Universitá degli Studi di Napoli
"Parthenope"

Scuola interdipartimentale delle Scienze, dell'Ingegneria e della Salute

Dipartimento di Scienze e Tecnologie

Corso di laurea in Informatica

Tesi di laurea in Calcolo Parallelo e Distribuito

SIMULAZIONE NUMERICA SU GPU DI UN MODELLO DI VEGETAZIONE

NUMERICAL SIMULATION OF A VEGETATION MODEL ON GPU

Relatore:
Prof.ssa Livia Marcellino

Correlatore:
Prof. Pasquale De Luca

Candidato:
Vincenzo Iannucci
Matricola 0124002093

Anno Accademico 2021-2022

Ho deciso di dedicare questo spazio dell'elaborato alle persone che mi sono state accanto nel corso del mio percorso di studi. In primis, un grazie speciale al mio relatore Livia Marcellino e correlatore Pasquale De Luca per gli indispensabili consigli e le conoscenze che mi hanno trasmesso durante la stesura della tesi. Ringrazio i miei compagni di corso Mario, Pasquale e Michele per aver condiviso parte del percorso universitario con me. Ringrazio i miei amici, Alessandro, Fabrizio, Attilio. In questi anni ho imparato a conoscervi e meglio e mi sono reso conto di quanto siete delle persone fantastiche e di quanto la nostra amicizia sia speciale. Grazie per avermi regalato momenti di allegria, emozione e condivisione indimenticabili, che mi porterò nel cuore per sempre. Un grazie speciale a Lorenzo, che con la sua dolcezza e pazienza mi ha insegnato ad essere meno melodrammatico e più sorridente. Il tuo sostegno mi è stato fondamentale in alcuni frangenti e questo non lo dimenticherò mai, sei una persona speciale. Grazie a mia madre e alla mia famiglia per aver accolto le mie scelte ed avermi sostenuto in tutti questi anni, non sempre facili. Desidero inoltre dedicare questo lavoro alla memoria di mio nonno, che mi ha insegnato il valore dell'impegno e della determinazione. Anche se non è più con noi, so che sarebbe stato fiero di me per aver raggiunto questo traguardo. Mi sento in dovere di dedicare questo lavoro a chi purtroppo sempre più spesso decide di farla finita per aver fallito durante il percorso universitario. Se c'è una cosa che ho imparato è che la vita è un bene prezioso e non va sprecata a causa di uno o più fallimenti. Essi sono una naturale conseguenza della vita e vanno accettati, anche se a volte può risultare difficile e doloroso farlo. Con il tempo ho imparato che solo fallendo e cadendo in basso si può risalire ed arrivare in alto. Infine, grazie a me stesso, per non essermi arreso, per aver imparato dai fallimenti e per essere stato costante nello studio. Se sono arrivato fin qui oggi è anche merito delle tante notti insonni passate sui libri e dei sacrifici che ho fatto.

Sommario

Lo scopo della tesi è proporre uno schema parallelo per l'accelerazione di un algoritmo numerico che studia un problema di vegetazione in zone semiaride, modellato da un sistema di equazioni differenziali alle derivate parziali (PDE). Per farlo si è dapprima studiato il modello matematico proposto e la sua discretizzazione per il problema esposto. Successivamente, è stato formulato un algoritmo sequenziale che risolve il problema ed infine è stato implementato il software sequenziale. Dall'analisi delle prestazioni del software sequenziale e dalle successive osservazioni è stato progettato un algoritmo parallelo con una apposita strategia di parallelizzazione, che mira a calcolare contemporaneamente gli stage dell'equazione relativa alla discretizzazione temporale della PDE. Il software parallelo è stato implementato utilizzando l'ambiente CUDA per le GPU (Graphics Processing Unit) NVIDIA. La tesi è corredata di esperimenti numerici che mostrano il guadagno in termini di prestazioni della strategia proposta.

Abstract

The purpose of the thesis is to propose a parallel scheme for the acceleration of a numerical algorithm that studies a vegetation problem in semi-arid areas, modeled by a system of Partial Differential Equation (PDE). To do that, the proposed mathematical model and its discretization for the problem were first studied. Subsequently, a sequential algorithm that solves the problem was formulated and finally the sequential software was implemented. From the analysis of the performance of the sequential software and from the subsequent observations, a parallel algorithm was designed with a suitable parallelization strategy, which aims to simultaneously calculate the stages of the equation relating to the time discretization of the PDE. The parallel software was implemented using the CUDA environment for NVIDIA Graphics Processing Units (GPUs). Numerical experiments, showing the performance gain of the proposed strategy, are provided.

Introduzione

Il lavoro svolto si concentra sulla soluzione numerica efficiente di ***PDE (Partial Differential Equation)*** che modellano un particolare problema di vegetazione, per mezzo di metodi numerici paralleli che utilizzano ***GPU (Graphics Processing Unit)***.
Più precisamente, verrà preso in considerazione un sistema di PDE che rappresenta un modello di vegetazione in zone climatiche semi-aride, e costituisce un esempio del principio di autorganizzazione in ecologia. Tale sistema è stato introdotto per studiare come due specie erbacee competono per la stessa risorsa limitata (l'acqua) riuscendo a sopravvivere. [5]
Per risolvere numericamente questo tipo di PDE, che è funzione spazio-tempo, è necessario discretizzare spazialemnte il problema generando così un sistema di ***ODE (Ordinary Differential Equation)***, che deve essere poi discretizzato rispetto al tempo. Una risoluzione efficiente prevede che il metodo impiegato non sia troppo costoso, cioè che coinvolga il minor numero possibile di valutazioni di funzioni e inversioni di matrici. Dunque, bisogna scegliere un metodo che sia numericamente stabile e non sia pesante computazionalmente.
Un buon compromesso nella scelta del metodo di discretizzazione rispetto al tempo è rappresentato da una classe di metodi espliciti chiamati ***peer methods***. I peer methods sono metodi numerici basati su più passi (stage) che fanno parte della classe dei ***General Linear Methods (GLM)*** [12] e hanno delle proprietà per cui il loro uso può risultare molto conveniente. Infatti, in questi metodi, tutte le fasi sono calcolate con la stessa accuratezza della soluzione che avanza, e questo assicura che i peer methods non soffrano di riduzione dell'ordine di accuratezza della soluzione finale.
L'attività svolta si è concentrata in primo luogo sulla scrittura di un algoritmo basato sul modello numerico che dati in input i parametri del modello restituisca in output la soluzione numerica del sistema (1.1). Tale algoritmo è stato poi

implementato in linguaggio di programmazione C, in un ambiente di calcolo sequenziale, organizzando l'algoritmo in più macromoduli. Successivamente, si sono osservate le prestazioni del software e individuate le parti più critiche per cui fosse utile e necessario pensare ad una strategia di parallelizzazione. È stato quindi formulato un algoritmo parallelo, con relativa strategia di parallelizzazione, la quale mira proprio ad eseguire in parallelo i due stage (passi) dell'equazione relativa al calcolo dei peer methods. L'algoritmo parallelo è stato poi è stato implementato in ambiente ad alte prestazioni, sfruttando le architetture ***GPU (Graphics Processing Units)*** e impiegando l'ambiente ***CUDA*** [14]. Ricordiamo che oggigiorno la scelta delle GPU diventa quasi obbligatoria grazie alla possibilità che offrono di risolvere problemi di grandi dimensioni, mediante opportuni algoritmi paralleli, in un tempo di esecuzione ridotto. Sono state usate le API in linguaggio C che CUDA mette a disposizione per la creazione del software parallelo. La scelta del linguaggio C è da ascriversi non solo al fatto che è stato ampiamente trattato durante il corso di studi ma anche al fatto che offre un controllo a basso livello della memoria, permettendo di utilizzare in maniera efficiente le risorse delle GPU. Infine, sono state analizzate e confrontate le performance osservando come il software parallelo abbia prestazioni nettamente migliori rispetto al sequenziale.

Il resto della tesi è organizzata come segue. Nel capitolo 1 verrà presentato il problema di vegetazione, il modello matematico e i metodi di discretizzazione impiegati per passare al modello numerico. Verranno presentati in maniera più approfondita i peer methods e infine l'algoritmo sequenziale che risolve il problema di vegetazione. Il capitolo si conclude con i test effettuati sull'implementazione sequenziale dell'algoritmo. Nel capitolo 2 si parlerà delle GPU e dell'ambiente CUDA, con qualche esempio di codice in linguaggio C e una breve presentazione delle funzioni di libreria più usate. Nel capitolo 3 si discuterà della strategia di parallelizzazione scelta e dell'algoritmo parallelo. Nel capitolo 4 verranno effettuati i test del codice parallelo e confrontati con i risultati ottenuti dalla controparte sequenziale, confermando l'efficienza dell'approccio parallelo.

Indice

Elenco delle figure

Elenco degli algoritmi

Listings

Elenco delle tabelle

Capitolo 1

Un problema di vegetazione

Nel seguente capitolo verrà trattato un procedimento di discretizzazione che consente di passare dal modello matematico, che rappresenta il problema di vegetazione in esame, alla formulazione di un algoritmo per l'ambiente sequenziale e alla sua implementazione in ambiente CPU standard.

1.1 Il modello matematico

Consideriamo il seguente sistema di PDE:

$$\begin{cases} \dfrac{\partial u_1}{\partial t} = \dfrac{\partial^2 u_1}{\partial x^2} + w u_1(u_1 + H u_2) - B_1 u_1 - S u_1 u_2, \\ \dfrac{\partial u_2}{\partial t} = D\dfrac{\partial^2 u_2}{\partial x^2} + F w u_2(u_1 + H u_2) - B_2 u_2, \\ \dfrac{\partial w}{\partial t} = d\dfrac{\partial^2 w}{\partial x^2} + A - w - w(u_1 + u_2)(u_1 + H u_2). \end{cases} \tag{1.1}$$

Ove $(x, t) \in [x_0, X] \times [t_0, T]$, con le seguenti condizioni iniziali:

$$\begin{cases} u_1(x, t_0) = U_{1,0}(x), \\ u_2(x, t_0) = U_{2,0}(x), \\ w(x, t_0) = W_0(x). \end{cases} \tag{1.2}$$

Esso rappresenta un modello di vegetazione in zone climatiche semi-aride ed è stato introdotto per studiare come due specie erbacee competono per la stessa risorsa limitata (l'acqua) riuscendo a sopravvivere. Per capire meglio come funziona il modello, si può immaginare che la funzione u_1 descriva la densità dell'erba

presente sul suolo x all'istante t, che la funzione u_2 rappresenti la stessa cosa ma per gli alberi, e che w sia la quantità di acqua disponibile. La coesistenza delle due specie erbacee è uno stato metastabile, ovvero si presenta come soluzione stabile del sistema per tempi molto lunghi. Tuttavia, per $t \to \infty$ una delle due piante muore.

