Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

INTERPOLAZIONE NUMERICA: UN APPROCCIO CUDA-PARALLELO

Relatore
Prof. Moreno Marzolla

Presentata da
Andrea Ingargiola

Anno Accademico 2019/2020

Humans are very good at making algorithms work eventually.

- Usama Fayyad

Sommario

In un mondo ideale si avrebbero a disposizione calcolatori dotati di memoria e potenza computazionale infinita, in grado di lavorare con numeri reali dalle infinite cifre decimali e di eseguire in un istante anche gli algoritmi più complessi. Nonostante l'avanzamento tecnologico incrementi sempre di più le prestazioni degli strumenti in nostro possesso, le macchine attuali sono in ogni caso soggette a leggi della fisica incompatibili con il concetto di "risorse infinite". Questo limite comporta, oltre all'impossibilità di lavorare direttamente con la matematica a numeri reali, anche parecchie difficoltà nell'esecuzione di algoritmi che, benché noti e risolvibili in linea teorica, non possono arrivare a fornire un risultato finale in tempi utili all'uomo.

È qui che entrano in gioco discipline come il **Calcolo Numerico** e l'**High Performance Computing**. Per quanto diverse, sono accomunate da un obbiettivo comune: risolvere problemi complessi tenendo conto dei limiti tecnologici attualmente esistenti.

Il calcolo numerico introduce il concetto di "**approssimazione**", utile per trasformare problemi matematici continui in algoritmi approssimati a cifre finite, eseguibili quindi da macchine con risorse limitate come i calcolatori attuali. L'HPC invece affronta l'esecuzione di algoritmi dal costo computazionale ottimo intrinsecamente alto utilizzando tecniche di **programmazione parallela** che, benché difficili da implementare, possono diminuire i tempi di esecuzione di interi ordini di grandezza.

In questa tesi verranno prese in esame le possibilità offerte dalla programmazione concorrente in CUDA C nell'implementazione di algoritmi compu-

tazionalmente costosi nel dominio del Calcolo Numerico. In particolare, verrà esaminato uno dei principali algoritmi di interpolazione polinomiale: il metodo delle basi di Lagrange.

Gli algoritmi di **interpolazione numerica** permettono di ricostruire funzioni matematiche a partire da un insieme di punti che costituiscono un piccolo sottoinsieme del dominio preso in esame. Nello specifico, l'interpolazione polinomiale serve a trasformare funzioni complesse da calcolare in semplici polinomi attraverso cui è possibile ottenere un'approssimazione accettabile del codominio della funzione originale. Questa trasformazione avviene mediante algoritmi dal costo computazionale nell'ordine di $O(n^2)$, dove n è il numero di punti su cui si vuole costruire il polinomio interpolante.

Tramite la progettazione e l'implementazione di un'**apposita libreria in CUDA C** verranno analizzate le possibilità offerte dall'algoritmo parallelo rispetto alla sua controparte seriale sia nelle tempistiche di esecuzione, che nel numero di operazioni svolte al secondo.

L'obbiettivo di questa tesi è quello di **massimizzare l'ottimizzazione dell'algoritmo** cercando di sfruttare quanto più possibile le peculiarità dell'architettura parallela CUDA.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Fondamenti di programmazione parallela

Le tecniche di progettazione e programmazione parallela costituiscono l'argomento di studio principale dell'high performance computing e consistono nel suddividere un programma tra più unità di computazione per ridurne i tempi di esecuzione.

1.1.1 Tassonomia delle architetture parallele

Le architetture parallele possono essere classificate in base a diversi criteri, ad esempio il soggetto del parallelismo [**1**]. Se si differenziano le operazioni svolte dalle unità di esecuzione, come ad esempio nell'elaborazione in pipeline, si tratta di un'architettura **task-parallel**. Se invece l'operazione svolta dalle varie unità di esecuzione è la stessa ma a cambiare sono i dati che ognuna di esse deve elaborare ci si trova di fronte a un'architettura di tipo **data-parallel**.

Un ulteriore criterio di classificazione delle architetture parallele è il grado di condivisione della memoria: quanto tutte le unità di elaborazione condividono la stessa area di memoria si parla di architettura a **memoria condivisa**. Questo tipo di architettura rende necessaria una gestione più attenta degli accessi in memoria in modo da evitare race condition.

Una **race condition** è un errore di scrittura di una variabile in memoria condivisa dovuto all'aggiornamento concorrente da parte di più di un'unità di esecuzione, rendendo il risultato finale non deterministico e dettato esclusivamente dalla politica di schedulazione. Quando invece non esiste una memoria condivisa e le unità di elaborazione devono coordinarsi attraverso uno scambio di messaggi si parla di architettura a **memoria distribuita**; la velocità di questo genere di parallelismo è determinata in larga parte dal volume dello scambio di messaggi e dal tempo di attesa di tale scambio (overhead). Come vedremo, CUDA è un ibrido di entrambe queste filosofie, presentando componenti logici che possono lavorare sia in un modo che nell'altro a seconda del contesto.

1.1.2 Pattern di programmazione parallela

Pur non essendo un paradigma di programmazione a sé stante, anche la programmazione parallela ha soluzioni comuni a problemi frequenti (pattern di progettazione)[**2**]. Quelli che torneranno utili in questa tesi sono due: embarassingly parallel e riduzione (figure 1.1 e 1.2).

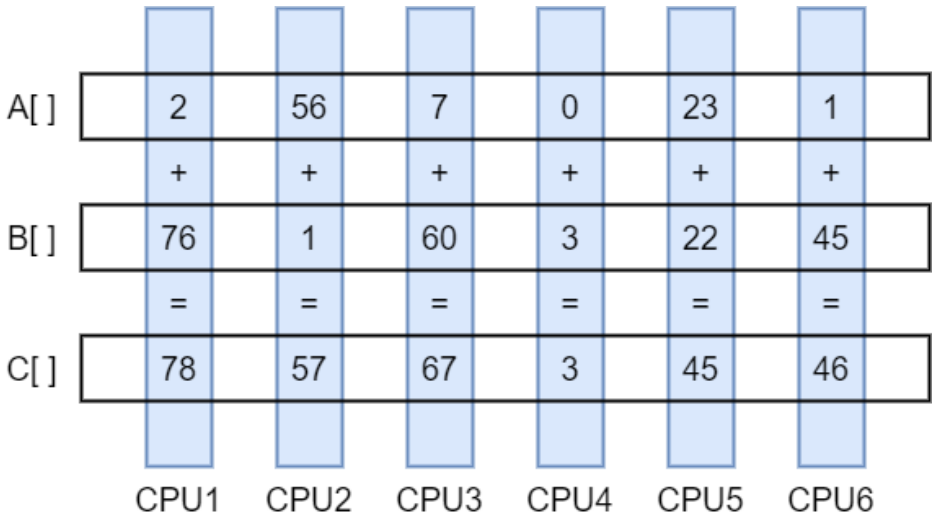

Figura 1.1: Esempio di somma tra array embarassingly parallel

Il pattern **embarassingly parallel** si applica quando la parallelizzazione risulta banale, con un'ovvia distribuzione dell'input tra le unità di elaborazione che non richieda comunicazione ulteriore tra esse. Un classico esempio è la somma tra array: se il numero di elementi è uguale o inferiore al numero di unità di elaborazione ogni unità si occupa della somma tra gli i-esimi elementi dei due array, riducendo il tempo di esecuzione da n (numero di elementi da sommare) a 1.

La **riduzione** invece consiste nell'applicazione di un operatore associativo a tutti gli elementi di un array. Esempi classici di operatori associativi sono operazioni matematiche come somma e prodotto e operazioni di confronto come la ricerca del valore minimo o massimo.

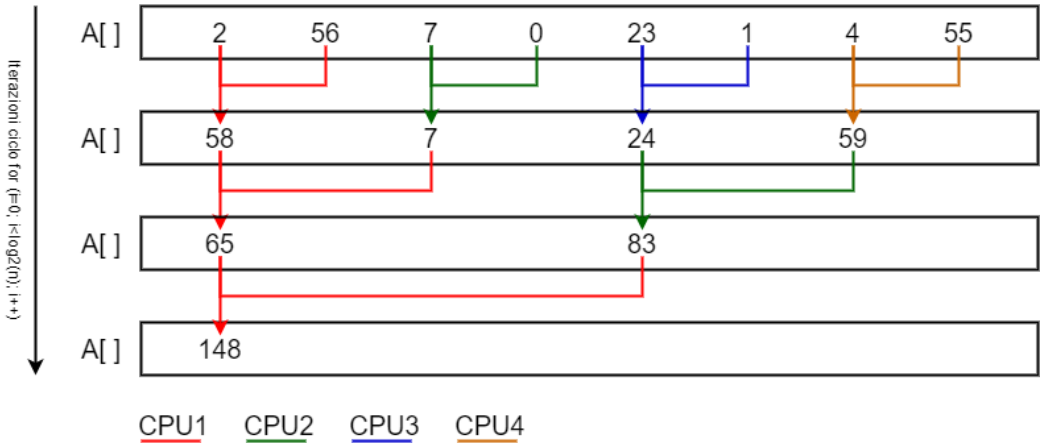

Figura 1.2: Esempio di sommatoria con riduzione

1.1.3 Misurazione delle prestazioni

L'efficacia di un'implementazione parallela si misura in base al miglioramento nei tempi di esecuzione (speedup) e nel throughput. Nello specifico, lo **speedup** rappresenta il miglioramento dei tempi di esecuzione del programma in funzione della quantità di unità di calcolo utilizzate e si calcola tramite questa formula:

$$S(p) = \frac{T_{seriale}}{T_{parallelo}}$$

dove:

- $T_{seriale}$ è il tempo di esecuzione ottenuto eseguendo il codice (già predisposto alla parallelizzazione) con una sola unità di calcolo.
- $T_{parallelo}$ è tempo di esecuzione del programma eseguito con p unità di computazione parallele.

Il **throughput** invece è il numero di elementi processati al secondo in funzione della dimensione dell'input.

1.2 Architettura CUDA

CUDA è un'architettura hardware proprietaria usata nell'ambito del calcolo in parallelo tramite scheda video che sfrutta sia il paradigma di computazione parallela a memoria condivisa, che quello a memoria distribuita [**3**]. Nel contesto dell'architettura CUDA, la macchina ospitante e la sua CPU si chiamano **Host**, mentre le schede video ospitate si chiamano **Device**.

Durante l'esecuzione di un programma, nel momento in cui l'host inizializza e richiede il lancio di un **kernel CUDA** attraverso un'interfaccia API proprietaria, questo viene eseguito sul device selezionato in modo indipendente e asincrono rispetto alla CPU dell'host.

1.2.1 Gestione del parallelismo

L'architettura logica CUDA si compone di diversi livelli, ognuno caratterizzato dal proprio tipo di parallelismo, da una differente disponibilità di memoria e da diversi gradi di isolamento (figura 1.3).

