la definizione di rapporto incrementale, il calcolo della derivata prima avviene nel seguente modo:

$$v_{i+1} = v_i + v'h + o(h^2)$$

Da cui segue

$$v' = \frac{v_{i+1} - v_i}{h} + o(h) \tag{1.1.3}$$

Il calcolo della derivata seconda richiede l'utilizzo congiunto delle espansioni in avanti e all'indietro di Taylor, troncate al terzo ordine:

$$v_{i+1} = v_i + v'h + v''\frac{h^2}{2!} + v'''\frac{h^3}{3!} + o(h^4) \quad \text{(espansione in avanti)}$$
$$v_{i-1} = v_i - v'h + v''\frac{h^2}{2!} - v'''\frac{h^3}{3!} + o(h^4) \quad \text{(espansione all'indietro)}$$

Sommando e risolvendo rispetto a v'':

$$v'' = \frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} + o(h^2) \tag{1.1.4}$$

É quindi possibile ora riproporre il problema 1.1.1 in forma discretizzata:

$$-\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} + \sigma v_i = f_i \tag{1.1.5}$$

con condizioni di contorno:

$$v_0 = v_N = 0 \tag{1.1.6}$$

Lo schema appena descritto viene definito ""**Metodo delle differenze finite**"" ed é utilizzato da ogni metodo risolutivo qui trattato. Il problema 1.1.5 può essere riscritto sotto forma matriciale $\boldsymbol{Av} = \boldsymbol{f}$, dove $\boldsymbol{A}$, $\boldsymbol{v}$ e $\boldsymbol{f}$ sono matrici cosí composte:

$$\boldsymbol{A} = \frac{1}{h^2}\begin{vmatrix} 2+\sigma h^2 & -1 & & & & & \\ -1 & 2+\sigma h^2 & -1 & & & & \\ & -1 & 2+\sigma h^2 & -1 & & & \\ & & & ... & & & \\ & & & & -1 & 2+\sigma h^2 & -1 \\ & & & & & -1 & 2+\sigma h^2 \end{vmatrix} \tag{1.1.7}$$

$$\boldsymbol{v} = \begin{vmatrix} v_1 \\ v_2 \\ v_3 \\ ... \\ v_{N-2} \\ v_{N-1} \end{vmatrix} \tag{1.1.8}$$

$$\boldsymbol{f} = \begin{vmatrix} f_1 \\ f_2 \\ f_3 \\ ... \\ f_{N-2} \\ f_{N-1} \end{vmatrix} \tag{1.1.9}$$

La possibilità di rappresentare sotto forma matriciale l'equazione da risolvere fornisce un sistema di equazioni che sarà alla base dei metodi risolutivi successivamente descritti.

Per fissare bene le idee verrà ora proposto un problema a due dimensioni

$$-u_{xx} - u_{yy} + \sigma u(x,y) = f(x,y), \qquad 0 < x < 1, 0 < y < 1 \quad \sigma > 0 \tag{1.1.10}$$

$$u = 0, \quad x = 0, x = 1, y = 0, y = 1 \tag{1.1.11}$$

dove con u_{xx} si intende la derivata parziale seconda rispetto ad x. In figura Figura 1.2 viene mostrata la griglia di punti creata per effettuare la discretizzazione dell'equazione 1.1.10. In questo caso vengono utilizzati due intervalli h_x, h_y il cui valore dipende

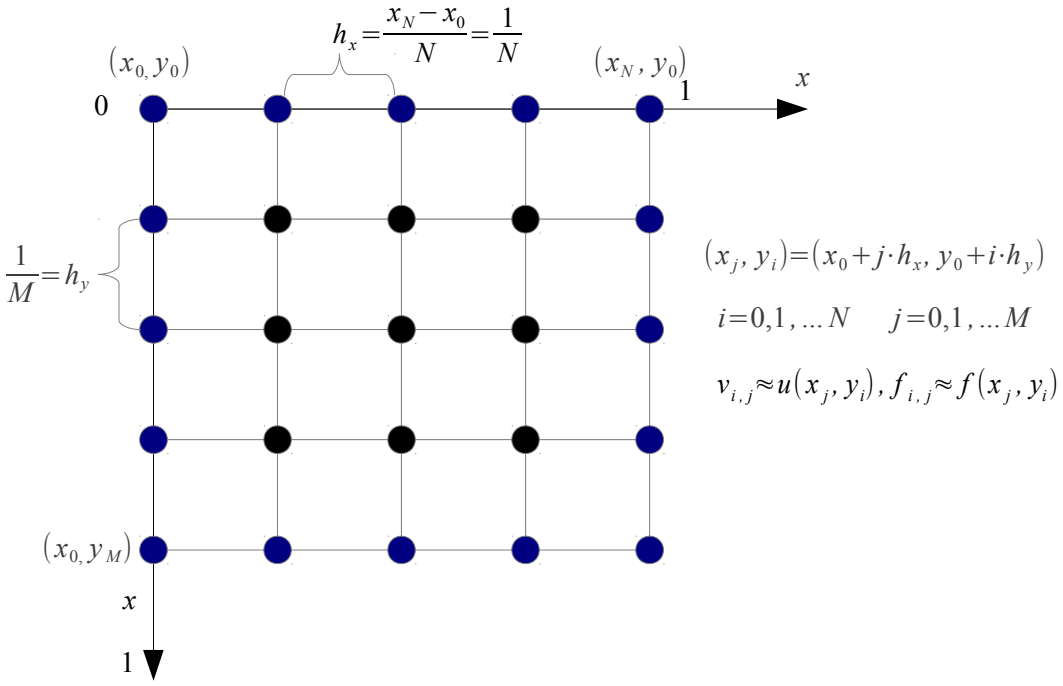

Figura 1.2: Griglia 2D.

dal numero di punti lungo gli assi x e y.

Dopo aver effettuato il passo di discretizzazione si ottiene il seguente risultato:

$$-\frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{h_x^2} - \frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{h_y^2} + \sigma v_{i,j} = f_{i,j} \tag{1.1.12}$$

dove l'indice i é relativo all'asse y, l'indice j all'asse x. Anche il problema a due dimensioni 1.1.10 può essere riscritto come un sistema di equazioni e sotto la forma matriciale $\boldsymbol{Av} = \boldsymbol{f}$, dove in questo caso le espressioni di $\boldsymbol{A}$, $\boldsymbol{v}$ e $\boldsymbol{f}$ sono più complesse rispetto al caso ad una dimensione.

Nella sezione successiva verranno introdotti i primi metodi per poter risolvere PDE chiamati **Metodi Iterativi**.

1.2 Metodi Iterativi

1.2.1 Introduzione

Si consideri un sistema lineare

$$\boldsymbol{Au} = \boldsymbol{f} \tag{1.2.1}$$

e sia $\boldsymbol{v}$ un'approssimazione di $\boldsymbol{u}$; vengono qui introdotte due importanti definizioni:

Errore:

$$\boldsymbol{e} = \boldsymbol{u} - \boldsymbol{v} \tag{1.2.2}$$

con norme:

$$\|\boldsymbol{e}\|_\infty = max|e_i|, \quad \|\boldsymbol{e}\|_2 = \sqrt{\sum_{i=1}^{N} e_i^2}$$

Residuo:

$$\boldsymbol{r} = \boldsymbol{f} - \boldsymbol{Av} \tag{1.2.3}$$

con norme:

$$\|\boldsymbol{r}\|_\infty = max|r_i|, \quad \|\boldsymbol{r}\|_2 = \sqrt{\sum_{i=1}^{N} r_i^2}$$

Entrambe ci danno informazioni su quanto la soluzione approssimata $\boldsymbol{v}$ disti dalla soluzione reale $\boldsymbol{u}$.

Utilizzando le 1.2.2 e 1.2.3 é quindi possibile riscrivere nel seguente modo la 1.2.1

$$\boldsymbol{Au} = \boldsymbol{f}$$

$$\boldsymbol{A}(\boldsymbol{v} + \boldsymbol{e}) = \boldsymbol{f}$$

$$\boldsymbol{Av} + \boldsymbol{Ae} = \boldsymbol{f}$$

$$\boldsymbol{Ae} = \boldsymbol{f} - \boldsymbol{Av}$$

Da cui la fondamentale **equazione del residuo**:

$$\boldsymbol{Ae} = \boldsymbol{r} \tag{1.2.4}$$

dove $\boldsymbol{r}$ é dato dalla 1.2.3. É qui importante sottolineare che tale relazione é valida solo in quanto il problema da risolvere é **lineare**; in caso contrario devono esser fatte considerazioni diverse, come sarà descritto nel paragrafo finale del presente capitolo. La 1.2.4 é di fondamentale importanza in quanto dopo aver trovato la soluzione $\boldsymbol{e}$ del sistema lineare $\boldsymbol{Ae} = \boldsymbol{r}$ é possibile effettuare la correzione della soluzione approssimata:

$$\boldsymbol{u} = \boldsymbol{v} + \boldsymbol{e} \tag{1.2.5}$$

Si introducono ora i **Metodi Iterativi** che partendo da una stima iniziale della soluzione, perfezionano tale stima avvicinandola alla soluzione reale, applicando un numero di passi teoricamente infinito, da cui l'aggettivo ""*Iterativi*"".

Si consideri il seguente problema monodimensionale:

$$u(x)'' = f(x), \qquad 0 < x < 1 \tag{1.2.6}$$

$$u(0) = u(1) = 0. \tag{1.2.7}$$

che discretizzato diventa

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = f_i \tag{1.2.8}$$

e risolvendo rispetto a v_i

$$v_i = \frac{v_{i+1} + v_{i-1} - h^2 f_i}{2}$$

É quindi ora possibile applicare un primo metodo iterativo, il metodo di **Jacobi**, di cui viene fornito lo pseudo-codice:

Algorithm 1: Jacobi

$v^{old} = array()$
$v^{new} = array()$
$f = array()$
for $k = 1 \rightarrow max\_iteration$ **do**
  for $i = 1 \rightarrow N - 1$ **do**
    $v^{new}[i] \leftarrow v^{old}[i+1] + v^{old}[i-1] - h^2 * f[i]$ {Rilassamento}
  end for
end for

Tale metodo tiene traccia per ogni punto **interno** (sui punti al limite la soluzione é già nota) alla griglia di due valori, la vecchia soluzione e la nuova soluzione, così come mostrato nell'Algoritmo 1. Una volta calcolata la soluzione approssimata per ogni punto dell'intervallo, la vecchia soluzione viene sovrascritta dalla nuova. Relativamente al calcolo della soluzione, lo spazio occupato é quindi doppio rispetto alla dimensione della griglia in numero di punti. Nel caso di griglie di grandi dimensioni ciò può pesare in maniera significativa sulle prestazioni. Altro limite é dovuto al fatto che le nuove approssimazioni non possono essere utilizzate appena computate, ma vi é la necessità di aspettare di aver iterato su tutti i punti della griglia.

Poichè scopo di questo lavoro é risolvere PDE in maniera rapida, un'occhio di riguardo và riservato alla possibilità di parallelizzare gli algoritmi che verranno mano a mano descritti. Il fatto che il metodo di Jacobi per ogni punto conservi due valori, la soluzione vecchia e quella aggiornata, se da un lato porta ad una maggiore occupazione di memoria, d'altro canto rende l'algoritmo facilmente parallelizzabile; ogni passo di

rilassamento, istruzione in cui la nuova soluzione viene computata, utilizzando unicamente i valori del vettore contenente la vecchia soluzione e non del vettore con la soluzione aggiornata, se eseguito in parallelo, non porta a modifiche nel risultato finale.

Diverse considerazioni devono essere fatte relativamente al metodo di **Gauss-Seidel**, Algoritmo 2.

Algorithm 2: Gauss-Seidel

$v = array()$
$f = array()$
for $k = 1 \rightarrow max\_iteration$ **do**
  for $i = 1 \rightarrow N - 1$ **do**
    $v[i] \leftarrow v[i+1] + v[i-1] - h^2 * f[i]$ {Rilassamento}
  end for
end for

Tale metodo una volta computata la nuova soluzione, la utilizza immediatamente nelle successive iterazioni. Viene di conseguenza eliminato il problema dell'occupazione doppia di memoria causato dal metodo di Jacobi, ma nel farlo viene introdotto un altro fattore: l'ordine in cui vengono eseguiti i passi di rilassamento é qui importante. In caso di utilizzo di un'architettura parallela, essendo l'ordine di esecuzione dei passi di rilassamento deciso dallo scheduler, non vi é la possibilità di prevedere il risultato finale, introducendo un inaccettabile fattore di imprevedibilità. Tale algoritmo é quindi da considerarsi non parallelizzabile.

Viene quindi introdotta una versione modificata, e sopratutto parallelizzabile, del metodo di Gauss-Seidel: il **Red-Black Gauss-Seidel**, rappresentato dall'Algoritmo 3.

Andando ad analizzare il passo di rilassamento del metodo di Gauss-Seidel é possibile notare come l'equazione associata ad un punto x_i dipenda unicamente dagli immediati vicini x_{i+1} e x_{i-1}; in conclusione i punti identificati da un indice pari, dipendono uni-

Algorithm 3: Red-Black Gauss-Seidel

```
v = array()
f = array()
for k = 1 → max_iteration do
  for i = 1 → N − 1 do
    if isEven(i) then
      v[i] ← v[i + 1] + v[i − 1] − h^2 ∗ f[i] {Rilassamento}
    end if
  end for
  for i = 1 → N − 1 do
    if isOdd(i) then
      v[i] ← v[i + 1] + v[i − 1] − h^2 ∗ f[i] {Rilassamento}
    end if
  end for
end for
```

camente dai vicini di indice dispari e viceversa. L'algoritmo Red-Black Gauss-Seidel sfruttando questa proprietà é facilmente parallelizzabile eseguendo prima le iterazioni sui punti pari, completamente indipendenti fra loro, e quindi le iterazioni fra i punti dispari, anch'esse fra loro indipendenti, il tutto per un numero k di volte, stabilito a priori, sufficiente in teoria a garantire la convergenza verso la soluzione reale.
Relativamente al caso in 2D in figura Figura 1.3 viene mostrato come applicare l'algoritmo Red-Black Gauss-Seidel; un punto é considerato pari (punti rossi) se la somma degli indici i e j genera un numero pari, dispari (punti neri) se la somma genera un numero dispari (dispari). Sono stati messi in evidenza con un colore diverso i punti al contorno, in quanto l'algoritmo non modifica gli stessi, essendo già nota la soluzione su tali punti.

1.2.2 Risultati

L'algoritmo Red-Black Gauss-Seidel, risulta quindi essere il miglior metodo iterativo per la risoluzione di PDE, ed é quindi ora necessario studiarne l'effettiva efficacia.
Come banco di lavoro per effettuare valutazioni sul suddetto metodo viene proposta

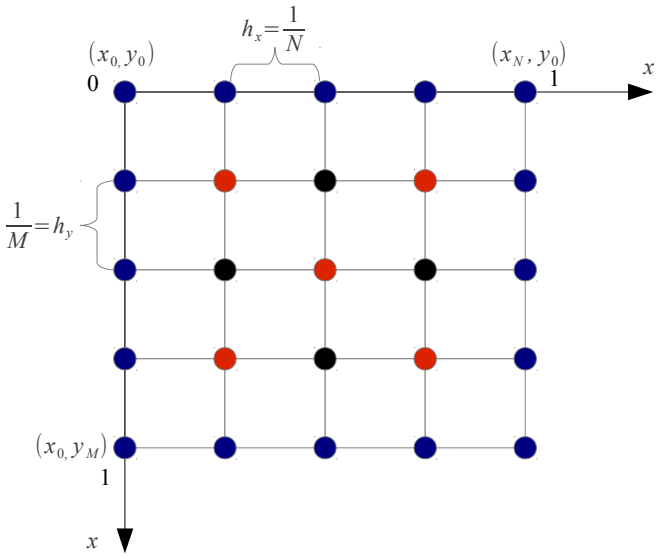

Figura 1.3: Griglia 2D - Red Black Gauss Seidel.

la seguente semplice equazione:

$$-u(x)'' = 0 \tag{1.2.9}$$

riscrivibile sotto forma di sistema

$$\boldsymbol{Au} = 0 \tag{1.2.10}$$

Poichè in questo caso la soluzione é nota, ossia pari a $\boldsymbol{u} = 0$ e cosí anche l'errore, $\boldsymbol{e} = \boldsymbol{u} - \boldsymbol{v} = -\boldsymbol{v}$, é possibile partire da una qualsiasi stima iniziale. Essendo il problema proposto un caso monodimensionale di **equazione di Laplace**, $\nabla u = 0$, la soluzione cercata sarà una funzione armonica (per definizione una funzione é armonica se é derivabile parzialmente due volte e soddisfa l'equazione di Laplace); sapendo inoltre che la **serie di Fourier**, $f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx}$, é costituita da una somma pesata di armoniche, anche dette **modi di Fourier**, ne deriva che una possibila stima iniziale possa essere composta dai seguenti modi di Fourier:

$$v_j = \sin \frac{jk\pi}{n}, \qquad 0 \leq j \geq n, \quad 1 \leq k \geq n-1 \tag{1.2.11}$$

con v_j componente j-esima del vettore $\boldsymbol{v}$ (associata al punto j sulla griglia) e k **numero d'onda**, numero di oscillazioni che un'onda compie nell'unità di spazio.
Il numero d'onda é di fondamentale importanza, in quanto l'efficacia dei Metodi Iterativi dipende fortemente da esso. Viene quindi introdotta un'importante distinzione:

- i modi caratterizzati da un numero d'onda k, tale che $1 \leq k < \frac{n}{2}$, vengono definiti modi a bassa frequenza, o **smooth** e corrispondono ad onde con poche oscillazioni per unità di tempo.

- i modi caratterizzati da $\frac{n}{2} \leq k \geq n-1$ sono invece definiti modi ad alta frequenza, od oscillatori.

Come banco di prova viene utilizzata una griglia a 64 punti partendo con tre differenti stime iniziali costituite dai modi **v1**, **v3** e **v6**, le cui j-esime componenti sono rappresentate dalla 1.2.12.

$$\begin{aligned} v_j &= \sin \frac{j\pi}{n} \\ v_j &= \sin \frac{3j\pi}{n} \\ v_j &= \sin \frac{6j\pi}{n} \end{aligned} \tag{1.2.12}$$

I risultati ottenuti nei tre casi sono mostrati in Figura 1.4, dove sulle ascisse é presente il numero di iterazioni eseguite e sulle ordinate la norma $\|e\|_\infty$. Dal grafico risulta evidente come i metodi iterativi si comportino bene nell'eliminare le componenti dell'errore a frequenza più alta, mentre le componenti a bassa frequenza vengono smorzate con poca efficacia anche dopo molte iterazioni. A tale proposito si dice che i metodi iterativi abbiano la cosiddetta ""**Smoothing Property**"". A seguito di altri esperimenti é possibile dare le seguenti conclusioni: i Metodi Iterativi hanno il vantaggio di essere facilmente implementabili, così come si evince dagli algoritmi sopra descritti; al fronte di tale facilità di implementazione, lo svantaggio principale nel

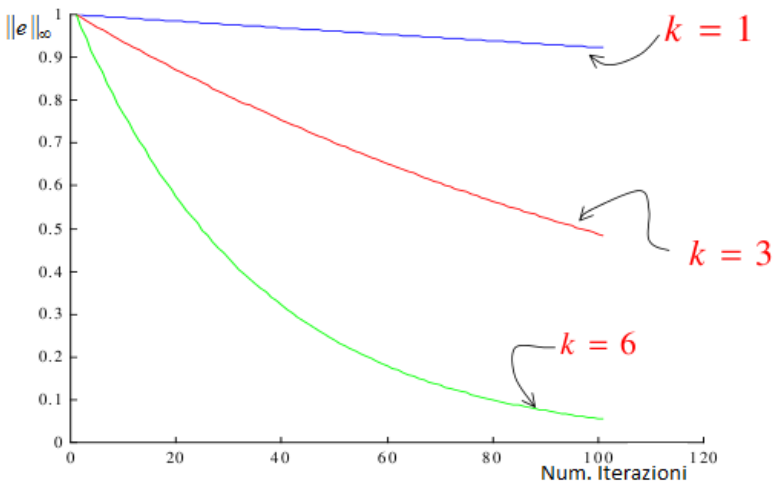

Figura 1.4: Efficienza Metodi Iterativi.

loro utilizzo corrisponde ad una poca efficacia nell'eliminare le componenti a bassa frequenza degli errori inevitabilmente presenti nelle stime iniziali; di conseguenza non vengono forniti risultati soddisfacenti, ossia soluzioni prossime a quelle reali.

É proprio che a causa di questi limiti che vengono poste le fondamenta per l'introduzione dei **Metodi MultiGrid**, i quali, sebbene utilizzino i metodi iterativi, li inglobano in algoritmi piú complessi per fronteggiare e superare le restrizioni riscontrate negli stessi.

Il successivo paragrafo é interamente dedicato alla descrizione di tali metodi, fulcro del lavoro svolto.

1.3 Metodi MultiGrid

1.3.1 Griglia Densa, Griglia Rada

Come il nome suggerisce, i **metodi MultiGrid** lavorano su piú griglie; quello che resta da definire é di che tipologia sono tali griglie. Per introdurre all'argomento verrà qui proposto un esempio; si consideri un'onda con $k = 4$, dove k é il numero d'onda,

su una griglia Ω^h (l'intervallo fra ogni punto é pari ad h) costituita da $n = 16$ punti (senza considerare x_0). Tale onda poichè $1 \leq k < \frac{n}{2}$ é definitiva ""smooth"", ossia non oscillatoria. Si consideri ora la griglia Ω^{2h} (l'intervallo fra ogni punto é pari ad $2h$) con $n = \frac{16}{2} = 8$ punti, legata dalla precedente tramite la seguente relazione:

$$x_{2i}^h = x_i^{2h}$$

ossia i punti della griglia meno densa (**coarse grid**), corrispondono ai punti pari della griglia densa (**fine grid**). Portando la forma d'onda precedente dalla griglia densa a quella meno densa, il risultato che si ottiene é quello in Figura 1.5. Si può notare

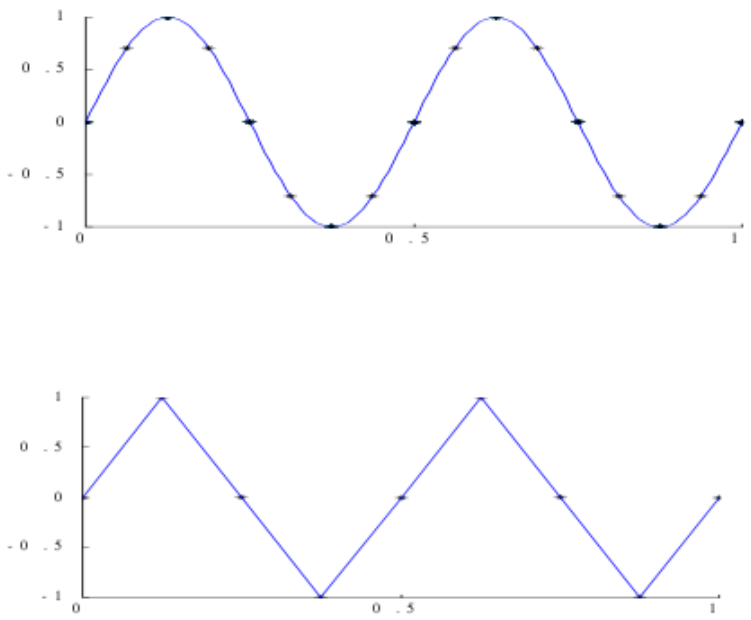

Figura 1.5: Trasferimento Onda: da griglia fine a griglia rada.

quindi che l'onda sia diventata più oscillatoria sulla griglia rada, di quanto non lo fosse sulla griglia fine. Si consideri ora un'onda con un'alto numero d'onda, $\frac{n}{2} \leq k \geq n - 1$; si ripete qui lo stesso procedimento di prima portando l'onda dalla griglia fine a quella rada, Figura 1.6. Dal grafico si evince che in questo caso le componenti oscillatorie si trasformano in componenti smooth, fenomeno chiamato **Aliasing**.