1.2 Discretizzazione spaziale

Consideriamo l'equazione (1.1). Il primo passo consiste nell'approssimare numericamente le derivate parziali rispetto allo spazio utilizzando il **MOL (Method Of Lines)**, che usando il metodo delle differenze finite centrali del secondo ordine (***central finite differences***) permette di dividere l'intervallo spaziale $[x_0, X]$ in $M-1$ sottointervalli:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}, \quad u = u_1, u_2, w, \quad \Delta x = \frac{X - x_0}{M-1}$$

Dunque si passa da un sistema di PDE, costituito da derivate parziali e avente come variabili indipendenti il tempo e lo spazio ad un sistema di ODE, costituito da derivate ordinarie:

$$\begin{cases} \dfrac{\partial U_1}{\partial t} = \dfrac{1}{\Delta x^2} L_{Diff} U_1 + F_1 \\ \dfrac{\partial U_2}{\partial t} = \dfrac{D}{\Delta x^2} L_{Diff} U_2 + F_2 \\ \dfrac{\partial W}{\partial t} = \dfrac{d}{\Delta x^2} L_{Diff} W + F_3 \end{cases} \tag{1.3}$$

Per semplicità di notazione, è stata omessa la dipendenza dal tempo di tutte le funzioni coinvolte. La notazione usata è la seguente:

$$x_m = x_0 + m\Delta x; \quad m = 0, \ldots, M-1; \quad x_{M-1} = X \tag{1.4}$$

e:

$$u_1^m = u_1(x_m, t), \quad u_2^m = u_2(x_m, t), \quad w^m = w(x_m, t)$$

$$U_1 = (u_1^m(t))_{m=0}^{M-1}, \quad U_2 = (u_2^m(t))_{m=0}^{M-1}, \quad W = (w^m(t))_{m=0}^{M-1}$$

con $U_1, U_2, W \in \mathbb{R}^M$.

$$\frac{\partial U_1}{\partial t} = \Big(\frac{\partial u_1^m}{\partial t}\Big)_{m=0}^{M-1}, \quad \frac{\partial U_2}{\partial t} = \Big(\frac{\partial u_2^m}{\partial t}\Big)_{m=0}^{M-1}, \quad \frac{\partial W}{\partial t} = \Big(\frac{\partial w^m}{\partial t}\Big)_{m=0}^{M-1}$$

$$F_1 = (w^m u_1^m (u_1^m + H u_2^m) - B_1 u_1^m - S u_1^m u_2^m)_{m=0}^{M-1}$$
$$F_2 = (F w^m u_2^m (u_1^m + H u_2^m) - B_2 u_2^m)_{m=0}^{M-1}$$
$$F_3 = (A - w^m - w^m (u_1^m + u_2^m)(u_1^m + H u_2^m))_{m=0}^{M-1}$$

Al fine di completare la procedura di discretizzazione spaziale, se si assume che u_1, u_2, w siano nulli al di fuori del loro dominio, risulta che la matrice L_{Diff} è tridiagonale ed è data da:

$$L_{Diff} = \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix} \in \mathbb{R}^{M,M} \tag{1.5}$$

Utilizzando condizioni al contorno periodiche, la matrice L_{Diff} deve essere leggermente modificata, ponendo $L_{Diff}(1, M) = L_{Diff}(M, 1) = 1$.

Infine, compattando ulteriormente il sistema di ODE discrete (1.3), si ottiene:

$$y'(t) = L \cdot y(t) + NL(y(t)) \tag{1.6}$$

dove:

$$y(t) = [U_1, U_2, W]^T \in \mathbb{R}^{3M} \quad \text{e} \quad NL(y(t)) = [F_1, F_2, F_3]^T \tag{1.7}$$

mentre la matrice L assume la forma:

$$L = \frac{1}{\Delta x^2} \begin{pmatrix} L_{Diff} & 0 & 0 \\ 0 & DL_{Diff} & 0 \\ 0 & 0 & dL_{Diff} \end{pmatrix} \in \mathbb{R}^{3M,3M}. \tag{1.8}$$

Il passo successivo mira a discretizzare il sistema di ODE (1.6) mediante peer methods.

1.3 Discretizzazione rispetto al tempo con peer methods

L'equazione (1.6) rappresenta un sistema di ODE in forma vettoriale. Per discretizzare rispetto al tempo il sistema di ODE si può fare affidamento su due tipi principali di metodi: **espliciti** e **impliciti**. Nei metodi espliciti, la soluzione al passo temporale successivo viene stimata direttamente dai valori della soluzione al passo temporale corrente. Nei metodi impliciti, la soluzione al passo temporale successivo viene stimata risolvendo un'equazione non lineare che coinvolge sia il passo temporale corrente che quello successivo. La scelta del metodo di discretizzazione temporale **dipende dal problema specifico da risolvere**, nonché dal compromesso desiderato tra accuratezza ed efficienza computazionale. Per risolvere il sistema di ODE sarebbe preferibile utilizzare un metodo che non sia troppo costoso, cioè che coinvolga il minor numero possibile di valutazioni di funzioni e inversioni di matrici. Dunque, bisogna scegliere un metodo che sia numericamente stabile e non sia pesante computazionalmente. Di solito si fa uso di metodi impliciti come quello di ***Runge-Kutta*** [10, 11], Rosenbrock o IMEX. Tuttavia i metodi implici richiedono ad ogni passo di integrazione e per ogni stage di calcolare un sistema di equazioni lineari o non lineari, che richiedono di eseguire numericamente delle inversioni di matrici la cui dimensione è proporzionale alla dimensione del problema. Dunque, i metodi impliciti sono sì costosi ma sono gli unici che soddisfano i requisiti di stabilità richiesti dal problema. D'altro canto, i metodi espliciti non comportano la risoluzione di sistemi di equazioni, ma richiedono di usare un intervallo temporale molto piccolo e per questo motivo tali metodi impiegano un tempo considerevolmente elevato per convergere. Un buon compromesso tra accuratezza ed efficienza computazionale è rappresentato da una classe di metodi espliciti denominata **peer methods** [[15]-[24]]. I peer methods sono una classe di metodi numerici basati su più passi (stage) che risolvono sistemi di ODE del primo ordine, nella forma generale di Cauchy $y(t) = f(t, y(t))(t \in [t_0, T])$ con condizione iniziale $y_0 = y(t_0) \in \mathbb{R}^d$. Si consideri la discretizzazione temporale:

$$t_n = t_0 + nh; \quad n = 0 \dots N; \quad t_N = T \tag{1.9}$$

dell'intervallo di integrazione $[t_0, T]$ relativo alla (1.1), e quindi alla (1.6). I peer methods basati su stage, espliciti, con s-stage e con dimensione del passo fissa h, hanno la seguente forma:

$$Y_{n,i} = \sum_{j=1}^{s} b_{ij} Y_{n-1,j} + h \sum_{j=1}^{s} a_{ij} f(t_{n-1,j}, Y_{n-1,j}), \quad n = 0, \ldots, N-1 \tag{1.10}$$

$$Y_{n,i} \approx y(t_{n,i}), \qquad t_{n,i} = t_n + hc_i \qquad i = 1, \ldots, s.$$

dove c_i sono i nodi in $[0, 1]$. Quindi, la soluzione progressiva $Y_{n,s}$ è l'approssimazione numerica di $y(t_n + h)$, cioè l'ultima fase calcola la soluzione numerica nei punti della griglia. I coefficienti nelle matrici $A = (a_{ij})_{i,j=1}^{s}$ e $B = (b_{ij})_{i,j=1}^{s}$ caratterizzano il peer methods utilizzato. Per scegliere i coefficienti delle matrici dobbiamo richiamare qualche risultato teorico. Per prima cosa, i peer methods si dicono ottimisticamente zero stabili imponendo:

$$(b_{ij})_{i,j=1}^{s-1} = 0, \qquad \text{e} \qquad b_{is} = 1, \forall i = 1, \ldots, s.$$

Questa scelta è legata alla stabilità dei peer methods vicino all'origine [6]. Inoltre, i coefficienti della matrice A vengono assegnati imponendo l'ordine di consistenza dei peer methods [6], ovvero annullando un numero necessario di residui, definiti come:

$$h\Delta_i := y(t_{n,i}) - \sum_{j=1}^{s} b_{ij} y(t_{n-1,j}) - h \sum_{j=1}^{s} a_{ij} y'(t_{n-1,j}), \qquad i = 1, \ldots, s.$$

dove y si estende su opportune basi polinomiali. Si noti che l'i-esimo residuo misura l'errore tra l'i-esimo stage e il suo valore esatto. Quindi, assumendo peer methods a s-stage di ordine $p = s$, i coefficienti $(a_{ij})_{i,j=1}^{s}$ devono soddisfare la relazione [6]:

$$A = (CV_0 D^{-1}) V_1^{-1} - B(C - I_s) V_1 D^{-1} V_1^{-1}$$

dove $V_0 = (c_i^{j-1})_{i,j=1}^{s}, V_1 = ((c_i - 1)^{j-1})_{i,j=1}^{s}, C = diag(c_i), D = diag(1, \ldots, s)$ e I_s è la matrice identità di dimensione s. In particolare, nello schema proposto, si sceglie di utilizzare i valori per $s = 2$ per quanto riguarda il numero di stage. Con queste condizioni, otteniamo:

$$A = \begin{pmatrix} 0 & 0 \\ -1/2 & 3/2 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}, \quad (c_1, c_2) = (0, 1) \tag{1.11}$$

per $s = 2$.

Indicando con $\mathcal{F}$ le valutazioni delle funzioni degli stage in ogni punto discreto $t_{n,i}$, cioè,

$$\mathcal{F}(Y^{[n]}) = (f(t_{n,1}, Y_{n,1}))$$
$$\mathcal{F}(Y^{[n]}) = (f(t_{n,2}, Y_{n,2}))$$

è possibile rappresentare i peer methods (1.10) in forma vettoriale:

$$Y^{[n]} = (B \otimes I_d)Y^{[n-1]} + h(A \otimes I_d)\mathcal{F}(Y^{[n-1]}) \tag{1.12}$$

dove d la dimensione della soluzione e I_d è la matrice identità di ordine d. Infine, possiamo derivare l'intera procedura numerica ponendo nella formula (1.12) $d = 3M$ e:

$$\mathcal{F}(Y^{[n-1]}) = (LY_{n-1,1} + NL(Y_{n-1,1}))$$
$$\mathcal{F}(Y^{[n-1]}) = (LY_{n-1,2} + NL(Y_{n-1,2})) \tag{1.13}$$

dove L e NL sono come in (1.7) e (1.8), e applicando lo schema al problema di vegetazione nella forma ODE (1.6).
Per capire meglio come i peer methods possono essere implementati prima in sequenziale e poi in parallelo, li si può riscrivere in maniera più estesa come segue:

$$\begin{aligned} Y_{n,1} &= b_{11}Y_{n-1,1} + \ldots + b_{1s}Y_{n-1,s} + ha_{11}f(t_{n-1,1}, Y_{n-1,1}) + \ldots + ha_{1s}f(t_{n-1,s}, Y_{n-1,s}), \\ Y_{n,2} &= b_{21}Y_{n-1,1} + \ldots + b_{2s}Y_{n-1,s} + ha_{21}f(t_{n-1,1}, Y_{n-1,1}) + \ldots + ha_{2s}f(t_{n-1,s}, Y_{n-1,s}). \end{aligned} \tag{1.14}$$

Da questa scrittura risulta evidente che tutti gli stage nell'intervallo $[t_n, t_n + h]$ sono funzione di tutti gli stage nell'intervallo $[t_n - h, t_n]$. Quindi la soluzione progressiva $y(t_n + h) \approx Y_{n,s}$.

1.4 Algoritmo sequenziale

Le discussioni precedenti ci consentono di introdurre un algoritmo che richiama i principali passaggi numerici sopra descritti. L'algoritmo 1 mostra le operazioni necessarie per trovare la soluzione numerica del problema descritto, basata sulla discretizzazione discussa precedentemente.