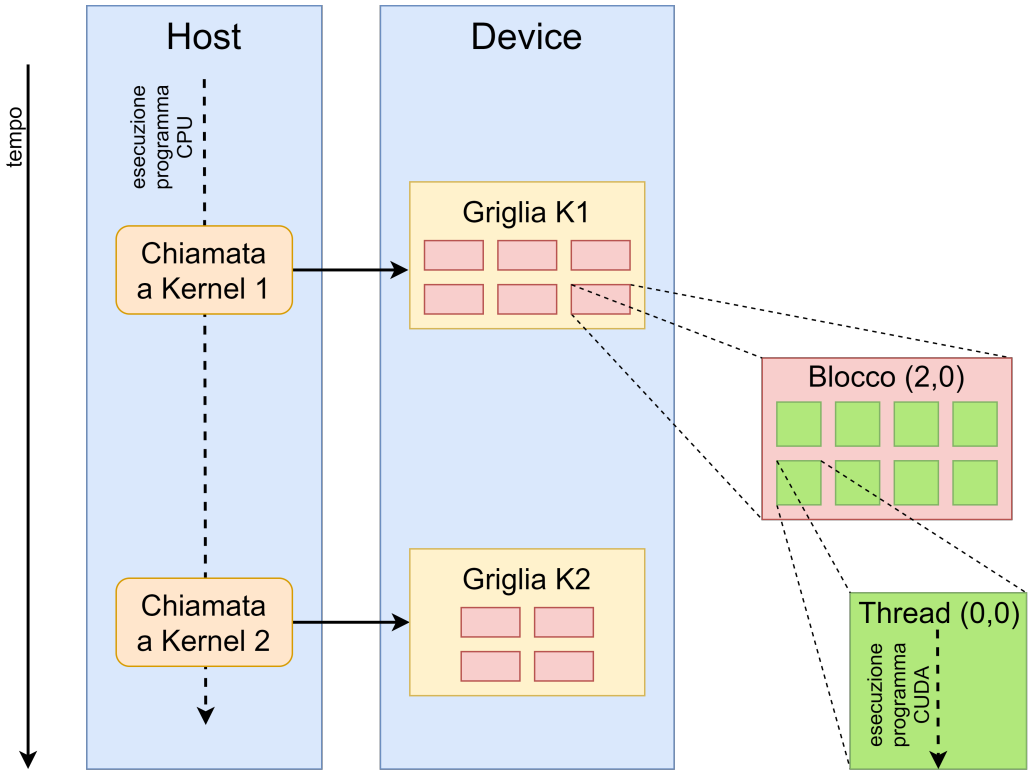

Figura 1.3: Esploso del parallelismo CUDA

Il livello più basso da considerare è quello del **thread**, l'unità fondamentale del parallelismo. Un thread CUDA è un flusso di esecuzione indipendente dagli altri che può utilizzare sia variabili locali che variabili in memorie condivise. Per quanto logicamente divisi, i thread vengono eseguiti dal device sempre a gruppi di 32 chiamati **warp** e sincronizzati in modo implicito.

I thread sono raggruppati in **blocchi**, a loro volta posizionati in una **griglia** inizializzata all'avvio del kernel. La divisione in blocchi consente un grado in più di sincronizzazione tra thread tramite la funzione `synchthread()`, che inserisce una barriera che tutti i thread dello stesso blocco devono raggiungere prima di procedere con l'esecuzione del resto del programma. Questo rende vantaggioso parallelizzare il lavoro a livello di blocco, in modo da ridurre al minimo i tempi di attesa dovuti alle eventuali dipendenze tra operazioni su uno stesso elemento del dominio.

È interessante notare come l'API di CUDA non metta a disposizione funzioni specifiche per ottenere la sincronizzazione in modo esplicito tra tutti i thread della griglia appartenenti a blocchi diversi. Per conseguirla viene sfruttato il fatto che l'esecuzione di kernel differenti sia asincrona rispetto alla CPU e seriale nel device. La serializzazione dei kernel in esecuzione sul device comporta che all'inizio di un nuovo kernel tutti i blocchi del kernel precedente abbiano terminato l'esecuzione: in questo modo è stata ottenuta una sincronizzazione implicita tra blocchi diversi (appartenenti allo stesso kernel).

1.2.2 Gestione della memoria

L'architettura CUDA mette a disposizione più di un tipo di memoria condivisa, il cui utilizzo da parte di un programmatore dipende dalla velocità di accesso e dal grado di condivisione desiderati (figura 1.4). La memoria condivisa più capiente è la memoria globale, esterna alla griglia e dal contenuto persistente tra un kernel e l'altro.

I blocchi dispongono di una memoria accessibile solo dai thread appartenenti allo stesso blocco chiamata shared memory. Questa memoria risulta vantaggiosa per il caching manuale dei dati usati più frequentemente dai thread a causa della velocità di accesso estremamente più alta rispetto alla memoria globale.

Parte della progettazione del programma realizzato in questa tesi si è concentrata sull'ottimizzazione degli accessi alla memoria, potenziale collo di bottiglia in caso di algoritmi che durante la loro esecuzione consultano i dati in ingresso molte volte, portando a una minimizzazione delle operazioni di I/O sulla memoria globale. Durante l'analisi delle scelte ingegneristiche che hanno portato alla versione finale del codice verranno prese in esame le "buone pratiche" di progettazione CUDA che ne costituiscono la motivazione.

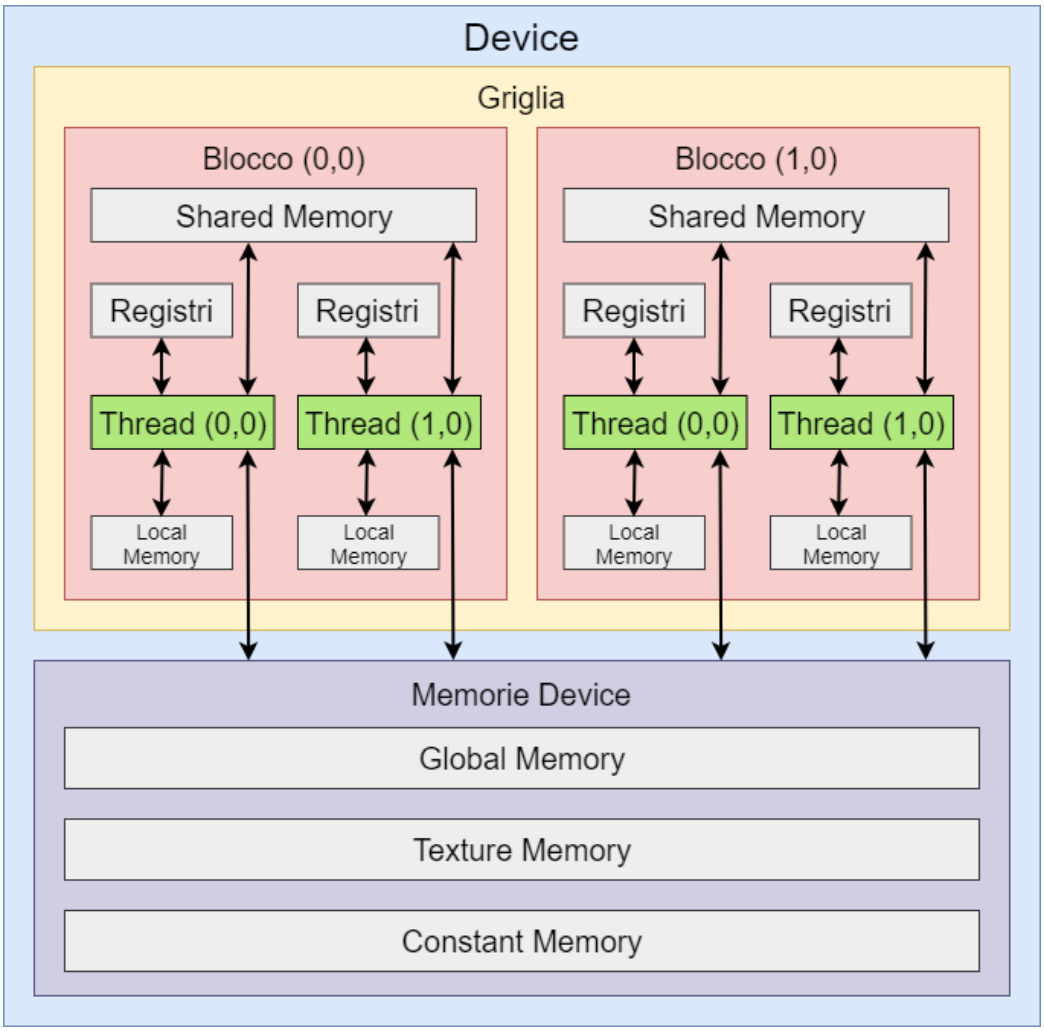

Figura 1.4: Schema della memoria

1.3 Interpolazione polinomiale

[4] L'obbiettivo principale dell'interpolazione è trovare una funzione interpolante $\Phi(x)$ che, date $n+1$ coppie di valori (x_j, y_j) dette **nodi** comprese in un intervallo $[\alpha, \beta]$, verifichi la condizione

$$\Phi(x_j) = y_j \quad \forall j \in \{0, \ldots, n\}$$

Una volta trovata, questa funzione $\Phi(x)$ può essere usata per calcolare il codominio di un insieme di ascisse $\{x'_1, ..., x'_S\}$ compreso nell'intervallo originale dei nodi. **Il numero S è la cardinalità del dominio da interpolare.**

Una funzione interpolante è **polinomiale** se $\Phi(x)$ è un polinomio algebrico e si dice di **grado n** se costruito su $n+1$ nodi. Spesso n è un numero "piccolo" nell'ordine delle decine, mentre S è un numero "grande" nell'ordine delle centinaia o delle migliaia: in figura 1.5 è possibile vedere un polinomio interpolante di grado $n = 9$ (e $n = 15$, a sinistra) calcolato in $S = 301$ punti scelti a campione nell'intervallo $[\alpha, \beta] = [-5, 5]$.

Tutti gli algoritmi di interpolazione, dati gli stessi nodi, convergono verso lo stesso polinomio. A fare la differenza nella precisione della ricostruzione sono la quantità e la distribuzione dei nodi.

1.3.1 Scelta dei nodi

Il criterio più intuitivo per la scelta dei nodi è quello di prenderne più possibili nell'intervallo utilizzando la **distribuzione equispaziata**.

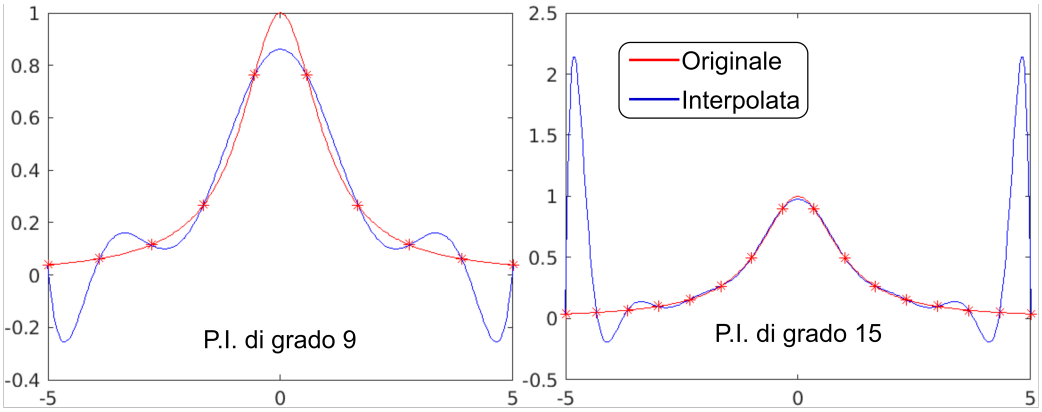

Figura 1.5: Funzione di Runge originale e interpolata con nodi equispaziati, aumentando il grado del polinomio.