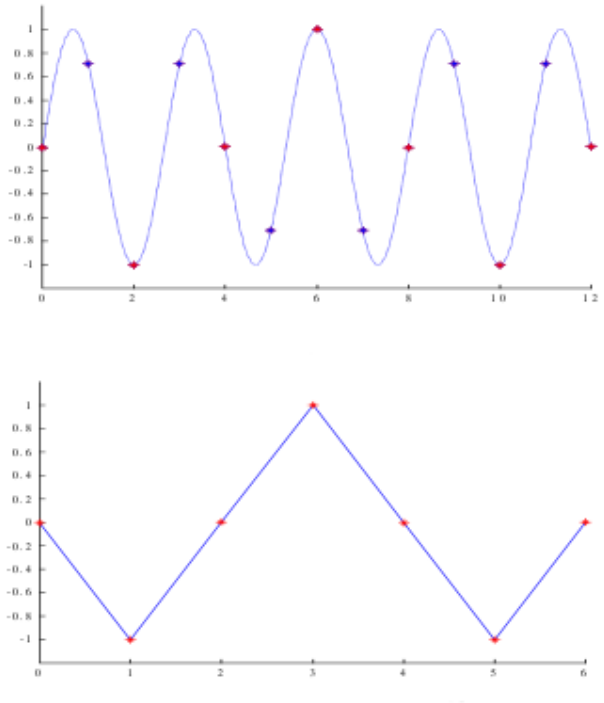

Figura 1.6: Trasferimento Onda: da griglia rada a griglia fine.

Riassumendo quando detto relativamente al trasferimento di un'onda da una griglia densa ad una meno densai:

- Le componenti smooth diventano oscillatorie
- Le componenti oscillatorie diventano smooth
- I Metodi Iterativi sono efficaci con le componenti oscillatorie, ma inadeguati relativamente alle componenti smooth

Dopo questo riepilogo é quindi possibile giungere ad un primo miglioramento dei Metodi Iterativi:

- Si esegue un certo numero di passi di rilassamento sulla griglia densa, riducendo così le componenti oscillatorie.

- L'onda viene studiata su una griglia meno densa: componenti smooth diventano oscillatorie
- Si eseguono un certo numero di passi di rilassamento sulla griglia meno densa, riducendo così le componenti oscillatorie (agendo in realtà sulle componenti smooth della griglia densa)
- L'onda viene posta sulla griglia densa (componenti smooth diventano oscillatorie), eseguendo altri passi di rilassamento

Tale algoritmo porta alla superazione di alcune delle restrizioni dei Metodi Iterativi, ma lascia in sospeso alcune questioni;

1. Come si ottiene la stima iniziale?
2. Come é possibile effettuare i passi di rilassamento sulla griglia più rada?
3. Se una volta tornati alla griglia più densa rimangono ancora componenti smooth, come si ci si deve comportare?
4. In che modo é possibile effettuare trasferimenti griglia fine $\rightarrow$ griglia rada e griglia rada $\rightarrow$ griglia fine?

1. Come si ottiene la stima iniziale?

Per rispondere al primo quesito si può pensare di non considerare unicamente la griglia immediatamente meno densa di quella iniziale, ma si può procedere ricorsivamente, dimezzando (nel caso monodimensionale) di volta in volta la dimensioni in numero di punti fino ad arrivare ad una griglia costituita unicamente da due punti esterni, x_0, x_2 ed un solo punto interno x_1. Poichè i punti esterni si trovano sul contorno del dominio, sicuramente si conosce la soluzione esatta sugli stessi grazie alle condizioni di contorno

specificate nel problema; di conseguenza con un solo passo di rilassamento e' possibile ottenere la soluzione sul punto interno. Per verificare quanto appena affermato basti pensare alla seguente equazione:

$$-u(x)'' = 0$$

che in forma discretizzata diventa

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{H^2} = 0$$

e il relativo passo di rilassamento

$$v[i] \leftarrow v[i+1] + v[i-1] - H^2 * f[i]$$

ossia sulla griglia meno densa Ω^H

$$v[1] \leftarrow v[2] + v[0] - H^2 * 0$$

dove v[2] e v[0] sono noti dalle condizioni al contorno.
Si tenga quindi in mente la possibilità di ottenere un buona stima iniziale partendo direttamente dalla griglia più rada.

2. Come é possibile effettuare i passi di rilassamento sulla griglia più rada?

Nel paragrafo precedente si era parlato della cosiddetta equazione del residuo, la 1.2.4, qui riproposta per comodità

$$\boldsymbol{Ae} = \boldsymbol{r} = \boldsymbol{f} - \boldsymbol{Av}$$

Viene qui sottolineata una fondamentale relazione fra la soluzione approssimata e l'errore:

> *Effettuare dei passi di rilassamento sull'equazione* $\boldsymbol{Au} = \boldsymbol{f}$ *con una certa stima iniziale* $\boldsymbol{v}$*, é equivalente ad effettuare passi di rilassamento sull'equazione* $\boldsymbol{Ae} = \boldsymbol{r}$ *con stima iniziale* $\boldsymbol{e} = 0$

Grazie a questo risultato, si puó quindi pensare di lavorare sulle griglie rade con l'equazione dei residui, ottenendo così un'approssimazione dell'errore, per poi trasferire tale errore alla griglia più densa, dove eseguire ulteriori passi di rilassamento per perfezionare la soluzione ottenuta. Si tenga bene in mente anche questo risultato.

3. Se una volta tornati alla griglia più densa rimangono ancora componenti smooth, come si ci si deve comportare?

É necessario costruire un algoritmo che si muova ricorsivamente percorrendo più volte il percorso griglia densa→ griglia rada e griglia rada→ griglia densa per assicurare la riduzione massima possibile di ogni componenete dell'errore.

4. In che modo possibile effettuare trasferimenti fine→rada e rada→fine?

Gli operatori di trasferimento fra griglie saranno argomento della prossima sezione.

1.3.2 Operatori di Trasferimento: Restrizione, Interpolazione

Poichè i metodi MultiGrid si basano sull'utilizzo di più griglie vi é la necessità di definire degli operatori che permettano il trasferimento di informazioni fra una griglia e l'altra. Vengono quindi qui introdotti due operatori, **Restrizione** e **Interpolazione**, la cui rappresentazione é diversa a seconda della dimensione del dominio dell'equazione da risolvere. Verranno qui unicamente presentate le espressioni di tali operatori nei caso 1D, 2D e 3D.

L'operazione di Restrizione corrisponde al trasferimento di informazioni da una griglia densa ad una meno densa. La sua rappresentazione é la seguente:

$$\boldsymbol{I}_h^{2h}\boldsymbol{v}^h = \boldsymbol{v}^{2h} \tag{1.3.1}$$

Poichè i punti di una griglia rada, corrispondono ai punti pari della griglia immediatamente più densa una prima operazione di Restrizione che si potrebbe attuare é

la cosiddetta **Iniezione**. Così come mostrato in Figura 1.7 tale operazione prende semplicemente le soluzioni presenti sui punti pari della griglia densa, e le trasferisce nella griglia rada, senza attuare alcuna modifica. Quindi:

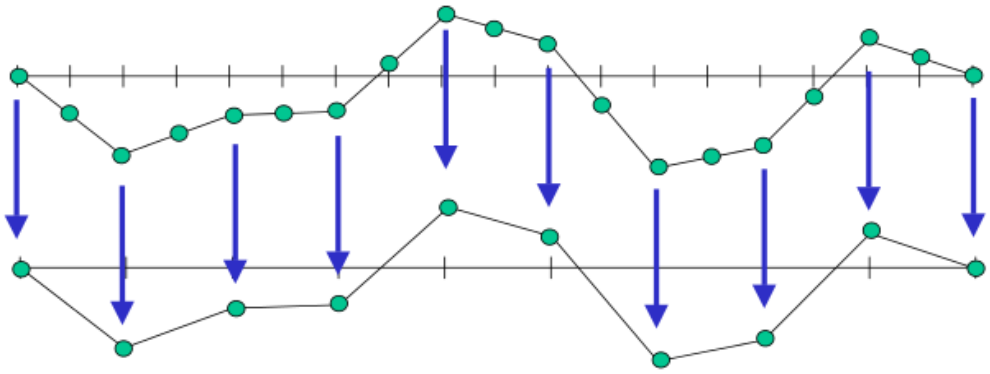

Figura 1.7: Iniezione 1D.

$$v_j^{2h} = v_{2j}^h \tag{1.3.2}$$

Non sempre però la semplice Iniezione offre risultati soddisfacenti, in determinate occasioni é preferibile utilizzare qualcosa di più complesso; si introduce quindi la cosidetta Interpolazione ""**Full Weighting**"", il cui nome deriva dal fatto che i punti della griglia meno densa vengono ottenuti effettuando una media pesata dei vicini punti della griglia densa. La relativa espressione nel caso monodimensionale é la seguente:

$$v_j^{2h} = \frac{1}{4}(v_{2j-1}^h + 2v_{2j}^h + v_{2j+1}^h), \qquad 1 \leq j \geq \frac{n}{2} - 1 \tag{1.3.3}$$

e coinvolge tre punti (3^1), ognuno con il proprio peso, così come mostrato in Figura 1.8. É di fondamentale importanza sottolineare come x_0 e x_N, punti al contorno del dominio, vengano copiati dalla griglia densa alla griglia rada senza effettuare alcuna modifica sul relativo valore della soluzione approssimata; anche sulle griglie meno dense le condizioni al contorno stabilite sulla griglia più densa continuano a rimanere

valide. Resta quindi da vedere come é possibile trasferire informazioni da una griglia

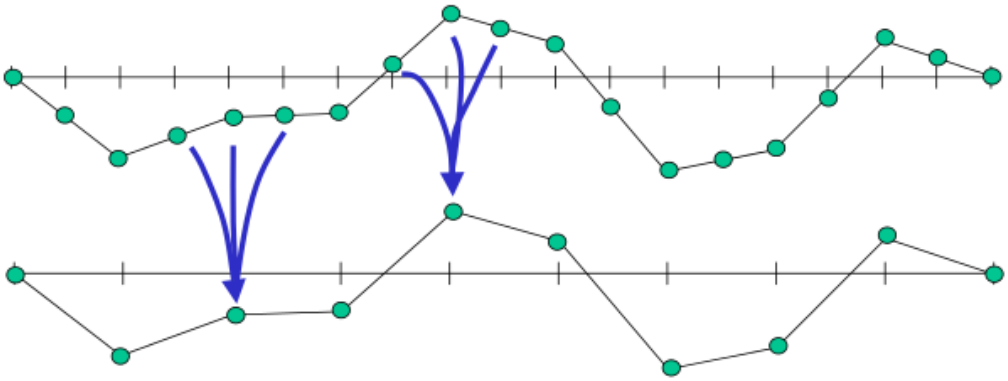

Figura 1.8: Full-Weighting 1D.

densa ad una meno densa nei casi a due o tre dimensioni.

Relativamente alla Restrizione Full Weighting in due dimensioni, il valore definito sui punti interni della griglia rada si ottiene effettuando una media pesata che coinvolge tutti gli immediati vicini sulla griglia fine del punto in considerazione. Una rappresentazione della Restrizione Full Weighting nel caso a due dimensioni e' mostrata in Figura 1.9, dove é associato un peso ad ogni punto della griglia densa che contribuisce al valore della soluzione nel punto sulla griglia meno densa. L'espressione relativa é

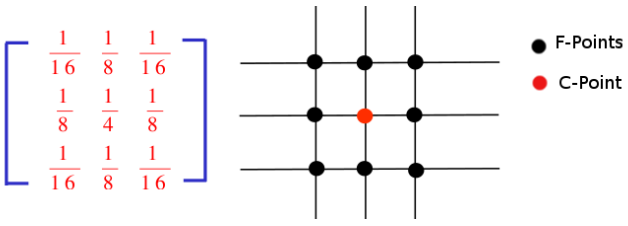

Figura 1.9: Full-Weighting 2D.

quindi la seguente

$$v_{ij}^{2h} = \frac{1}{16}(v_{2i-1,2j-1}^{h} + v_{2i-1,2j+1}^{h} + v_{2i+1,2j-1}^{h} + v_{2i+1,2j+1}^{h} + 2(v_{2i,2j-1}^{h} + v_{2i,2j+1}^{h} + v_{2i-1,2j}^{h} + v_{2i+1,2j}^{h}) + 4v_{2i,2j}^{h}), \qquad 1 \leq i,j \geq \frac{n}{2} - 1 \quad (1.3.4)$$

e coinvolge nove (3^2) punti.

La Restrizione in tre dimensioni viene effettuata tramite una media pesata che coinvolge in tutto ventisette ((3^3)) punti). Una rappresentazione della stessa é mostrata in Figura 1.10. Ogni punto della griglia fine é stato etichettato da un numero che

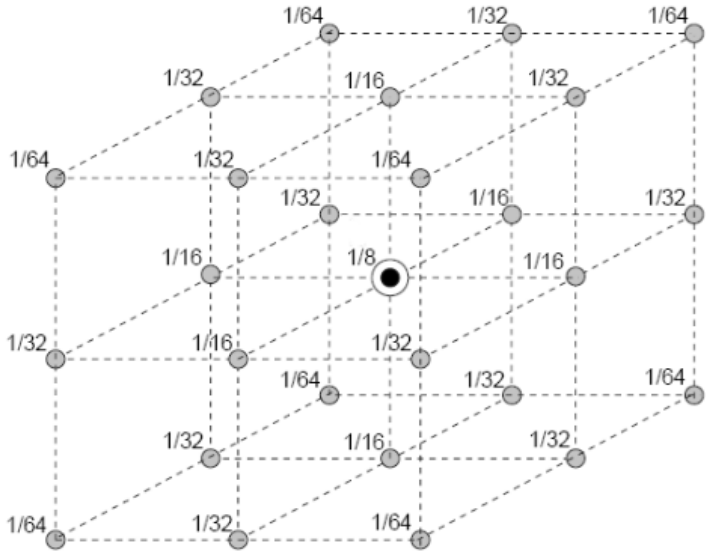

Figura 1.10: Full-Weighting 3D.

corrisponde al peso con cui ciascun punto figura nell'espressione della Restrizione 3D,

come si può notare nella 1.3.5.

$$\begin{aligned} v_{ijk}^{2h} = \frac{1}{64}((&v_{2i-1,2j-1,2k+1}^{h} + v_{2i-1,2j+1,2k+1}^{h} + v_{2i-1,2j+1,2k-1}^{h} + v_{2i-1,2j-1,2k-1}^{h} \\ &+ v_{2i+1,2j-1,2k+1}^{h} + v_{2i+1,2j+1,2k+1}^{h} + v_{2i+1,2j+1,2k-1}^{h} + v_{2i+1,2j-1,2k-1}^{h}) \\ &+ 2(v_{2i-1,2j,2k+1}^{h} + v_{2i-1,2j+1,2k}^{h} + v_{2i-1,2j,2k-1}^{h} + v_{2i-1,2j-1,2k}^{h} \\ &+ v_{2i,2j+1,2k+1}^{h} + v_{2i,2j+1,2k+1}^{h} + v_{2i,2j+1,2k-1}^{h} + v_{2i,2j-1,2k-1}^{h} \\ &+ v_{2i+1,2j,2k+1}^{h} + v_{2i+1,2j+1,2k}^{h} + v_{2i+1,2j,2k-1}^{h} + v_{2i+1,2j-1,2k}^{h}) \\ &+ 4(v_{2i,2j,2k+1}^{h} + v_{2i,2j+1,2k}^{h} + v_{2i,2j,2k-1}^{h} + v_{2i,2j-1,2k}^{h} + v_{2i-1,2j,2k}^{h} + v_{2i,2j+1,2k}^{h}) \\ &+ 8(v_{2i,2j,2k}^{h})) \end{aligned} \quad (1.3.5)$$

Si conclude quindi la descrizione della Restrizione nei tre casi affrontati in questo lavoro. Proseguendo nell'aumento delle dimensioni, per un problema d-dimensionale figureranno in tutto 3^d punti nell'espressione dell'operatore di Restrizione.
L'Interpolazione é l'operazione inversa alla Restrizione e permette il trasferimento di informazioni da una griglia rada ad una densa. La relativa espressione é la seguente

$$\boldsymbol{I}_{2h}^{h}\boldsymbol{v}^{2h} = \boldsymbol{v}^{h} \quad (1.3.6)$$

dove rispetto all'espressione della Restrizione, la 1.3.1, vengono scambiati di posto il pedice e l'apice in quanto il verso dell'operazione é opposto. Anche in questo caso verrà data una descrizione dell'operatore nei casi 1D, 2D e 3D. Nel caso d-dimensionale devono essere fornite 2^d espressioni per poter definire completamente l'Interpolazione, in quanto ogni indice puó assumere valore pari o dispari.
Relativamente al caso monodimensionale i punti identificati da indice pari vengono copiati dalla griglia rada a quella densa, mentre ai punti contrassegnati da indice dispari viene assegnato un valore equivalente alla media dei due punti adiacenti sulla

griglia rada, così come mostrato nella 1.3.7 e nella Figura 1.11

$$\begin{aligned} v_{2j}^{h} &= v_{j}^{2h} \\ v_{2j+1}^{h} &= \frac{1}{2}(v_{j}^{2h} + v_{j+1}^{2h}) \end{aligned} \tag{1.3.7}$$

Nel caso di problema bidimensionale si ha a che fare con due indici, i e j rappresentati

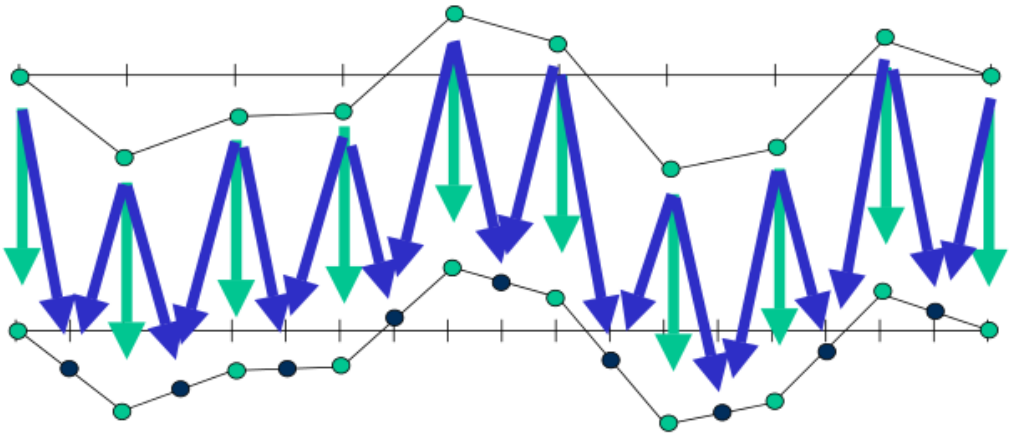

Figura 1.11: Interpolazione 1D.

rispettivamente ordinate e ascisse. I casi da discutere sono quindi quattro:

- i pari, j pari
- i dispari, j pari
- i pari, j dispari
- i dispari, j dispari

Tali casi sono rappresentati dalla 1.3.8.

$$\begin{aligned} v_{2i,2j}^{h} &= v_{i,j}^{2h} \\ v_{2i+1,2j}^{h} &= \frac{1}{2}(v_{i,j}^{2h} + v_{i+1,j}^{2h}) \\ v_{2i,2j+1}^{h} &= \frac{1}{2}(v_{i,j}^{2h} + v_{i,j+1}^{2h}) \\ v_{2i+1,2j+1}^{h} &= \frac{1}{4}(v_{i,j}^{2h} + v_{i+1,j}^{2h} + v_{i,j+1}^{2h} + v_{i+1,j+1}^{2h}) \end{aligned} \tag{1.3.8}$$

In Figura 1.12 é mostrata una rappresentazione simile a quella in Figura 1.9; in essa ad ogni punto F della griglia fine, viene associato il peso con cui il punto sulla griglia meno densa (al centro della figura) contribuisce al valore di F. Si riporta infine il

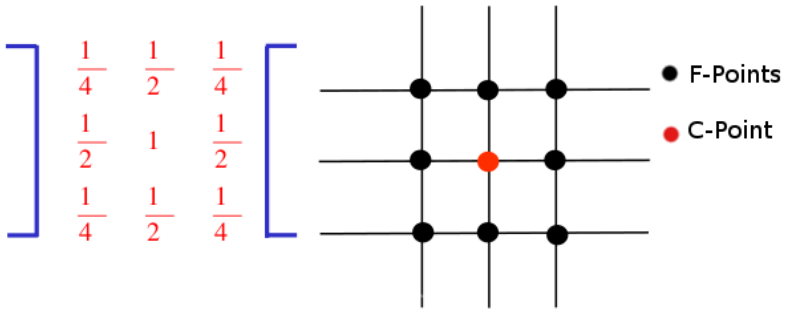

Figura 1.12: Interpolazione 2D.

caso a tre dimensioni. L'espressione risultate per i motivi descritti precedentemente riguarda in tutto $2^3 = 8$ casi, come si evince dalla 1.3.9.

$$
\begin{aligned}
v^h_{2i,2j,2k} &= v^{2h}_{i,j,k} \\
v^h_{2i,2j+1,2k} &= \frac{1}{2}(v^{2h}_{i,j,k} + v^{2h}_{i,j+1,k}) \\
v^h_{2i+1,2j,2k} &= \frac{1}{2}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k}) \\
v^h_{2i+1,2j+1,2k} &= \frac{1}{4}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k} + v^{2h}_{i,j+1,k} + v^{2h}_{i+1,j+1,k}) \\
v^h_{2i,2j,2k+1} &= \frac{1}{2}(v^{2h}_{i,j,k} + v^{2h}_{i,j,k+1}) \\
v^h_{2i,2j+1,2k+1} &= \frac{1}{4}(v^{2h}_{i,j,k} + v^{2h}_{i,j+1,k} + v^{2h}_{i,j,k+1} + v^{2h}_{i,j+1,k+1}) \\
v^h_{2i+1,2j,2k+1} &= \frac{1}{4}(v^{2h}_{i,j,k} + v^{2h}_{i+1,j,k} + v^{2h}_{i,j,k+1} + v^{2h}_{i+1,j,k+1}) \\
v^h_{2i+1,2j+1,2k+1} &= \frac{1}{8}(v^{2h}_{i,j,k} + v^{2h}_{i,j,k+1} + v^{2h}_{i,j+1,k+1} + v^{2h}_{i,j+1,k} \\
&\quad + v^{2h}_{i+1,j,k} + v^{2h}_{i+1,j,k} + v^{2h}_{i+1,j+1,k+1} + v^{2h}_{i+1,j+1,k})
\end{aligned}
\tag{1.3.9}
$$

In Figura 1.13 viene invece mostrata una rappresentazione con tre situazioni possibili (resta fuori il caso in cui ogni indice risulta essere pari)

- un indice dispari: i punti coinvolti hanno peso $\frac{1}{2}$ (tre casi)

- due indici dispari: i punti coinvolti hanno peso $\frac{1}{4}$ (tre casi)
- tre indice dispari: i punti coinvolti hanno peso $\frac{1}{8}$ (un caso)

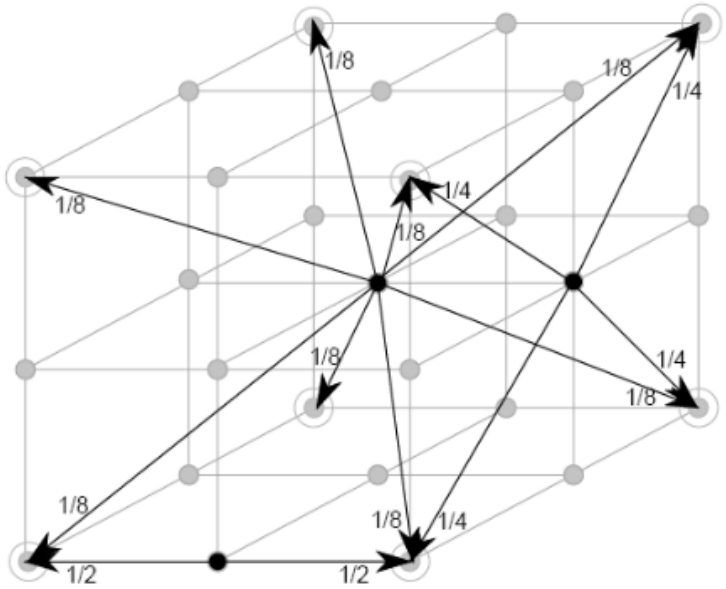

Figura 1.13: Interpolazione3D.