Algorithm 1 Algoritmo sequenziale peer methods

Input $s, x_0, X, N, t_0, T, M, y_0, a, B_1, B_2, H, F, S, d, D, k, \Delta t$. **Output** Y

// **Initialization**

1: $t\_span = [t_0, T]$
2: time discretization $t_n (n = 0, \ldots, N)$
3: $N = (t\_span[2] - t\_span[1])/\Delta t$
4: $x\_span = [x_0, X]$
5: space discretization $x_m (m = 0, \ldots, M-1)$
6: $\Delta x = (x\_span[2] - x\_span[1])/(M-1)$

// **Spatial discretization**

7: defineLMatrix$(L, \Delta x)$

// **Time discretization by peer methods**

8: $s = 2$ // set the number of stages

// Initialization with time step $n = 0$

9: `set` $t_{0,1} = t_0 + h \cdot c_1$
10: `set` $t_{0,2} = t_0 + h \cdot c_2$
11: `compute` $Y_{0,1}$ and $Y_{0,2}$ using Runge-Kutta 4th order method
12: `evaluate` $\mathcal{F}((Y_{0,1}))$ and $\mathcal{F}((Y_{0,2}))$ as in (1.13)

// Main loop: loop on time steps

13: **for** $n = 1, \ldots, N$ **do**
14: `set` $t_{n,1} = t_n + h \cdot c_1$
15: `set` $t_{n,2} = t_n + h \cdot c_2$
16: `compute` $Y_{n,1}$ and $Y_{n,2}$ as in (1.14)
17: `evaluate` $\mathcal{F}((Y_{n,1}))$ and $\mathcal{F}((Y_{n,2}))$ as in (1.13)
18: **end for**

- `Righe 2-7`: l'algoritmo inizializza i parametri necessari per la discretizzazione del tempo e dello spazio. I parametri iniziali sono:
  - t_0 tempo iniziale;
  - T tempo finale;
  - x_0 spazio iniziale;
  - X spazio finale;

  - M dimensione della griglia spaziale;
  - N dimensione della griglia temporale;
  - k indice che rappresenta la k-esima diagonale, se posto a 0 rappresenta la diagonale principale;
  - Δt distanza tra un valore e l'altro dell'intervallo temporale discretizzato;
  - s numero di stage;
  - a, B_1, B_2, H, F, S parametri di $NL(y(t))$ dell'equazione (1.6);
  - d, D parametri di $L \cdot y(t)$ dell'equazione (1.6).

- `Riga 9`: `defineLMatrix` rappresenta il primo macro-modulo dell'algoritmo sequenziale. Per discretizzare rispetto alla coordinata spaziale, dobbiamo definire la matrice tridiagonale a blocchi L (1.8) come segue:

  1. Definire la matrice identità `eye` di dimensione $M \times M$
  2. Definire due matrici $M \times M$ con la $k+1$ e $k-1$ diagonale unitaria, dove $k = 0$ rappresenta la diagonale principale.
  3. Sommare le tre matrici calcolate ai passi precedenti per ottenere la matrice L_{Diff} (1.5)
  4. Impostare $L_{Diff}(M, 1) = L_{Diff}(1, M) = 1$
  5. Moltiplicare la matrice L_{Diff} per $1/\Delta x^2$
  6. Moltiplicare L_{Diff} rispettivamente per gli scalari D e d (1.15) passati in input, ottenendo le matrici DL_{Diff} e dL_{Diff}
  7. Costruire la matrice tridiagonale a blocchi L avente sulla diagonale principale le matrici L_{Diff}, DL_{Diff} e dL_{Diff}

- `Righe 11-22`: dopo aver discretizzato rispetto la coordinata spaziale adesso non ci resta che discretizzare rispetto al tempo grazie all'utilizzo dei peer methods. Per prima cosa si calcola il valore di Y quando $n = 0$ e per tutti gli stage s, ossia $Y_{0,i} \quad (i = 1 \dots s)$, usando il metodo esplicito ad un passo di Runge-Kutta del quarto ordine, che garantisce che il peer methods mantenga lo stesso ordine di accuratezza quando calcola il successivo valore $Y_{n,i}$. Successivamente, per poter applicare l'equazione (1.10) e quindi (1.12),

bisogna definire la funzione $\mathcal{F}(Y^{[n]})$ come in (1.13). Al passo $n+1$, questa funzione diverrà $\mathcal{F}(Y^{[n-1]})$ e potrà essere usata nella formula del peer methods. Infatti alle righe `18-22` si trova il loop principale del metodo, che altro non fa che eseguire le stesse operazioni eseguite alle righe `14-16`, stavolta per $n = 1 \dots N$.

Si osserva che, una piccola dimensione del passo di discretizzazione comporta una maggiore dimensione delle griglie spazio-temporali discretizzate. Si avrà dunque un notevole aumento della complessità computazionale della procedura, che condurrà inevitabilmente a tempi di esecuzioni via via più grandi per quanto riguarda l'intero software. Nella sezione 1.5 verranno analizzati i tempi derivanti dall'esecuzione dell'software sequenziale al variare di N.

1.5 Test sequenziale

Nella sezione 1.4 è stata analizzata l'idea dietro l'algoritmo sequenziale che successivamente è stato implementato in linguaggio C. I risultati dei test svolti verranno analizzati in questa sezione.

Di seguito si riporta in una tabella i tempi per i due macro-moduli al variare di N per quanto riguarda l'algoritmo sequenziale. I valori dati in input all'algoritmo sono:

$$t_0 = 0, \quad T = 50, \quad x_0 = -50, \quad X = 50, \quad M = 64, \quad d = 500 \tag{1.15}$$

$$a = 1.5, \quad B_1 = 0.45, \quad B_2 = 0.3611, \quad F = H = D = 0.802, \quad S = 0.0002.$$

Come si può evincere dalla tabella, i tempi per quanto riguarda il primo macromodulo, che si occupa di definire la matrice L, hanno un andamento quasi costante. Questo è dovuto in primo luogo al fatto che le vere e proprie operazioni di calcolo sono poche e si tratta per lo più di operazioni di definizione e in secondo luogo al fatto che il primo macromodulo dipende solo da M che è costante. Per quanto riguarda il secondo macromodulo, quello relativo all'applicazione del peer method, i tempi degradono all'aumentare di N in quanto esso dipende fortemente da quest'ultimo valore. Si può concludere affermando che il macromodulo "più pesante" in termini computazionali è proprio il secondo.

	Execution times (s)		
N	defineLMatrix module time	PeerMethods module time	Total time
1×10^5	1.1970×10^{-3}	5.7743×10^1	5.7744×10^1
2×10^5	1.1970×10^{-3}	1.1534×10^2	1.1535×10^2
4×10^5	1.1954×10^{-3}	2.3117×10^2	2.3118×10^2
8×10^5	1.2000×10^{-3}	4.6446×10^2	4.6445×10^2
1×10^6	-	-	-

Tabella 1.1: Confronto tempi dell'algoritmo sequenziale dei vari macro-moduli e tempo totale di esecuzione per $M = 64$ e N variabile

	Execution times (s)			
N	RungeKutta4th sub-module time	Sherratt sub-module time	computeY sub-module time	PeerMethods module time
1×10^5	3.2775×10^{-3}	2.7116×10^{-4}	1.2223×10^{-5}	5.7744×10^1
2×10^5	3.0250×10^{-3}	2.7069×10^{-4}	1.2236×10^{-5}	1.1535×10^2
4×10^5	3.180×10^{-3}	2.7110×10^{-4}	1.2708×10^{-5}	2.3118×10^2
8×10^5	2.8115×10^{-3}	2.7243×10^{-4}	1.2714×10^{-5}	4.6445×10^2
1×10^6	-	-	-	-

Tabella 1.2: Confronto tempi dei sottomoduli del macro-modulo relativo al calcolo dei peer methods per $M = 64$ e N variabile

Poiché il secondo macromodulo è quello più pesante, ciò che è stato fatto è parallelizzarlo sfruttando il potere computazionale delle GPU e la programmazione parallela in ambiente CUDA, di modo da riucire a ridurre il tempo di esecuzione totale del software. Nel capitolo successivo verranno presentate le GPU e l'ambiente CUDA e nel capitolo 3 verrà spiegato l'approccio usato per parallelizzare il secondo macromodulo.

Capitolo 2

GPU (Graphics Processing Unit)

2.1 Cosa sono le GPU

Con il termine ***GPU (Graphics Processing Unit)*** si intendono dei microprocessori [1] nati per essere adoperati principalmente nel campo della ***computer grafica*** [2]: rendering e operazioni grafiche pricipalmente. Grazie alla loro capacità di eseguire calcoli in parallelo, ben presto furono utilizzate anche in applicazioni di tipo ***general purpose***, ossia applicazioni che non coinvolgessero eslusivamente solo il campo della computer grafica. In particolar modo, data la loro elevata capacità computazionale, le GPU si prestano molto bene in ambito scientifico, dove sono utilizzate per risolvere problemi particolarmente complessi che richiederebbero tempi considerevolmente lunghi per essere risolti dai normali processori.

2.2 Architettura GPU vs architettura CPU

Le ***GPU (Graphics Processing Unit)*** sono specializzata per il calcolo ad alta intensità e altamente parallelo - esattamente ciò che riguarda il rendering grafico - e quindi progettata in modo tale che più transistor siano dedicati all'elaborazione

[1] Tipologia particolare di processore; più precisamente è un circuito elettronico dedicato all'elaborazione di istruzioni di dimensioni molto ridotte

[2] Insieme di tecniche e algoritmi informatici per la generazione e la modifica di immagini e video digitali.

dei dati piuttosto che alla memorizzazione nella cache di quest'ultimi e al controllo del flusso, come accade ad esempio nelle ***CPU (Central Processing Unit)*** [3], rendendo difatti più veloci ed efficienti le GPU rispetto alle CPU. Questa

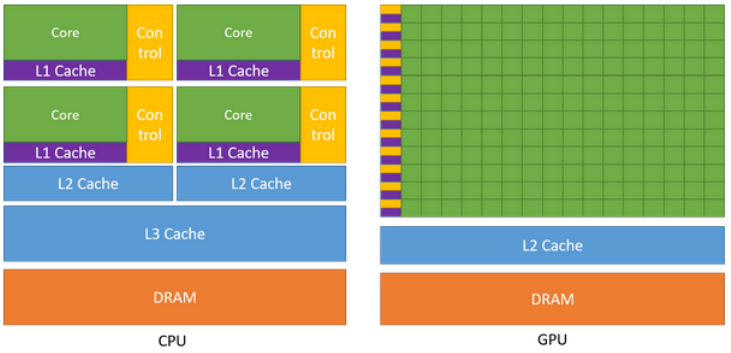

Figura 2.1: Architettura CPU vs architettura GPU

differenza di capacità tra la GPU e la CPU esiste perché sono progettate con obiettivi diversi in mente. Mentre la CPU è progettata per eccellere nell'eseguire una sequenza di operazioni, chiamate ***thread***, il più velocemente possibile e può eseguire alcune decine di questi thread in parallelo, la GPU è progettata per eccellere nell'eseguirne migliaia in parallelo. In generale, un'applicazione ha un mix di parti parallele e parti sequenziali, quindi i sistemi sono progettati con un mix di GPU e CPU per massimizzare le prestazioni complessive. Le applicazioni con un elevato grado di parallelismo possono sfruttare la natura massicciamente parallela della GPU per ottenere prestazioni più elevate rispetto alla CPU[1].