È immediatamente visibile però come per certe funzioni l'aumento di nodi (con questa distribuzione) non corrisponda necessariamente a un aumento della precisione del polinomio interpolante, soprattutto in prossimità degli estremi (figura 1.5). All'aumentare dei numero di nodi di un polinomio infatti aumenta linearmente anche il numero di operazioni da eseguire per costruirlo, portando a un accumulo di errori dovuto alle approssimazioni dei numeri a virgola mobile che rende il polinomio numericamente instabile, dando origine al **fenomeno di Runge**. Ne consegue che per ottenere un livello di precisione accettabile è necessario utilizzare un polinomio interpolante di grado inferiore (ma il più vicino possibile) a una certa "soglia". Un modo intelligente per incrementare questa soglia senza aumentare l'instabilità numerica è quello di passare dalla distribuzione equispaziata a quella di Čebyšëv. Nei **nodi di Čebyšëv** le ascisse vengono calcolate mediante questa formula:

$$x_j = \cos \frac{(2j+1)\pi}{2(n+1)} \quad \forall j \in \{0, \ldots, n\}$$

Il cambio di distribuzione trasforma l'andamento dell'errore in funzione del grado da lineare a logaritmica, e rende sensato lavorare con polinomi di grado "alto". Nella figura 1.6 è possibile notare come a parità di n il polinomio costruito usando la distribuzione di Čebyšëv sia molto più accurato rispetto a quello costruito con la distribuzione dei nodi equispaziata, mentre nella figura 1.7 risulta evidente come questa precisione continui ad aumentare all'aumentare dei nodi, arrivando a valori di n uguali o maggiori di 50 senza accumulare errori [**5**].

La libreria oggetto di questa tesi non calcola i nodi interpolanti, prendendoli semplicemente in ingresso. Tuttavia, è stato ritenuto utile analizzare il fenomeno di Runge e la distribuzione dei nodi di Čebyšëv per definire l'ordine di grandezza dell'input, in modo da dimensionare correttamente la griglia CUDA e garantire un buon margine di flessibilità nei casi d'uso.

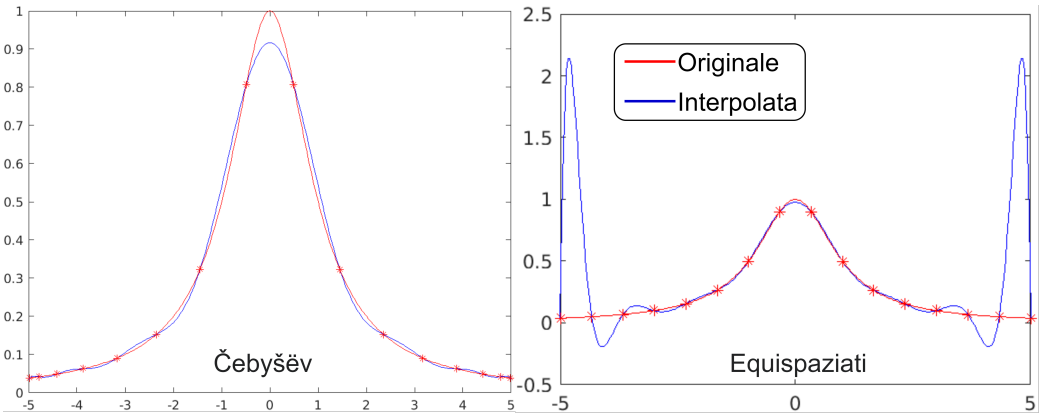

Figura 1.6: Funzione di Runge interpolata ($n = 15$) con nodi equispaziati e di Čebyšëv

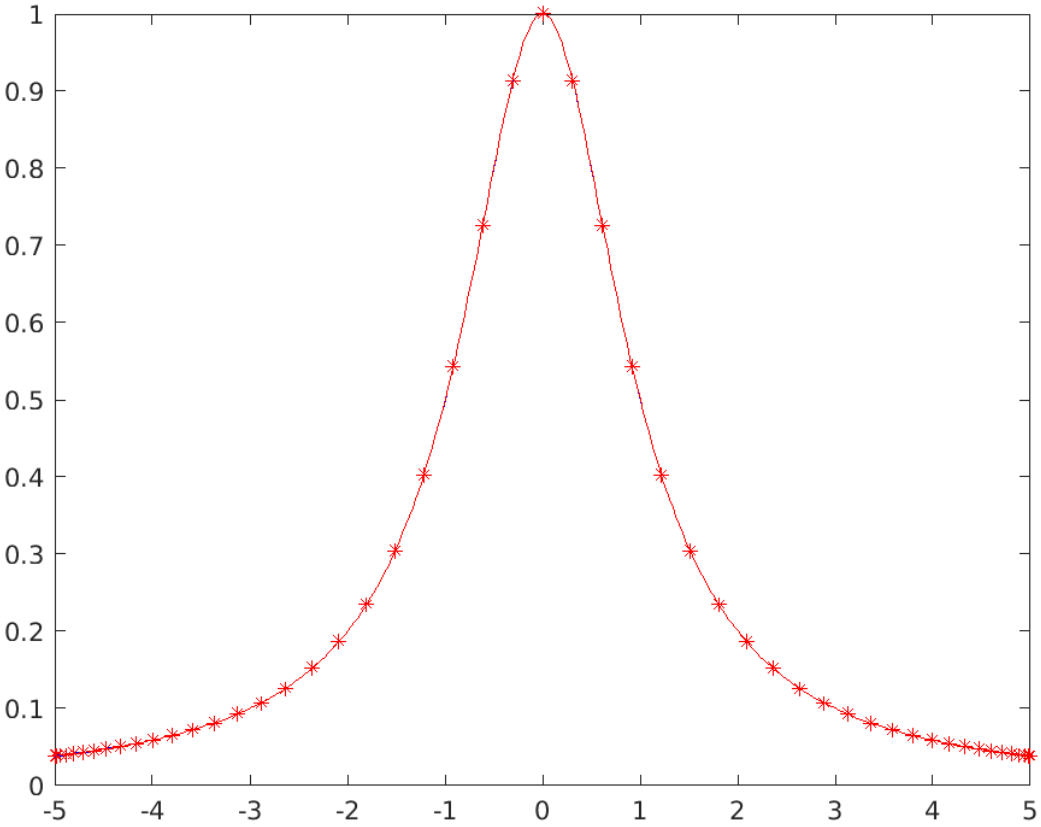

Figura 1.7: Funzione di Runge interpolata con $n = 50$ e nodi di Čebyšëv

1.3.2 Algoritmo delle basi di Lagrange

Un **polinomio fondamentale di Lagrange** di grado n (figura 1.8) è una funzione matematica costruita su uno (il j-esimo) degli $n+1$ nodi in cui:

$$\begin{cases} L_j(x_i) = 0 \quad \forall i \in \{0, ..., n\} : i \neq j \\ L_j(x_j) = 1 \end{cases} \tag{1.1}$$

Per costruire un polinomio fondamentale di Lagrange che restituisca 1 sul j-esimo nodo si usa questa formula:

$$L_j(x') = \prod_{i=0, i\neq j}^{n} \gamma_{ij} \tag{1.2}$$

con

$$\gamma_{ij} = \frac{x' - x_i}{x_j - x_i} \tag{1.3}$$

dove:

- x_j e x_i sono le ascisse del j-esimo e l'i-esimo nodo.
- x' è l'ascissa in cui calcolare $L_j(x')$ per trovare y'.

L'operazione che consente di assemblare il polinomio interpolante è la **somma tra tutti gli $n+1$ polinomi fondamentali di Lagrange** costruiti sui nodi. Ricordando che l'obbiettivo del polinomio interpolante è ottenere $\Phi(x_j) = y_j \quad \forall j \in \{0, \ldots, n\}$ e tenendo presente che $L_j(x_j) = 1$ è immediatamente intuibile che oltre a sommare tra loro i vari L_j sia necessario anche moltiplicare L_j per la corrispondente y_j (cioè l'ordinata del j-esimo nodo). La formula del **polinomio interpolante di Lagrange** è quindi

$$\Phi(x') = \sum_{j=0}^{n} y_j L_j(x') \tag{1.4}$$

che estesa diventa

$$\Phi(x') = \sum_{j=0}^{n} \left(y_j \prod_{i=0, i\neq j}^{n} \frac{x' - x_i}{x_j - x_i} \right) \tag{1.5}$$

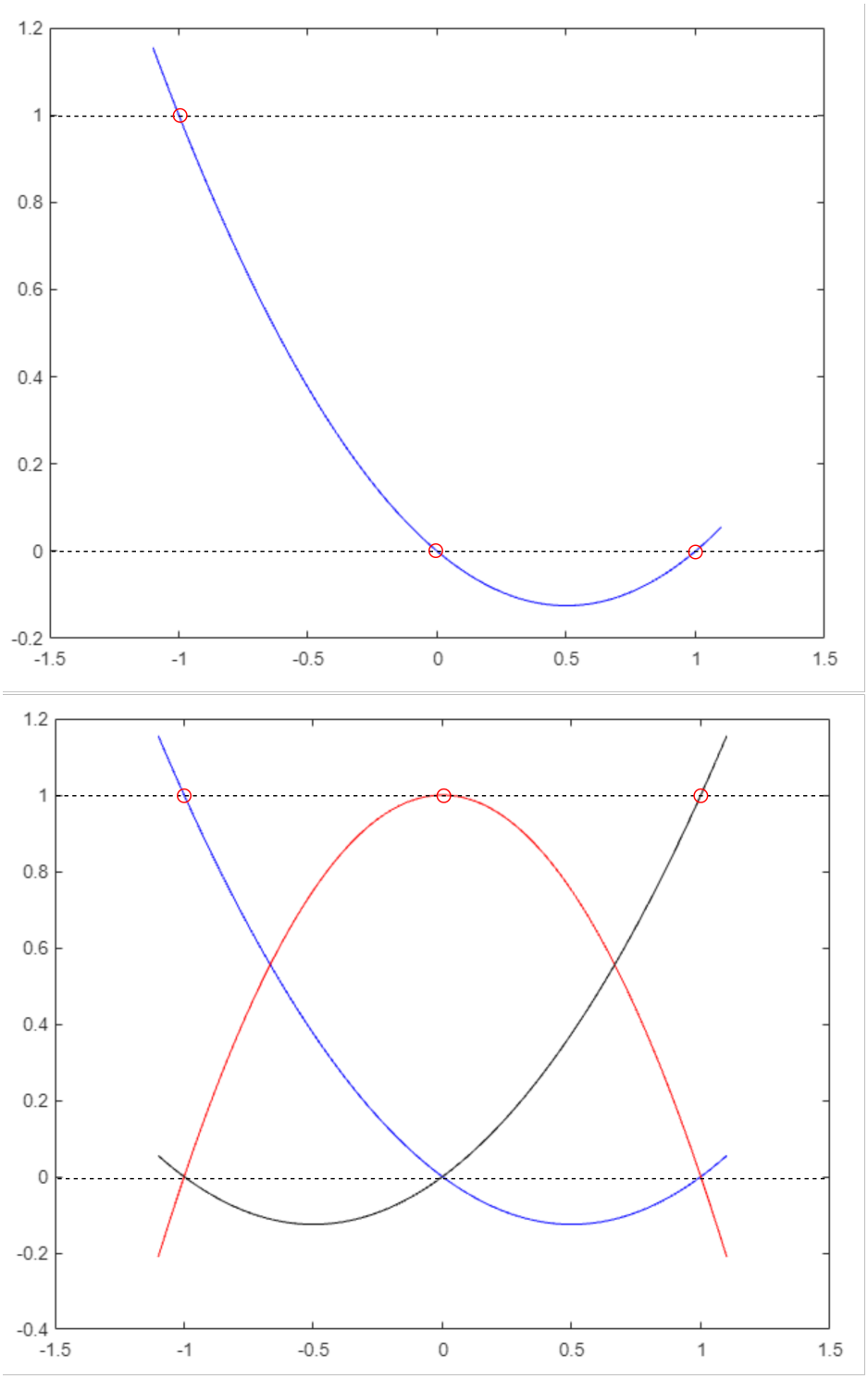

Figura 1.8: Esempio di polinomi fondamentali di Lagrange con $n = 2$ e $x = \{-1, 0, 1\}$. $L_0(x')$ in blu, $L_1(x')$ in rosso e $L_2(x')$ in nero.