Si é quindi conclusa la trattazione sugli operatori di trasferimento fra griglie, e quindi sono state poste tutte le fondamenta per poter introdurre i metodi MultiGrid.

1.3.3 Algoritmi MultiGrid

Grazie agli operatori di Restrizione e Interpolazione é ora possibile studiare algoritmi per la risoluzione di PDE più performanti relativamente ai metodi descritti nei precedenti paragrafi.

Viene qui introdotto un primo metodo per tale scopo. Si ricordi il risultato enunciato nel paragrafo 1.3.1, ossia lavorare sull'equazione dei residui, $\boldsymbol{Ae} = \boldsymbol{r} = \boldsymbol{f} - \boldsymbol{Av}$ con stima iniziale $\boldsymbol{e} = 0$ é equivalente a lavorare con l'equazione di partenza $\boldsymbol{Au} = \boldsymbol{f}$ con arbitraria stima iniziale $\boldsymbol{v}$; é quindi possibile realizzare un primo algoritmo:

- Partire dalla griglia più densa, Ω^h ed effettuare ν_1 passi di rilassamento sull'equazione originale $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima iniziale $\boldsymbol{v}^h$
- Computare il residuo, $\boldsymbol{r}^h = \boldsymbol{f}^h - \boldsymbol{A}^h\boldsymbol{v}^h$ sulla griglia Ω^h e tramite l'operazione di Restrizione, $\boldsymbol{I}_h^{2h}\boldsymbol{r}^h = \boldsymbol{r}^{2h}$, trasferirlo sulla griglia meno densa, Ω^{2h}
- Effettuare ν_1 passi di rilassamento sull'equazione dei residui $\boldsymbol{A}^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$ con stima iniziale $\boldsymbol{e}^{2h} = 0$
- Tramite l'operazione di Interpolazione trasferire l'errore da Ω^{2h} a Ω^h, $\boldsymbol{I}_{2h}^h\boldsymbol{e}^{2h} = \boldsymbol{e}^h$
- Correggere la soluzione sulla griglia più fine, $\boldsymbol{v}^h = \boldsymbol{v}^h + \boldsymbol{e}^h$
- Effettuare ν_2 passi di rilassamento sull'equazione originale $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima $\boldsymbol{v}^h$

Tale algoritmo viene chiamato ""**Two Grid Correction Scheme**"", ossia schema di correzione a due griglie. Tale metodo utilizza i due operatori di trasferimento fra griglie, oltre all'equazione dei residui il cui scopo é quello di correggere la soluzione una volta stimato l'errore (calcolato nella griglia meno densa). Le costanti ν_1 e ν_2 vengono stabilite a priori in base a premesse teoriche o a precedenti esperimenti.
Questo metodo é sicuramente migliorabile, in quanto rimangono alcuni questioni irrisolte:

1. Poichè il primo passo dell'algoritmo viene effettuato sulla griglia più densa, é tuttora necessario effettuare una stima iniziale della soluzione
2. Non é ancora possibile trovare una soluzione esatta all'equazione del residuo, in quanto le incognite del sistema sono più di una (la griglia Ω^{2h} ha un numero di punti interni diverso da 1)

3. Alla fine dell'algoritmo é ancora possibile trovare alcune componenti dell'errore, per i due precedenti punti.

Il metodo Two Grid Correction Scheme può essere migliorato pensando di trasferire ricorsivamente informazioni a griglie di dimensioni sempre minori, in modo tale da trovare il valore esatto dell'errore sulla griglia con minor numero di punti, Ω^H, e quindi interpolare tale errore di griglia in griglia effettuando ogni volta la correzione della soluzione. Questa estensione dello schema di correzione a due griglie viene definito **V-Cycle**, ciclo a V.

Prima di procedere nella descrizione dell'algoritmo, bisogna fornire alcune precisazioni relativamente alle dimensioni delle griglie, e al numero delle stesse; ogni griglia ha un numero di punti lungo un asse (per semplicità si può imporre lo stesso numero di punti lungo gli altri assi) pari a $2^k + 1$, x_0 compreso, in modo da effettuare la divisione del dominio in un numero di intervalli pari ad una potenza di 2; in base alla dimensione della griglia più densa, si stabilisce il numero totale di griglie; se ad esempio si prende $k = 6$, quindi Ω^h ha dimensione pari a $2^6 + 1 = 65$ punti lungo un asse, il numero totale di griglie sarà pari a $\log((2^6 + 1) - 1) = 6$, dove Ω^H ha una sola incognita, un solo punto interno, su cui non é nota la soluzione. Le dimensioni della griglia più densa vengono stabilite in base alla grandezza dell'intervallo su cui si vuol effettuare il calcolo; maggiore é l'ampiezza, maggiore deve essere il numero di punti di cui é costituita la griglia più fine, in modo da ottenere una buona approssimazione della soluzione.

Nel descrivere l'Algoritmo V-Cycle, vengono introdotte alcune semplificazioni: nell'equazione del residuo, $\boldsymbol{A}^{2h}\boldsymbol{e}^{2h} = \boldsymbol{r}^{2h}$, é possibile rinominare $\boldsymbol{e}^{2h}$ in $\boldsymbol{v}^{2h}$, in quanto costituisce un vettore di soluzioni, e $\boldsymbol{r}^{2h}$ in $\boldsymbol{f}^{2h}$, in quanto costituisce un vettore di termini noti. Tale modifica non si rispecchia solo nella notazione, ma, come si vedrà

successivamente, per ogni griglia vengono mantenuti solo due vettori: vettore soluzione approssimata (che a seconda della griglia considerata può contenere l'errore) e vettore termini noti (che a seconda della griglia considerata può contenere il residuo). A seguire viene riportata la struttura dell'algoritmo V-Cycle.

- Applica ν_1 volte il passo di rilassamento all'equazione $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima iniziale $\boldsymbol{v}^h$
- Calcola il Residuo $\boldsymbol{r}^h = \boldsymbol{f}^h - \boldsymbol{A}^h\boldsymbol{v}^h$ e trasferiscilo alla griglia Ω^{2h}
- Applica ν_1 volte il passo di rilassamento all'equazione $\boldsymbol{A}^{2h}\boldsymbol{u}^{2h} = \boldsymbol{f}^{2h}$ con stima iniziale $\boldsymbol{v}^{2h} = 0$
- Calcola il Residuo $\boldsymbol{r}^{2h} = \boldsymbol{f}^{2h} - \boldsymbol{A}^{2h}\boldsymbol{v}^{2h}$ e trasferiscilo alla griglia Ω^{4h}
- ...
- Calcola il Residuo $\boldsymbol{r}^{H-1} = \boldsymbol{f}^{H-1} - \boldsymbol{A}^{H-1}\boldsymbol{v}^{H-1}$ e trasferiscilo sulla griglia Ω^H
- Calcola la soluzione reale dell'equazione $\boldsymbol{A}^H\boldsymbol{v}^H = \boldsymbol{f}^H$ con stima iniziale $\boldsymbol{v}^H = 0$
- Trasferisci errore $\boldsymbol{v}^H$ su griglia Ω^{H-1} e applica correzione $\boldsymbol{v}^{H-1} = \boldsymbol{v}^{H-1} + \boldsymbol{I}_H^{H-1}\boldsymbol{v}^H$
- Applica ν_2 volte il passo di rilassamento all'equazione $\boldsymbol{A}^{H-1}\boldsymbol{u}^{H-1} = \boldsymbol{f}^{H-1}$ con stima iniziale $\boldsymbol{v}^{H-1}$
- ...
- Trasferisci errore $\boldsymbol{v}^{2h}$ su griglia Ω^h e applica correzione $\boldsymbol{v}^h = \boldsymbol{v}^h + \boldsymbol{I}_{2h}^h\boldsymbol{v}^{2h}$
- Applica ν_2 volte il passo di rilassamento all'equazione $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima iniziale $\boldsymbol{v}^h$

É bene sottolineare che dal punto di vista concettuale, solo su Ω^h si lavora effettivamente sull'equazione originale e non sull'equazione del residuo. Fondamentale notare come nel caso di risoluzione dell'equazione del residuo, la stima iniziale dell'errore deve essere pari a zero non solo sui punti interni alla griglia, ma anche sul contorno e su tutti gli altri punti su cui é nota la soluzione esatta, grazie alle condizioni fornite all'inizio del problema.

Utilizzando la ricorsione é possibile riscrivere il V-Cycle in una forma più compatta

$$\boldsymbol{v}^h = V^h(\boldsymbol{v}^h, \boldsymbol{f}^h)$$

- Rilassa ν_1 volte l'equazione $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima iniziale $\boldsymbol{v}^h$
- **IF** $(\Omega^h = \Omega^H)$ (la griglia corrente é la meno densa)

  Rilassa ν_2 volte l'equazione $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima iniziale $\boldsymbol{v}^h$
- **ELSE**

  $\boldsymbol{f}^{2h} = \boldsymbol{I}_h^{2h}(\boldsymbol{f}^h - \boldsymbol{A}^h\boldsymbol{v}^h)$

  $\boldsymbol{v}^{2h} \leftarrow 0$

  $\boldsymbol{v}^{2h} \leftarrow V^{2h}(\boldsymbol{v}^{2h}, \boldsymbol{f}^{2h})$

  Rilassa ν_2 volte l'equazione $\boldsymbol{A}^h\boldsymbol{u}^h = \boldsymbol{f}^h$ con stima iniziale $\boldsymbol{v}^h$

Una rappresentazione grafica dell'algoritmo é mostrata in Figura 1.14 dove sono state evidenziate le operazioni compiute ad ogni livello e al passaggio da un livello all'altro. Nella griglia Ω^h é stata utilizzata una notazione differente relativamente all'equazione del residuo in quanto, poichè tale griglia é costituita da un solo punto interno e quindi da una solo incognita, é possibile trovare la soluzione esatta dell'equazione.

Tramite il V-Cycle si sfruttano tutti i vantaggi derivanti dall'utilizzo di più griglie,

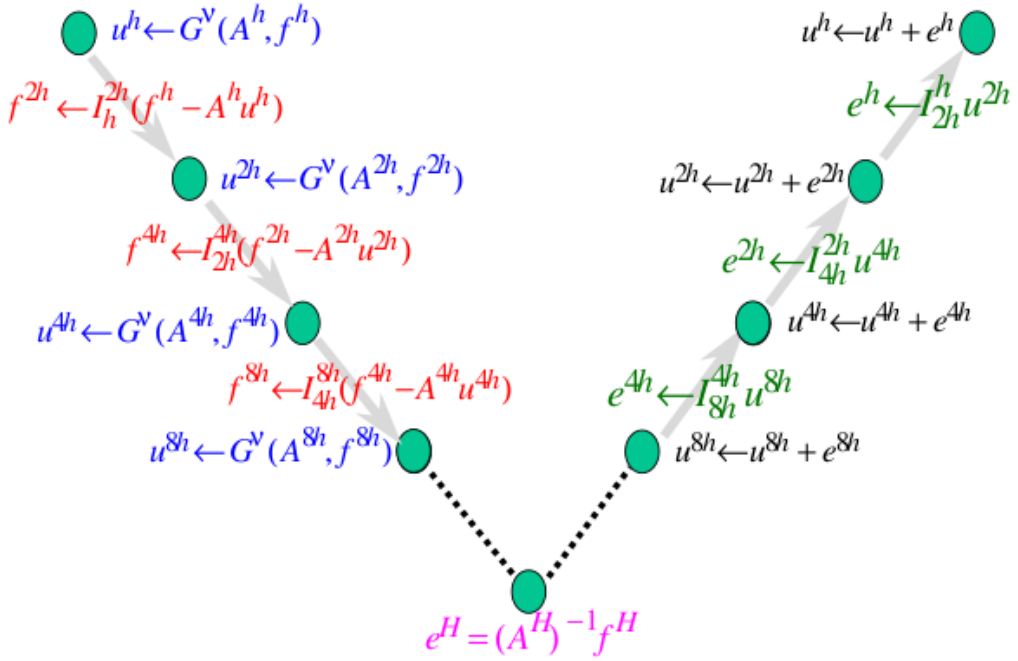

Figura 1.14: V-Cycle.

come descritto precedentemente, e quindi non sussistono più i problemi dei metodi iterativi o dello schema di correzione a due griglie. Nonostante ciò l'efficienza dell'algoritmo dipende in ogni caso da una buona stima iniziale. Nei paragrafi precedenti si era sottolineato di come sia possibile ottenere una buona stima iniziale risolvendo esattamente l'equazione del residuo nella griglia meno densa, Ω^H, da cui si viene a conoscenza dell'errore esatto da interpolare ed utilizzare nelle altre griglie. Il prossimo algoritmo presentato si basa su queste conclusioni per ottenere una buona stima iniziale e quindi garantire la convergenza verso una buona approssimazione della soluzione reale. A seguire viene presentata la struttura del cosiddetto ""**Full MultiGrid V-Cycle**""

- Trasferisci il vettore della soluzione nota fino a livello inferiore, Ω^{kh} applicando l'operatore di Restrizione: $\boldsymbol{f}^{2h} = \boldsymbol{I}_h^{2h}\boldsymbol{f}^h, \boldsymbol{f}^{4h} = \boldsymbol{I}_{2h}^{4h}\boldsymbol{f}^{2h}, ...\boldsymbol{f}^H = \boldsymbol{I}_{H-1}^H\boldsymbol{f}^{H-1}$

- Trova la soluzione esatta all'equazione del residuo sul livello Ω^H
- Esegui interpolazione stima iniziale $\boldsymbol{v}^{H-1} \leftarrow \boldsymbol{I}_H^{H-1}\boldsymbol{v}^H$
- Esegui ν_0 volte un V-Cycle a partire dal livello Ω^{H-1}
- ...
- Esegui interpolazione stima iniziale $\boldsymbol{v}^{2h} \leftarrow \boldsymbol{I}_{4h}^{2h}\boldsymbol{v}^{4h}$
- Esegui ν_0 volte un V-Cycle a partire dal livello Ω^{2h}
- Esegui interpolazione stima iniziale $\boldsymbol{v}^h \leftarrow \boldsymbol{I}_{2h}^{h}\boldsymbol{v}^{2h}$
- Esegui ν_0 volte un V-Cycle a partire dal livello Ω^h

Una rappresentazione grafica dell'algoritmo appena descritto é mostrata in Figura 1.15, dove ad ogni livello vengono eseguiti $v_0 = 1$ passi di V-Cycle. In figura

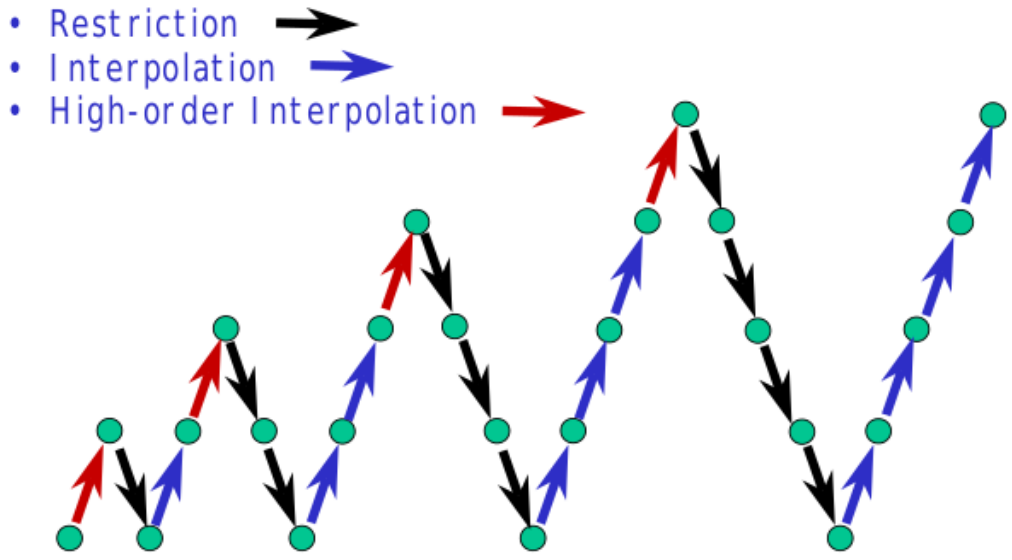

Figura 1.15: Full MultiGrid V-Cycle.

sono evidenziate le operazioni di Restrizione, Interpolazione, dove l'ultima operazione

é stata identificata con un colore differente (rosso), nel caso in cui essa porti al livello superiore la stima ottenuto dal V-Cycle appena eseguito. Anche per il Full MultiGrid V-Cycle é possibile dare un'espressione ricorsiva più compatta della precedente.

$$\boldsymbol{v}^h = FMG^h(\boldsymbol{f}^h)$$

- **IF** $(\Omega^h = \Omega^H)$ (la griglia corrente é la meno densa)

  $\boldsymbol{v}^h \leftarrow 0$

  $\boldsymbol{v}^h \leftarrow V^h(\boldsymbol{v}^h, \boldsymbol{f}^h)$ ν_0 volte

- **ELSE**

  $\boldsymbol{f}^{2h} \leftarrow \boldsymbol{I}_h^{2h} \boldsymbol{f}^h$

  $\boldsymbol{v}^{2h} = FMG^{2h}(\boldsymbol{f}^{2h})$

  Applica correzione $\boldsymbol{v}^h \leftarrow \boldsymbol{I}_{2h}^h \boldsymbol{v}^{2h}$

  $\boldsymbol{v}^h \leftarrow V^h(\boldsymbol{v}^h, \boldsymbol{f}^h)$

I passi iniziali del Full MultiGrid V-Cycle garantiscono la partenza dalla miglior stima iniziale possibile, eliminando le problematiche derivanti da stime arbitrarie eccessivamente distanti dalla soluzione reale; l'esecuzione ripetuta di V-Cycle di livello in livello invece garantisce una maggior riduzione delle componenti dell'errore, eliminando di fatto la possibilità di trovare componenti residue di errore alla fine dell'algoritmo.
É quindi possibile concludere che l'algoritmo appena descritto, il Full MultiGrid V-Cycle, costituisce il miglior metodo per risolvere PDE lineari.

1.3.4 Casi Particolari: PDE non lineari, Condizioni al contorno di Neumann

Nei paragrafi precedenti sono stati trattati vari metodi per risolvere PDE lineari, in caso di condizioni al contorno di Dirichlet, ossia condizioni specificanti il valore della soluzione lungo i bordi del dominio. In questa sezione verranno sottolineate le differenze nel caso di PDE non lineari e di utilizzo di condizioni al contorno di Neumann, specificanti il valore di alcune derivate della funzione lungo i bordi del dominio.

PDE non lineari

Nel caso di PDE lineari i vari metodi ruotavano sull'utilizzo dell'equazione del residuo, definita da

$$\boldsymbol{Au} - \boldsymbol{Av} = \boldsymbol{f} - \boldsymbol{Av} \rightarrow \boldsymbol{Ae} = \boldsymbol{r} \tag{1.3.10}$$

Sia data ora la seguente PDE non lineare

$$\boldsymbol{A(u)} = \boldsymbol{f} \tag{1.3.11}$$

dove con la notazione $\boldsymbol{A(u)}$ si vuole sottolineare la non linearità dell'equazione. Per un sistema non lineare la relazione 1.3.10 non é valida in quanto

$$\boldsymbol{A(u)} - \boldsymbol{A(v)} \neq \boldsymbol{A(e)} \tag{1.3.12}$$

É quindi necessario lavorare con un'equazione del residuo diversa rispetto al caso lineare

$$\boldsymbol{A(u)} - \boldsymbol{A(v)} = \boldsymbol{f} - \boldsymbol{A(v)} = \boldsymbol{r} \tag{1.3.13}$$

La risoluzione di PDE non lineari può essere effettuata, come nel caso lineare, con metodi iterativi. Fra i metodi iterativi il **Metodo di Newton** é considerato il migliore.

Si consideri l'equazione

$$F(x) = 0$$

Applicando l'espansione di Taylor troncata al primo ordine si ottiene

$$F(x+s) = F(x) + sF'(x) + o(s^2)$$

se $x + s$ é una soluzione dell'equazione, allora

$$0 = F(x) + sF'(x)$$

per cui

$$s = -\frac{F(x)}{F'(x)}$$

da cui deriva l'iterazione

$$\begin{aligned} x &\leftarrow x + s \\ x &\leftarrow x - \frac{F(x)}{F'(x)} \end{aligned} \tag{1.3.14}$$

Si consideri ora il sistema

$$\boldsymbol{A(u)} = 0$$

Applicando l'espansione di Taylor a $\boldsymbol{A(v+e)}$ si ottiene

$$\boldsymbol{A(v+e)} = \boldsymbol{A(v)} + \boldsymbol{J(v)e} + o(\boldsymbol{e}^2)$$

dove $\boldsymbol{J(v)}$ é la Matrice Jacobiana definita da

$$\boldsymbol{J(v)} = \begin{vmatrix} \frac{\partial f_1}{u_1} & \frac{\partial f_1}{u_2} & \dots & \frac{\partial f_1}{u_N} \\ \frac{\partial f_2}{u_1} & \frac{\partial f_2}{u_2} & \dots & \frac{\partial f_2}{u_N} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_N}{u_1} & \frac{\partial f_N}{u_2} & \dots & \frac{\partial f_N}{u_N} \end{vmatrix} \tag{1.3.15}$$

se $\boldsymbol{v} + \boldsymbol{e}$ é soluzione del sistema allora ne deriva che

$$0 = \boldsymbol{A}(\boldsymbol{v}) + \boldsymbol{J}(\boldsymbol{v})\boldsymbol{e} + o(\boldsymbol{e}^2)$$

$$\boldsymbol{e} = -[\boldsymbol{J}(\boldsymbol{v})]^{-1}\boldsymbol{A}(\boldsymbol{v})$$

ottenendo l'iterazione:

$$\begin{aligned} \boldsymbol{v} &\leftarrow \boldsymbol{v} + \boldsymbol{e} \\ \boldsymbol{v} &\leftarrow \boldsymbol{v} - [\boldsymbol{J}(\boldsymbol{v})]^{-1}\boldsymbol{A}(\boldsymbol{v}) \end{aligned} \tag{1.3.16}$$

dove per ogni componente j del vettore $\boldsymbol{v}$ viene eseguito il passo di iterazione 1.3.14 Avvalendosi del risultato appena ottenuto é possibile riscrivere l'equazione del residuo nel caso non lineare, la 1.3.13, nel seguente modo:

$$\begin{aligned} \boldsymbol{A}(\boldsymbol{u}) - \boldsymbol{A}(\boldsymbol{v}) &= \boldsymbol{f} - \boldsymbol{A}(\boldsymbol{v}) = \boldsymbol{r} \\ \boldsymbol{A}\boldsymbol{v} + \boldsymbol{J}(\boldsymbol{v})\boldsymbol{e} - \boldsymbol{A}\boldsymbol{v} &= \boldsymbol{r} \\ \boldsymbol{J}(\boldsymbol{v})\boldsymbol{e} &= \boldsymbol{r} \end{aligned} \tag{1.3.17}$$

Di conseguenza si può applicare un metodo iterativo, come per esempio il Gauss-Seidel, in cui il passo di rilassamento é $\boldsymbol{v} \leftarrow \boldsymbol{v} + \boldsymbol{e}$, ossia

$$\boldsymbol{v} \leftarrow \boldsymbol{v} + [\boldsymbol{J}(\boldsymbol{v})]^{-1}\boldsymbol{r} \tag{1.3.18}$$

Oltre al metodo iterativo appena descritto é anche possibile sviluppare un algoritmo in tutto e per tutto simile al Full MultiGrid V-Cycle, con le dovute modifiche del caso. Si riprenda in considerazione l'equazione dei residui, relativa alla griglia Ω^{2h}

$$\boldsymbol{A^{2h}}(\boldsymbol{u^{2h}}) - \boldsymbol{A^{2h}}(\boldsymbol{v^{2h}}) = \boldsymbol{r^{2h}}$$

utilizzando le seguenti relazioni:

$$\boldsymbol{u^{2h}} = \boldsymbol{v^{2h}} + \boldsymbol{e^{2h}}$$

$$\boldsymbol{v^{2h}} = \boldsymbol{I_h^{2h}}\boldsymbol{v^h}$$

$$\boldsymbol{r^{2h}} = \boldsymbol{I_h^{2h}}\boldsymbol{r^h} = \boldsymbol{I_h^{2h}}(\boldsymbol{f^h} - \boldsymbol{A^h}\boldsymbol{v^h})$$

si riscrive la 1.3.4 nella forma:

$$\boldsymbol{A^{2h}(I_h^{2h}v^h + e^{2h}) = A^{2h}(I_h^{2h}v^h) + I_h^{2h}(f^h - A^h v^h)} \tag{1.3.19}$$

dove l'unica incognita é $\boldsymbol{e^{2h}}$.