2.3 Il ruolo di NVIDIA

Nel 1999 NVIDIA inventa l'unità di elaborazione grafica o ***GPU (Graphics Processing Unit)***, un'idea che rivoluzionerà l'intero settore. Il lancio di GeForce 256 la descrive come prima GPU del mondo, coniando un termine che NVIDIA definisce come *"un processore a chip singolo con motori integrati di trasformazione, illuminazione, creazione/clipping di triangoli e rendering in grado di elaborare*

[3]Componente di un calcolatore (detta anche processore) che carica le istruzioni dei programmi in memoria, le interpreta e manipola i dati di conseguenza.

un minimo di 10 milioni di poligoni al secondo". Le attuali GPU elaborano oltre 7 miliardi di poligoni al secondo[2]. Nel corso degli anni, NVIDIA si è affermata come azienda leader nel settore, sviluppando architetture per le GPU via via più complesse e performanti. Oggi NVIDIA opera principalmente nel campo della ***computer vision*** [4], come ad esempio l'addestramento di veicoli per la guida autonoma.

2.4 Nvidia CUDA (Compute Unified Device Architecture)

Nel 2006 NVIDIA crea ***CUDA (Compute Unified Device Architecture)***[2], un'architettura sviluppata con lo scopo di utilizzare le proprie GPU dalla serie GeForce 8 in poi per eseguire eaborazioni che non siano quelle tradizionalmente legate all'ambito della computer grafica. In questo modo, le GPU diventano totalmente programmabili e viene aggiunto il supporto a linguaggi di programmazione ad alto livello come ad esempio C e C++. Grazie all'ambiente di sviluppo CUDA è possibile utilizzare specifiche API per programmare le GPU. Ovviamente, ci sono delle specifiche API per ogni linguaggio supportato da CUDA.

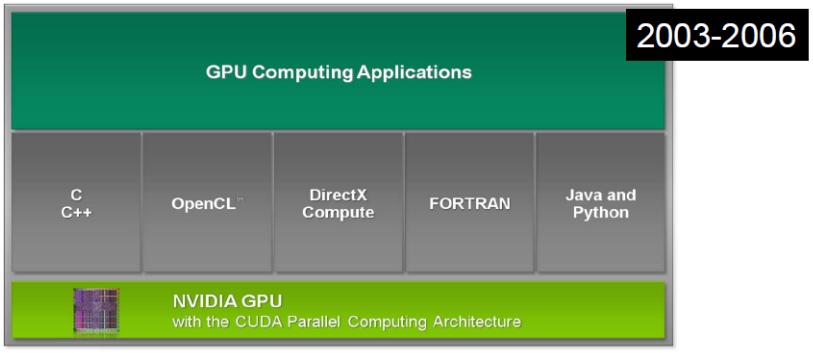

Figura 2.2: Alcuni dei linguaggi supportati dall'ambiente di sviluppo CUDA.

[4]Un campo dell'intelligenza artificiale (IA) che permette ai computer e ai sistemi di ricavare informazioni significative da immagini digitali, video e altri input visivi - e intraprendere azioni o formulare delle segnalazioni sulla base di tali informazioni.

2.4.1 Modello di programmazione CUDA

Il modello di programmazione CUDA considera CPU e GPU come due macchine distinte, denominate ***host*** e ***device***. Un'applicazione CUDA combina parti sequenziali (eseguite dall'host) e parti parallele (eseguite dal device). La parte di codice che lavora in parallelo è chiamata ***kernel***. L'host invoca i kernel configurando il device per l'esecuzione in parallelo, passandogli alcuni parametri. Il device può eseguire solo un kernel alla volta.

Thread, block e grid

L'unità base del parallelismo in CUDA è chiamata ***thread***: più thread eseguiti sul device eseguono lo stesso flusso di istruzioni su dati differenti, creando quello che NVIDIA ha ribattezzato come paradigma ***SIMT (Single Instruction Multiple Threads***, simile al paradigma ***SIMD (Single Instruction Multiple Data)*** della ***tassonomia di Flynn***[5]. I thread sono raggruppati in blocchi (***blocks***), questi blocchi vengono eseguiti sullo stesso ***Streaming Multiprocessor (SM)*** e condividono una memoria condivisa che d'ora in avanti chiameremo ***shared memory***. I blocchi sono raggruppati in una griglia (***grid***), tutti i blocchi del device condividono una ***global memory*** (la memoria della GPU). Nella Figura 2.3 possiamo osservare una possibile configurazione di thread e blocchi in una griglia.

Struttura base di un'applicazione CUDA

La struttura base di un'applicazione può essere riassunta nel seguente modo:

- Prima di chiamare il kernel, l'host trasferisce i dati da elaborare sulla GPU;
- L'host chiama il kernel, passandogli gli argomenti appropiati di modo da permetterne la sua esecuzione;
- Dopo che i dati sono stati elaborati, vengono riportati sull'host dalla memoria del device.

[5]La tassonomia di Flynn è un sistema di classificazione delle architetture dei calcolatori che classifica i sistemi di calcolo a seconda della molteplicità del flusso di istruzioni e del flusso dei dati che possono gestire

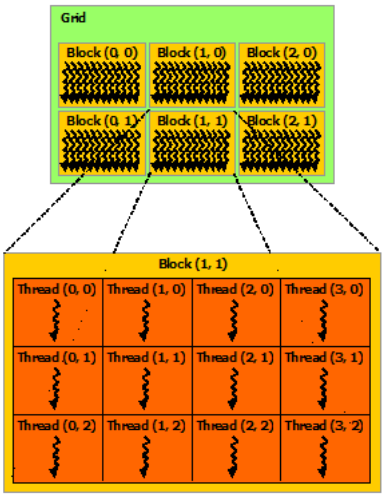

Figura 2.3: Una possibile configurazione di un kernel CUDA. Una griglia 2D con 2×3 blocchi con 3×4 thread per ogni blocco.

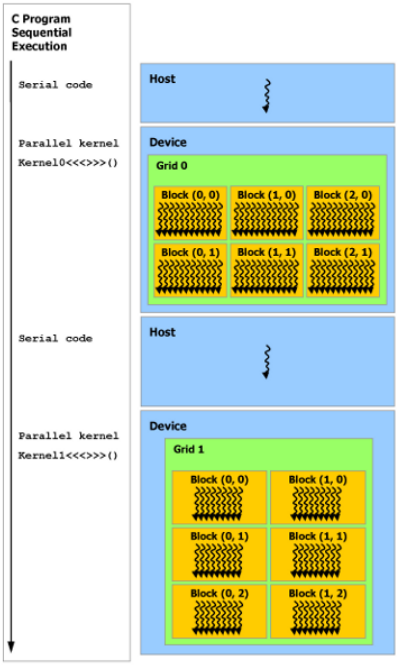

Figura 2.4: Struttura base di un'applicazione cuda che evidenzia le parti sequenziali e quelle parallele.

2.4.2 Organizzazione della memoria

La memoria del dispositivo è suddivisa in diverse tipologie distinguibili dalla latenza nel tempo di accesso e da quale unità è accessibile:

- ***Constant memory***: area di sola lettura, per la lettura accelerata da parte di tutti i thread.
- ***Texture memory***: area di sola lettura, ottimizzata per la lettura e accessibile da tutti i thread.
- ***Global memory***: area di lettura/scrittura, esterna ai multiprocessori e condivisa tra i thread. In questo spazio, controllato dall'host, si trovano le variabili trasferite dall'host al dispositivo e viceversa. Il tempo di accesso a questa memoria è molto elevato, ma poiché essa costituisce l'interfaccia immediata con la memoria RAM della CPU, i dati devono passare per forza dalla global memory.

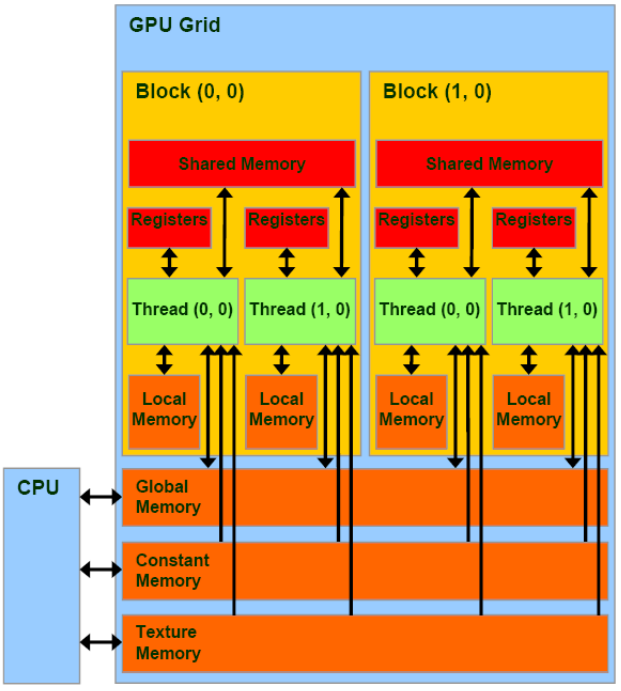

Figura 2.5: Organizzazione della memoria nell'architettura CUDA.

2.4.3 Shared memory e coalescenza

Alla classificazione della memoria della sezione precedente che si concentra sulle memorie esterne ai blocchi, possiamo aggiungere un'altra classificazione, quella delle memorie che troviamo all'interno di ciascun blocco (come riferimento si tenga presente la Figura 2.5):

- ***Register memory***: memoria a bassa latenza [6], privata per ogni singolo processore e accessibile da un solo thread.
- ***Local memory***: spazio privato per ogni singolo thread in cui sono memorizzate le variabili locali.
- ***Shared memory***: area di accesso a bassa latenza condivisa tra tutti i thread dello stesso blocco.

La chiave per ottenere prestazioni soddisfacenti è minimizzare gli accessi alla global memory sfruttando il più possibile la shared memory. La memoria condivisa può essere utilizzata sia come spazio privato che come spazio condiviso per la comunicazione tra i thread dello stesso blocco. Sfruttando questa capacità, i thread di uno stesso blocco possono collaborare per caricare dalla global memory alla shared memory i dati che devono essere elaborati. Successivamente provvederanno ad elaborare i dati sfruttando la minor latenza offerta dalla shared memory. Dunque i passi che ciascun thread deve eseguire sono:

- caricare i dati dalla global memory alla shared memory;
- sincronizzare i thread di un blocco
- elaborare i dati nella shared memory
- sincronizzare nuovamente i thread per assicurarsi che tutti i thread abbiano eseguito i calcoli
- trasferire i dati dalla shared alla global memory

[6] La latenza rappresenta i cicli di clock della memoria necessari a reperire un dato in essa memorizzato.

Per ottenere accessi alla memoria più veloci, i 16 KB destinati alla shared memory e condivisi tra tutti gli streaming multiprocessor, sono divisi in 16 banchi ciascuno da 1 KB, uno per ogni processore. Come è facile intuire, per sfruttare questa memoria sono necessarie modifiche sostanziali al codice e una completa riprogettazione degli algoritmi. Tuttavia il lavoro viene ripagato da performance nettamente migliori rispetto all'algoritmo che non fa uso della shared memory.
I trasferimenti di dati da e verso la memoria sono governati dalla ***coalescenza***. Si indica con questo termine una serie di restrizioni che consentono di unire più accessi alla memoria in un'unica transazione. Gli accessi al kernel sono chiamati coalescenti se i thread con identificatori[7] consecutivi accedono a posizioni di memoria contigue utilizzando la stessa istruzione. Ciò deriva da una speciale configurazione di thread che possono essere raggruppati in ***warp***, gruppi di 32 thread che eseguono tutti la stessa istruzione.