Capitolo 2

Progettazione Concettuale

In questo capitolo adatteremo l'algoritmo delle basi di Lagrange all'architettura CUDA, decidendo dove e come applicare i pattern di progettazione parallela e quale paradigma di programmazione utilizzare.

La prima scelta di design riguarda proprio quest'ultimo punto: per sfruttare al meglio il parallelismo massivo di CUDA e garantire il massimo grado di interoperabilità del codice si è deciso di utilizzare il paradigma di **programmazione procedurale di ANSI C**, escludendo utilizzi avanzati di C++ che permetterebbero di usare funzioni dinamiche generate a tempo di esecuzione attraverso cui calcolare $\Phi(x')$ di tutto il dominio da interpolare. Bisogna quindi escludere la possibilità, offerta dai linguaggi di programmazione parzialmente o totalmente funzionali, di rendere x' l'input di un polinomio già calcolato e dare per scontato che si stia lavorando esclusivamente con delle costanti numeriche. Questo comporta che **si debba ricalcolare il polinomio interpolante per tutti gli *S* punti da interpolare**, facendo esplodere i tempi di esecuzione dell'algoritmo.

Poste queste condizioni, la formula estesa con un dominio da interpolare di S punti assume questa forma:

$$y'_s = \sum_{j=0}^{n} \left(y_j \prod_{i=0, i\neq j}^{n} \frac{x'_s - x_i}{x_j - x_i} \right) \quad \forall s \in \{1, \ldots, S\} \tag{2.1}$$

La differenza tra x' nell'equazione 1.5 e x'_s nell'equazione 2.1 è che:

- x' rappresenta una variabile x simbolica e $y' = \Phi(x')$ è un **polinomio**.
- x'_s rappresenta l's-esima ascissa nota da interpolare, e $y'_s = \Phi(x'_s)$ è un **numero**.

2.1 Algoritmo seriale

A questo punto possiamo tradurre la formula 2.1 in uno pseudocodice seriale:

input : n nodi (x, y) e S ascisse $xdom$ da interpolare
output: S ordinate interpolate salvate nell'array res

```
1  for s ← 1 to S do
2      somma ← 0;
3      prodotto ← 1;
4      for j ← 1 to n do
5          for i ← 1 to n do
6              if i ≠ j then
7                  parziale ←(xdom[ s ] - x[ i ]) / (x[ j ] - x[ i ]);
8              else
9                  parziale ← 1;
10             end if
11             prodotto ← prodotto * parziale;
12         end for
13         somma ← somma + prodotto;
14         prodotto ← 1;
15     end for
16     res[s] ← somma;
17 end for
```

in cui:

- Le righe da 6 a 10 rappresentano il calcolo degli elementi atomici.

$$\gamma_{ijs} = \begin{cases} \frac{x'_s - x_i}{x_j - x_i} & : i \neq j \\ 1 \quad \text{altrimenti} \end{cases} \tag{2.2}$$

- Il *ciclo for* dalla riga 5 alla riga 12 rappresenta il calcolo del j-esimo polinomio fondamentale di Lagrange per l's-esimo nodo.

$$L_j(x'_s) = \prod_{i=0, i\neq j}^{n} \gamma_{ijs} \tag{2.3}$$

- Il *ciclo for* dalla riga 4 alla riga 15 rappresenta il calcolo della sommatoria tra tutti i polinomi di Lagrange calcolati nel punto x'_s. Al termine di questo ciclo si ha il risultato $\Phi(x'_s)$ (salvato alla riga 16).

$$\Phi(x'_s) = \sum_{j=0}^{n} y_j L_j(x'_s) \tag{2.4}$$

- Ogni iterazione del ciclo più esterno in s è una x'_s diversa; il ciclo in sé rappresenta il "$\forall s \in \{1, ..., S\}$" della formula 2.1.

2.2 Architettura parallela ideale

2.2.1 Analisi delle dipendenze

Possiamo osservare come i γ_{ijs} siano **elementi atomici** della formula, in quanto il loro valore non dipende dalle iterazioni precedenti di nessun ciclo. Similmente, ogni iterazione del ciclo più esterno in s non interagisce con nessun elemento delle iterazioni passate, azzerando ogni volta le variabili "somma" e "prodotto". Possiamo dire quindi che sia il calcolo degli elementi γ_{ijs} che quello delle diverse $y'_s = \Phi(x'_s)$ sia di tipo **embarassingly parallel**.

Esistono però una serie di dipendenze "**part-of**" tra i diversi elementi di questa formula. È vero infatti che:

- Ogni γ_{ijs} è **parte di** un $L_j(x'_s)$ in quanto elemento della sua produttoria (equazione 2.3).
- Ogni $L_j(x'_s)$ è **parte di** una $y'_s = \Phi(x'_s)$ in quanto elemento della sommatoria (equazione 2.4).

Ogni $y'_s = \Phi(x'_s)$ può quindi essere pensato come un insieme di γ_{ijs}, che si sommano o moltiplicano tra loro a seconda degli indici i e j.

2.2.2 Griglia CUDA e divisione in fasi

Il raggruppamento dei γ_{ijs} (calcolabili in parallelo) in S insiemi (a loro volta calcolabili in parallelo) è il perno su cui si basa la versione CUDA del programma (figura 2.1). L'idea è quella di mantenere la gerarchia "part-of" basata sull'indice s assegnando

- Ogni punto da interpolare x'_s a un **blocco**.
- Ogni valore γ_{ijs} a un **thread**.

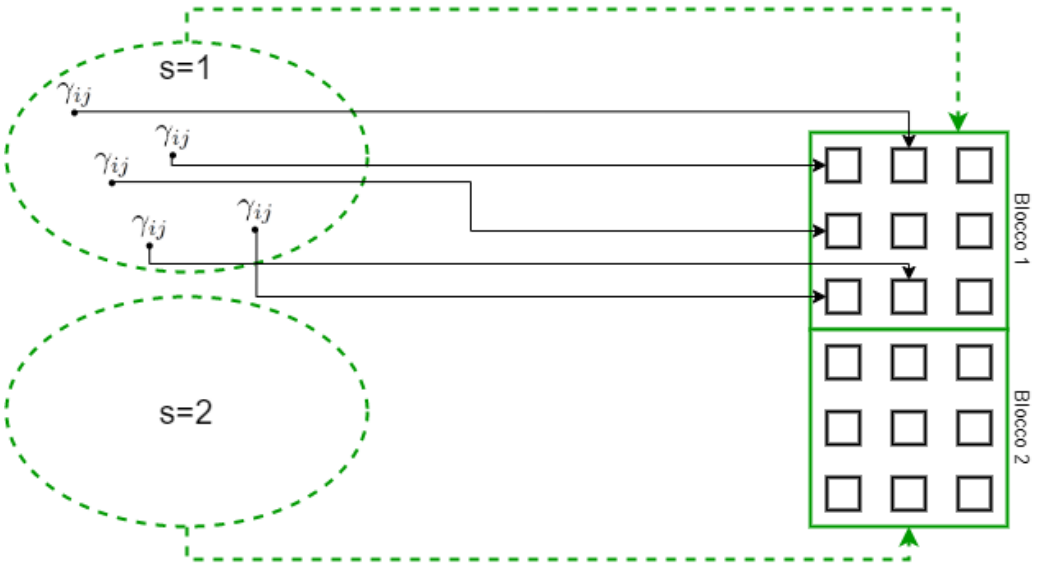

Figura 2.1: Associazione blocco-y'_s e thread-γ_{ijs}.

In questo modo la griglia sarà grande S blocchi, ogni blocco avrà $(n+1)^2$ thread e ogni thread calcolerà un solo valore γ, eseguendo $S(n+1)^2$ operazioni contemporaneamente.

Sfortunatamente parte delle dipendenze *part-of* comportano una serializzazione del calcolo di alcuni elementi della formula 2.1, come gli $L_j(x'_s)$ e i risultati y'_s. Risulta evidente come gli $L_j(x'_s)$ richiedano, per essere calcolati mediante produttoria (equazione 2.3), tutti i γ_{ijs} **già calcolati**. Similmente anche il risultato finale y'_s richiede, prima di essere trovato tramite sommatoria, che tutti i valori $L_j(x'_s)$ siano noti (equazione 2.4). I thread coinvolti nel calcolo della sommatoria e della produttoria verranno organizzati secondo il pattern **riduzione**.

Questa serializzazione porta alla seguente **suddivisione in fasi**:

- Fase 1: calcolo di tutti i valori atomici γ_{ijs}.
- Fase 2: calcolo di tutti i polinomi fondamentali di Lagrange $L_j(x'_s)$ mediante riduzione-produttoria.
- Fase 3: calcolo di tutte le S y'_s mediante riduzione-sommatoria.

Essendo un'architettura parallela *ideale* per il momento stiamo assumendo di avere blocchi di grandezza infinita: vedremo nei prossimi capitoli che non è così, ma prima di considerare anche i limiti hardware di CUDA è necessario approfondire quanto detto finora.

2.2.3 Fase 1: calcolo degli elementi atomici

I dati in nostro possesso all'inizio della computazione sono questi:

- Array di $n+1$ x e y dei nodi di interpolazione.
- Array di S x' da interpolare.

L'obbiettivo di questa fase è parallelizzare il calcolo degli elementi γ_{ijs} secondo la formula 2.2. Questo corrisponde alla parallelizzazione della porzione di pseudocodice che va dalla riga 6 alla riga 10.

Considerando il raggruppamento descritto nel capitolo 2.2.2, verrà inizializzata una griglia di S blocchi di forma quadrata da $(n+1)^2$ thread ciascuno. Aver scelto una forma dei blocchi quadrata ci consente di sfruttare le coordinate dei thread all'interno del blocco per ottenere tutte le combinazioni possibili di i e j, senza ripetizioni e facendo calcolare un solo valore a ogni thread. L'indice s corrisponderà all'indice del blocco nella griglia, in modo che ogni blocco calcoli gli elementi γ_{ijs} appartenenti a una x'_s diversa.

La figura 2.2 rappresenta graficamente la griglia durante la fase 1 con $n = 2$ (quindi 3 nodi) e $S = 4$.