É quindi possibile sviluppare un algoritmo identico nella forma al Full MultiGrid V-Cycle del caso lineare, dove però l'equazione del residuo utilizzata nei livelli inferiori é definita dalla 1.3.19; tale metodo viene definito "**Full Approximation Scheme**".

Condizioni al contorno di Neumann

Durante la descrizione dei metodi MultiGrid é stato trattato sempre il caso in cui le condizioni al contorno fornivano il valore della soluzione lungo i bordi del dominio, ossia condizioni di Dirichlet. Le condizioni di Neumann forniscono invece il valore della derivata della soluzione lungo il contorno, ossia sono del tipo:

$$u_x = f(x), \qquad \forall x \in \partial\Omega \tag{1.3.20}$$

Tali condizioni introducono delle modifiche nelle griglie su cui lavorare.

Si consideri il seguente problema

$$\begin{aligned} -u''(x) &= f(x), \qquad 0 < x < 1 \\ u'(0) &= u'(1) = 0 \end{aligned} \tag{1.3.21}$$

Per poter risolvere tale equazioni con condizioni al contorno di Neumann é necessario introdurre due punti ulteriori oltre il bordo del dominio, x_{-1} e x_{N+1}, così come mostrato in Figura 1.16. Tali punti vengono definiti ""**Ghost Points**"". Si sottolinea che l'intervallo fra un punto e l'altro rimane inviariato, ossia pari a $h = \frac{1}{N}$. Viene cosí

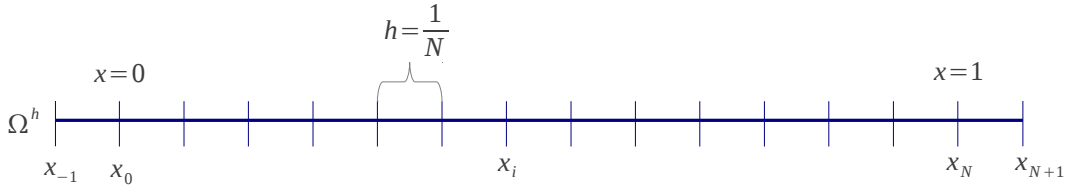

Figura 1.16: Ghost Points 1D.

definito il seguente sistema di equazioni:

$$-\frac{u_{j+1} - 2u_j + u_{j+1}}{h^2} = f_j, \qquad 0 \leq j \leq N+1 \tag{1.3.22}$$

$$\frac{u_1 - u_{-1}}{2h} = 0 \tag{1.3.23}$$

$$\frac{u_{N+2} - u_N}{2h} = 0 \tag{1.3.24}$$

Dove per descrivere il valore della derivata della funzione $u(x)$ nei ""ghost points"" x_{-1} e x_{N+1} é stata utilizzata la formula delle **differenze centrali di Taylor**.

Capitolo 2

Ambiente di Sviluppo

2.1 CUDA

Il presente capitolo é dedicato alla descrizione dell'ambiente di sviluppo utilizzato per implementare i metodi risolutivi di PDE, precedentemente descritti, e le motivazioni alla base di tale scelta.

Nel capitolo precedente si é visto come il miglior metodo per la risoluzione di una PDE é il Full MultiGrid V-Cycle; tale metodo consta di diverse componenti, ognuna delle quali operanti su griglie di una, due o tre dimensioni. Risulta evidente che l'aggiornamento di ogni singolo elemento di tali griglie ne richiede la scansione intera, ed é quindi il collo di bottiglia dell'algoritmo. Si ha quindi la necessità di parallelizzare le operazioni sugli elementi delle griglie, nel tentativo di ridurre i tempi di esecuzioni, ed é per questo scopo che sono state utilizzate le librerie CUDA.

2.1.1 Architettura

CUDA (**C**ompute **U**nified **D**evice **A**rchitecture) e costituisce un'architettura di calcolo parallelo incentrata sul paradigma **GPGPU**, **G**eneral **P**urpose computing on **G**raphics **P**rocessing **U**nits, ossia l'utilizzo del processore grafico per scopi diversi

dalla gestione di grafica tridimensionale cui solitamente dedicato. Dal grafico in Figura 2.1 é possibile notare un confronto in termini di GFLOPS, *Giga Floating Point Operations Per Second*, fra Cpu e Gpu. Il distacco prestazionale mostrato in figura

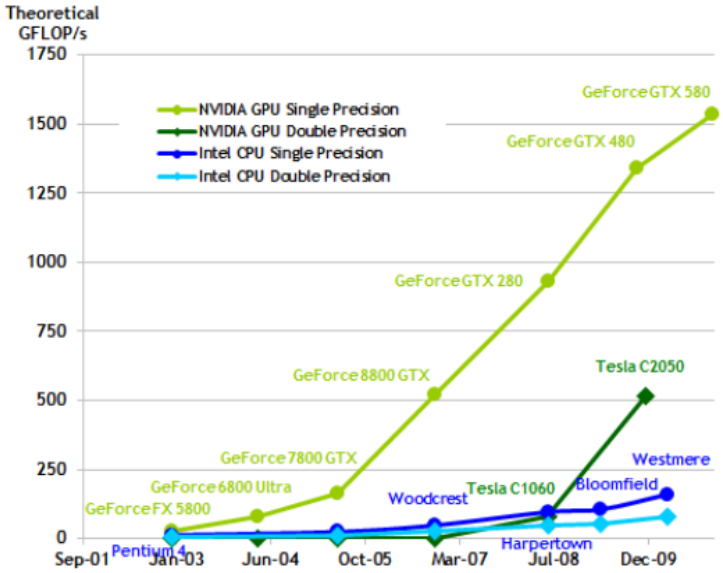

Figura 2.1: Gpu e Cpu al confronto: GFLOPS.

é motivato dai differenti domini di applicazione che ne ha determinato le relative architetture; il rendering tridimensionale coinvolge l'esecuzione paralella di migliaia di operazioni, estremamente ripetitive, la Gpu si é quindi evoluta come un'architettura fortemente parallela in cui la maggior parte dei transistor sono dedicati al processamento di dati, anzichè alla gestione della cache e del controllo di flusso; nella Cpu la situazione é diametralmente opposta, in quanto poichè utilizzata nei domini applicativi più disparati, grande attenzione deve essere dedicata al caching dei dati e alla logica di controllo. Una rappresentazione grafica di quanto appena detta la si può ritrovare in Figura 2.2 Le Gpu con supporto CUDA sono costituite da un determinato numero di unità definite **Multiprocessori**, in numero variabile a seconda del modello; ogni multiprocessore é costituito a sua volta da sottounità, definite "Core", ognuna delle

Figura 2.2: Gpu e Cpu al confronto: Schema architetturale.

quali può lanciare un certo numero di thread in parallelo.

2.1.2 Struttura Applicazione Cuda

Un'applicazione Cuda non é interamente costruita per girare su Gpu; essa si compone di una parte seriale eseguita sulla Cpu (Host) ed una parte parallela eseguita sulla Gpu (Device) definita **Kernel**. Un Kernel é rappresentabile come una **griglia** divisa in **blocchi**; un blocco viene assegnato unicamente ad un multiprocessore, sebbene ad un multiprocessore possano esser assegnati più di un blocco. A questo livello di astrazione si parla di parallelismo a grana grossa, dove con tale termine si identifica l'esecuzione parallela di processi con limitata comunicazione interprocesso. Ogni blocco é divisibile in un certo numero di **thread**, ognuno dei quali appartiene ad un solo blocco; a questo livello di astrazione si parla di parallelismo a grana fine, dove, diversamente dal caso precedente, la comunicazione fra i processi operanti in parallelo, i thread, non ha restrizione alcuna. In Figura 2.3 é possibile notare una rappresentazione grafica di quanto appena descritto. Poichè il codice di un'applicazione Cuda é costituito da porzioni eseguibile solamente da un'unità fra Host e Device, vi é la necessità di fornire uno strumento di comunicazione tra gli stessi, ed é quindi stata introdotta la "Global Memory", memoria lenta situata sulla Ram, accedibile da Device e Host. Quest'ultimo non può però accedere in alcun modo alla memoria privata

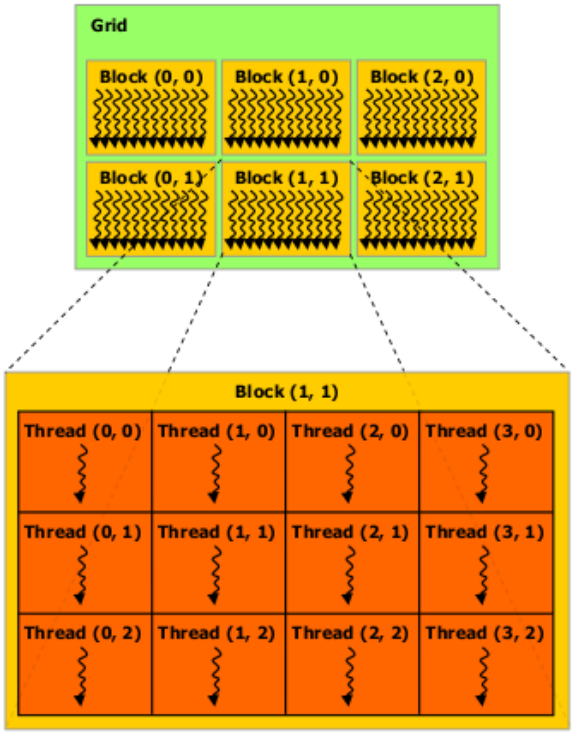

Figura 2.3: Kernel: Griglia di Blocchi di Thread.

di ogni multiprocessore, la ""Shared Memory"", veloce memoria situata direttamente sul chip, riservata ai thread di blocchi dello stesso multiprocessore.
Come già è stato detto, un Kernel é considerato come un griglia divisa in blocchi in cui ciascun blocco é ulteriormente suddivisibile in thread. In termini spaziali, la dimensione massima del Kernel é limitata a due, mentre per i blocchi il limite é di tre dimensioni; la situazione é diversa relativamente alle Gpu dedicate al *High-performance computing* (*HPC*), in cui un Kernel può essere strutturato come una griglia tridimensionale, portando a grandi benefici in termini di gestione della memoria, come verrà successivamente descritto.
Si assuma ora di dover risolvere il seguente problema: *aggiornarne gli elementi di un dato vettore d-dimensionale*. Seguono due programmi implementati su Host (Cpu) e Device (Gpu) che risolvono tale problema:

- Cpu: Esegui una serie di cicli annidati, in cui ad ogni iterazione viene aggiornato l'elemento identificato dagli indici correnti. Le iterazioni sono eseguite in maniera **sequenziale**.

- Gpu: Dato il vettore iniziale, dividilo in blocchi di elementi. Ogni elemento viene assegnato univocamente ad un thread. Al lancio del Kernel ogni thread esegue in **paralello** l'operazione di aggiornamento dell'operazione.

É quindi possibile dare le seguenti conclusioni: il programma eseguito su Cpu viene eseguito con tempo d'esecuzione $O(N^d)$, dipendente dalla dimensione del vettore in numero di elementi; il programma eseguito su Gpu impiega un tempo d'esecuzione pari invece a $O(1)$, ossia costante. É da sottolineare che in realtà vi é un limite al numero di thread eseguibili in parallelo, quindi la stima effettuata costituisce il caso ottimistico in cui tale limite non viene superato. Questo semplice esempio mette in

luce le grandi potenzialità dell'architettura Cuda.

2.1.3 Organizzazione dei Dati: Linearizzazione

Nell'esempio fornito alla fine del paragrafo precedente si é parlato di assegnazione di elementi di un vettore ai thread dei blocchi costituenti il Kernel. Per evitare che un blocco venga aggiornato più volte, con conseguente inconsistenza e imprevedibilità dei risultati, é necessario che la relazione fra gli elementi del vettore e i thread sia di tipo 1 a 1: ogni elemento viene assegnato unicamente ad un thread. Allo scopo vengono fornite delle variabili relative alla griglia, ai blocchi e ai thread:

- *gridDim*: dimensione griglia in numero di blocchi, fino a tre dimensioni nell'ultima generazione di Gpu
- *blockDim*: dimensioni blocco in numero di thread su un asse, massimo tre dimensioni
- *blockIdx*: indice blocco corrente relativamente ad un asse
- *threadIdx*: indice thread corrente relativamente ad un asse e al blocco a cui appartiene

L'utilizzo congiunto delle variabili sopra descritte porta all'assegnazione di un indice univoco ad ogni thread, requisito fondamentale per mantenere la consistenza dei dati da modificare.
La struttura di un'applicazione Cuda richiede di "linearizzare" i vettori a più dimensioni contenenti i dati da elaborare, come mostrato in Figura 2.4 relativamete al caso bidimensionale. Come si può notare, le righe della matrice fornita in ingresso vengono

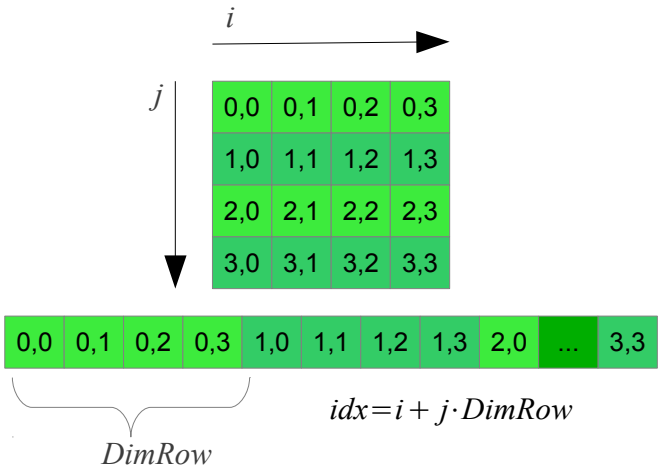

Figura 2.4: Linearizzazione Matrice.

affiancate andando a costituire un vettore monodimensionale. Per ottenere l'indice idx di un elemento a partire dagli indici i e j é necessario innanzitutto calcolare lo spiazzamento all'interno della riga, ossia i, quindi sommarvi il numero di elementi nelle righe che precedono la corrente, $y \cdot DimRow$; come mostrato in figura la formula risultante é

$$idx = i + j \cdot DimRow \tag{2.1.1}$$

dove $DimRow$ é la dimensione di ogni riga in numero di elementi (si suppone che sia uguale per ogni riga).

Relativamente all'esempio precedente é possibile elaborare gli elementi della matrice da un Kernel strutturato come una griglia di 2x2 blocchi, ognuno composto da 2x2 thread, come in Figura 2.5 La formula utilizzata per il calcolo é equivalente alla 2.1.1, dove però gli indici i e j sono stati sostituiti rispettivamente da ix e iy, dei quali é necessario effettuare il calcolo. L'indice ix identifica la posizione assoluta (e non relativa al blocco) lungo le ascisse di un thread; il calcolo avviene considerando lo spiazzamento lungo le ascisse all'interno del blocco, *threadIdx.x*, sommato al numero totale di thread precedenti il thread in questione (sempre lungo l'asse x), ossia

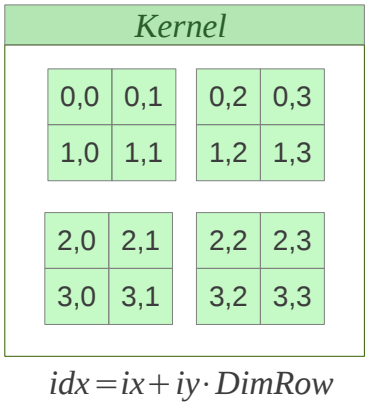

Figura 2.5: Linearizzazione Matrice - Kernel.

$blockIdx.x \cdot blockDim.x$, ottenendo quindi

$$ix = threadIdx.x + blockIdx.x \cdot blockDim.x \tag{2.1.2}$$

Il calcolo dell'indice iy avviene nello stesso modo, considerando gli spiazzamenti relativi alle ordinate.

$$iy = threadIdx.y + blockIdx.y \cdot blockDim.y \tag{2.1.3}$$

Il risultato finale é quindi

$$idx = ix + iy \cdot DimRow \tag{2.1.4}$$

In caso in cui il numero degli elementi della matrice sia tale per cui teoricamente non sarebbe possibile dividere il Kernel in blocchi aventi ugual numero di thread, è necessario effettuare delle modifiche; poichè non é lecito assegnare un numero diverso di thread ad ogni blocco, si presenta la situazione in cui ad uno o più blocchi vengono assegnati elementi in realtà non presenti nel vettore in ingresso, identificati ossia da indici ""out of bound"", portando ad accessi non autorizzati in altre aree di memoria; una soluzione a tale problema corrisponde ad imporre una condizione secondo la quale l'indice calcolato all'inizio del Kernel, deve essere inferiore rispetto al numero totale

di elementi nel vettore.

Nel caso tridimensionale le linearizzazione é un'operazione più complessa, in quanto entra in gioco un terzo indice, k. In Figura 2.6 é mostrata una schematizzazione dell'operazione in un dominio tridimensionale. Come nel caso bidimensionale é pos-

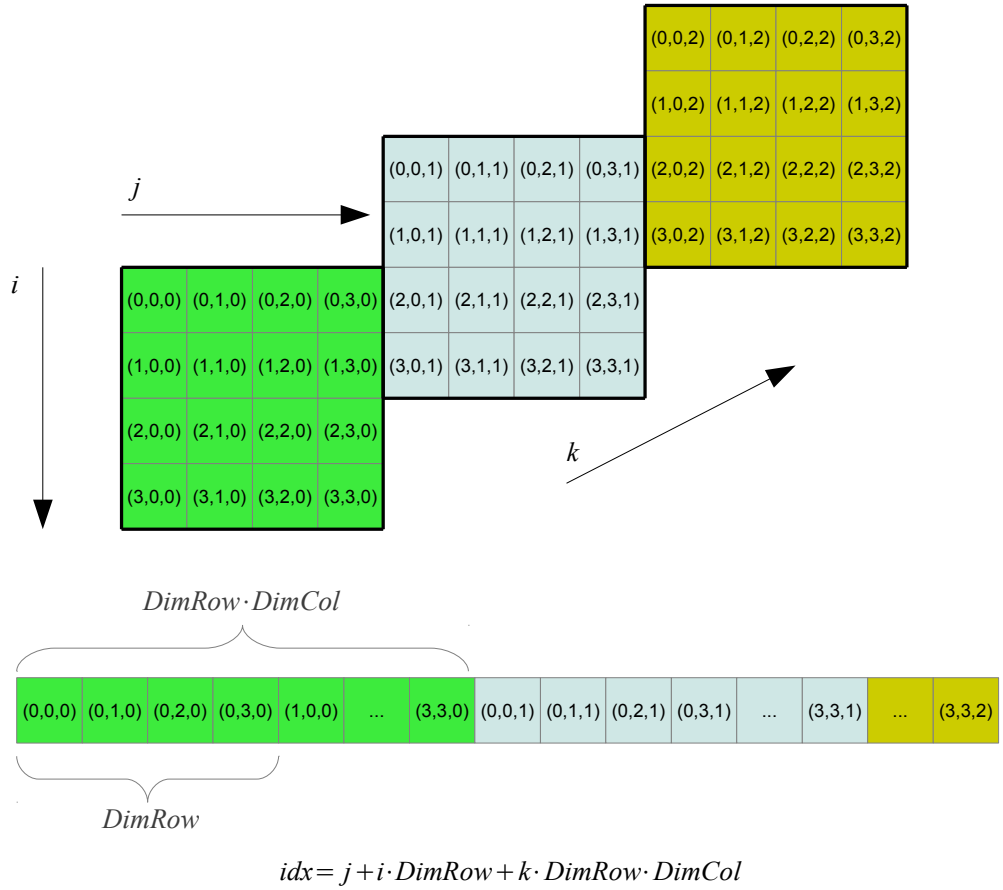

Figura 2.6: Linearizzazione 3D.

sibile lanciare un Kernel per elaborare i singoli elementi del vettore 3D linearizzato, sostituendo come nel caso precedente gli indici i, j, k rispettivamente con ix, iy, ik; nel caso in cui si sia forniti di una Gpu dedicate al *HPC*, supportante Kernel divisi in una griglia tridimensionale, il calcolo del terzo indice avviene nel modo usuale

$$iz = threadIdx.z + blockIdx.z \cdot blockDim.z \tag{2.1.5}$$

In tali condizioni non si é soggetti ad alcuna restrizione (almeno finchè si resta in un dominio a tre dimensioni).

In caso di Gpu non dedite al *HPC*, di cui un normale utente é usualmente dotato, la situazione può complicarsi notevolmente a seconda delle necessità. Poichè il numero massimo ammissibile di thread lungo l'asse z é pari a 64, nel caso in cui non vi é la necessità di avere più di tale numero di elementi lungo tale asse, il calcolo dell'indice iz avviene nel seguente semplificato modo:

$$iz = threadIdx.z \tag{2.1.6}$$

Usualmente si ha però a che fare con griglie con un numero di elementi lungo l'asse z maggiore del limite di 64 thread, di conseguenza é necessario trovare una soluzione alternativa.

Poichè in questo lavoro tale problema si é venuto a presentare é stata pensata la seguente soluzione: preso in considerazione un vettore tridimensionale, lo si pensi come organizzato in matrici su più livelli lungo l'asse z; l'idea é di accorpare tutti gli elementi della stessa matrice, lungo l'asse x dei blocchi della griglia (il Kernel); ogni riga del Kernel (divisa in thread appartenenti a blocchi diversi) costituisce quindi una matrice del vettore tridimensionale. Per poter comprendere tale procedura, si faccia riferimento alla Figura 2.7.