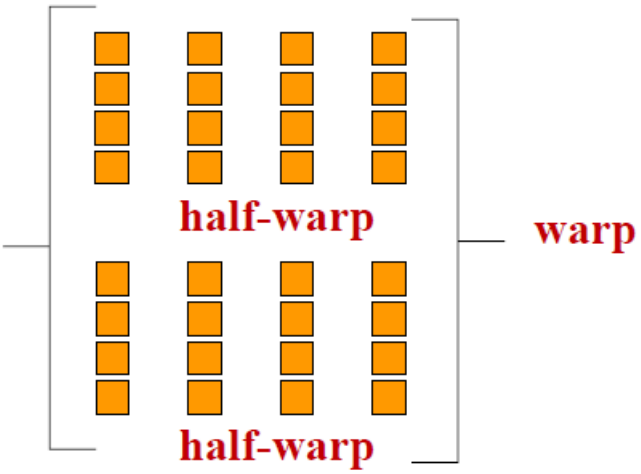

Figura 2.6: Un warp di thread (32 thread) diviso un due warp da 16 thread ciascuno.

[7]Ogni CUDA thread ha un proprio identificatore (id) che lo identifica nel blocco.

2.5 API di CUDA per il linguaggio C

CUDA mette a disposizione delle API per il linguaggio C, che rendono semplice e intuitivo l'allocazione di dati sul device, il trasferimento di dati da host a device e viceversa, la creazione e la configurazione di kernel e molto altro. Nelle sezioni successive verrà analizzata la struttura base di programma CUDA, concentrandosi in particolar modo su quali sono le funzioni di libreria che vengono più spesso utilizzate e come i dati devono essere allocati. Inoltre viene proposto un esempio banale di kernel che effettua la somma puntuale di due vettori di interi.

2.5.1 Struttura base di un programma CUDA

Un programma in ***CUDA C*** deve necessariamente includere l'header `<cuda.h>`, ove sono contenute tutte le funzioni di libreria necessarie al programma. La struttura base di un programma può essere riassunta nei seguenti passi:

- dichiarazione delle variabili dell'host
- dichiarazione delle variabili del device
- allocazione dei dati dell'host con `malloc`
- allocazione dei dati del device con `cudaMalloc`
- inizializzazione dei dati dell'host
- copia dei dati da host a device con la funzione `cudaMemcpy`
- invocazione kernel: configurazione dei parametri
- copia dei risultati da device a host con la funzione `cudaMemcpy`
- deallocazione dei dati dell'host con la funzione `freee`
- deallocazione dei dati del device con la funzione `cudaFree`

Tutte le operazioni in CUDA sono fatte per mezzo di puntatori. Vettori e matrici devono essere allocati dinamicamente e in particolare, le matrici devono essere allocate come se fossero dei vettori. Esempio: se la matrice A ha dimensioni $M \times N$, deve essere allocato un array di dimensioni $M \times N$.

```c
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>
#define M 10
#define N 4
int main(int argn, char **argv) {
    double *a;
    a = (double *)malloc(M * N * sizeof(double));
    if (a == NULL) {
        fprintf(stderr, "Errore allocazione.\n");
        exit(EXIT_FAILURE);
    }
    exit(0);
}
```

Listing 2.1: Esempio allocazione matrice dell'host con CUDA

Di conseguenza, per accedere ad un elemento della matrice `a` non si usa la seguente sintassi `a[i][j]` ma la sintassi per accedere ad un elemento in posizione `i,j` è `a[i * N + j]`, dove N è il numero di colonne della matrice è costituisce la cosiddetta ***leading dimension***[8].
Di seguito si riporta un banale programma cuda che non fa altro che copiare un array da host a device e poi di nuovo da device ad host.

Librerie da includere:

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

Listing 2.2: Librerie richieste dall'ambiente CUDA

[8]La leading dimension per una matrice è un incremento utilizzato per trovare il punto iniziale per gli elementi della matrice in ogni colonna successiva della matrice.[3]

Funzione main del programma:

```
int main(int argn, char **argv) {
    int N; // total number of array elements
    int *A_host; // array of host
    int *A_device; // array of the device
    int *copy; // array for copy data from the device
    int size; // size in byte of each array

    if (argn == 1) {
        N = 20;
    } else {
        N = atoi(argv[1]);
        printf("**********\tFIRST EXAMPLE\t**********\n");
        printf("copy of %d elements from CPU to GPU and vice
   versa\n\n", N);
    }

    // size in byte for each array
    size = N * sizeof(int);

    // host data allocation
    A_host = (int *)malloc(size);
    if (A_host == NULL) {
        printf("Malloc error.\n");
        exit(EXIT_FAILURE);
    }
    copy = (int *)malloc(size);
    if (copy == NULL) {
        printf("Malloc error.\n");
        exit(EXIT_FAILURE);
    }
```

```
30
31      // device data allocation
32      cudaMalloc((void **)&A_device, size);
33      // host data initialization
34      initializeArray(A_host, N);
35
36      // copy data from host to device
37      cudaMemcpy(A_device, A_host, size, cudaMemcpyHostToDevice);
38      // copy result from device to host
39      cudaMemcpy(copy, A_device, size, cudaMemcpyDeviceToHost);
40
41      printf("array of host\n");
42      stampaArray(A_host,N);
43      printf("array copied from device\n");
44      stampaArray(copy,N);
45      //accuracy test
46      equalArray(copy, A_host,N);
47      //host data de-allocation
48      free(A_host);
49      free(copy);
50      //device data de-allocation
51      cudaFree(A_device);
52
53      exit(0);
54  }
```

Listing 2.3: Inizializzazione e copia di una matrice host-device e device-host

Funzioni helper:

```
1  void initializeArray(int *array, int n) {
2      int i;
3      for (i = 0; i < n; i++)
```

```
4           array[i] = i;
5   }
6
7   void stampaArray(int* array, int n) {
8       int i;
9       for(i = 0; i < n; i++)
10          printf("%d ", array[i]);
11      printf("\n");
12  }
13
14  void equalArray(int *a, int *b, int n) {
15      int i = 0;
16      while (a[i] == b[i])
17          i++;
18      if (i < n)
19          printf(" The results of the host and the device are
        different \n");
20      else
21          printf(" The results of the host and the device are the
        same \n");
22  }
```

Listing 2.4: Funzioni utilizzate nel precedente listato

2.5.2 Funzioni di libreria più utilizzate

Dopo aver illustrato un semplice esempio di programma CUDA C, analizziamo un po' più nel dettaglio quali sono le funzioni di libreria più utilizzate.

Allocazione della memoria su GPU

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

- `devPtr` - puntatore all'area di memoria da allocare sul device
- `size` - la dimensione in byte dell'area da allocare

Deallocazione della memoria su GPU

```
1    cudaError_t cudaFree(void *devPtr);
```

- `devPtr` - puntatore all'area di memoria da deallocare sul device

Scambio di dati tra CPU e GPU

```
1    cudaError_t cudaMemcpy(
2        void *dest,
3        void *src,
4        size_t nBytes,
5        enum cudaMemcpyKind
6    );
```

- `dest` - puntatore all'area di memoria in cui copiare
- `src` - puntatore all'area di memoria da cui copiare
- `nBytes` - la dimensione in byte dell'area di memoria da copiare
- `cudaMemcpyKind` - indica la direzione della copia. È una variabile che può assumere i seguenti valori:
  - `cudaMemcpyHostToHost` copia i dati da host a host
  - `cudaMemcpyHostToDevice` copia i dati da host a device
  - `cudaMemcpyDeviceToHost` copia i dati da device a host
  - `cudaMemcpyDeviceToDevice` copia i dati da device a device

2.5.3 Creazione e configurazione di un kernel

Di seguito si riporta l'esempio di un kernel che effettua la somma puntuale di due vettori di `float` e la memorizza in un terzo vettore. Per semplicità si omettono tutte le parti relative all'allocazione e alla deallocazione di variabili.

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

// Kernel definition
__global__ void VecAdd(float *A, float *B, float *C) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    C[index] = A[index] + B[index];
}