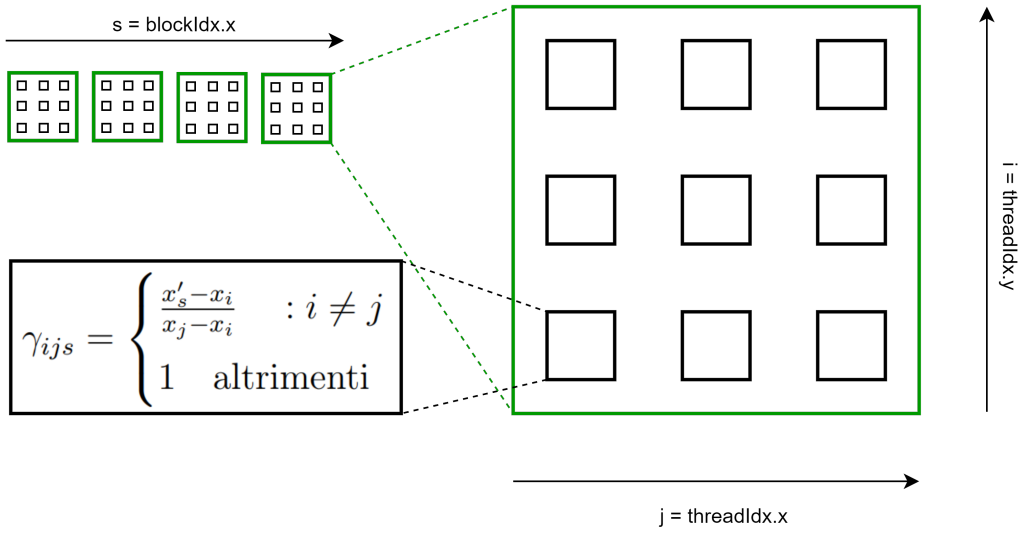

Figura 2.2: Fase 1 - Ogni thread calcola un γ diverso.

2.2.4 Fase 2: calcolo degli L_j tramite riduzione-produttoria

Una volta calcolati tutti i γ_{ijs} è possibile procedere con il calcolo parallelo di tutti gli $S(n+1)$ $L_j(x'_s)$. In questa fase si sta parallelizzando la linea 11 dello pseudocodice, distribuendo le istanze del ciclo in j lungo le colonne dei blocchi. Ricordiamo che $L_j(x'_s)$ è un **numero** che rappresenta il j-esimo polinomio fondamentale di Lagrange calcolato nell's-esima x' (da interpolare). Essendo l'indice j corrispondente all'indice $threadIdx.x$, ogni colonna di thread corrisponderà a una riduzione-produttoria diversa.

Il calcolo degli L_j è *embarassingly parallel* perché ogni colonna ha già tutti i dati di cui ha bisogno per trovare il risultato (secondo la formula 2.3). In questo modo è possibile **far avvenire tutte le riduzioni-produttoria contemporaneamente** all'interno del blocco. Data la forma della griglia, questo è vero per ogni blocco. Durante questa fase quindi vengono eseguite un totale di $S(n+1)$ riduzioni in parallelo (figura 2.3).

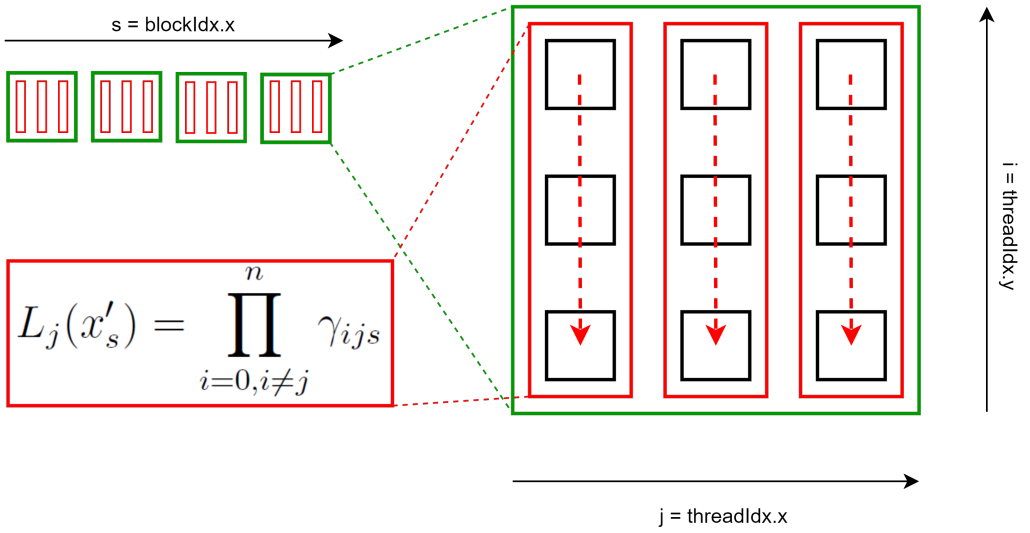

Figura 2.3: Fase 2 - Ogni colonna di thread calcola un $L_j(x'_s)$ diverso.

2.2.5 Fase 3: calcolo dei $\Phi(x'_s)$ tramite riduzione-sommatoria

Al termine della fase 2 nella prima riga di ogni blocco sono presenti i $L_j(x'_s)$ che, moltiplicati per la j-esima y dei nodi, vanno a costituire gli elementi della sommatoria (equazione 2.4) corrispondente alla riga 13 dello pseudocodice. Durante la fase 3 i thread delle prime righe dei blocchi effettueranno le S riduzioni-sommatoria che troveranno le y' (figura 2.4). Ogni sommatoria appartiene a un solo y'_s, e pertanto è *embarassingly parallel* rispetto alle altre.

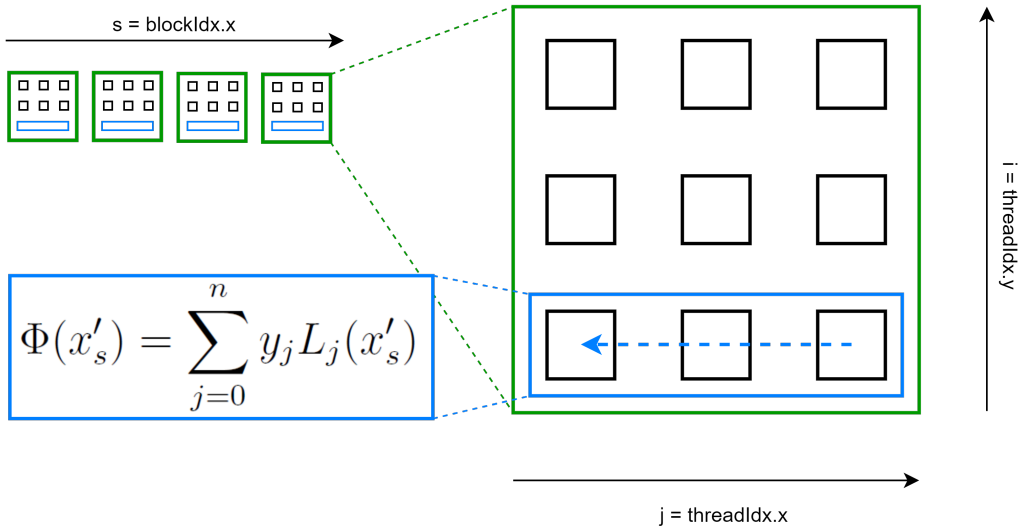

Figura 2.4: Fase 3 - la prima riga di ogni blocco calcola la sua sommatoria.

Al termine della fase 3 avremo il risultato della sommatoria alle coordinate locali $(0, 0)$ di ogni blocco che, essendo assegnato biunivocamente a ogni coppia (x'_s, y'_s), corrisponde anche al risultato finale y'_s interpolata (figura 2.5).

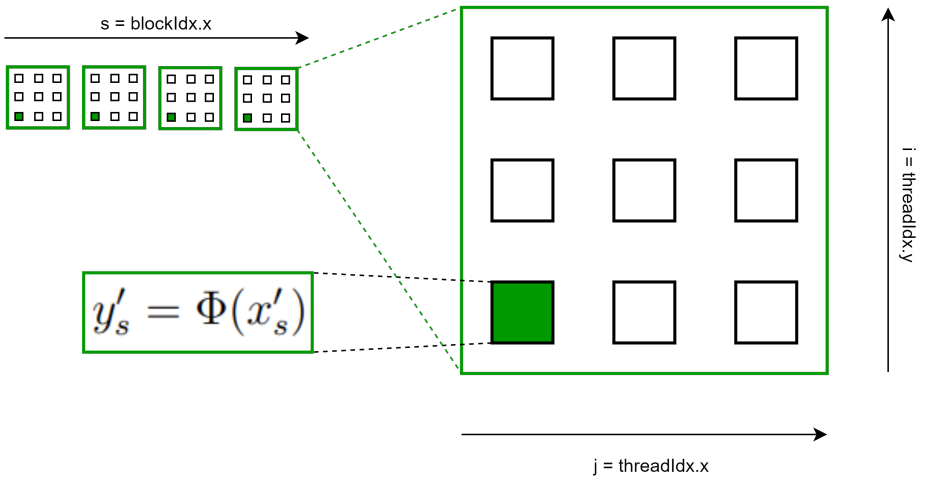

Figura 2.5: Risultato finale.

2.3 Architettura parallela reale

L'architettura CUDA è basata sull'hardware della scheda video, e in quanto tale presenta limiti tecnici riconducibili alla quantità di memoria finita disponibile su chip. Per la costruzione e il testing del programma oggetto di questa tesi è stata utilizzata una scheda video Nvidia GTX 960, con le seguenti caratteristiche:

Cuda capability	5.x
Massimo numero di thread per blocco	1024
ThreadIdx.x massimo	1024
ThreadIdx.y massimo	1024
BlockIdx.x massimo	2'147'483'647
BlockIdx.y massimo	65'535
Grandezza massima di shared memory per blocco	49'152 byte
Grandezza global memory	2 GB
Larghezza di banda bus global memory	128 Byte

Tabella 2.1: Specifiche GPU

Il limite più impattante, comune a tutte le architetture CUDA con compute capability superiore alla 2.x, è quello dei **1024 thread per blocco**, che con blocchi di forma quadrata comportano lati di grandezza massima pari a 32. Dovendo processare $(n+1)^2$ valori per blocco, questo limite rende impossibile usare (senza modifiche) l'architettura descritta nel capitolo precedente per calcolare polinomi interpolanti di grado $n > 31$, precludendo la larga fetta di casi d'uso in cui i benefici dell'elaborazione parallela sarebbero maggiori.

Nella figura 2.6, ponendo l'ampiezza di lato massima = 3 (invece di 32) e $n = 4$ (quindi 5 nodi) è possibile vedere come:

- I γ_{ijs} con indice $i >= blockDim.y$ o $j >= blockDim.x$ non saranno calcolati da nessun thread durante la fase 1 (**limite verticale**).
- Gli $L_j(x'_s)$ con indice $j >= blockDim.x$ non saranno calcolati da nessuna colonna di thread durante la fase 2, facendo mancare alcuni elementi alla sommatoria della fase 3 (**limite orizzontale**).