Nell'esempio che ci si presta a descrivere sono stati utilizzati blocchi delle dimensioni di $4x4$ thread. La dimensione del Kernel in termini di blocchi di thread é così calcolata:

$$\begin{aligned} numBlocksX &= \lceil \frac{sizeX * sizeY}{threadsPerBlock.x} \rceil \\ numBlocksY &= \lceil \frac{sizeZ}{threadsPerBlock.y} \rceil \end{aligned} \tag{2.1.7}$$

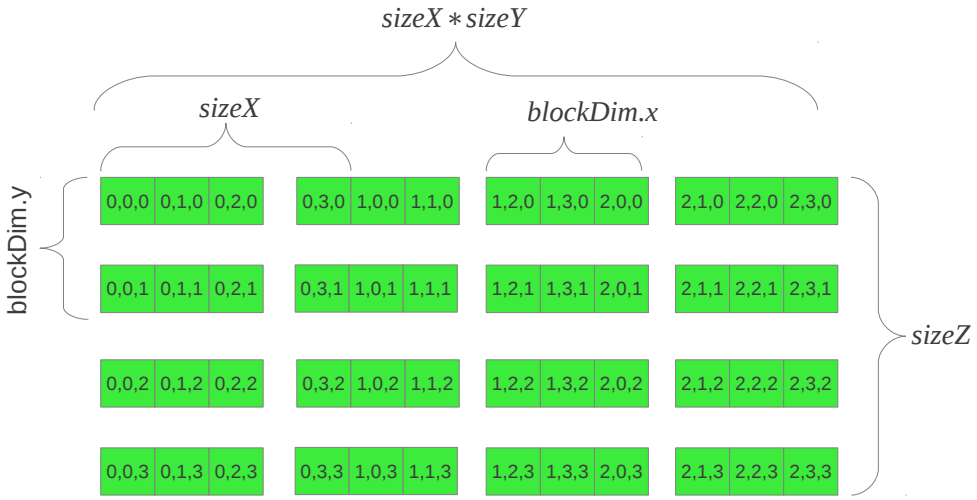

Figura 2.7: Organizzazione Kernel: caso 3D.

dove é stata utilizzata l'operazione di arrotondamento ad intero superiore, in quanto per far sì che ogni blocco possa essere gestito da un thread, devono essere presenti **almeno** tanti thread quanti gli elementi da modificare. Una volta all'interno del Kernel é quindi necessario calcolare gli indici ix, iy e iz di ogni singolo elemento.

L'indice iy, ossia l'indice della riga a cui appartiene l'elemento nel vettore originale, é calcolato considerando la posizione assoluta lungo l'asse x nel Kernel, data da $threadIdx.x + blockIdx.x * blockDim.x$, e divendola per le dimensioni di una singola riga (effettuando una divisione fra interi), ossia per $sizeX$, ottenendo

$$iy = \frac{threadIdx.x + blockIdx.x * blockDim.x}{sizeX} \tag{2.1.8}$$

Relativamente all'elemento $(1, 2, 0)$ ad esempio, il calcolo dell'indice iy si ottiene utilizzando la 2.1.8:

$$iy_{(1,2,0)} = \frac{threadIdx.x + blockIdx.x * blockDim.x}{sizeX} = \frac{0 + 2 * 3}{4} = 1$$

Il calcolo dell'indice ix, indice della colonna a cui appartiene l'elemento nel vettore originale, avviene sottraendo all'indice della posizione assoluta lungo l'asse x nel Ker-

nel, ossia nuovamente $threadIdx.x + blockIdx.x * blockDim.x$, il numero di righe che precedono tale elemento nel vettore originale, $iy * sizeX$, ottenendo:

$$ix = threadIdx.x + blockIdx.x * blockDim.x - iy * sizeX \tag{2.1.9}$$

Riconsiderando nuovamente l'elemento $(1, 2, 0)$, l'indice ix pari a 2 viene calcolato nel seguente modo:

$$ix_{(1,2,0)} = threadIdx.x + blockIdx.x * blockDim.x - iy * sizeX = 0 + 2 * 3 - 1 * 4 = 2$$

Resta infine da calcolare l'indice ik; in maniera molto più semplice rispetto agli altri indici, la relativa espressione é la seguente:

$$iz = blockIdx.y * blockDim.y + threadIdx.y \tag{2.1.10}$$

ossia semplicemente si calcola la posizione assoluta nel Kernel lungo l'asse y. Prendendo ad esempio in considerazione l'elemento $(0, 3, 2)$ il calcolo del relativo indice iz si effettua come a seguire:

$$iz_{(0,3,2)} = blockIdx.y * blockDim.y + threadIdx.y = 1 * 2 + 0 = 2;$$

La procedura utilizzata rappresenta una delle tante soluzioni al problema di gestire un vettore tridimensionale avendo a disposizione unicamente una griglia a due dimensioni, quale é il Kernel.

2.1.4 Ottimizzazioni memoria: coalescenza

Per poter sfruttare al meglio le potenzialità delle librerie Cuda, é necessario utilizzare alcune ottimizzazione relative all'utilizzo della memoria. Si introduce qui il concetto di *coalescenza*; rendere coalescenti gli accessi alla memoria, significa riuscire ad accorparne il maggiore numero possibile in un'unica transazione del controller di memoria

(batching). L'unità da tenere in considerazione in questo caso é *l'half warp* ossia il gruppo di 16 thread eseguiti in parallelo su di un multiprocessore. Affinchè gli accessi siano coalescenti si devono verificare le seguenti condizioni:

- Lettura contigua di un'area di memoria grande 64/128/256 byte: ogni thread dell'half warp legge una word/double-word /quad-word (16 thread x 4/8/16 Byte)
- L'indirizzo iniziale di una regione deve essere multiplo della grandezza della regione
- Il k-esimo thread di un half-warp deve accedere al k-esimo elemento di un blocco, sebbene alcuni thread possano non partecipare alla lettura

Per poter ottenere la condizione di coalescenza, le librerie Cuda forniscono una funzione utile allo scopo: ""cudaMallocPitch"". Si consideri il caso di una matrice in cui ogni riga é costituita da 15 elementi; poichè ad ogni ciclo di clock vengono letti 16 elementi (nel caso di half warp) si presenta la seguente situazione:

- Una transazione da 16x4=64 *Byte* per leggere la prima riga
- Una transazione da 64 *Byte* per leggere il primo elemento della seconda riga
- Una transazione da 64 *Byte* per leggere i restanti elementi della seconda riga

Per evitare questo disallineamento e portarsi nella situazione in cui ad ogni transazione corrisponde la lettura di tutti e soli gli elementi di una riga, si utilizza la già citata *cudaMallocPitch*, la quale restituisce un valore definito **Pitch**, che corrisponde alle dimensioni in **Byte** che una riga dovrebbe avere per avere accessi allineati. Nel caso dell'esempio precedente il pitch ha valore di 64 *Byte*.

L'utilizzo della funzione *cudaMallocPitch* porta a cambiamenti nel modo in cui accedere agli elementi di una matrice linearizzata; la 2.1.4 viene qui sostituita dalla 2.1.11:

$$idx = ix + iy \cdot matPitch \tag{2.1.11}$$

dove $matPitch = pitch/sizeof(float)$ nel caso di matrice costituita da elementi di tipo di dato *float*; il significato di questa variabile é quindi la dimensione che una riga dovrebbe avere, per allineare gli accessi, misurata in numero di elementi e non in Byte, come nel caso del Pitch.

2.1.5 MultiGrid in paralello

Nel capitolo precedente si é visto come il *Full MultiGrid V-Cycle* sia costituito da varie componenenti, qui riproposte:

- Gestione delle griglie
- Gestione operatori di trasferimento: Restrizione, Interpolazione
- Algoritmo di Rilassamento: Red-Black Gauss-Seidel
- Calcolo del residuo
- Correzione soluzione approssimata

Si noti come in ogni punto uno o più cicli annidati (a seconda delle dimensioni del problema) vengono eseguiti; ad ogni iterazione, in base al contesto, viene aggiornato l'elemento identificato dagli indici correnti, sulla griglia da modificare. Dalla struttura appena descritta é quindi evidente come le Cuda entrino in gioco trasformando le iterazioni sequenziali fra i vari elementi in operazioni parallele in cui ogni thread aggiorna l'elemento della griglia assegnatogli.

A seconda dell'operazione effettuata, la modifiche degli elementi di una griglia deve essere eseguita sotto determinate condizioni:

- l'indice calcolato non deve portare ad un accesso illegale di memoria (nel caso di thread in numero maggiore del numero degli elementi da modificare)
- gli elementi del vettore corrispondenti ai punti sul bordo della griglia, nel caso dell'algoritmo di rilassamento, non devono essere modificati
- in caso siano presenti condizioni iniziali che specifichino il valore della soluzione in punti interni alla griglia, gli elementi del vettore corrispondenti non devono essere modificati

Capitolo 3

Applicazioni

In questo capitolo verranno presentate le applicazioni dei metodi MultiGrid con il supporto delle librerie Cuda per la risoluzione di diverse PDE, la maggior parte delle quali di grande interesse nell'area della Teoria del Controllo. Per avere una visione complessiva, sono stati affrontati problemi ad una, due e tre dimensioni, in particolare:

- ODE - Ordinary differential equation (1D)
- Equazione di Lyapunov (2D)
- Controllo ottimo a minimo tempo (2D)
- Poisson (3D)

In questa sezione verranno descritti i metodi risolutivi nei casi sopracitati, senza però entrare nel merito delle prestazioni degli algoritmi risolutori utilizzati; il capitolo successivo sarà dedicato a tale argomento.

3.1 Equazione differenziale ordinaria 1D

Relativamente al caso monodimensionale, é stata risolta una cosiddetta "" *ODE* "", acronimo stante per **O**rdinary **D**ifferential **Eq**uation, ossia equazione differenziale

ordinaria; in tale tipologia di equazioni figurano unicamente funzioni ad una variabile, e quindi non sono presenti derivate parziali. In particolare é stato risolto il seguente problema di Cauchy:

$$u' - \frac{u(x)}{e^x + 1} = e^x$$

$$\begin{aligned} u(x_0) &= \frac{e^{x_0} + x_0 - 3}{1 + e^{-x_0}} \\ u(x_N) &= \frac{e^{x_N} + x_N - 3}{1 + e^{-x_N}} \end{aligned} \tag{3.1.1}$$

Per semplicità sui contorni del dominio su cui effettuare il calcolo é stata forzata la soluzione reale, nota e pari a

$$u(x) = \frac{e^x + x - 3}{1 + e^{-x}} \tag{3.1.2}$$

La suddetta equazione é stata utilizzata unicamente con lo scopo di fornire un'ulteriore piattaforma di confronto prestazionale fra Cpu e Gpu, nel caso ad una dimensione; non é legata in particolare ad alcun fenomeno fisico reale e figura come esempio di equazione differenziale lineare di primo ordine in dispense di Analisi 2 [4].

Discretizzando l'equazione, nei termini descritti nel capitolo 2, si ottiene il seguente passo di rilassamento:

$$v_j = \frac{v_{j+1}(e^{x_j} + 1) - e^{x_j} h_x (e^{x_j} + 1)}{e^{x_j} + 1 + h_x} \tag{3.1.3}$$

Poichè il denominatore non é mai nullo (e^{x_j} e h_x sono sempre maggiori di zero), e poichè é stato forzato il valore della soluzione sul contorno, é possibile calcolare la 3.1.1 su qualsiasi dominio.

3.2 Equazione di Lyapunov

Relativamente al caso a due dimensioni, uno dei problemi affrontati riguarda la risoluzione della **equazione di Lyapunov**; prima di poter descrivere la PDE risultante,

é necessario fornire alcune spiegazioni preliminari per dare un contesto al problema risolto.

Per descrivere l'evoluzione nel tempo di sistemi fisici, economici, sociali ecc., é possibile ricorrere a dei modelli matematici che ne caratterizzano il comportamento sotto forma di equazioni differenziali in funzione delle interazione che questi sistemi hanno con l'ambiente circostante. Matematicamente, questo può essere rappresentato dalla seguente equazione differenziale

$$\dot{x}(t) = f(x(t), u(t)), \qquad x(t) \in \mathbb{R}^n, \quad u(t) \in \mathbb{R}^p, t \in \mathbb{R} \tag{3.2.1}$$

dove $u(t)$ rappresenta l'ingresso, esogeno al sistema, x(t) lo stato, che ne descrive unicamente la configurazione, e che può essere definito come dalla la 3.2.2

$$x(t) = \varphi(t, x_0, u(\cdot)) \tag{3.2.2}$$

ove x_0 Ã¨ la condizione iniziale, e ϕ é quindi la soluzione del problema di Cauchy corrispondente. Si introduce ora una nuova definizione: uno stato x_e si dice **stato d'equilibro** se la vale la seguente relazione

$$\varphi(t, x_e, u_e) = x_e \qquad \forall t \geq 0, \quad t \in \mathbb{T} \tag{3.2.3}$$

per un dato u_e costante, che implica

$$f(x_e, u_e) = 0 \tag{3.2.4}$$

Un stato di equilibrio può godere delle seguenti proprietà:

- **Stabilità**: é sempre possibile trovare delle condizioni iniziali diverse da x_e tali che l'evoluzione dello stato Ã¨ contenuto in un intorno di dimensioni fissate a piacere del punto x_e

- **Attrattività**: lo stato del sistema converge asintoticamente a x_e
- **Stabilità Asintotica**: x_e gode delle proprietà di stabilità e attrattività

Poichè le precedenti proprietà sono fondamentali per l'analisi di un sistema dinamico, diversi metodi sono stati sviluppati per poter determinate le proprietà di un punto di equilibrio. Il **Metodo Diretto di Lyapunov**; permette di dedurre tali proprietà senza dover passare per il calcolo della risposta dello stato in forma esplicita (ovvero la soluzione in forma chiusa della 3.2.1), non sempre possibile; ciò é reso possibile grazie all'utilizzo di una particolare funzione, $V(x)$, definita ""*funzione candidata di Lyapunov*"", il cui studio fornisce le informazioni ricercate sulla natura del punto d'equilibrio.
Prima di procedere nella descrizione del metodo é necessario introdurre altre nozioni. Sia

$$\mathbb{B}_\rho(w) = v \in \mathbb{V} : ||v - w|| < \rho \tag{3.2.5}$$

l'insieme dei vettori v la cui differenza rispetto al vettore w ha norma minore di ρ, definito come ""*intorno aperto di w di raggio ρ*"".
Dato un $\bar{x} \in \mathbb{R}^n$, una funzione $V(\cdot)\, \mathbb{R}^n \to \mathbb{R}$ si dice

- **semidefinita positiva in** $\bar{x}$, $V(\cdot) \succeq 0$ in $\bar{x}$ se

$$V(\cdot) \succeq 0 \iff V(\bar{x}) = 0 \quad \text{e} \quad \exists \rho > 0 : V(x) \geq 0, \quad \forall x \in \mathbb{B}_\rho(\bar{x}) \tag{3.2.6}$$

- **definita positiva in** $\bar{x}$, $V(\cdot) \succ 0$ in $\bar{x}$ se

$$V(\cdot) \succ 0 \iff V(\bar{x}) = 0 \quad \text{e} \quad \exists \rho > 0 : V(x) > 0, \quad \forall x \in \mathbb{B}_\rho(\bar{x}), \quad x \neq \bar{x} \tag{3.2.7}$$

- **globalmente semidefinita positiva in** $\bar{x}$, se

$$V(\bar{x}) = 0 \quad \text{e} \quad V(x) \geq 0, \forall x \in \mathbb{R}^n \tag{3.2.8}$$

- **globalmente definita positiva in** $\bar{x}$, se

$$V(\bar{x}) = 0 \quad \text{e} \quad V(x) > 0, \forall x \in \mathbb{R}^n, \quad x \neq \bar{x} \tag{3.2.9}$$

- **radialmente illimitata** se

$$\lim_{||x|| \to +\infty} V(x) = +\infty \tag{3.2.10}$$

Nel contesto del metodo diretto di Lyapunov generalmente vengono utilizzate funzioni in **forma quadratica in x**, ovvero nella forma:

$$V(x) = x'Px, \qquad P \in \mathbb{R}^{nxn} : P = P' \tag{3.2.11}$$

dove la condizione su P impone che essia sia una matrice **simmetrica**.
Data quindi una funzione $V(x) = (x - \bar{x})'P(x - \bar{x})$ per lo studio delle proprietÃ del punto $\overline{x}$, valgono le seguenti proprietà:

1. $V(\cdot)$ é globalmente semidefinita positiva in $\bar{x}$ se e solo se $P \geq 0$
2. $V(\cdot)$ é globalmente definita positiva in $\bar{x}$ e radialmente illimitata se e solo se $P > 0$

Per poter verificare le proprietà di P vi é la necessità di studiare la struttura della matrice e in particolare i relativi minori principali; i minori principali sono i determinanti delle sottomatrici ottenute estraendo da P le sottomatrici di dimensione jXj i cui elementi appartengono alle j righe e alle j colonne con stesso indice delle righe prese

in considerazione, con $j = 1, ..., n$. Nel caso di $n = 3$ per $j = 2$ i minori principali sono:

$$det\begin{vmatrix} p11 & p12 \\ p21 & p22 \end{vmatrix}, \quad det\begin{vmatrix} p11 & p13 \\ p31 & p33 \end{vmatrix}, \quad det\begin{vmatrix} p22 & p23 \\ p32 & p33 \end{vmatrix} \tag{3.2.12}$$

Si dicono invece minori principali annidati, i determinanti delle n sottomatrici ottenute prendendo gli elementi delle prime j righe e delle j colone corrispondenti, , con $j = 1, ..., n$.

Valgono le seguenti proprietà per una matrice P simmetrica:

- $P \geq 0$ se e solo se tutti i minori principali d P sono non negativi
- $P > 0$ se e solo se tutti i minori principali annidati di P sono positivi.

Per funzioni $V(\cdot)$ non quadratiche devono essere fatte osservazioni diverse da quanto descritto precedentemente: $V(\cdot) \succ 0$ in $\bar{x}$ se $V(\bar{x}) = 0$ e in $\bar{x}$ vi é un minimo locale stretto, in formule (localmente in $\overline{x}$):

$$(V(\bar{x}) = 0, \quad \nabla V(\bar{x}) = 0, \quad \nabla^2 V(\bar{x}) > 0) \rightarrow V(\cdot) = 0 \text{ in } \bar{x} \tag{3.2.13}$$

dove

$$\nabla V(\cdot) = \begin{vmatrix} \frac{\partial V(\cdot)}{\partial x_1} & \frac{\partial V(\cdot)}{\partial x_2} & ... & \frac{\partial V(\cdot)}{\partial x_n} \end{vmatrix} \tag{3.2.14}$$

$$\nabla^2 V(\cdot) = \begin{vmatrix} \frac{\partial V^2(\cdot)}{\partial x_1^2} & \frac{\partial V^2(\cdot)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial V^2(\cdot)}{\partial x_1 \partial x_n} \\ \frac{\partial V^2(\cdot)}{\partial x_2 \partial x_1} & \frac{\partial V^2(\cdot)}{\partial x_2^2} & \cdots & \frac{\partial V^2(\cdot)}{\partial x_2 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial V^2(\cdot)}{\partial x_n \partial x_1} & \frac{\partial V^2(\cdot)}{\partial x_n \partial x_2} & \cdots & \frac{\partial V^2(\cdot)}{\partial x_n^2} \end{vmatrix} \tag{3.2.15}$$

Come già detto, il Metodo Diretto di Lyapunov sfrutta una funzione $V(x)$ per determinare le proprietà di stabilità di un punto d'equilibrio x_e; in particolare si considera la variazione (derivata) di $V(x)$ lungo il moto del sistema, ovvero il prodotto scalare:

$$\dot{V}(x) = \nabla V(x) \cdot f(x) \tag{3.2.16}$$

Dopo aver dato le dovute definizioni é possibile enunciare il **Teorema di Lyapunov**: *Dato uno stato di equilibrio x_e, sia $V(\cdot) \succ 0$ in x_e, se:*

1. $\dot{V}(\cdot) \preceq 0$ in $x_e allora x_e$ é stabile
2. $\dot{V}(\cdot) \prec 0$ in $x_e allora x_e$ é asintoticamente stabile
3. $\dot{V}(\cdot) \succ 0$ in $x_e allora x_e$ é instabile

Ogni $V(\cdot)$ che rispetti la condizione $V(\cdot) \succ 0$ in x_e viene definita ""*Funzione Candidata di Lyapunov*""; se inoltre viene rispettata anche una delle tre condizioni sopra elencate, si parla allora di ""*Funzione di Lyapunov*"".

Per un sistema non lineare il teorema enunciato fornisce condizioni solo sufficienti, mentre per un sistema lineare le stesse sono sia sufficienti che necessarie; ciò significa che in quest'ultimo caso la ricerca di una funzione di Lyapunov può essere ristretta a funzioni in forma quadratica in x, ovvero alla ricerca di una opportuna matrice P.

Si consideri un sistema lineare $\dot{x}(t) = Ax(t)$ e sia $V(x) = x'Px, P = P' > 0$ una funzione candidata di Lyapunov con la relativa derivata $\dot{V}(x)$; effettuando le dovute sostituzioni si ottiene

$$\dot{V}(x) = \dot{x}'Px + xP\dot{x}' = x'A'Px + x'PAx = x'(A'P + PA)x \tag{3.2.17}$$

e quindi affinchè sia $\dot{V}(x) \prec 0$, condizione necessaria e sufficiente per la stabilità asintotica é $x'(A'P + PA)x \prec 0$. L'**equazione di Lyapunov** é quindi:

$$A'P + PA = -Q \tag{3.2.18}$$

dove P é l'incognita e Q, $Q = Q'$, é un parametro libero. Trasponendo i membri della 3.2.18 si ottiene

$$A'P' + P'A = -Q \tag{3.2.19}$$

ossia P e P' soddisfano la stessa equazione. Si deduce quindi che se $Q = Q'$, allora la soluzione P, se esiste ed é unica, é una matrice simmetrica. Di conseguenza é possibile enunciare il seguente teorema:

Il sistema lineare $\dot{x} = Ax(t)$ é globalmente asintoticamente stabile se e solo se per una scelta arbitraria di $Q = Q' > 0$, esiste una soluzione $P = P' > 0$ dell'equazione di Lyapunov.

Dopo aver introdotto le dovute definizioni ed enunciati alcuni teoremi di base legati alla stabilità, é infinte possibile trattare il problema risolto.

Sia dato il seguente sistema lineare a due dimensioni

$$\dot{x} = Ax, \qquad x_0 = x(0), \quad x \in \mathbb{R}^2 \tag{3.2.20}$$

riscrivibile in forma estesa

$$\begin{vmatrix} \dot{x_1} \\ \dot{x_2} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} \tag{3.2.21}$$

La matrice A deve essere scelta tale che il polinomio caratteristico sia un *polinomio di Hurwitz*, caratterizzato da radici λ_i (corrispondenti agli autovalori di A) tutte a parte reale minore di zero, $Re(\lambda_i < 0)$; ciò costituisce una condizione necessaria e sufficiente per la stabilità asintotica.

Partendo dal sistema dinamico 3.2.20 é stata risolta la seguente equazione di Lyapunov:

$$\dot{V}(x) = \nabla V(x) \cdot f(x) = -\alpha V(x) \tag{3.2.22}$$

dove Q é stata scelta pari al valore della soluzione cercata di P, moltiplicato per una certa costante $\alpha > 0$, il cui valore é noto una volta trovata la soluzione al sistema di equazioni risultate, illustrato più avanti. Risolvendo il prodotto scalare tra ∇V e $f(x)$ si ottiene la seguente PDE:

$$\frac{\partial V}{\partial x_1}(a_{11}x_1 + a_{12}x_2) + \frac{\partial V}{\partial x_2}(a_{21}x_1 + a_{22}x_2) = -\alpha V(x) \tag{3.2.23}$$

Per poter completare il problema é necessaria la conoscenza delle condizioni al contorno; in questo caso, poichè la soluzione della 3.2.23 si può ottenere facilmente risolvendo il seguente sistema lineare di equazioni

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}^T \begin{vmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{vmatrix} + \begin{vmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{vmatrix} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = -\alpha \begin{vmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{vmatrix} \tag{3.2.24}$$

é possibile imporre la soluzione reale lungo il contorno dell'intervallo su cui effettuare il calcolo.