int main() {
    ...
    // Set up the parameters of the kernel
    dim3 nThreadsPerBlock, nBlocks;
    nBlocks.x = 3;
    nThreadsPerBlock.x = N / nBlocks.x + ((N % nBlocks.x == 0) ?
   0 : 1);
    // Kernel invocation with N threads
    VecAdd <<< nBlocks, nThreadsPerBlock >>> (A, B, C);
    ...
}
```

Listing 2.5: Configurazione di un kernel per la somma di due vettori.

Le funzioni CUDA hanno lo stesso prototipo delle solite funzioni C precedute da uno di questi qualificatori:

- `__global__` : per le funzioni chiamate dall'host ed eseguite sul device (i kernel)
- `__device__` : per le funzioni chiamate dal device ed eseguite sul device
- `__host__`: (opzionale) per le funzioni eseguite dalla CPU

Il tipo di dato `dim3` è un tipo predefinito di CUDA che rappresenta un vettore di tre interi; questi tre campi sono accessibili con le notazioni `.x`, `.y`, `.z` e servono per indicare il numero dei fili o il numero dei blocchi in una particolare direzione; gli eventuali campi non dichiarati vengono automaticamente impostati a 1.

- `nBlocks` è il numero di blocchi della griglia, può essere 1D o 2D
- `nThreadsPerBlock` è il numero di thread per un singolo blocco (1D, 2D o 3D)

Esempio

Supponiamo di usare $N = 12$. Fissando il numero di blocchi a 3 come nell'esempio 2.5.3, il numero di thread per ciascun blocco sarà pari a 4, poiché $12/3 = 4$. Nel caso N non fosse stato esattamente divisibile per il numero di blocchi, al numero di thread sarebbe stato aggiunto un 1 e alcuni thread non avrebbero partecipato al calcolo della somma.

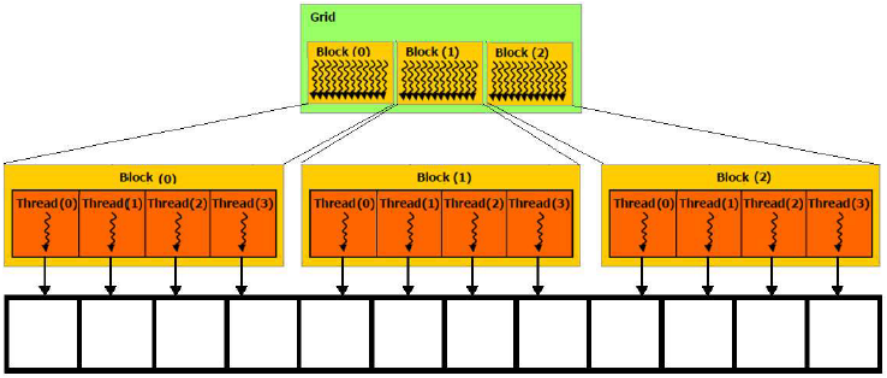

Figura 2.7: Configurazione del kernel CUDA `VecAdd` con N = 12 e numero di blocchi pari a 3.

Capitolo 3

Approccio parallelo e implementazione in ambiente GPU

3.1 Approccio parallelo

Nel capitolo 1 sezione 1.4 è stato analizzato l'algoritmo sequenziale per la risoluzione della PDE (Partial Differential Equation) proposta in questo lavoro. A causa dell'elevato tempo di calcolo richiesto dalla procedura sequenziale per trovare la soluzione dell'equazione (1.10) all'aumentare di N (si rimanda alla sezione 1.5 per approfondimenti), si è scelto di parallelizzare il modulo relativo al calcolo di tale equazione. Prima di passare direttamente alla presentazione del codice parallelo e della strategia di parallelizzazione adottata, vengono illustrare le operazioni preliminari svolte per la stesura dell'algoritmo parallelo.

3.1.1 Operazioni preliminari

Il primo passo effettuato nel processo di parallelizzazione dell'algoritmo sequenziale 1 è stato quello di ottimizzare il modo in cui è memorizzata la matrice L definita in (1.8). Tale matrice è una matrice tridiagonale a blocchi, che presenta un gran numero di valori nulli che non influiscono in alcun modo nelle operazioni in cui è coinvolta la matrice, rendendo quindi inutile salvarli in memoria. L'approccio che si utilizza in questi casi (e che è stato utilizzato) è quello di memorizzare la matrice L in formato ***CSR (Compressed Sparse Row)***. Il formato CSR è pensato per memorizzare una matrice con un gran numero di zeri in maniera compatta, eliminando tutti i valori nulli e memorizzando solo i valori non

nulli (chiamati non-zero values, talvolta abbreviato in nnz). Il formato prevede di dichiarare tre array che chiameremo rispettivamente `row_ptrs`, `col_ids` e `vals`. L'array `row_ptr` fornisce la somma cumulativa degli elementi diversi da zero in ogni riga e quindi ha dimensione $m + 1$ per una matrice sparsa $m \times n$. L'array `col_ids` fornisce l'indice della colonna e l'array `vals` fornisce i valori per ciascuno degli elementi diversi da zero. Pertanto, il formato CSR richiede $2 \times nnz + m + 1$ di spazio di allocazione. Se il numero di nnz è particolarmente ridotto, come nel caso della matrice L, si ha una diminuzione sia dello spazio di allocazione e sia nel numero di operazioni che vengono svolte.
Di seguito si riporta un esempio del formato CSR applicato a una matrice tridiagonale a blocchi A di 4×4 elementi e con gli indici che partono da 0.

$$A = \begin{bmatrix} -2 & 1 & 0 & 1 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 1 & 0 & 1 & -2 \end{bmatrix}$$

$$\texttt{row\_ptrs} \quad [0 \quad 3 \quad 6 \quad 9 \quad 12]$$

$$\texttt{col\_idxs} \quad [0 \quad 1 \quad 3 \quad 0 \quad 1 \quad 2 \quad 1 \quad 2 \quad 3 \quad 0 \quad 2 \quad 3]$$

$$\texttt{vals} \quad [-2 \quad 1 \quad 1 \quad 1 \quad -2 \quad 1 \quad 1 \quad -2 \quad 1 \quad 1 \quad 1 \quad -2]$$

3.2 Algoritmo parallelo

Lo pseudo-codice parallelo 2 evidenzia i passi principali della procedura numerica in un ambiente GPU. I dettagli e le operazioni principali sono elencati nei seguenti passaggi:

Algorithm 2 Algoritmo parallelo peer methods

Input $s, x_0, X, N, t_0, T, M, y_0, N, L, a, B_1, B_2, H, F, S, d, D, \Delta t$. **Output** Y

// **Initialization**
1: $t\_span = [t_0, T]$
2: time discretization $t_n (n = 0, \ldots, N)$
3: $N = (t\_span[2] - t\_span[1])/\Delta t$
4: $x\_span = [x_0, X]$
5: space discretization $x_m (m = 0, \ldots, M-1)$
6: $\Delta x = (x\_span[2] - x\_span[1])/(M-1)$
// **Sequential spatial discretization**
7: building of L_{Diff} and L as in (1.5), (1.8)
8: convert L from dense to sparse matrix in CSR format
// **Parallel time discretization using peer method**
9: $s = 2$
// GPU computation start
10: `transfer` Input data from Host-To-Device
// Initialization with time step $n = 0$
11: `parallel computing` of $Y_{0,1}$ and $Y_{0,2}$ using Runge-Kutta 4th order method
12: `parallel evaluation` of $\mathcal{F}((Y_{0,1}))$ and $\mathcal{F}((Y_{0,2}))$ as in (1.13)
// Main loop: loop on time steps
13: **for** $n = 1 \ldots N$ **do**
14: `parallel computing` $Y_{n,1}$ and $Y_{n,2}$ as in (1.14)
15: `parallel evaluation` of $\mathcal{F}((Y_{n,1}))$ and $\mathcal{F}((Y_{n,2}))$ as in (1.13)
16: **end for**
17: `transfer` Output data from Device-To-Host
// GPU computation end

- `Righe 2-7`: l'algoritmo inizializza i parametri necessari per la discretizzazione del tempo e dello spazio così come nella controparte sequenziale 1.

- `Righe 9-10`: dopo aver definito la matrice L così come è stato fatto nell'algoritmo 1, la si converte in formato CSR (Compressed Sparse Row). L'algoritmo per la definizione della matrice L in formato CSR è il seguente:

Algorithm 3 Definizione matrice in formato CSR

Input $A, nrows, ncols, nnz$
Output $row\_ptrs, col\_idxs, values$