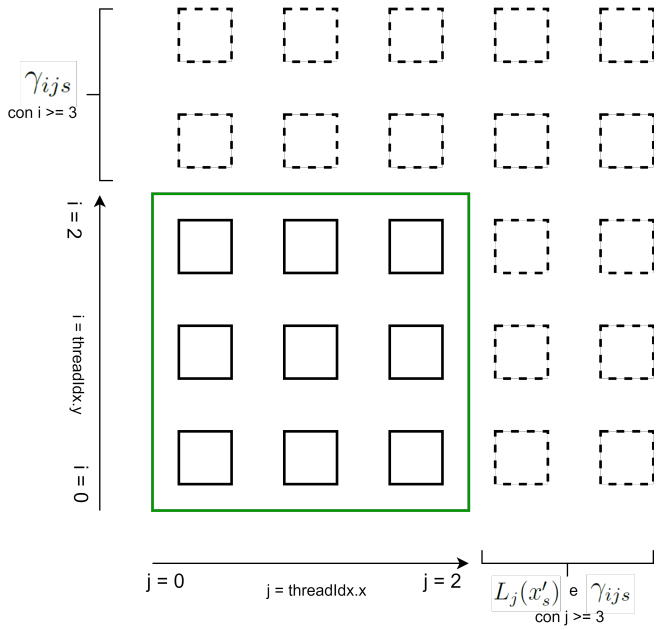

Figura 2.6: Limite verticale e limite orizzontale.

2.3.1 Limite verticale e cambio semantica dei thread

Il primo passo verso la compatibilità del programma con polinomi interpolanti di grado elevato è quello di gestire il surplus di γ_{ijs} da calcolare sulla verticale. Per farlo è necessario far calcolare a un thread più di un valore, scegliendo il pattern giusto affinché lo stesso valore non venga calcolato più volte.

Al posto di complicare ulteriormente l'architettura inserendo il pattern *master-worker* (che assegnerebbe dinamicamente i valori da calcolare ai primi thread disponibili) si è scelto di assegnare in modo arbitrario ai thread tutti i valori il cui indice i sia multiplo di $blockDim.y$ con uno scostamento di $threadIdx.y$. In altre parole il blocco diventa una finestra scorrevole sull'asse y in cui **ogni thread "consuma" tutti i γ_{ijs} con $i - threadIdx.y$ multipla di $blockDim.y$**, moltiplicandoli di volta in volta e ottenendo una produttoria parziale già al termine della fase 1.

Il meccanismo di finestra scorrevole viene implementato attraverso un **ciclo while** (eseguito da ogni thread) simile al seguente pseudocodice:

```
1 i ← threadIdx.y;
2 parziale ← 1;
3 while i ≤ n do
4 |   γ_ijs ← valore secondo equazione 2.2;
5 |   parziale ← parziale * γ_ijs;
6 |   i ← i + blockDim.y;
7 end while
```

Formalmente il risultato finale di un thread a fine fase 1 passa da $\text{thread}_{ijs} = \gamma_{ijs}$ a quello mostrato nella seguente formula:

$$\begin{aligned} \text{thread}_{ijs} &= \gamma_{ijs} \cdot \gamma_{(i+blockDim.y)js} \cdot \gamma_{(i+2 \cdot blockDim.y)js} \cdot \gamma_{(i+3 \cdot blockDim.y)js} \cdots \\ &= \prod_{\substack{k=0 \\ i+k \cdot blockDim.y <= n}}^{\lfloor (n+1)/blockDim.y \rfloor} \gamma_{(i+k \cdot blockDim.y)js} \end{aligned} \tag{2.5}$$

Nella figura 2.7 viene mostrata la distribuzione dei calcoli tra i thread durante la fase 1 modificata ponendo $blockDim.y = 3$ e $n = 4$: possiamo notare come ad esempio il thread (0,0) calcoli la produttoria parziale tra gli elementi $\gamma_{0,0,s}$ e $\gamma_{3,0,s}$. Se invece si fosse posto $n = 10$ (11 nodi), il thread (0,0) avrebbe calcolato la produttoria tra $\gamma_{0,0,s}$, $\gamma_{3,0,s}$, $\gamma_{6,0,s}$ e $\gamma_{9,0,s}$, mentre il thread (0,1) (quello immediatamente sopra) avrebbe calcolato $\gamma_{1,0,s}$, $\gamma_{4,0,s}$, $\gamma_{7,0,s}$ e $\gamma_{11,0,s}$.

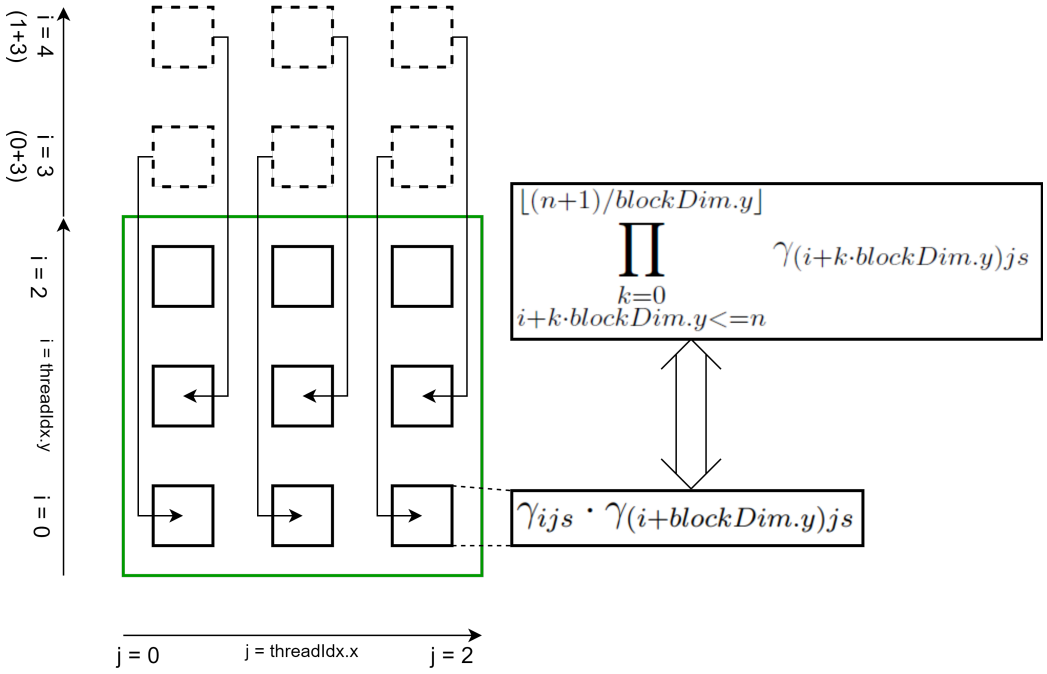

Figura 2.7: Al termine della fase 1 ogni thread ha eseguito una produttoria parziale.

Una volta compressi i valori in più sulla verticale all'interno della griglia, **la fase 2 può svolgersi nello stesso modo descritto dalla figura 2.3**. Questo perché durante la fase 1 abbiamo semplicemente pre-moltiplicato tra loro alcuni elementi della produttoria della fase 2 **senza ripetizioni** e includendo **tutti i valori** γ_{ijs}.

2.3.2 Limite orizzontale e cambio semantica dei blocchi

Riprendendo l'esempio in figura 2.6 ci si può rendere conto che un solo blocco, basando l'indice j sull'indice $threadIdx.x$, con un dominio troppo grande non è più in grado di coprire i $j >= blockDim.x$.

È quindi necessario trasformare l'associazione y'_s-blocco da "uno a uno" a "uno a molti" dove i "molti" sono un **insieme di blocchi in cui ogni blocco prende in carico una certa quantità di** $L_j(x'_s)$ sull'asse x e al termine della fase 3 trova il proprio risultato parziale della sommatoria. L'entità blocco non è più sinonimo di $y'_s = \Phi(x'_s)$ nella sua interezza, ma solo di un sottoinsieme di addendi della sua riduzione-sommatoria finale (figura 2.8).

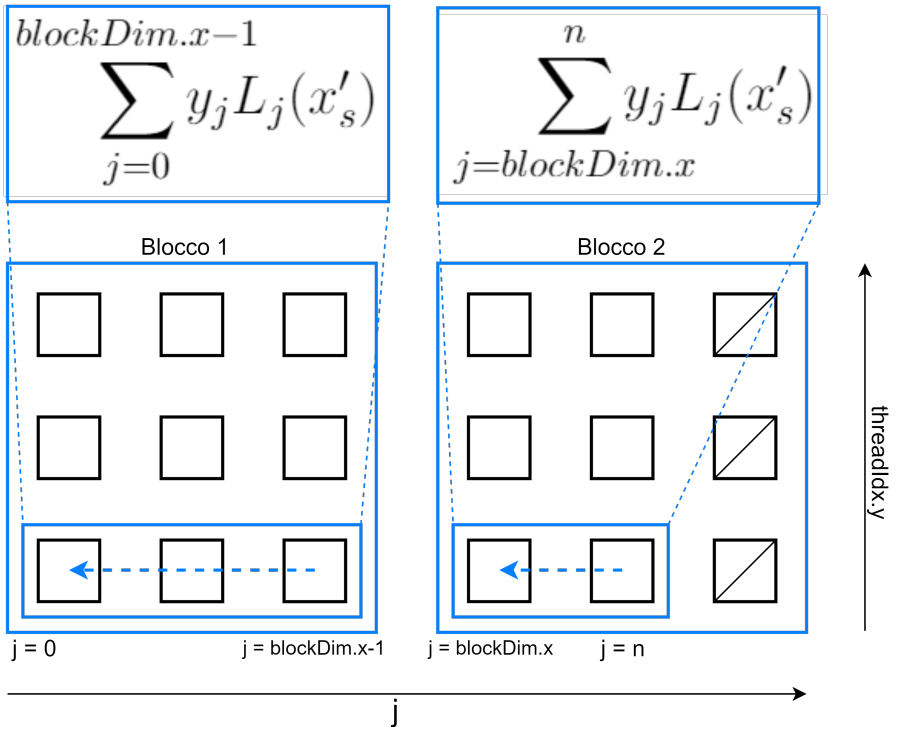

Figura 2.8: Fase 3 modificata distribuendo la sommatoria tra più blocchi.

2.3.3 Fase 4: riduzione inter-blocco

Questa trasformazione implica due cose:

- L'introduzione di un'**ulteriore riduzione inter-blocco** per ottenere il risultato finale.
- La **modifica della forma della griglia**, che è stata fino ad ora un semplice array di blocchi in cui ogni blocco corrispondeva a un punto da interpolare.

La riduzione inter-blocco può essere considerata una vera e propria "**fase 4**" e, dovendo utilizzare dati elaborati da blocchi diversi, necessita di una sincronizzazione dell'intera griglia. Come già visto nell'introduzione, un modo per sincronizzare più blocchi tra loro e assicurarsi che tutti i dati necessari a questa fase siano stati calcolati è quello di **inizializzare due kernel separati e sequenziali**. In questo modo al termine del primo kernel, che avrà eseguito le prime tre fasi e ne avrà salvato il risultato in delle apposite strutture dati, potrà partire il kernel che eseguirà la fase 4. L'implementazione di questa fase verrà discussa nei prossimi capitoli, mentre in figura 2.9 è possibile vedere in modo concettuale quale sia il suo compito.