Poichè si intende risolvere la 3.2.23 in forma numerica, per poi applicare i metodi MultiGrid descritti nel primo capitolo, é necessario discretizzare l'equazione:

$$\frac{u_{i,j+1} - u_{i,j}}{h_x}(a_{11}x_j + a_{12}y_i) + \frac{u_{i+1,j} - u_{i,j}}{h_y}(a_{21}x_j + a_{22}y_i) + \alpha u_{i,j} = 0 \tag{3.2.25}$$

Ponendo

$$\begin{aligned} K_1 &= (a_{11}x_j + a_{12}y_i) \\ K_2 &= (a_{21}x_j + a_{22}y_i) \end{aligned} \tag{3.2.26}$$

e risolvendo rispetto a $u_{i,j}$ si ottiene il seguente passo di rilassamento:

$$u_{i,j} = \frac{u_{i,j+1}h_yK_1 + u_{i+1,j}h_xK_2}{-\alpha h_x h_y + K_1 h_y + K_2 h_x} \tag{3.2.27}$$

Dall'ultima espressione si può notare come i punti pari dipendano unicamente dai punti dispari e viceversa per questi ultimi; di conseguenza é possibile applicare l'algoritmo di Red-Black Gauss Seidel come componenete iterativa del Full MultiGrid V-Cycle.

3.3 Controllo ottimo a minimo tempo

Relativamente al secondo problema affrontata nel caso bidimensionale, esso verrà trattato unicamente dal punto di vista teorico; nel capitolo dedicato ai test, la relativa

equazione non figura, in quanto i risultati finali non costituivano una valida approssimazione della soluzione reale.

Il suddetto problema riguarda la teoria del **Controllo Ottimo**; tale campo riguarda tutti gli algoritmi di controllo necessari per la stabilizzazione di un sistema dinamico, minimizzando determinate caratteristiche.

Un sistema di controllo é rappresentato dalla seguenze equazioni differenziale ordinaria:

$$\dot{x} = f(t, x, u), \qquad x(t_0) = x_0 \tag{3.3.1}$$

dove x é lo stato del sistema che assume valori in $\mathbb{R}^m$, u il controllo che assume valori in un insieme di controllo $U \subset \mathbb{R}^m$, t_0 il tempo iniziale, x_0 lo stato iniziale.

Un'altra nozione importante é il **funzionale di costo**:

$$J(u) = \int_{t_0}^{t_f} L(t, x(t), u(t))dt + K(t_f, x_f) \tag{3.3.2}$$

dove L e K sono chiamati rispettivamente *costo istantaneo* e *costo terminale*. J viene definito funzionale, in quanto essa dipende da una funzione, $u(t)$, così come per definizione di funzionale.

Si consideri il problema di portare una massa puntiforme all'origine di un determinato sistema di riferimento inerziale, in cui si deve trovare in condizione di riposo (con velocitÃ nulla nell'origine), applicando una certa accelerazione e cercando di minimizzare il tempo impiegato. Si può pensare di considerare non solo il costo funzionale (tempo impiegato) partendo da un fissato stato iniziale, ma il costo funzionale partendo da un qualsiasi stato iniziale, ossia

$$J(t, x, u) = \int_{t}^{t_f} L(s, x(s), u(s))ds + K(x(t_f)) \tag{3.3.3}$$

Si introduce quindi la **Funzione Valore**:

$$V(t,x) = \inf_{u\in[t,t_f]} J(t,x,u) \tag{3.3.4}$$

dove $V(t,x)$ é stata definita tramite la nozione di limite inferiore anzichè minimo, in quanto non é certo che esista un controllo ottimo.

Per completare il problema é necessario specificare la seguente condizione al contorno:

$$V(t_f,x) = K(x), \qquad \forall x \in \mathbb{R}^n \tag{3.3.5}$$

É possibile dimostrare che la 3.3.4 sia equivalente alla seguente PDE:

$$-\frac{\partial V(t,x)}{\partial t} = \inf_{u\in U}\{L(t,x,u) + < \nabla V(t,x), f(t,x,u) >\} \tag{3.3.6}$$

definita **Equazione di Hamilton-Jacobi-Bellman**.

Si consideri nuovamente il problema di portare una massa nell'origine del sistema, in cui il controllo é rappresentato dall'accelerazione ovvero

$$\begin{cases} \dot{x_1} = x_2 \\ \dot{x_2} = u \end{cases} \tag{3.3.7}$$

Riscrivendo la 3.3.6, con il costo istantaneo L avente valore 1 in quanto si considera l'integrale del tempo (la funzione integranda é quindi pari ad 1) si ottiene il seguente risultato

$$-\frac{\partial V(t,x)}{\partial t} = \inf_{u\in[-1,1]}\{1 + \frac{\partial V}{\partial x1}x2 + \frac{\partial V}{\partial x2}u\} \tag{3.3.8}$$

con condizione al contorno

$$V(t_f,0) = 0, \qquad \forall x \in \mathbb{R}^n \tag{3.3.9}$$

ossia il costo per portare la massa all'origine partendo dall'origine é chiaramente nullo. Poichè la misura da minimizzare é il tempo, si puó considerare il problema come se

avesse un dominio bidimensionale anzichè tridimensionale. Infatti a prescindere da quanto tempo é già passato, in qualunque istante, il tempo per arrivare in $x = 0$ da uno stato x non cambia, rimane costante relativamente alla soluzione ottima. Si può di conseguenza riscrivere la 3.3.8 nella forma

$$0 = \inf_{u\in[-1,1]}\{1 + \frac{\partial V}{\partial x1}x2 + \frac{\partial V}{\partial x2}u\} \tag{3.3.10}$$

Affinchè la precedente equazione abbia valore minimo il controllo u deve essere pari a

$$u = -sign(\frac{\partial V}{\partial x2}) = \begin{cases} 1, & \frac{\partial V}{\partial x2} < 0 \\ -1, & \frac{\partial V}{\partial x2} > 0 \\ ? & \frac{\partial V}{\partial x2} = 0 \end{cases} \tag{3.3.11}$$

ossia il problema é in realtà un problema in cui la legge di controllo ottima é di tipo "Bang Bang", in quanto u può assumere due soli valori opposti; Nel caso $\frac{\partial V}{\partial x2} = 0$ la u può assumere teoricamente qualsiasi valore. Dopo aver specificato l'espressione della u si può quindi esprimere la 3.3.10 nella forma

$$0 = \inf_{u\in[-1,1]}\{1 + \frac{\partial V}{\partial x1}x2 - |\frac{\partial V}{\partial x2}|\} \tag{3.3.12}$$

Come già nel caso dell'equazione di Lyapunov é necessario ora applicare il passo di discretizzazione, per poter poi applicare i metodi MultiGrid per la risoluzione della PDE. Discretizzando e risolvendo rispetto alla soluzione $v_{i,j}$ si ottiene il passo di rilassamento

$$\begin{aligned} v_{i,j} &= \frac{v_{i,j+1}h_{x2} - v_{i+1,j}h_{x1} + h_{x1}h_{x2}}{h_{x2} - h_{x1}} & \frac{\partial V}{\partial x2} > 0 \\ v_{i,j} &= \frac{v_{i,j+1}h_{x2} + v_{i+1,j}h_{x1} + h_{x1}h_{x2}}{h_{x2} + h_{x1}} & \frac{\partial V}{\partial x2} < 0 \\ v_{i,j} &= \frac{h_x + v_{i,j+1}y_i}{y_i} & \frac{\partial V}{\partial x2} = 0 \end{aligned} \tag{3.3.13}$$

Facendo il confronto con l'equazione di Lyapunov si può notare un problema di fondo: le condizioni al contorno forniscono il valore della soluzione in un solo punto del bordo;

per i metodi numerici precedentemente descritti ciò costituisce un problema. Con lo scopo di risolvere tale situazione si può pensare di imporre una trasformazione che dia le dovute condizioni al contorno, come la seguente trasformazione esponenziale:

$$W(x) = 1 - e^{-V(x)} \tag{3.3.14}$$

Si può dimostrare che la W soddisfi la seguente PDE:

$$\begin{aligned} &W(x) + sup_{u\in[-1,1]}\{< -f(x,u), \nabla W >\} = 1 \\ &W(0) = 0 \\ &W(x) = 1, \qquad \forall x \in \partial D \end{aligned} \tag{3.3.15}$$

ossia

$$\begin{aligned} &W(x) + \frac{\partial V}{x1}x2 + |\frac{\partial V}{x2}| = 1 \\ &W(0) = 0 \\ &W(x) = 1, \qquad \forall x \in \partial D \end{aligned} \tag{3.3.16}$$

dove u ha assunto l'espressione 3.3.11 La seconda condizione imposta porta ad una modifica della soluzione; per evitare che il risultato finale sia compromesso, é necessario prendere un intervallo abbastanza grande su cui effettuare il calcolo, tale che l'approssimazione della soluzione in un intorno dell'origine sia da considerare valida. Dopo la trasformazione il passo di rilassamento diventa

$$\begin{aligned} w_{i,j} &= \frac{h_x h_y + w_{i,j+1} h_y yi - h_x w_{i+1,j}}{h_x h_y + h_y yi - h_x}, \quad && \frac{\partial W}{\partial x_2} > 0 \\ w_{i,j} &= \frac{h_x h_y + w_{i,j+1} h_y yi + h_x w_{i+1,j}}{h_x h_y + h_y yi + h_x}, \quad && \frac{\partial W}{\partial x_2} < 0 \\ w_{i,j} &= \frac{h_x + w_{i,j+1} y_i}{h_x + yi} && \frac{\partial W}{\partial x_2} = 0 \end{aligned} \tag{3.3.17}$$

3.4 Poisson 3D

Come ultima piattaforma di test, si é pensato di risolvere un'equazione in tre dimensioni, in particolare **l'equazione di Poisson**. La forma di tale equazione é la

seguente:

$$\nabla^2 u(x,y,z) = f(x,y,z) \tag{3.4.1}$$

dove l'operatore ∇^2 viene definito operatore di Laplace ed é pari a

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{3.4.2}$$

In particolare l'equazione risolta é la seguente:

$$\begin{aligned} &\nabla^2 \phi(x,y,z) = f(x,y,z) = -3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z) \\ &\phi(x,y,z) = 0, \qquad \text{su } \partial\Omega, \quad \Omega = (0,1)^3 \end{aligned} \tag{3.4.3}$$

la cui soluzione reale é nota ed é pari a

$$\phi(x,y,z) = \sin(\pi x)\sin(\pi y)\sin(\pi z) \tag{3.4.4}$$

Applicando il passo di discretizzazione e risolvendo rispetto a v (soluzione approssimata di ϕ) si ottiene

$$\begin{aligned} v_{i,j,k} = ((&v_{i-1,j,k}h_x^2h_z^2 + v_{i+1,j,k}h_x^2h_z^2) + (v_{i,j-1,k}h_y^2h_z^2 + v_{i,j+1,k}h_y^2h_z^2) + \\ &(v_{i,j,k-1}h_y^2h_x^2 + v_{i,j,k+1}h_y^2h_x^2) - (f_{i,j,k}h_x^2h_y^2h_z^2)) \cdot \frac{1}{2(h_x^2h_z^2 + h_y^2h_z^2 + h_y^2h_x^2)} \end{aligned} \tag{3.4.5}$$

L'equazione appena descritta può essere risolta tramite le Cuda (nel caso di griglie la cui dimensione superi il limite di thread lungo l'asse z di un blocco) utilizzando la procedura descritta alla fine del paragrafo 2.1.2. Tale procedura é evitabile solo se dotati di Gpu con supporto a Kernel tridimensionali, oppure nel caso di griglie di dimensioni lungo l'asse z contenute (minori di 64).

Capitolo 4

Gpu e Cpu al confronto

Il presente capitolo é interamente votato alla descrizione dei risultati ottenuti nella risoluzione delle PDE descritte del capitolo 3.. La piattaforma hardware utilizzata nei test é così composta:

- Processore: Pentium Dual Core Processor E5400: 2,7Ghz, 800Fsb
- Memoria Ram: 2Gb
- Scheda video: Geforce GTX 550Ti 1Gb, 192 Cuda Cores

I test effettuati mettono al confronto le prestazioni degli algoritmi MultiGrid eseguiti da Cpu e Gpu; in particolare é stato misurato il tempo di esecuzione in relazione alla grandezza della griglia più densa, in termini di punti lungo un asse (ogni asse é dotato dello stesso numero di punti). Per comodità si specificano qui nuovamente dettagli riguardanti le griglie; il numero di punti presenti su un asse é pari a 2^k+1, x_0 compreso, avendo così un numero di intervalli pari ad una potenza di 2; fissato un k, per cui Ω^h ha dimensione pari a 2^k+1, il numero totale di griglie é pari allo stesso k. Da notare come le dimensioni della griglia più densa dipendano dalla grandezza dell'intervallo su cui effettuare il calcolo, in maniera direttamente proporzionale; a maggiore ampiezza

del dominio, per garantire una buona approssimazione deve corrispondere un maggior numero di punti. Si sottolinea quindi che per calcolatori dotati di poca memoria ram, i risultati migliori si ottengono per domini non troppo grandi (una decina di unità).
Altri fattori da tenere in considerazione relativamente al calcolo del tempo d'esecuzione, sono i seguenti parametri:

- ν_0: numero di V-Cycle eseguiti ad ogni livello nel *Full MultiGrid V-Cyle*
- ν_1: numero di passi di rilassamento da eseguire sul livello corrente prima di spostarsi al livello inferiore (discesa nel V-Cycle)
- ν_2: numero di passi di rilassamento da eseguire sul livello corrente prima di spostarsi al livello superiore (risalita nel V-Cycle)

La modifica del primo parametro può influire in maniera evidente sul tempo d'esecuzione dell'algoritmo, anche effettuando incrementi di qualche unità; generalmente ponendolo pari ad 1 si ottengono già risultati soddisfacenti.
Gli altri due parametri vengono posti generalmente allo stesso valore, il quale dipende fortemente dal numero di punti in cui é strutturata la griglia e dalla grandezza del dominio su cui si effettua il calcolo; ad un maggior numero di punti corrisponde un maggior numero di passi di rilassamento da effettuare su ognuno di essi. Tali parametri variano anche di equazione in equazione, quindi ripetendo più volte l'esperimento si può trovare il valore ideale.
I primi test sono stati effettuati sull'equazione presentata nel paragrafo 3.1

$$u' - \frac{u(x)}{e^x + 1} = e^x \tag{4.0.1}$$

I parametri utilizzati sono i seguenti:

- Dominio: $[0, 1]$

- Dimensioni griglia: 257, 513, 1025, 2049, 4097, 8193 punti.
- $\nu_0 = 2$: Ogni ciclo a V nel FMG V-Cyle viene eseguito due volte
- $\nu_1 = \nu_2 = 1000$: 1000 passi di rilassamento, prima di scendere al livello inferiore, e prima di salire al livello superiore; tale numero é necessario affinchè si ottenga una buona approssimazione nel caso di griglia più densa

I risultati ottenuti sono mostrati in Figura 4.1. Il grafico mostra come inizialmente

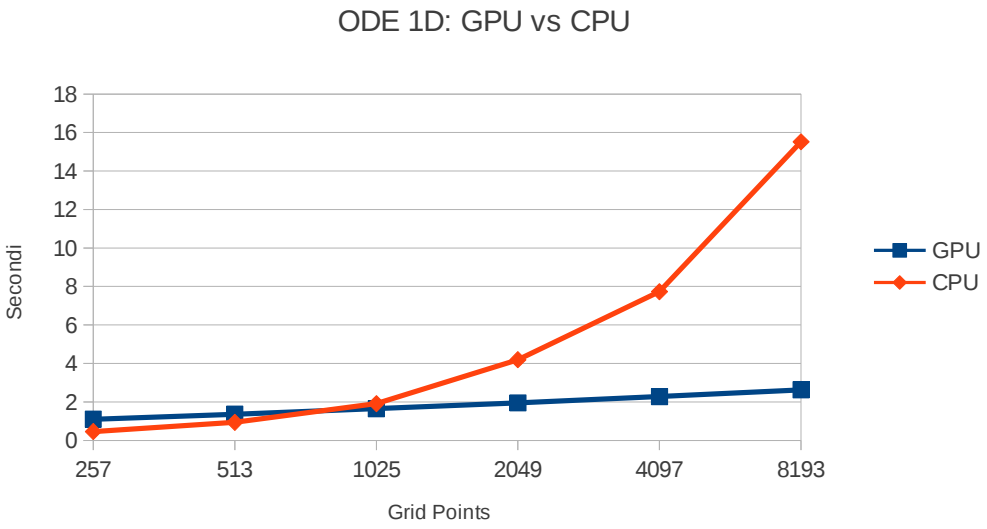

Figura 4.1: Equazione differenziale ordinaria 1D: GPU vs CPU.

i tempi di esecuzione di Cpu e Gpu siano confrontabili; la Cpu é leggermente piú veloce, in quanto nel caso dell'applicazione CUDA deve essere gestito lo scambio di dati fra Cpu e Gpu, cosa che non accade nel caso di sola Cpu. Quando il numero di punti lungo l'asse x raggiunge dimensioni notevoli, la Gpu con la sua architettura fortemente parallela, registra tempi d'esecuzione inferiori di alcuni secondi rispetto alla Cpu. In realtà in tale situazione un gran numero di punti non é necessario, in quanto si ottiene una buona approssimazione con griglie poco dense; in questo conte-

sto (su un dominio ristretto) quindi l'utilizzo della Gpu non porta a grandi vantaggi. Diversamente accade nelle altre equazioni.

Relativamente al caso in due dimensioni, é stata risolta l'equazione di Lyapunov:

$$\frac{\partial V}{\partial x_1}(a_{11}x_1 + a_{12}x_2) + \frac{\partial V}{\partial x_2}(a_{21}x_1 + a_{22}x_2) = -\alpha V(x) \tag{4.0.2}$$

I parametri utilizzati sono i seguenti:

- Dominio: $[0, 20]^2$ (stesso intervallo su asse x e y)
- Dimensioni griglia: 65, 129, 257, 513, 1025, 2049, 4097 punti per asse
- $\nu_0 = 2$: Ogni ciclo a V nel FMG V-Cyle viene eseguito due volte
- $\nu_1 = \nu_1 = 500$: 500 passi di rilassamento, prima di scendere al livello inferiore, e prima di salire al livello superiore; tale numero é necessario affinchè si ottenga una buona approssimazione nel caso di griglia più densa di 2049 punti.

I risultati ottenuti sono mostrati in Figura 4.2 Dalla figura si evince che inizialmente Gpu e Cpu hanno praticamente le stesse prestazioni durante l'esecuzione dell'algoritmo. Le potenziliatà dell'architettura fortemente parallela della Gpu, capace di trasformare i cicli di iterazione lungo i punti delle griglie in attività di elaborazione parallele, vengono messe in mostra non appena si comincia ad utilizzare griglie più dense costituite da un alto numero di punti. Relativamente al grafico, si nota come non sono stati effettuati calcoli con la CPU, nel caso di griglia più densa di dimensioni per asse pari a 2049, in quanto richiedente un elevato tempo d'esecuzione; l'andamento generale é comunque evidente ed immaginabile osservando la curva.
Il grafico in Figura 4.2 riporta informazioni relative unicamente alle prestazioni di Gpu

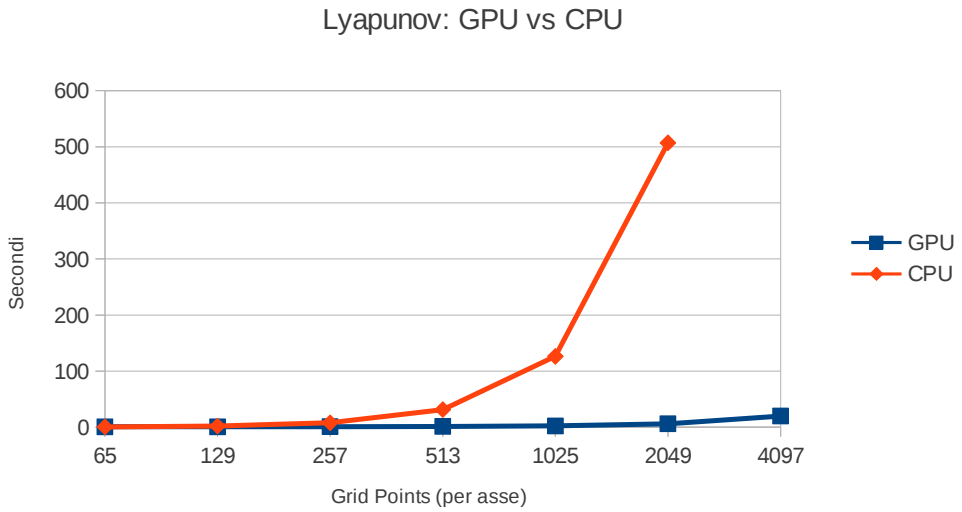

Figura 4.2: Equazione di Lyapunov: GPU vs CPU.

e Cpu durante l'esecuzione dell'algoritmo; é però necessario che tali elevate prestazioni, nel caso della Gpu, siano accompagnate da buone approssimazioni della soluzione reale; in Figura 4.3 é riportato un grafico che pone in relazione l'errore assoluto medio con il numero di punti per asse relativo alla griglia più densa.

L'ultima problema risolto é rappresentato dall'equazione di Poisson:

$$\nabla^2\phi(x,y,z) = f(x,y,z) = -3\pi^2 \sin(\pi x)\sin(\pi y)\sin(\pi z) \tag{4.0.3}$$

I test eseguiti su Cpu e Gpu utilizzano i seguenti parametri:

- Dominio: $[0,1]^3$ (stesso intervallo su asse x, y e z)
- Dimensioni griglia: 9, 17, 33, 65, 129, 257 punti per asse
- $\nu_0 = 2$: Ogni ciclo a V nel FMG V-Cyle viene eseguito due volte
- $\nu_1 = \nu_1 = 3000$: 3000 passi di rilassamento, prima di scendere al livello inferiore, e prima di salire al livello superiore;

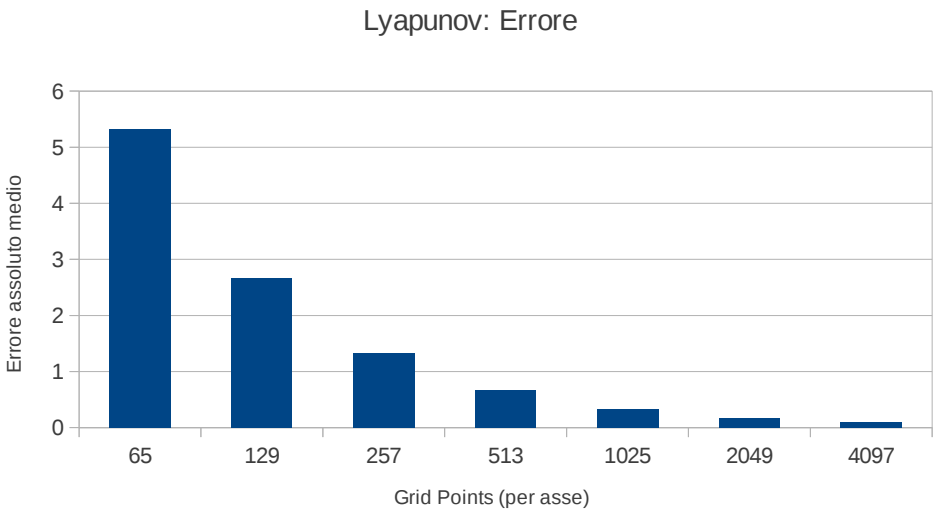

Figura 4.3: Equazione di Lyapunov: Errore.