```
1:  allocate row_ptrs[1 ... nrwos + 1]
2:  allocate col_idxs[1 ... nnz]
3:  allocate values[1 ... nnz]
4:  cumulative_sum = 0
5:  for i = 1 ... nrows do
6:      row_ptrs[i] = cumulative_sum
7:      for j = 1 ... ncols do
8:          col_idxs[cumulative_sum] = j
9:          values[cumulative_sum] = A[i, j]
10:         cumulative_sum = cumulative_sum + 1
11:     end for
12: end for
13: row_ptrs[nrows] = cumulative_sum
```

  - `Righe 1-4`: inizializzazione e allocazione delle strutture dati ossia row_ptrs, col_idxs, $values$.
  - `Righe 5-12`: viene costruita la matrice sparsa. Il ciclo for più esterno costruire l'array row_ptrs, quello più interno col_idxs e $values$ salvando rispettivamente gli indici di colonna e i valori diversi da zero all'interno di queste due strutture dati.

- `Righe 16-17`: si procede con il calcolo di $Y_{0,i}$ dove $i = 1 \dots s$, utilizzando il metodo Runge-Kutta del quarto ordine, che è stato parallelizzato distribuendo le operazioni di algebra lineare su un opportuno numero di thread. Anche la valutazione della funzione $\mathcal{F}$ è stata parallelizzata, distribuendo le operazioni tra un opportuno numero di thread.

Algorithm 4 Algoritmo di Runge-Kutta parallelo

Input h, t_0, y_0
Output y

// Compute Y_1
1: $Y_1 = y_0$
// Compute Y_2
2: `parallel evaluation` of $\mathcal{F}(Y_1)$
3: `parallel product` $h/2$ by $\mathcal{F}(Y_1)$
4: $Y_2 =$ `parallel sum` $y_0 + \mathcal{F}(Y_1)$
// Compute Y_3
5: `parallel evaluation` of $\mathcal{F}(Y_2)$
6: `parallel product` $h/2$ by $\mathcal{F}(Y_2)$
7: $Y_3 =$ `parallel sum` $y_0 + \mathcal{F}(Y_2)$
// Compute Y_4
8: `parallel evaluation` of $\mathcal{F}(Y_3)$
9: `parallel product` h by $\mathcal{F}(Y_3)$
10: $Y_4 =$ `parallel sum` $y_0 + \mathcal{F}(Y_3)$
// Make a sort of weighted arithmetic mean
11: $y = y_0 + h \cdot (1/6 \cdot \mathcal{F}(Y_1) + 1/3 \cdot \mathcal{F}(Y_2) + 1/3 \cdot \mathcal{F}(Y_3) + 1/6 \cdot \mathcal{F}(Y_4))$

- `Righe 20-21`: ancora una volta il loop principale non fa altro che eseguire le stesse operazioni eseguite al di fuori del loop, dato che ogni step è funzione del precedente come è stato ampiamente discusso nel capitolo 1. Si calcola in parallelo $Y_{n,i}$ dove $i = 1 \dots s$ ed $n = 1 \dots N$ secondo l'equazione (1.10), eseguendo parallelamente gli s stage della variabile j. La valutazione della funzione $\mathcal{F}$ è praticamente la stessa effettuata alla riga `17`.

Nella paragrafo successivo verrà analizzata in dettaglio la strategia di parallelizzazione utilizzata per il calcolo dell'equazione (1.10) in ambiente CUDA (riga 20 dell'algoritmo parallelo 2), in quanto per la sua importanza merita un approfondimento in più.

3.2.1 Strategia di parallelizzazione kernel dell'equazione principale

Per capire qual è il processo che ha portato alla parallelizzazione dell'equazione (1.10) in ambiente CUDA, la si può riscrivere in forma più compatta come segue:

$$Y_{n,i} = b_{ij} Y_{n-1,j} + h a_{ij} f(t_{n-1,j}, Y_{n-1,j}) \tag{3.1}$$

$$\text{con } n = 2 \dots N \quad i = 1 \dots s \quad j = 1 \dots s$$

Poiché $s = 2$, andiamo ad esplicitare la j e otteniamo due equazioni:

$$\begin{aligned} Y_{n,1} &= b_{i1} Y_{n-1,1} + h a_{i1} f(t_{n-1,1}, Y_{n-1,1}) \\ Y_{n,2} &= b_{i2} Y_{n-1,2} + h a_{i2} f(t_{n-1,2}, Y_{n-1,2}) \end{aligned} \tag{3.2}$$

Una volta ottenuta l'equazione in questa forma, bisogna capire come implementare i vari attori che la compongono. Sia d la dimensione del problema, Y può essere visto come una matrice di dimensione $s \cdot d \times N$, ove i primi d valori rappresentano la soluzione $Y_{n,1}$ e i restanti d valori rappresentano $Y_{n,2}$. A e B sono due matrici 2×2 la cui configurazione è quella in (1.11) e la funzione f viene calcolata seguendo l'equazione (1.6) e poiché la dimensione del problema è d, sarà un vettore di d elementi.

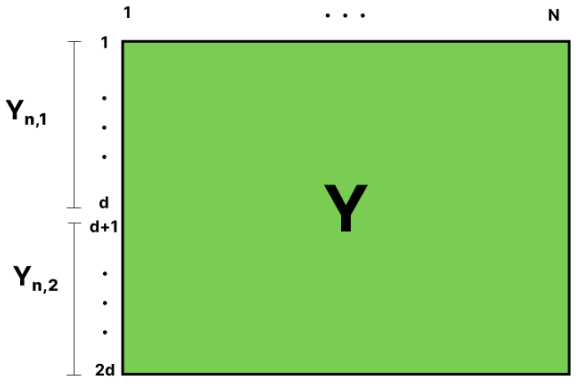

Figura 3.1: Struttura della matrice Y.

L'implementazione della strategia di parallelizzazione consiste nel creare un kernel apposito per il calcolo di $Y_{n,1}$ e $Y_{n,2}$ per ogni $n = 2, \dots, N$ e nello sfruttare l'architettura CUDA, che ci consente di organizzare i thread in blocchi (come

spiegato nel capitolo 2). Sia i thread che i blocchi possono essere bidimensionali, cioè è possibile scegliere una griglia di thread all'interno di una griglia di blocchi. I thread presenti nello stesso blocco godono di alcuni vantaggi, come ad esempio possono condividere alcuni registri in modo da eseguire in maniera più efficienti le operazioni assegnate. L'implementazione della strategia di parallelizzazione proposta mira a fare in modo che la griglia con cui viene lanciato il kernel sia bidimensionale, ossia ci siano fissi due blocchi lungo l'asse y e un numero di blocchi lungo l'asse delle x che può essere arbitrario. Anche il numero di thread all'interno di ciascun blocco può essere arbitrario, purché non sia in numero maggiore rispetto agli elementi della matrice Y su cui bisogna operare. La distribuzione del lavoro tra i vari thread e i vari blocchi avviene dividendo la dimensione del problema d per il prodotto tra il numero di thread e il numero di blocchi. Disponendo di una griglia bidimensionale di blocchi, considerando che ciascun blocco ha una coppia di identificativi chiamati rispettivamente `blockIdx.x` e `blockIdx.y`, i blocchi con `blockIdx.y` $= 0$ calcoleranno l'equazione (3.2) per $i = 1$ e i blocchi con `blockIdx.y` $= 1$ calcoleranno la medesima equazione per $i = 2$ concorrentemente. Osservando la figura 3.2 si può notare che ciascun stage i opera su porzioni differenti della matrice Y e questo fa sì che non possano verificarsi race condition sugli elementi della matrice.

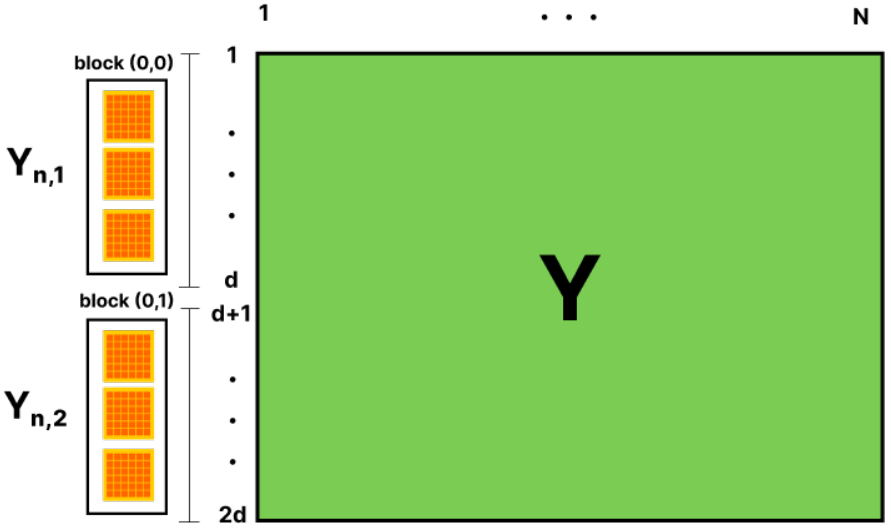

Figura 3.2: Distribuzione dei valori della matrice Y fra thread e blocchi.

Capitolo 4

Test, risultati e conclusioni

4.1 Confronto tempi GPU vs CPU

In questa sezione andremo a confrontare i tempi dell'algoritmo sequenziale con quelli dell'algoritmo parallelo per ciascun macro modulo che è stato individuato nel problema (1.1). Verrà mostrato quindi come l'accelerazione parallela dell'algoritmo 1 ha portato ad una notevole riduzione nei tempi di esecuzione sia dei vari macro moduli e sia del software nel suo complesso. Per il resto della sezione, consideriamo il problema (1.1) con i seguenti parametri di input $A = 1.5$, $B_1 = 0.45$, $B_2 = 0.3611$, $F = 0.802$, $H = 0.802$, $S = 0.0002$, $d = 500$, $D = 0.802$, e gli intervalli di spazio e tempo pari rispettivamente a $[-50, 50]$ e $[0, 50]$. Inoltre, gli esperimenti sono stati eseguiti su un computer con le seguenti specifiche tecniche:

- 4 Intel(R) Core(TM) i7 860 CPU @2.80 GHz, 4 core, 8 thread per core, 8 GB di RAM
- 1 GPU NVIDIA Quadro K5000, 1536 cuda core, 4 GB GDDR5, 173 GB/s di banda

Si noti che in ciascuna tabella, moltiplicando il numero di chiamate delle funzioni per il tempo medio si ottiene il tempo totale di esecuzione della specifica funzione.

4.1.1 Thread × block: 16 × 12

	RungeKutta4th GPU	RungeKutta4th CPU
N	Tempo medio(s)	Tempo medio(s)
1×10^5	4.6160×10^{-5}	3.2775×10^{-3}
2×10^5	4.6208×10^{-5}	3.0250×10^{-3}
4×10^5	4.6000×10^{-5}	3.1805×10^{-3}
8×10^5	4.6400×10^{-5}	2.8115×10^{-3}
1×10^6	4.6336×10^{-5}	-
3×10^6	-	-

Tabella 4.1: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazione 16 thread e 12 blocchi

	Sherratt GPU		Sherratt CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	2×10^5	6.4120×10^{-6}	2×10^5	2.7116×10^{-4}
2×10^5	4×10^5	6.4160×10^{-6}	4×10^5	2.7050×10^{-4}
4×10^5	8×10^5	6.4270×10^{-6}	28×10^5	2.7170×10^{-4}
8×10^5	1×10^5	6.4280×10^{-6}	1×10^6	2.7243×10^{-4}
1×10^6	3×10^6	6.4290×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.2: Confronto tempi di esecuzione modulo Sherratt su GPU vs CPU per la configurazione 16 thread e 12 blocchi

	computeY GPU		computeY CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	1×10^5	4.1140×10^{-6}	1×10^5	1.2223×10^{-5}
2×10^5	2×10^5	4.1730×10^{-6}	2×10^5	1.2236×10^{-5}
4×10^5	4×10^5	4.1150×10^{-6}	4×10^5	1.2708×10^{-5}
8×10^5	8×10^5	4.0730×10^{-6}	8×10^5	1.2714×10^{-5}
1×10^6	1×10^6	4.0800×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.3: Confronto tempi di esecuzione modulo computeY su GPU vs CPU per la configurazione 16 thread e 12 blocchi

	Tempo totale modulo (s)	
N	GPU	CPU
1×10^5	8.0828	5.7394×10
2×10^5	1.5699×10	1.1457×10^2
4×10^5	3.1088×10	2.2945×10^2
8×10^5	6.3654×10	4.6385×10^2
1×10^6	1.2801×10^2	-
3×10^6	-	-

Tabella 4.4: Tempo di esecuzione GPU vs CPU per 16 thread e 12 blocchi

4.1.2 Thread × block: 20 × 10

	RungeKutta4th GPU	RungeKutta4th CPU
N	Tempo medio(s)	Tempo medio(s)
1×10^5	4.6192×10^{-5}	3.2775×10^{-3}
2×10^5	4.6544×10^{-5}	3.0250×10^{-3}
4×10^5	4.6224×10^{-5}	3.1805×10^{-3}
8×10^5	4.6464×10^{-5}	2.8115×10^{-3}
1×10^6	4.6320×10^{-5}	-
3×10^6	-	-

Tabella 4.5: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazionee 20 thread e 10 blocchi

	Sherratt GPU		Sherratt CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	2×10^5	7.1000×10^{-6}	2×10^5	2.7116×10^{-4}
2×10^5	4×10^5	7.0930×10^{-6}	4×10^5	2.7050×10^{-4}
4×10^5	8×10^5	7.1060×10^{-6}	8×10^5	2.7170×10^{-4}
8×10^5	1×10^6	7.0930×10^{-6}	1×10^6	2.7243×10^{-4}
1×10^6	3×10^6	7.0930×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.6: Confronto tempi di esecuzione modulo Sherratt su GPU vs CPU per la configurazione 20 thread e 10 blocchi

	computeY GPU		computeY CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	1×10^5	4.3480×10^{-6}	1×10^5	1.2223×10^{-5}
2×10^5	2×10^5	4.3930×10^{-6}	2×10^5	1.2236×10^{-5}
4×10^5	4×10^5	4.3840×10^{-6}	4×10^5	1.2708×10^{-5}
8×10^5	8×10^5	4.3700×10^{-6}	8×10^5	1.2714×10^{-5}
1×10^6	1×10^6	4.3950×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.7: Confronto tempi di esecuzione modulo computeY su GPU vs CPU per la configurazione 20 thread e 10 blocchi

	Tempo totale modulo (s)	
N	GPU	CPU
1×10^5	8.2201	5.7394×10
2×10^5	1.6060×10	1.1457×10^2
4×10^5	3.2633×10	2.2945×10^2
8×10^5	6.4645×10	4.6385×10^2
1×10^6	1.2850×10^2	-
3×10^6	-	-

Tabella 4.8: Tempo di esecuzione GPU vs CPU per 20 thread e 10 blocchi

4.1.3 Thread × block: 32 × 6

	RungeKutta4th GPU	RungeKutta4th CPU
N	Tempo medio(s)	Tempo medio(s)
1×10^5	4.6032×10^{-5}	3.2775×10^{-3}
2×10^5	4.6160×10^{-5}	3.0250×10^{-3}
4×10^5	4.6319×10^{-5}	3.1805×10^{-3}
8×10^5	4.6208×10^{-5}	2.8115×10^{-3}
1×10^6	4.6143×10^{-5}	-
3×10^6	-	-