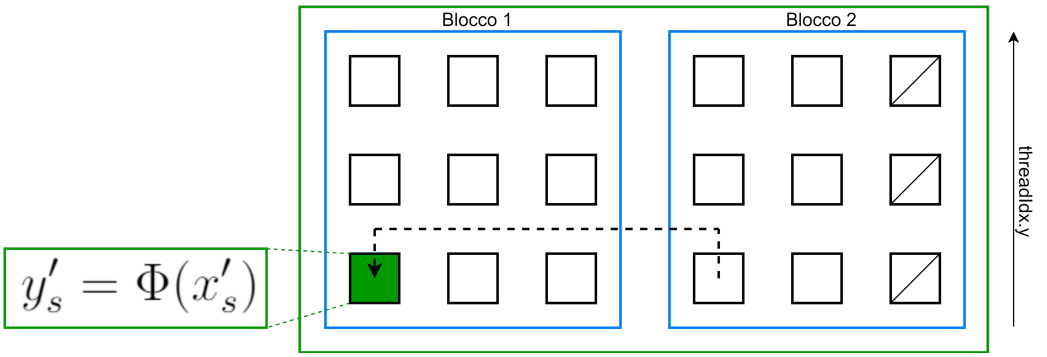

Figura 2.9: Un gruppo di blocchi è assegnato a una y'_s, che al termine della fase 4 viene trovata mettendo insieme i risultati parziali della fase 3.

2.3.4 Forma della griglia

Come avrete notato, la forma della griglia e gli indici sull'asse orizzontale delle figure 2.8 e 2.9 sono stati volutamente mantenuti vaghi: sappiamo che questi ultimi corrispondono all'indice j della sommatoria (ciclo a riga 4 nello pseudocodice) ma non abbiamo ancora definito *come* j venga associato ai thread. Questo perché a causa dei limiti di CUDA relativi alla dimensione della griglia è necessario definire l'associazione in modo controintuitivo.

Come è possibile vedere nella tabella 2.1 il numero di blocchi posizionabili sull'asse x è estremamente più alto rispetto a quelli dell'asse y. Sappiamo anche che nell'interpolazione con le basi di Lagrange n è nell'ordine delle decine (massimo centinaia) mentre il numero di punti da interpolare S è un numero grande a piacere nell'ordine delle migliaia, se non milioni. Risulta quindi più coerente con l'architettura CUDA far coincidere:

- **L'indice s** con **la coordinata x del blocco** (*blockIdx.x*).
- **L'indice del blocco all'interno del proprio gruppo** (assegnato al calcolo di una sola y_s) **alla sua coordinata y**.

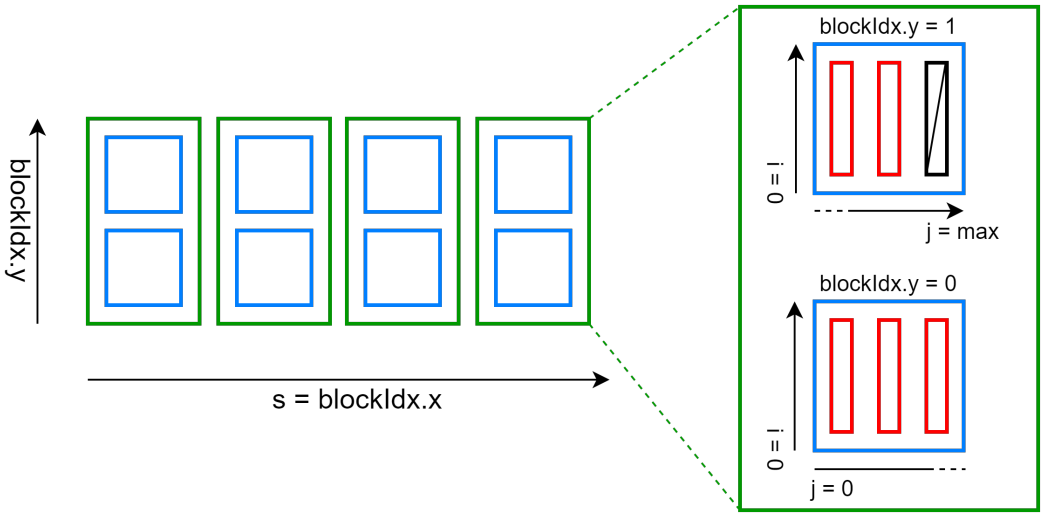

Figura 2.10: Forma della griglia con S=4, $\Psi = 2$ e $n = 4$ (5 nodi).

La forma che ne risulterà nel caso d'uso medio è "larga e bassa":

- "Larga" perché i punti da interpolare saranno presumibilmente migliaia.
- "Bassa" perché **la quantità Ψ di "blocchi per y_s"** sarà nell'ordine delle unità o delle decine, dato che $\Psi = \frac{n+1}{32}$ dove n è "piccolo" e 32 è dato dall'ampiezza massima del lato del blocco sull'asse x con una forma dei blocchi quadrata.

Questo posizionamento comporta un algoritmo di **scelta degli indici j dei thread sull'asse x** che tenga conto anche dell'altezza del blocco, corrispondendo essa all'indice del "blocco nel gruppo".

$$j = threadIdx.x + blockIdx.y * blockDim.x$$

2.4 Fase 0: calcolo delle costanti ottimizzato

All'interno di questa architettura, anche tra y'_s diverse, vengono eseguite le stesse operazioni sulle stesse costanti durante il calcolo dei γ_{ijs} nella fase 1 (figura 2.7). In questa sezione vedremo come facendo qualche modifica alla formula matematica sia possibile **ridurre il numero di operazioni ridondanti** effettuate, riducendo il carico di lavoro sui singoli thread. La formula, ricordiamo, è questa:

$$\gamma_{ijs} = \frac{x'_s - x_i}{x_j - x_i}$$

Possiamo notare come due thread nella stessa posizione in blocchi diversi (quindi a parità di i e j) effettuino calcoli ridondanti che non dipendono da s. Un obbiettivo di ottimizzazione auspicabile è quello di rendere le operazioni all'interno di un blocco uniche all'interno dell'intera griglia, e quindi **dipendenti esclusivamente dall'indice s**.

Dimostrazione. Separiamo x'_s dalle costanti ridondanti a più thread:

$$\begin{aligned}
\gamma_{ijs} &= \frac{x'_s - x_i}{x_j - x_i} \\
&= \frac{x'_s}{x_j - x_i} - \frac{x_i}{x_j - x_i} \\
&= \frac{x'_s}{t_{ji}} - \frac{x_i}{t_{ji}} && (\text{con } t_{ji} = x_j - x_i) \\
&= \frac{x'_s}{t_{ji}} - z_{ji} && (\text{con } z_{ji} = \frac{x_i}{t_{ji}}) \\
&= x_s \cdot \frac{1}{t_{ji}} - z_{ji}
\end{aligned}$$

□

Per farlo è necessario effettuare delle modifiche alla formula matematica attraverso la tecnica del cambio di variabile. In questo modo non solo è possibile pre-calcolare tutte le costanti indipendenti da s in un kernel apposito (costituendo di fatto una **fase 0**), ma anche ottimizzare gli accessi alla memoria in occasione del loro utilizzo.

2.5 Architettura delle chiamate a Kernel

Dopo aver diviso la computazione parallela in fasi, è necessario analizzare quanti e quali kernel seriali il programma dovrà inizializzare durante un suo ciclo di esecuzione.

Il **primo kernel** da eseguire sarà quello che si occuperà della fase 0, durante la quale andranno calcolati i due **vettori di costanti** t e z con $(n+1)^2$ valori ciascuno. Per farlo sarà sufficiente inizializzare una griglia quadrata contenente $(n+1)^2$ thread e lasciare che ognuno di essi, in base ai propri indici x e y globali, calcoli i corrispettivi t_{ji} e z_{ji} (con $j = x$ e $i = y$).

Il **secondo kernel** a questo punto potrà procedere con tutte le fasi che comportano **calcoli intra-blocco**, quindi quelle da 1 a 3. In questo caso la griglia avrà blocchi quadrati di dimensione $maxThreadIdx.x \times maxThreadIdx.y$ e forma rettangolare (come descritto in figura 2.10).

Il **terzo kernel** infine terminerà la riduzione finale unendo i risultati parziali delle Ψ righe tramite una **sommatoria inter-blocco**.

Per garantire la massima ottimizzazione dei tempi di esecuzione nei casi d'uso in cui il dominio da interpolare risulti troppo grande o variabile per essere processato tutto in una volta, è stata data la possibilità di adottare **due modelli di esecuzione** attraverso differenti chiamate a funzione di libreria. Nel primo (figura 2.11) la computazione viene divisa tra due chiamate a funzioni: una che effettua solo la fase di pre-calcolo delle costanti e un'altra che procede con le successive, da ripetere tutte le volte necessarie per calcolare l'intero dominio senza effettuare operazioni ridondanti. Nel secondo invece tutte le fasi vengono eseguite da una sola funzione C, utile quando il calcolo di un codominio interpolato con uno specifico polinomio interpolante viene effettuato *una-tantum*.

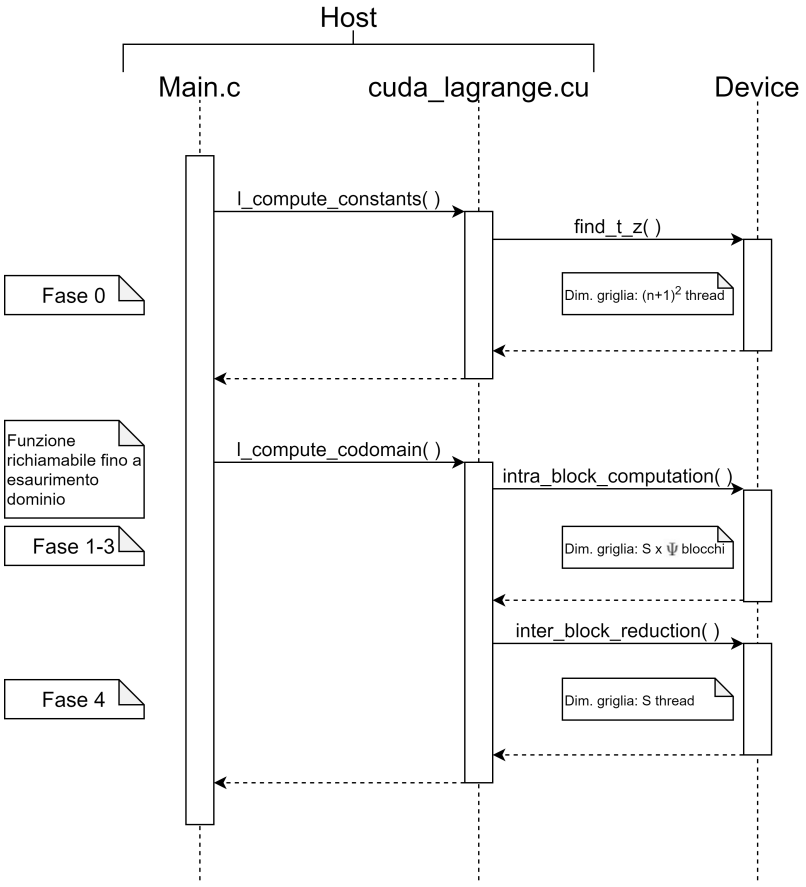

Figura 2.11: Diagramma UML di sequenza con chiamate a libreria separate.

2.6 Architettura parallela finale

Nella figura 2.12 è possibile vedere uno **schema riassuntivo dell'architettura parallela** del programma alla luce dei limiti implementativi di CUDA. Il caso d'uso rappresentato è quello usato anche per le figure 2.7, 2.8 e 2.9, con una forma della griglia (durante le fasi 1-3) corrispondente a quella della figura 2.10 e una distribuzione di fasi e chiamate a kernel corrispondente a quella descritta in figura 2.11.

In definitiva, le **fasi del parallelismo** sono 5:

0. Calcolo dei **vettori di costanti** t e z secondo la dimostrazione 2.4 usando una griglia di $(n+1)^2$ thread in cui ogni thread trova una tupla (t_{ij}, z_{ij}).