Facendo il confronto con il caso a due dimensioni dell'equazione di Lyapunov, si nota subito come ci siano delle sostanziali differenze; il numero di punti per asse é diminuito drasticamente, in quanto il numero totale di punti risulta essere comunque molto elevato; il numero di passi di rilassamento da eseguire ad ogni livello nel V-Cycle é salito a 3000, in quanto empiricamente si é visto che sotto a tale valore non si otteneva una buona approssimazione. Il fatto che il dominio sia stato ridotto é invece legato intrinsecamente alla PDE risolta, poichè le condizioni al contorno fornite dal problema, sono relative a tali punti.

In Figura 4.4 é mostrato un grafico relativo alle prestazioni della Gpu e della Cpu al confronto, nella risoluzione dell'equazione di Poisson in tre dimensioni. Come nel caso dell'equazione di Lyapunov, non tutti i test sono stati eseguiti per la Cpu, in particolare quelli relativi a 129 e 257 punti per asse; come é evidente dal grafico, tali test richiedono tempo d'esecuzione molto elevati.

Dai test mostrati risulta evidente come l'utilizzo della Gpu, nel contesto dei metodi

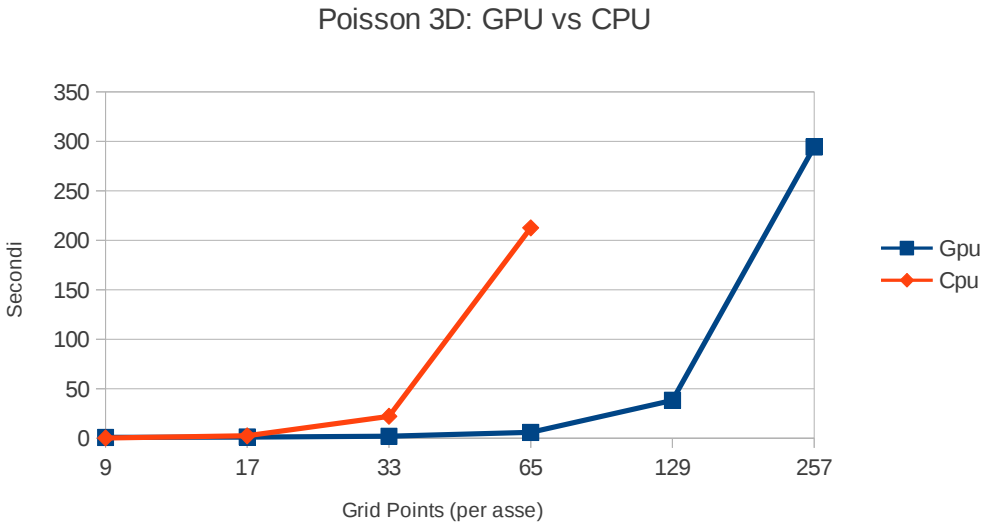

Figura 4.4: Equazione di Poisson 3D.

MultiGrid, porta a grandi vantaggi dal punto di vista del tempo d'esecuzione. Poichè il numero di punti da utilizzare deve essere direttamente proporzionale alle dimensioni del dominio, risulta chiaro come il calcolo della soluzione lungo intervalli molto ampi possa essere eseguito unicamente mediante il supporto della Gpu; essa non rimane più quindi uno strumento utile nell'eseguire i calcoli, ma diventa indispensabile per ottenere soluzioni adeguate in tempi accettabili.

Capitolo 5

Conclusioni e sviluppi futuri

Il capitolo precedente ha messo in evidenza come l'utilizzo congiunto di metodi MultiGrid e librerie Cuda, possa essere una scelta vincente per abbattere i tempi richiesti per il calcolo numerico nella risoluzione di PDE. E' stato dimostrato numericamente come la parallelizzazione (ove possibile) degli algoritmi risolutori, porti a sostanziali vantaggi in termini prestazionali. É quindi importante sottolineare come sia possibile compiere un ulteriore passo in tale senso; la griglia su cui si stà effettuando il calcolo, può essere divisa in più sottogriglie di uguali dimensioni, ognuna delle quali assegnata ed elaborate da Gpu diverse. La capacità di calcolo aumenta quindi in maniera proporzionale al numero di Gpu impiegate. In un tale contesto la gestione delle condizioni al contorno diventa un'operazione ancora più delicata di quanto non lo sia già; punti che in precedenza erano da considerarsi interni alla griglia, vanno ora a trovarsi al limite fra due sottogriglie, e vanno quindi maneggiati con cautela (la cattiva gestione delle condizioni al contorno é una delle principali cause di fallimento degli algoritmi risolutori).

Si conclude quindi questo lavoro, nella speranza che venga riutilizzato in futuro, ampliando la gamma di problemi risolvibili, introducendo ad esempio PDE non lineari, qui trattate unicamente dal punto di vista teorico, come l'equazione di**Grad-**

Shafranov di cui si é parlato nell'introduzione.

Appendice A - MultiGrid 1D

```
__global__ void CUDASetGrids(float* d_v, float* d_f, int sizeX, float h_x, float x_a,
    float x_b)
{
        int posX = blockDim.x * blockIdx.x + threadIdx.x;

        if(posX >= sizeX)
                return;

        float xj = x_a + posX*h_x;
        d_f[posX] = exp(xj);


        if(posX == 0)
        {
                d_v[posX] = (exp(x_a) + x_a - 3)/(1 + exp(-x_a));
                return;
        }
        if(posX == sizeX - 1)
        {
                d_v[posX] = (exp(x_b) + x_b - 3)/(1 + exp(-x_b));
                return;
        }
}

void MultiGrid1D::VCycle(int gridID, int v1, int v2)
{
        Grid1D* fine = grids1D[gridID];
        Relax(fine, v1);
        if(gridID != numGrids - 1)
        {
                float* residual = CalculateResidual(fine);
                int residualSizeX = fine->sizeX;
                Grid1D* coarse = grids1D[gridID+1];
                Restrict(residual, residualSizeX, coarse->d_f, coarse->sizeX);

                Set(coarse->d_v, coarse->sizeX, 0.0f, true);

                VCycle(gridID+1, v1, v2);

                int fsizeX = fine->sizeX;
                float* fine_error;
                cudaMalloc(&fine_error, fsizeX*sizeof(float));

                Interpolate(fine_error, fsizeX, coarse->d_v, coarse->sizeX);

                ApplyCorrection(fine->d_v, fsizeX, fine_error, fsizeX);
        }

        Relax(fine, v2);
}
```

```
50
51 void MultiGrid1D::FullMultiGridVCycle(int gridID, int v0, int v1, int v2)
52 {
53         Grid1D* fine = grids1D[gridID];
54         if(gridID != numGrids - 1)
55         {
56                 Grid1D* coarse = grids1D[gridID + 1];
57                 Restrict(fine->d_f, fine->sizeX, coarse->d_f, coarse->sizeX);
58                 FullMultiGridVCycle(gridID + 1, v0, v1, v2);
59                 Interpolate(fine->d_v, fine->sizeX, coarse->d_v, coarse->sizeX);
60         }
61         else
62                 Set(fine->d_v, fine->sizeX, 0.0f, false);
63
64         for(int i = 0; i < v0; i++)
65                 VCycle(gridID, v1, v2);
66 }
67
68 __global__ void CUDARestrict(float* fine, int fsizeX, float* coarse, int csizeX)
69 {
70         int cposX = blockDim.x * blockIdx.x + threadIdx.x;
71
72         if(cposX >= csizeX)
73                 return;
74
75         if(cposX == 0 || cposX == csizeX - 1)
76         {
77                 coarse[cposX] = fine[2*cposX];
78                 return;
79         }
80
81         float C = fine[2*cposX];
82         float E = fine[2*cposX+1];
83         float O = fine[2*cposX-1];
84
85         coarse[cposX] = (1/4.0f)*(O + 2*C + E);
86
87 }
88
89 __global__ void CUDAInterpolate(float* fine, int fsizeX, float* coarse, int csizeX)
90 {
91         int fposX = blockDim.x * blockIdx.x + threadIdx.x;
92
93         if(fposX >= fsizeX)
94                 return;
95
96         int cposX = fposX/2;
97
98         if(fposX == 0 || fposX == fsizeX - 1)
99                 return;
100
101         if(fposX%2 == 0)
102                 fine[fposX] = coarse[cposX];
103         else
104                 fine[fposX] = (1/2.0f)*(coarse[cposX] + coarse[cposX+1]);
105
106 }
107
108 __global__ void CUDARelax(float* d_v, float* d_f, float h_x, int sizeX, float x_a)
109 {
110         int posX = blockDim.x * blockIdx.x + threadIdx.x;
111
112         if(posX >= sizeX)
113                 return;
114
```

```
115         if(posX == 0 || posX == sizeX - 1)
116                 return;
117
118         if(posX%2 == 0)
119         {
120                 float xj = x_a + posX*h_x;
121                 d_v[posX] = (d_v[posX+1]*(exp(xj)+1) - d_f[posX]*h_x*(exp(xj) + 1))/(
                        exp(xj)+1+h_x);
122         }
123
124         __syncthreads();
125
126         if(posX%2 != 0)
127         {
128                 float xj = x_a + posX*h_x;
129                 d_v[posX] = (d_v[posX+1]*(exp(xj)+1) - d_f[posX]*h_x*(exp(xj) + 1))/(
                        exp(xj)+1+h_x);
130         }
131 }
132
133 __global__ void CUDASet(float* d_v, int sizeX, float value, bool modifyBoundaries)
134 {
135         int posX = blockDim.x * blockIdx.x + threadIdx.x;
136
137         if(posX >= sizeX)
138                 return;
139
140         if(!modifyBoundaries)
141                 if(posX == 0 || posX == sizeX - 1)
142                         return;
143
144         d_v[posX] = value;
145 }
146
147 __global__ void CUDACalculateResidual(float* d_v, float* d_f, float* d_residual,
        float h_x, int sizeX, float x_a)
148 {
149         int posX = blockDim.x * blockIdx.x + threadIdx.x;
150
151         if(posX >= sizeX)
152                 return;
153
154         if(posX == 0 || posX == sizeX - 1)
155         {
156                 d_residual[posX] = 0;
157                 return;
158         }
159
160         float xj = x_a + posX*h_x;
161         d_residual[posX]= d_f[posX] - (d_v[posX+1] - d_v[posX])/h_x - d_v[posX]/(exp(
                xj)+1);
162 }
163
164 __global__ void CUDAApplyCorrection(float* fine, float* error, int sizeX)
165 {
166         int posX = blockDim.x * blockIdx.x + threadIdx.x;
167
168         if(posX >= sizeX)
169                 return;
170
171         if(posX == 0 || posX == sizeX - 1)
172                 return;
173
174         fine[posX] = fine[posX] + error[posX];
175 }
```

Appendice B - MultiGrid 2D

```
1
2  __global__ void CUDASetBoundaries(float* d_v, float* d_f, size_t pitch, int sizeX,
       int sizeY, float h_x, float h_y, float x_a, float y_a)
3  {
4          int iy = blockDim.y * blockIdx.y + threadIdx.y;
5          int ix = blockDim.x * blockIdx.x + threadIdx.x;
6          int idx = iy * pitch + ix;
7
8          if(idx >= pitch*sizeY)
9                  return;
10
11         d_f[idx] = 0.0f;
12
13         if(ix == 0 || ix == sizeX - 1 || iy == 0 || iy == sizeY - 1)
14         {
15                 float yi = y_a + iy*h_y;
16                 float xj = x_a + ix*h_x;
17                 float sol = 2*xj*xj-4*xj*yi+2*yi*yi;
18                 d_v[idx] = sol;
19                 return;
20         }
21
22         d_v[idx] = 0.0f;
23 }
24
25 void MultiGrid2D::VCycle(int gridID, int v1, int v2)
26 {
27         Grid2D* fine = grids2D[gridID];
28         Relax(fine, v1);
29         if(gridID != numGrids - 1)
30         {
31                 float* residual = CalculateResidual(fine);
32                 int residualSize = fine->size;
33                 int residualPitch = fine->d_pitch;
34                 Grid2D* coarse = grids2D[gridID+1];
35                 Restrict(residual, residualSize, residualPitch, coarse->d_f, coarse->
                       size, coarse->d_pitch);
36
37                 Set(coarse->d_v, coarse->size, coarse->d_pitch, 0.0f, true);
38
39                 VCycle(gridID+1, v1, v2);
40
41                 int fsize = fine->size;
42                 size_t error_pitchByte;
43                 float* fine_error;
44                 cudaMallocPitch(&fine_error, &error_pitchByte, fsize*sizeof(float),
                       fsize);
45                 int error_pitch = error_pitchByte/sizeof(float);
46
```

```
47                  Interpolate(fine_error, fsize, error_pitch, coarse->d_v, coarse->size
                        , coarse->d_pitch);
48
49                  ApplyCorrection(fine->d_v, fsize, fine->d_pitch, fine_error, fsize,
                        error_pitch);
50          }
51
52          Relax(fine, v2);
53
54  }
55
56  void MultiGrid2D::FullMultiGridVCycle(int gridID, int v0, int v1, int v2)
57  {
58          Grid2D* fine = grids2D[gridID];
59          if(gridID != numGrids - 1)
60          {
61                  Grid2D* coarse = grids2D[gridID + 1];
62                  Restrict(fine->d_f, fine->size, fine->d_pitch, coarse->d_f, coarse->
                        size, coarse->d_pitch);
63                  FullMultiGridVCycle(gridID + 1,v0, v1, v2);
64                  Interpolate(fine->d_v, fine->size, fine->d_pitch, coarse->d_v, coarse
                        ->size, coarse->d_pitch);
65          }
66          else
67                  Set(fine->d_v, fine->size, fine->d_pitch, 0.0f, false);
68
69          for(int i = 0; i < v0; i++)
70                  VCycle(gridID, v1, v2);
71  }
72
73  __global__ void CUDARestrict(float* fine, int fsize, int f_pitch, float* coarse, int
        csize, int c_pitch)
74  {
75          int c_iy = blockDim.y * blockIdx.y + threadIdx.y;
76          int c_ix = blockDim.x * blockIdx.x + threadIdx.x;
77          int c_idx = c_iy * c_pitch + c_ix;
78
79          int f_iy = c_iy*2;
80          int f_ix = c_ix*2;
81          int f_idx = f_iy * f_pitch + f_ix;
82
83          if(c_idx >= c_pitch*csize)
84                  return;
85
86          if(c_ix == 0 || c_ix == csize - 1 || c_iy == 0 || c_iy == csize - 1)
87          {
88                  coarse[c_idx] = fine[f_idx];
89                  return;
90          }
91
92          f_idx = f_iy * f_pitch + f_ix;
93          float C = fine[f_idx];
94
95          f_idx = (f_iy-1)*f_pitch + f_ix;
96          float N = fine[f_idx];
97
98          f_idx = (f_iy+1)*f_pitch + f_ix;
99          float S = fine[f_idx];
100
101         f_idx = f_iy*f_pitch + (f_ix+1);
102         float E = fine[f_idx];
103
104         f_idx = f_iy*f_pitch + (f_ix-1);
105         float O = fine[f_idx];
106
```

```
107         f_idx = (f_iy-1)*f_pitch + (f_ix+1);
108         float NE = fine[f_idx];
109 
110         f_idx = (f_iy-1)*f_pitch + (f_ix-1);
111         float NO = fine[f_idx];
112 
113         f_idx = (f_iy+1)*f_pitch + (f_ix+1);
114         float SE = fine[f_idx];
115 
116         f_idx = (f_iy+1)*f_pitch + (f_ix-1);
117         float SO = fine[f_idx];
118 
119         coarse[c_idx] = (1/16.0f)*(NO+NE+SO+SE + 2*(O+E+N+S) + 4*C);
120 
121 }
122 
123 __global__ void CUDAInterpolate(float* fine, int fsize, int f_pitch, float* coarse,
        int csize, int c_pitch)
124 {
125         int f_iy = blockDim.y * blockIdx.y + threadIdx.y;
126         int f_ix = blockDim.x * blockIdx.x + threadIdx.x;
127         int f_idx = f_iy * f_pitch + f_ix;
128 
129         int c_iy = f_iy/2;
130         int c_ix = f_ix/2;
131         int c_idx = c_iy * c_pitch + c_ix;
132 
133         if(f_idx >= f_pitch*fsize)
134                 return;
135 
136         if(f_ix == 0 || f_ix == fsize - 1 || f_iy == 0 || f_iy == fsize - 1)
137                 return;
138 
139         if(f_iy%2 == 0 && f_ix%2 == 0)
140                 fine[f_idx] = coarse[c_idx];
141 
142         if(f_iy%2 != 0 && f_ix%2 == 0)
143         {
144                 c_idx = c_iy * c_pitch + c_ix;
145                 float N = coarse[c_idx];
146                 c_idx = (c_iy+1) * c_pitch + c_ix;
147                 float S = coarse[c_idx];
148                 fine[f_idx] = (1/2.0f)*(N + S);
149         }
150 
151         if(f_iy%2 == 0 && f_ix%2 != 0)
152         {
153                 c_idx = c_iy * c_pitch + c_ix;
154                 float O = coarse[c_idx];
155                 c_idx = c_iy * c_pitch + (c_ix+1);
156                 float E = coarse[c_idx];
157                 fine[f_idx] = (1/2.0f)*(O + E);
158         }
159 
160         if(f_iy%2 != 0 && f_ix%2 != 0)
161         {
162                 c_idx = c_iy * c_pitch + c_ix;
163                 float NO = coarse[c_idx];
164                 c_idx = (c_iy+1) * c_pitch + c_ix;
165                 float SO = coarse[c_idx];
166                 c_idx = c_iy * c_pitch + (c_ix+1);
167                 float NE = coarse[c_idx];
168                 c_idx = (c_iy+1) * c_pitch + (c_ix+1);
169                 float SE = coarse[c_idx];
170                 fine[f_idx] = (1/4.0f)*(NO + SO + NE + SE);
```

```
171         }
172
173 }
174
175 __global__ void CUDARelax(float* v, float* f, float h_x, float h_y, int size, int
        pitch, float x_a, float y_a, float* A, int alfa)
176 {
177         int iy = blockDim.y * blockIdx.y + threadIdx.y;
178         int ix = blockDim.x * blockIdx.x + threadIdx.x;
179         int idx = iy * pitch + ix;
180
181         if(idx >= pitch*size)
182                 return;
183
184         if(ix == 0 || ix == size - 1 || iy == 0 || iy == size - 1)
185                 return;
186
187         if((iy+ix)%2 == 0)
188         {
189                 float xj = x_a + ix*h_x;
190                 float yi = y_a + iy*h_y;
191                 float K1 = A[0]*xj + A[1]*yi;
192                 float K2 = A[2]*xj + A[3]*yi;
193                 float den = K1*h_y+K2*h_x-alfa*h_x*h_y;
194
195                 idx = iy * pitch + (ix+1);
196                 float E = v[idx];
197
198                 idx = (iy+1) * pitch + ix;
199                 float S = v[idx];
200
201                 idx = iy * pitch + ix;
202
203                 v[idx] = (h_y*K1*E + h_x*K2*S - 0)/(den);
204         }
205
206         __syncthreads();
207
208         if((iy+ix)%2 != 0)
209         {
210                 float xj = x_a + ix*h_x;
211                 float yi = y_a + iy*h_y;
212                 float K1 = A[0]*xj + A[1]*yi;
213                 float K2 = A[2]*xj + A[3]*yi;
214                 float den = K1*h_y+K2*h_x-alfa*h_x*h_y;
215
216                 idx = iy * pitch + (ix+1);
217                 float E = v[idx];
218
219                 idx = (iy+1) * pitch + ix;
220                 float S = v[idx];
221
222                 idx = iy * pitch + ix;
223
224                 v[idx] = (h_y*K1*E + h_x*K2*S - 0)/(den);
225         }
226 }
227
228 __global__ void CUDASet(float* v, int size, int pitch, float value, bool modifyBorder
        )
229 {
230         int iy = blockDim.y * blockIdx.y + threadIdx.y;
231         int ix = blockDim.x * blockIdx.x + threadIdx.x;
232         int idx = iy * pitch + ix;
233
```

```
234         if(idx >= pitch*size)
235                 return;
236
237         if(!modifyBorder)
238                 if(ix == 0 || ix == size - 1 || iy == 0 || iy == size - 1)
239                         return;
240
241         v[idx] = value;
242 }
243
244 __global__ void CUDACalculateResidual(float* v, float* f, float* residual, float h_x,
        float h_y, int size, int pitch, float x_a, float y_a, float* A, int alfa)
245 {
246         int iy = blockDim.y * blockIdx.y + threadIdx.y;
247         int ix = blockDim.x * blockIdx.x + threadIdx.x;
248         int idx = iy * pitch + ix;
249
250         if(idx >= pitch*size)
251                 return;
252
253         if(ix == 0 || ix == size - 1 || iy == 0 || iy == size - 1)
254         {
255                 residual[idx] = 0.0;
256                 return;
257         }
258
259         float xj = x_a + ix*h_x;
260         float yi = y_a + iy*h_y;
261         float K1 = A[0]*xj + A[1]*yi;
262         float K2 = A[2]*xj + A[3]*yi;
263
264         idx = iy * pitch + (ix+1);
265         float E = v[idx];
266
267         idx = (iy+1) * pitch + ix;
268         float S = v[idx];
269
270         idx = iy * pitch + ix;
271
272         residual[idx] = f[idx] - (h_y*K1*E + h_x*K2*S - v[idx]*(h_y*K1+h_x*K2-alfa*
            h_x*h_y))/(h_x*h_y);
273 }
274
275 __global__ void CUDAApplyCorrection(float* fine, float* error, int size, int pitch)
276 {
277         int iy = blockDim.y * blockIdx.y + threadIdx.y;
278         int ix = blockDim.x * blockIdx.x + threadIdx.x;
279         int idx = iy * pitch + ix;
280
281         if(idx >= pitch*size)
282                 return;
283
284         if(ix == 0 || ix == size - 1 || iy == 0 || iy == size - 1)
285                 return;
286
287         fine[idx] = fine[idx] + error[idx];
288 }
```