Tabella 4.9: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazione 32 thread e 6 blocchi

	Sherratt GPU		Sherratt CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	2×10^5	7.2060×10^{-6}	2×10^5	2.7116×10^{-4}
2×10^5	4×10^5	7.2100×10^{-6}	4×10^5	2.7050×10^{-4}
4×10^5	8×10^5	7.2010×10^{-6}	8×10^5	2.7170×10^{-4}
8×10^5	1×10^6	7.2040×10^{-6}	1×10^6	2.7243×10^{-4}
1×10^6	3×10^6	7.2130×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.10: Confronto tempi di esecuzione modulo Sherratt su GPU vs CPU per la configurazione 32 thread e 6 blocchi

	computeY GPU		computeY CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	1×10^5	4.6070×10^{-6}	1×10^5	1.2223×10^{-5}
2×10^5	2×10^5	4.6810×10^{-6}	2×10^5	1.2236×10^{-5}
4×10^5	4×10^5	4.7010×10^{-6}	4×10^5	1.2708×10^{-5}
8×10^5	8×10^5	4.7070×10^{-6}	8×10^5	1.2714×10^{-5}
1×10^6	1×10^6	4.6840×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.11: Confronto tempi di esecuzione modulo computeY su GPU vs CPU per la configurazione 32 thread e 6 blocchi

	Tempo totale modulo (s)	
N	GPU	CPU
1×10^5	8.4711	5.7394×10
2×10^5	1.6104×10	1.1457×10^2
4×10^5	3.3724×10	2.2945×10^2
8×10^5	6.5092×10	4.6385×10^2
1×10^6	1.3012×10^2	-
3×10^6	-	-

Tabella 4.12: Tempo di esecuzione GPU vs CPU per 32 thread e 6 blocchi

4.1.4 Thread × block: 40 × 5

	RungeKutta4th GPU	RungeKutta4th CPU
N	Tempo medio(s)	Tempo medio(s)
1×10^5	4.5888×10^{-5}	3.2775×10^{-3}
2×10^5	4.6432×10^{-5}	3.0250×10^{-3}
4×10^5	4.6496×10^{-5}	3.1805×10^{-3}
8×10^5	4.5839×10^{-5}	2.8115×10^{-3}
1×10^6	4.6496×10^{-5}	-
3×10^6	-	-

Tabella 4.13: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazione 40 thread e 5 blocchi

	Sherratt GPU		Sherratt CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	2×10^5	7.6420×10^{-6}	2×10^5	2.7116×10^{-4}
2×10^5	4×10^5	7.6610×10^{-6}	4×10^5	2.7050×10^{-4}
4×10^5	8×10^5	7.6380×10^{-6}	8×10^5	2.7170×10^{-4}
8×10^5	1×10^6	7.6400×10^{-6}	1×10^6	2.7243×10^{-4}
1×10^6	3×10^6	7.6370×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.14: Confronto tempi di esecuzione modulo Sherratt su GPU vs CPU per la configurazione 40 thread e 5 blocchi

	computeY GPU		computeY CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	1×10^5	4.6220×10^{-6}	1×10^5	1.2223×10^{-5}
2×10^5	2×10^5	4.6710×10^{-6}	2×10^5	1.2236×10^{-5}
4×10^5	4×10^5	4.6710×10^{-6}	4×10^5	1.2708×10^{-5}
8×10^5	8×10^5	4.6220×10^{-6}	8×10^5	1.2714×10^{-5}
1×10^6	1×10^6	4.6720×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.15: Confronto tempi di esecuzione modulo computeY su GPU vs CPU per la configurazione 40 thread e 5 blocchi

	Tempo totale modulo (s)	
N	GPU	CPU
1×10^5	8.3965	5.7394×10
2×10^5	1.6198×10	1.1457×10^2
4×10^5	3.4005×10	2.2945×10^2
8×10^5	6.8491×10	4.6385×10^2
1×10^6	1.3661×10^2	-
3×10^6	-	-

Tabella 4.16: Tempo di esecuzione GPU vs CPU per 40 thread e 5 blocchi

4.1.5 Thread × block: 64 × 3

	RungeKutta4th GPU	RungeKutta4th CPU
N	Tempo medio(s)	Tempo medio(s)
1×10^5	4.6320×10^{-5}	3.2775×10^{-3}
2×10^5	4.6032×10^{-5}	3.0250×10^{-3}
4×10^5	4.6512×10^{-5}	3.1805×10^{-3}
8×10^5	4.6143×10^{-5}	2.8115×10^{-3}
1×10^6	4.6479×10^{-5}	-
3×10^6	-	-

Tabella 4.17: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazione 64 thread e 3 blocchi

	Sherratt GPU		Sherratt CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	2×10^5	7.3010×10^{-6}	2×10^5	2.7116×10^{-4}
2×10^5	4×10^5	7.3000×10^{-6}	4×10^5	2.7050×10^{-4}
4×10^5	8×10^5	7.2790×10^{-6}	8×10^5	2.7170×10^{-4}
8×10^5	1×10^6	7.2980×10^{-6}	1×10^6	2.7243×10^{-4}
1×10^6	3×10^6	7.2760×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.18: Confronto tempi di esecuzione modulo Sherratt su GPU vs CPU per la configurazione 64 thread e 3 blocchi

	computeY GPU		computeY CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	1×10^5	4.7540×10^{-6}	1×10^5	1.2223×10^{-5}
2×10^5	2×10^5	4.7560×10^{-6}	2×10^5	1.2236×10^{-5}
4×10^5	4×10^5	4.7680×10^{-6}	4×10^5	1.2708×10^{-5}
8×10^5	8×10^5	4.7910×10^{-6}	8×10^5	1.2714×10^{-5}
1×10^6	1×10^6	4.7440×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.19: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazione 64 thread e 3 blocchi

	Tempo totale modulo (s)	
N	GPU	CPU
1×10^5	8.2453	5.7394×10
2×10^5	1.6627×10	1.1457×10^2
4×10^5	3.3992×10	2.2945×10^2
8×10^5	6.8016×10	4.6385×10^2
1×10^6	1.3274×10^2	-
3×10^6	-	-

Tabella 4.20: Tempo di esecuzione GPU vs CPU per 64 thread e 3 blocchi

4.1.6 Thread × block: 192 × 1

	RungeKutta4th GPU	RungeKutta4th CPU
N	Tempo medio(s)	Tempo medio(s)
1×10^5	4.5999×10^{-5}	3.2775×10^{-3}
2×10^5	4.5920×10^{-5}	3.0250×10^{-3}
4×10^5	4.6144×10^{-5}	3.1805×10^{-3}
8×10^5	4.6624×10^{-5}	2.8115×10^{-3}
1×10^6	4.6255×10^{-5}	-
3×10^6	-	-

Tabella 4.21: Confronto tempi di esecuzione modulo RungeKutta4th su GPU vs CPU per la configurazione 192 thread e 1 blocchi

	Sherratt GPU		Sherratt CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	2×10^5	7.5640×10^{-6}	2×10^5	3.2120×10^{-6}
2×10^5	4×10^5	7.5470×10^{-6}	4×10^5	3.1960×10^{-6}
4×10^5	8×10^5	7.5670×10^{-6}	8×10^5	3.2170×10^{-6}
8×10^5	1×10^6	7.5410×10^{-6}	1×10^6	3.1900×10^{-6}
1×10^6	3×10^6	7.5450×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.22: Confronto tempi di esecuzione modulo Sherratt su GPU vs CPU per la configurazione 192 thread e 1 blocchi

	computeY GPU		computeY CPU	
N	Chiamate	Tempo medio(s)	Chiamate	Tempo medio(s)
1×10^5	1×10^5	2.9850×10^{-6}	1×10^5	1.2223×10^{-5}
2×10^5	2×10^5	2.9650×10^{-6}	2×10^5	1.2236×10^{-5}
4×10^5	4×10^5	2.9860×10^{-6}	4×10^5	1.2708×10^{-5}
8×10^5	8×10^5	2.9870×10^{-6}	8×10^5	1.2714×10^{-5}
1×10^6	1×10^6	2.9850×10^{-6}	-	-
3×10^6	-	-	-	-

Tabella 4.23: Confronto tempi di esecuzione modulo computeY su GPU vs CPU per la configurazione 192 thread e 1 blocchi

	Tempo totale modulo (s)	
N	GPU	CPU
1×10^5	7.9331	5.7394×10
2×10^5	1.5908×10	1.1457×10^2
4×10^5	3.1818×10	2.2945×10^2
8×10^5	6.2973×10	4.6385×10^2
1×10^6	1.2658×10^2	-
3×10^6	-	-

Tabella 4.24: Tempo di esecuzione GPU vs CPU per 192 thread e 1 blocchi

4.2 Conclusioni

Nella tesi è stata presentata una strategia di parallelizzazione e una relativa implementazione in un ambiente GPU (Graphics Processing Units) per il calcolo della soluzione approssimata di un modello di vegetazione modellato da un sistema di PDE. La procedura numerica combina una discretizzazione spaziale mediante il metodo delle differenze finite centrali e l'uso di peer methods per la discretizzazione della variabile temporale. Sono stati forniti risultati numerici e test che mostrano il guadagno in termini di prestazioni dell'algoritmo proposto.

Bibliografia

[1] NVIDIA, *The Benefits of Using GPUs*, CUDA documentation https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

[2] NVIDIA, *Cronologia di NVIDIA, una storia di innovazione*. https://www.nvidia.com/it-it/about-nvidia/corporate-timeline/

[3] IBM, *How Leading Dimension Is Used for Matrices*, IBM documentation https://www.ibm.com/docs/en/essl/6.3?topic=matrices-how-leading-dimension-is-used

[4] Conte, D. et al. (2022). First Experiences on Parallelizing Peer Methods for Numerical Solution of a Vegetation Model. In: Gervasi, O., Murgante, B., Hendrix, E.M.T., Taniar, D., Apduhan, B.O. (eds) Computational Science and Its Applications – ICCSA 2022. ICCSA 2022. Lecture Notes in Computer Science, vol 13376. Springer, Cham. https://doi.org/10.1007/978-3-031-10450-3_33

[5] Eigentler, L., Sherratt, J.A. Metastability as a Coexistence Mechanism in a Model for Dryland Vegetation Patterns. Bull Math Biol 81, 2290–2322 (2019). https://doi.org/10.1007/s11538-019-00606-z

[6] Weiner, R., Biermann, K., Schmitt, B., Podhaisky, H.: Explicit two-step peer methods. Comput. Math. Appl. 55(4), 609–619 (2008)

[7] Butcher, J.C.: Implicit Runge-Kutta processes. Math. Comp. **18**, 50–64 (1964)

[8] Butcher, J.C.: Numerical Methods for Ordinary Differential Equations, 2nd edn. Wiley, Chichester (2008)

[9] Samir Hamdi et al. (2007) Method of lines. Scholarpedia, 2(7):2859. http://www.scholarpedia.org/article/Method_of_lines

[10] Butcher, J.C.: Implicit Runge-Kutta processes. Math. Comp. 18, 50–64 (1964)

[11] Butcher, J.C.: Numerical Methods for Ordinary Differential Equations, 2nd edn. Wiley, Chichester (2008)

[12] Butcher, J.C.: General linear methods. Acta Numer. 15, 157–256 (2006)

[13] Sanz-Serna, J.M., Verwer, J.G. and Hundsdorfer, W.H. Convergence and order reduction of Runge Kutta schemes applied to evolutionary problems in partial differential equations. Numer. Math. 50, 405–418 (1986). https://doi.org/10.1007/BF01396661

[14] https://developer.nvidia.com/cuda-zone

[15] Conte, D., Mohammadi, F., Moradi, L., Paternoster, B.: Exponentially fitted twostep peer methods for oscillatory problems. Comput. Appl. Math. 39(3), 1–19 (2020). https://doi.org/10.1007/s40314-020-01202-x

[16] Conte, D., Pagano, G., Paternoster, B.: Jacobian-dependent two-stage peer method for ordinary differential equations. In: Gervasi, O., et al. (eds.) ICCSA 2021, Part I. LNCS, vol. 12949, pp. 309–324. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86653-2_23

[17] Conte, D., Pagano, G., Paternoster, B.: Two-step peer methods with equation dependent coefficients. Comput. Appl. Math. 41(4), 140 (2022)

[18] Schmitt, B.A., Weiner, R., Jebens, S.: Parameter optimization for explicit parallel peer two-step methods. Appl. Numer. Math. 59(3–4), 769–782 (2009) 394 D. Conte et al.

[19] Schmitt, B.A., Weiner, R., Podhaisky, H.: Multi-implicit peer two-step W-methods for parallel time integration. BIT Numer. Math. 45(1), 197–217 (2005). https://doi.org/10.1007/s10543-005-2635-y

[20] Schmitt, B.A., Weiner, R., Erdmann, K.: Implicit parallel peer methods for stiff initial value problems. Appl. Numer. Math. 53(2–4), 457–470 (2005)

[21] Weiner, R., Schmitt, B.A., Podhaisky, H.: Parallel "Peer" two-stepW-methods and their application to MOL-systems. Appl. Numer. Math., 48(3–4), 425–439 (2004) 18. Schmitt, B.A.,Weiner, R.: Parallel two-stepW-methods with peer variables. SIAM J. Numer. Anal. 42(1), 265–282 (2004)

[22] Jebens, S., Weiner, R., Podhaisky, H., Schmitt, B.: Explicit multi-step peer methods for special second-order differential equations. Appl. Math. Comput. 202(2), 803–813 (2008)

[23] Klinge, M., Weiner, R., Podhaisky, H.: Optimally zero stable explicit peer methods with variable nodes. BIT Numer. Math. 58(2), 331–345 (2017). https://doi.org/ 10.1007/s10543-017-0691-8

[24] Rosenbrock, H.H.: Some general implicit processes for the numerical solution of differential equations. Comput. J. 5(4), 329–330 (1963)