Appendice C - MultiGrid 3D

```
1
2  __global__ void CUDAInitGrids(float* d_v, float* d_f, int* d_sizeXYZ, float h_x,
       float h_y, float h_z, float x_a, float y_a, float z_a)
3  {
4          int sizeX = d_sizeXYZ[0];
5          int sizeY = d_sizeXYZ[1];
6          int sizeZ = d_sizeXYZ[2];
7
8          float PI_D = CUDART_PI_F;
9
10         int posY = (threadIdx.x + blockIdx.x*blockDim.x)/sizeX;
11         int posX = (threadIdx.x + blockIdx.x*blockDim.x) - posY*sizeX;
12         int posZ = (blockIdx.y * blockDim.y) + threadIdx.y;
13
14         int idx = posX + posY * sizeX + posZ * sizeX * sizeY;
15
16         if(posY >= sizeY || posX >= sizeX || posZ >= sizeZ)
17                 return;
18
19         float x = x_a + posX*h_x;
20         float y = y_a + posY*h_y;
21         float z = z_a + posZ*h_z;
22
23         d_f[idx] = -3*PI_D*PI_D*sin(PI_D*x)*sin(PI_D*y)*sin(PI_D*z);
24
25         if(posX == 0 || posX == sizeX - 1 || posY == 0 || posY == sizeY - 1 || posZ
               == 0 || posZ == sizeZ - 1)
26         {
27                 d_v[idx] = 0.0f;
28                 return;
29         }
30 }
31
32 void MultiGrid3D::VCycle(int gridID, int v1, int v2)
33 {
34         Grid3D* fine = grids3D[gridID];
35         Relax(fine, v1);
36         if(gridID != numGrids - 1)
37         {
38                 float* residual = CalculateResidual(fine);
39                 int* residualSize = fine->d_sizeXYZ;
40                 Grid3D* coarse = grids3D[gridID+1];
41                 Restrict(residual, residualSize, coarse->d_f, coarse->d_sizeXYZ);
42                 Set(coarse->d_v, coarse->d_sizeXYZ, 0.0f, true);
43                 VCycle(gridID+1, v1, v2);
44
45                 int* fsize = fine->d_sizeXYZ;
46
47                 float* d_fine_error;
```

```
48                  cudaMalloc(&d_fine_error, fine->sizeX*fine->sizeY*fine->sizeZ*sizeof(
                        float));
49                  Interpolate(d_fine_error, fsize, coarse->d_v, coarse->d_sizeXYZ);
50
51                  ApplyCorrection(fine->d_v, fsize, d_fine_error, fsize);
52
53          }
54
55          Relax(fine, v2);
56  }
57
58  void MultiGrid3D::FullMultiGridVCycle(int gridID, int v0, int v1, int v2)
59  {
60          Grid3D* fine = grids3D[gridID];
61          if(gridID != numGrids - 1)
62          {
63                  Grid3D* coarse = grids3D[gridID + 1];
64                  Restrict(fine->d_f, fine->d_sizeXYZ, coarse->d_f, coarse->d_sizeXYZ);
65                  FullMultiGridVCycle(gridID + 1,v0, v1, v2);
66                  Interpolate(fine->d_v, fine->d_sizeXYZ, coarse->d_v, coarse->
                        d_sizeXYZ);
67          }
68          else
69                  Set(fine->d_v, fine->d_sizeXYZ, 0.0f, false);
70
71          for(int i = 0; i < v0; i++)
72                  VCycle(gridID, v1, v2);
73  }
74
75  __global__ void CUDARestrict(float* fine, int d_fsizeXYZ[], float* coarse, int
        d_csizeXYZ[])
76  {
77          int fsizeX = d_fsizeXYZ[0];
78          int fsizeY = d_fsizeXYZ[1];
79
80          int csizeX = d_csizeXYZ[0];
81          int csizeY = d_csizeXYZ[1];
82          int csizeZ = d_csizeXYZ[2];
83
84          int cposY = (threadIdx.x + blockIdx.x*blockDim.x)/csizeX;
85          int cposX = (threadIdx.x + blockIdx.x*blockDim.x) - cposY*csizeX;
86          int cposZ = (blockIdx.y * blockDim.y) + threadIdx.y;
87
88          int cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
89
90          int fposX = 2*cposX;
91          int fposY = 2*cposY;
92          int fposZ = 2*cposZ;
93          int fidx = fposX + fposY * fsizeX + fposZ * fsizeX * fsizeY;
94
95          if(cposY >= csizeY || cposX >= csizeX || cposZ >= csizeZ)
96                  return;
97
98
99          if(cposX == 0 || cposX == csizeX - 1 || cposY == 0 || cposY == csizeY - 1 ||
                cposZ == 0 || cposZ == csizeZ - 1)
100         {
101                 coarse[cidx] = fine[fidx];
102                 return;
103         }
104
105         fidx = fposX + fposY * fsizeX + fposZ * fsizeX * fsizeY;
106         float C_C = fine[fidx];
107         fidx = fposX + fposY * fsizeX + (fposZ+1) * fsizeX * fsizeY;
108         float N_C = fine[fidx];
```

```
109         fidx = fposX + fposY * fsizeX + (fposZ-1) * fsizeX * fsizeY;
110         float S_C = fine[fidx];
111         fidx = (fposX+1) + fposY * fsizeX + fposZ * fsizeX * fsizeY;
112         float E_C = fine[fidx];
113         fidx = (fposX-1) + fposY * fsizeX + fposZ * fsizeX * fsizeY;
114         float O_C = fine[fidx];
115         fidx = (fposX+1) + fposY * fsizeX + (fposZ+1) * fsizeX * fsizeY;
116         float NE_C = fine[fidx];
117         fidx = (fposX-1) + fposY * fsizeX + (fposZ+1) * fsizeX * fsizeY;
118         float NO_C = fine[fidx];
119         fidx = (fposX+1) + fposY * fsizeX + (fposZ-1) * fsizeX * fsizeY;
120         float SE_C = fine[fidx];
121         fidx = (fposX-1) + fposY * fsizeX + (fposZ-1) * fsizeX * fsizeY;
122         float SO_C = fine[fidx];
123
124         fidx = fposX + (fposY-1) * fsizeX + fposZ * fsizeX * fsizeY;
125         float C_N = fine[fidx];
126         fidx = fposX + (fposY-1) * fsizeX + (fposZ+1) * fsizeX * fsizeY;
127         float N_N = fine[fidx];
128         fidx = fposX + (fposY-1) * fsizeX + (fposZ-1) * fsizeX * fsizeY;
129         float S_N = fine[fidx];
130         fidx = (fposX+1) + (fposY-1) * fsizeX + (fposZ) * fsizeX * fsizeY;
131         float E_N = fine[fidx];
132         fidx = (fposX-1) + (fposY-1) * fsizeX + (fposZ) * fsizeX * fsizeY;
133         float O_N = fine[fidx];
134         fidx = (fposX+1) + (fposY-1) * fsizeX + (fposZ+1) * fsizeX * fsizeY;
135         float NE_N = fine[fidx];
136         fidx = (fposX-1) + (fposY-1) * fsizeX + (fposZ+1) * fsizeX * fsizeY;
137         float NO_N = fine[fidx];
138         fidx = (fposX+1) + (fposY-1) * fsizeX + (fposZ-1) * fsizeX * fsizeY;
139         float SE_N = fine[fidx];
140         fidx = (fposX-1) + (fposY-1) * fsizeX + (fposZ-1) * fsizeX * fsizeY;
141         float SO_N = fine[fidx];
142
143         fidx = fposX + (fposY+1) * fsizeX + fposZ * fsizeX * fsizeY;
144         float C_S = fine[fidx];
145         fidx = fposX + (fposY+1) * fsizeX + (fposZ+1) * fsizeX * fsizeY;
146         float N_S = fine[fidx];
147         fidx = fposX + (fposY+1) * fsizeX + (fposZ-1) * fsizeX * fsizeY;
148         float S_S = fine[fidx];
149         fidx = (fposX+1) + (fposY+1) * fsizeX + (fposZ) * fsizeX * fsizeY;
150         float E_S = fine[fidx];
151         fidx = (fposX-1) + (fposY+1) * fsizeX + (fposZ) * fsizeX * fsizeY;
152         float O_S = fine[fidx];
153         fidx = (fposX+1) + (fposY+1) * fsizeX + (fposZ+1) * fsizeX * fsizeY;
154         float NE_S = fine[fidx];
155         fidx = (fposX-1) + (fposY+1) * fsizeX + (fposZ+1) * fsizeX * fsizeY;
156         float NO_S = fine[fidx];
157         fidx = (fposX+1) + (fposY+1) * fsizeX + (fposZ-1) * fsizeX * fsizeY;
158         float SE_S = fine[fidx];
159         fidx = (fposX-1) + (fposY+1) * fsizeX + (fposZ-1) * fsizeX * fsizeY;
160         float SO_S = fine[fidx];
161
162         cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
163         coarse[cidx] = (1/8.0f)*(C_C) + (1/16.0f)*((N_C+E_C+S_C+O_C) + (C_N+C_S)) +
                (1/32.0f)*((NE_C+SE_C+SO_C+NO_C) + (N_N+E_N+S_N+O_N) + (N_S+E_S+S_S+O_S))
                 + (1/64.0f)*((NE_N+SE_N+SO_N+NO_N) + (NE_S+SE_S+SO_S+NO_S));
164
165 }
166
167 __global__ void CUDAInterpolate(float* fine, int d_fsizeXYZ[], float* coarse, int
        d_csizeXYZ[])
168 {
169         int fsizeX = d_fsizeXYZ[0];
170         int fsizeY = d_fsizeXYZ[1];
```

```
171         int fsizeZ = d_fsizeXYZ[2];
172
173         int csizeX = d_csizeXYZ[0];
174         int csizeY = d_csizeXYZ[1];
175
176         int fposY = (threadIdx.x + blockIdx.x*blockDim.x)/fsizeX;
177         int fposX = (threadIdx.x + blockIdx.x*blockDim.x) - fposY*fsizeX;
178         int fposZ = (blockIdx.y * blockDim.y) + threadIdx.y;
179
180         int fidx = fposX + fposY * fsizeX + fposZ * fsizeX * fsizeY;
181
182         int cposX = fposX/2;
183         int cposY = fposY/2;
184         int cposZ = fposZ/2;
185         int cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
186
187         if(fposY >= fsizeY || fposX >= fsizeX || fposZ >= fsizeZ)
188                 return;
189
190         if(fposX == 0 || fposX == fsizeX - 1 || fposY == 0 || fposY == fsizeY - 1 ||
                fposZ == 0 || fposZ == fsizeZ - 1)
191                 return;
192
193         if(fposY%2 == 0 && fposX%2 == 0 && fposZ%2 == 0)
194         {
195                 fine[fidx] = coarse[cidx];
196                 return;
197         }
198
199         if(fposY%2 == 0 && fposX%2 != 0 && fposZ%2 == 0)
200         {
201                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
202                 float O = coarse[cidx];
203                 cidx = (cposX+1) + cposY * csizeX + cposZ * csizeX * csizeY;
204                 float E = coarse[cidx];
205
206                 fine[fidx] = (1/2.0f)*(O + E);
207                 return;
208         }
209
210         if(fposY%2 != 0 && fposX%2 == 0 && fposZ%2 == 0)
211         {
212                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
213                 float N = coarse[cidx];
214                 cidx = cposX + (cposY+1) * csizeX + cposZ * csizeX * csizeY;
215                 float S = coarse[cidx];
216
217                 fine[fidx] = (1/2.0f)*(N + S);
218                 return;
219         }
220
221         if(fposY%2 != 0 && fposX%2 != 0 && fposZ%2 == 0)
222         {
223                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
224                 float NO = coarse[cidx];
225                 cidx = (cposX+1) + cposY * csizeX + cposZ * csizeX * csizeY;
226                 float NE = coarse[cidx];
227                 cidx = cposX + (cposY+1) * csizeX + cposZ * csizeX * csizeY;
228                 float SO = coarse[cidx];
229                 cidx = (cposX+1) + (cposY+1) * csizeX + cposZ * csizeX * csizeY;
230                 float SE = coarse[cidx];
231
232                 fine[fidx] = (1/4.0f)*(NO + NE + SO + SE);
233                 return;
234         }
```

```
235
236         if(fposY%2 == 0 && fposX%2 == 0 && fposZ%2 != 0)
237         {
238                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
239                 float S = coarse[cidx];
240                 cidx = cposX + cposY * csizeX + (cposZ+1) * csizeX * csizeY;
241                 float N = coarse[cidx];
242
243                 fine[fidx] = (1/2.0f)*(S + N);
244                 return;
245         }
246
247         if(fposY%2 == 0 && fposX%2 != 0 && fposZ%2 != 0)
248         {
249                 cidx = cposX + cposY * csizeX + (cposZ+1) * csizeX * csizeY;
250                 float NO = coarse[cidx];
251                 cidx = (cposX+1) + cposY * csizeX + (cposZ+1) * csizeX * csizeY;
252                 float NE = coarse[cidx];
253                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
254                 float SO = coarse[cidx];
255                 cidx = (cposX+1) + cposY * csizeX + cposZ * csizeX * csizeY;
256                 float SE = coarse[cidx];
257
258                 fine[fidx] = (1/4.0f)*(NO + NE + SO + SE);
259                 return;
260         }
261
262         if(fposY%2 != 0 && fposX%2 == 0 && fposZ%2 != 0)
263         {
264                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
265                 float NO = coarse[cidx];
266                 cidx = cposX + cposY * csizeX + (cposZ+1) * csizeX * csizeY;
267                 float NE = coarse[cidx];
268                 cidx = cposX + (cposY+1) * csizeX + cposZ * csizeX * csizeY;
269                 float SO = coarse[cidx];
270                 cidx = cposX + (cposY+1) * csizeX + (cposZ+1) * csizeX * csizeY;
271                 float SE = coarse[cidx];
272
273                 fine[fidx] = (1/4.0f)*(NO + NE + SO + SE);
274                 return;
275         }
276
277         if(fposY%2 != 0 && fposX%2 != 0 && fposZ%2 != 0)
278         {
279                 cidx = cposX + cposY * csizeX + cposZ * csizeX * csizeY;
280                 float USO = coarse[cidx];
281
282                 cidx = cposX + cposY * csizeX + (cposZ+1) * csizeX * csizeY;
283                 float UNO = coarse[cidx];
284
285                 cidx = (cposX+1) + cposY * csizeX + (cposZ+1) * csizeX * csizeY;
286                 float UNE = coarse[cidx];
287
288                 cidx = (cposX+1) + cposY * csizeX + cposZ * csizeX * csizeY;
289                 float USE = coarse[cidx];
290
291
292                 cidx = cposX + (cposY+1) * csizeX + cposZ * csizeX * csizeY;
293                 float DSO = coarse[cidx];
294
295                 cidx = cposX + (cposY+1) * csizeX + (cposZ+1) * csizeX * csizeY;
296                 float DNO = coarse[cidx];
297
298                 cidx = (cposX+1) + (cposY+1) * csizeX + (cposZ+1) * csizeX * csizeY;
299                 float DNE = coarse[cidx];
```

```
300
301                 cidx = (cposX+1) + (cposY+1) * csizeX + cposZ * csizeX * csizeY;
302                 float DSE = coarse[cidx];
303
304                 fine[fidx] = (1/8.0f)*(USO + UNO + UNE + USE  +  DSO + DNO + DNE +
                        DSE);
305
306         }
307
308 }
309
310 __global__ void CUDARelax(float* d_v, float* d_f, float h_x, float h_y, float h_z,
        int d_sizeXYZ[])
311 {
312         int sizeX = d_sizeXYZ[0];
313         int sizeY = d_sizeXYZ[1];
314         int sizeZ = d_sizeXYZ[2];
315
316         float h_x2 = h_x*h_x;
317         float h_y2 = h_y*h_y;
318         float h_z2 = h_z*h_z;
319
320         int posY = (threadIdx.x + blockIdx.x*blockDim.x)/sizeX;
321         int posX = (threadIdx.x + blockIdx.x*blockDim.x) - posY*sizeX;
322         int posZ = (blockIdx.y * blockDim.y) + threadIdx.y;
323
324         int idx = posX + posY * sizeX + posZ * sizeX * sizeY;
325
326
327         if(posY >= sizeY || posX >= sizeX || posZ >= sizeZ)
328                 return;
329
330         if(posX == 0 || posX == sizeX - 1 || posY == 0 || posY == sizeY - 1 || posZ
            == 0 || posZ == sizeZ - 1)
331                 return;
332
333         if((posY+ posX + posZ)%2 == 0)
334         {
335                 idx = (posX-1) + posY * sizeX + posZ * sizeX * sizeY;
336                 float O = d_v[idx];
337                 idx = (posX+1) + posY * sizeX + posZ * sizeX * sizeY;
338                 float E = d_v[idx];
339                 idx = posX + (posY-1) * sizeX + posZ * sizeX * sizeY;
340                 float N = d_v[idx];
341                 idx = posX + (posY+1) * sizeX + posZ * sizeX * sizeY;
342                 float S = d_v[idx];
343                 idx = posX + posY * sizeX + (posZ-1) * sizeX * sizeY;
344                 float D = d_v[idx];
345                 idx = posX + posY * sizeX + (posZ+1) * sizeX * sizeY;
346                 float U = d_v[idx];
347
348                 idx = posX + posY * sizeX + posZ * sizeX * sizeY;
349                 d_v[idx] = (O*(h_y2*h_z2)+E*(h_y2*h_z2) + N*(h_x2*h_z2)+S*(h_x2*h_z2)
                        + D*(h_x2*h_y2)+U*(h_x2*h_y2) - d_f[idx]*h_x2*h_y2*h_z2)/(2*(
                        h_y2*h_z2 + h_x2*h_z2 + h_x2*h_y2));
350         }
351
352         __syncthreads();
353
354         if((posY+ posX + posZ)%2 != 0)
355         {
356                 idx = (posX-1) + posY * sizeX + posZ * sizeX * sizeY;
357                 float O = d_v[idx];
358                 idx = (posX+1) + posY * sizeX + posZ * sizeX * sizeY;
359                 float E = d_v[idx];
```

```
360                 idx = posX + (posY-1) * sizeX + posZ * sizeX * sizeY;
361                 float N = d_v[idx];
362                 idx = posX + (posY+1) * sizeX + posZ * sizeX * sizeY;
363                 float S = d_v[idx];
364                 idx = posX + posY * sizeX + (posZ-1) * sizeX * sizeY;
365                 float D = d_v[idx];
366                 idx = posX + posY * sizeX + (posZ+1) * sizeX * sizeY;
367                 float U = d_v[idx];
368
369                 idx = posX + posY * sizeX + posZ * sizeX * sizeY;
370                 d_v[idx] = (O*(h_y2*h_z2)+E*(h_y2*h_z2) + N*(h_x2*h_z2)+S*(h_x2*h_z2)
                         + D*(h_x2*h_y2)+U*(h_x2*h_y2) - d_f[idx]*h_x2*h_y2*h_z2)/(2*(
                        h_y2*h_z2 + h_x2*h_z2 + h_x2*h_y2));
371         }
372
373 }
374
375 __global__ void CUDASet(float* d_v, int d_sizeXYZ[], float value, bool modifyBorder)
376 {
377         int sizeX = d_sizeXYZ[0];
378         int sizeY = d_sizeXYZ[1];
379         int sizeZ = d_sizeXYZ[2];
380
381         int posY = (threadIdx.x + blockIdx.x*blockDim.x)/sizeX;
382         int posX = (threadIdx.x + blockIdx.x*blockDim.x) - posY*sizeX;
383         int posZ = (blockIdx.y * blockDim.y) + threadIdx.y;
384
385         int idx = posX + posY * sizeX + posZ * sizeX * sizeY;
386
387         if(posY >= sizeY || posX >= sizeX || posZ >= sizeZ)
388                 return;
389
390         if(!modifyBorder)
391                 if(posX == 0 || posX == sizeX - 1 || posY == 0 || posY == sizeY - 1
                        || posZ == 0 || posZ == sizeZ - 1)
392                         return;
393
394         d_v[idx] = value;
395 }
396
397 __global__ void CUDASetTESTTEST(float* d_v, int d_sizeXYZ[], float value, bool
        modifyBorder)
398 {
399         int sizeX = d_sizeXYZ[0];
400         int sizeY = d_sizeXYZ[1];
401         int sizeZ = d_sizeXYZ[2];
402
403         int posY = (threadIdx.x + blockIdx.x*blockDim.x)/sizeX;
404         int posX = (threadIdx.x + blockIdx.x*blockDim.x) - posY*sizeX;
405         int posZ = (blockIdx.y * blockDim.y) + threadIdx.y;
406
407         int idx = posX + posY * sizeX + posZ * sizeX * sizeY;
408
409         if(posY >= sizeY || posX >= sizeX || posZ >= sizeZ)
410                 return;
411
412         d_v[idx] = posX + posY + posZ;
413 }
414
415
416 __global__ void CUDACalculateResidual(float* d_v, float* d_f, float* d_residual,
        float h_x, float h_y, float h_z, int d_sizeXYZ[])
417 {
418         int sizeX = d_sizeXYZ[0];
419         int sizeY = d_sizeXYZ[1];
```

```
420         int sizeZ = d_sizeXYZ[2];
421
422         float h_x2 = h_x*h_x;
423         float h_y2 = h_y*h_y;
424         float h_z2 = h_z*h_z;
425
426
427         int posY = (threadIdx.x + blockIdx.x*blockDim.x)/sizeX;
428         int posX = (threadIdx.x + blockIdx.x*blockDim.x) - posY*sizeX;
429         int posZ = (blockIdx.y * blockDim.y) + threadIdx.y;
430
431         int idx = posX + posY * sizeX + posZ * sizeX * sizeY;
432
433
434         if(posY >= sizeY || posX >= sizeX || posZ >= sizeZ)
435                 return;
436
437         if(posX == 0 || posX == sizeX - 1 || posY == 0 || posY == sizeY - 1 || posZ
                == 0 || posZ == sizeZ - 1)
438         {
439                 d_residual[idx] = 0.0;
440                 return;
441         }
442
443         idx = (posX-1) + posY * sizeX + posZ * sizeX * sizeY;
444         float O = d_v[idx];
445         idx = (posX+1) + posY * sizeX + posZ * sizeX * sizeY;
446         float E = d_v[idx];
447         idx = posX + (posY-1) * sizeX + posZ * sizeX * sizeY;
448         float N = d_v[idx];
449         idx = posX + (posY+1) * sizeX + posZ * sizeX * sizeY;
450         float S = d_v[idx];
451         idx = posX + posY * sizeX + (posZ-1) * sizeX * sizeY;
452         float D = d_v[idx];
453         idx = posX + posY * sizeX + (posZ+1) * sizeX * sizeY;
454         float U = d_v[idx];
455
456         idx = posX + posY * sizeX + posZ * sizeX * sizeY;
457         d_residual[idx] = d_f[idx] - ((O-2*d_v[idx]+E)/h_x2) - ((N-2*d_v[idx]-S)/h_y2
                ) - ((D-2*d_v[idx]-U)/h_z2);
458 }
459
460 __global__ void CUDAApplyCorrection(float* fine, float* error, int d_sizeXYZ[])
461 {
462         int sizeX = d_sizeXYZ[0];
463         int sizeY = d_sizeXYZ[1];
464         int sizeZ = d_sizeXYZ[2];
465
466         int posY = (threadIdx.x + blockIdx.x*blockDim.x)/sizeX;
467         int posX = (threadIdx.x + blockIdx.x*blockDim.x) - posY*sizeX;
468         int posZ = (blockIdx.y * blockDim.y) + threadIdx.y;
469
470         int idx = posX + posY * sizeX + posZ * sizeX * sizeY;
471
472
473         if(posY >= sizeY || posX >= sizeX || posZ >= sizeZ)
474                 return;
475
476         if(posX == 0 || posX == sizeX - 1 || posY == 0 || posY == sizeY - 1 || posZ
                == 0 || posZ == sizeZ - 1)
477                 return;
478
479         fine[idx] = fine[idx] + error[idx];
480 }
```

Elenco delle figure

Bibliografia

[1] William L. Briggs - Van Emden Henson - Stephen Fahrney McCormick, *"A MultiGrid Tutorial, Second Edition"*, Siam, 2000.

[2] Dragica Vasileska, *"Multi-Grid Method "*, Slide, 2009.

[3] NVIDIA Corporation, *"NVIDIA CUDA C Programming Guide Version 3.2"*, 2010.

[4] Roberto Tauraso, *"Analisi Matematica 2"*, Dispense, 2009.

[5] Osvaldo Maria Grasselli - Laura Menini - Sergio Galeani, *"Sistemi Dinamici"*, Ulrico Hoepli, 2008.

[6] Daniel Liberzone, *"Calculus of variations and optimal control theory, A Concise Introduction"*, 2010.

[7] Martino Bardi, Italo Capuzzo-Dolcetta, *"Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman Equations"*, Birkhäuser, 2008.