1. Calcolo dei **valori atomici** in cui ogni thread calcola (secondo l'equazione 2.2) e moltiplica tra loro i γ_{ijs} secondo la formula 2.5. All'inizio di questa fase viene inizializzata una griglia di dimensioni $S \times \Psi$ blocchi.

2. Calcolo degli $\boldsymbol{L_j(x'_s)}$ secondo la formula 2.3, mettendo insieme le produttorie parziali attraverso una riduzione per ogni colonna di thread ($S(n+1)$ riduzioni in parallelo).

3. Calcolo delle **sommatorie parziali** tramite una riduzione eseguita dalla prima riga di ogni blocco ($S\Psi$ riduzioni in parallelo).

4. Inizializzazione nuova griglia di S thread in cui ogni thread mette insieme gli Ψ risultati parziali di ogni $\Phi(x'_s)$, ottenendo le $\boldsymbol{y'_s}$ **interpolate**.

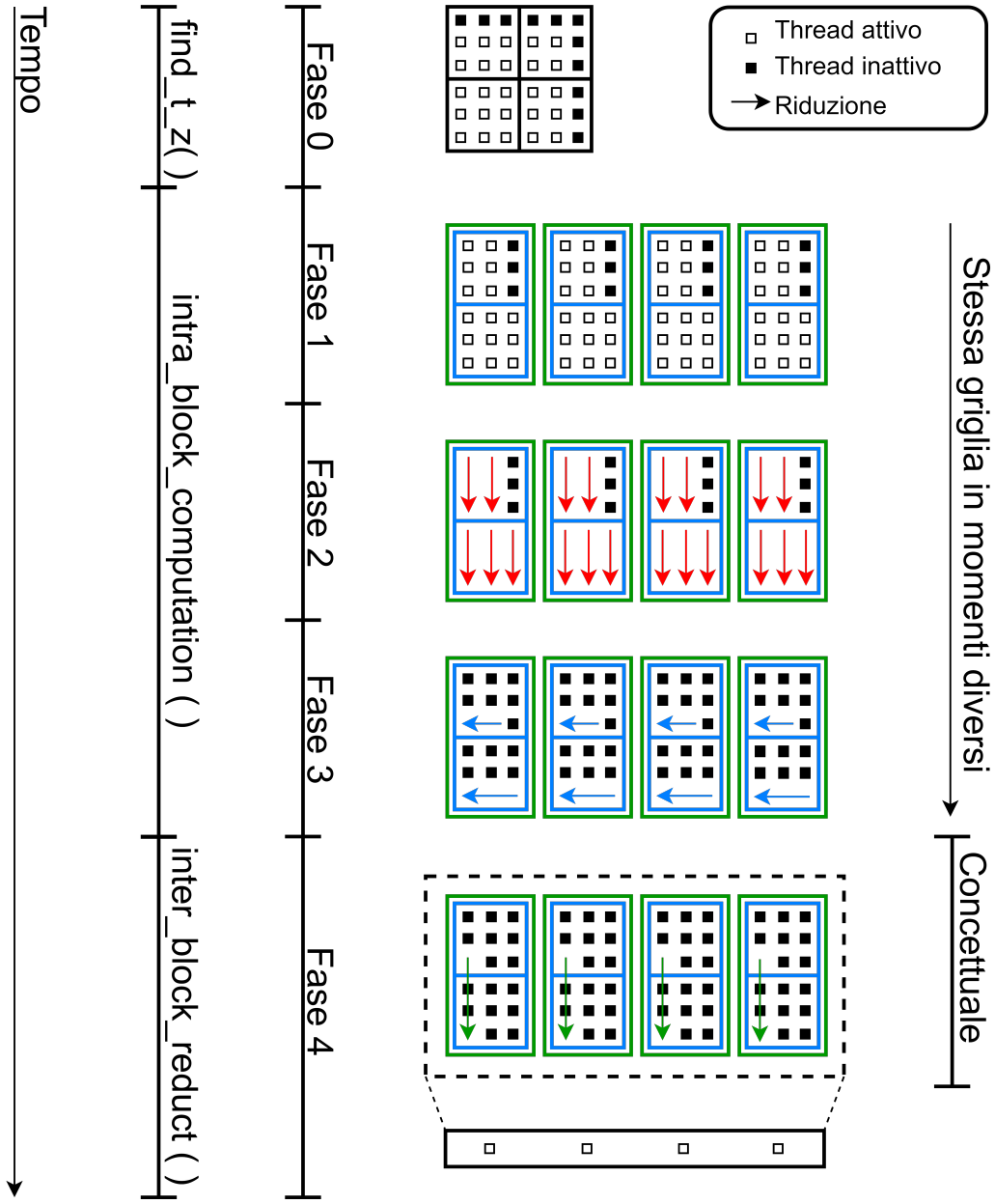

Figura 2.12: Architettura parallela finale con 5 nodi, $S = 4$ e $blockDim = 3$.

Capitolo 3

Implementazione

In questo capitolo tratteremo i dettagli implementativi della libreria in CUDA C, con una particolare attenzione alla gestione degli accessi in memoria, all'indicizzazione e all'implementazione delle riduzioni.

3.1 Buone pratiche di programmazione CUDA

[6] Per comprendere meglio la *ratio* dietro alle pratiche di programmazione esposte in questo capitolo bisogna tenere conto di due aspetti implementativi dell'architettura CUDA:

- La comunicazione tra i thread e la memoria globale (in caso di cache-miss) avviene tramite un bus limitato (128 byte, secondo la tabella 2.1) che **serializza gli accessi** nel caso vengano richieste porzioni di memoria troppo grandi o che non risiedono nella stessa *word* (figura 3.1).

- Il modello di esecuzione effettivo dei thread raggruppa *warp-size* thread alla volta, eseguendone contemporaneamente le istruzioni. In presenza di istruzioni condizionali che deviano il flusso di esecuzione come *if-else* o *switch*, nel caso solo parte dei thread dello warp percorrano una certa porzione di codice, **l'esecuzione delle ramificazioni verrà serializzata** (figura 3.2).

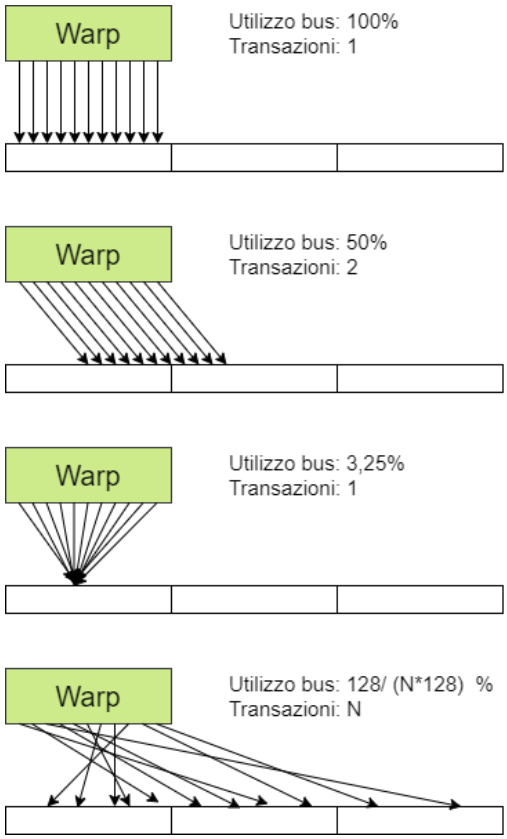

Figura 3.1: Esempio di accessi alla memoria globale in caso di cache-miss.

Nell'esempio in figura 3.1 possiamo notare come la percentuale di utilizzo e il numero di transazioni cambino in base alla quantità e alla distribuzione dei dati richiesti. Nello specifico, abbiamo posto che ogni thread dello warp stia richiedendo un intero di 4 byte, modificando di volta in volta le caratteristiche della richiesta:

- Caso A: indirizzi degli accessi allineati agli indici dei thread nello warp, caso ottimo.
- Caso B: indirizzi allineati ma dati salvati su due *word* differenti.
- Caso C: tutti i thread richiedono lo stesso intero contemporaneamente. Se utilizzato spesso per processare numerosi elementi di un array è

consigliabile copiare l'array in memoria shared, evitando di occupare il bus inutilmente.

- Caso D: accesso in cui gli indirizzi degli elementi processati e gli indici dei thread nello warp non sono correlati, caso pessimo.

Ne consegue che il modo giusto di interfacciarsi con array in memoria globale sia quello di **allineare gli indici** dei thread e gli indici degli array.

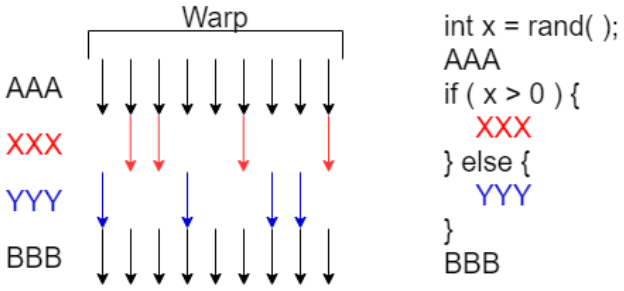

Figura 3.2: Esempio di esecuzione seriale delle ramificazioni di un if.

L'esecuzione seriale delle ramificazioni avviene solo se a divergere sono thread appartenenti a uno stesso warp. Per questo è considerata una buona pratica di programmazione quella di **impostare le condizioni delle deviazioni di flusso esclusivamente su indici di entità logiche pari o superiori allo warp** (come ad esempio gli indici del blocco), in modo da far prendere ai suoi thread la stessa "strada" e dimezzare i tempi di esecuzione di quella specifica porzione di codice. Nella figura 2.3 (che rappresenta l'implementazione della riduzione-produttoria) è possibile osservare come questa condizione venga rispettata: i thread basano la propria operazione all'interno della riduzione sul proprio indice di riga i (spiegato nei capitoli successivi), che con la forma quadrata del blocco (32×32) corrisponde anche all'indice dello warp a cui appartiene: in questo modo tutti i thread dello warp seguono lo stesso flusso di istruzioni, sia nel caso in cui debbano effettuare operazioni che quello in cui si debbano disattivare per il resto dell'esecuzione.

3.2 Strutture dati dell'host

Fatte queste premesse è doveroso presentare una lista dettagliata delle strutture dati utilizzate dalla libreria, in modo da comprendere meglio come nel resto dell'implementazione si sia cercato di seguire le buone pratiche descritte nel capitolo precedente. Essendo una libreria matematica che rende prioritaria l'accuratezza dei calcoli rispetto all'occupazione in memoria, tutti i tipi di dati numerici che riguardano elementi su cui effettuare operazioni matematiche sono *double* (numeri a virgola mobile a doppia precisione).

Le strutture che risiedono nella memoria RAM dell'host verranno lette e scritte una sola volta rispettivamente nella fase di preparazione e nella fase finale del programma attraverso la funzione `cudaMemCpy()`, che copia intere porzioni di memoria dell'host a quella del device. La copia ha come destinazione uno spazio di memoria globale precedentemente allocato a mano dalla funzione host `cudaMalloc()`. Queste strutture includono:

- Array x e y di coordinate (x, y) dei nodi.
- Array x'_s di ascisse da interpolare.
- Spazio riservato per l'array di ordinate interpolate y'_s, in cui verrà copiato il risultato finale dal device al termine della fase 4.

Gli array vengono passati alle funzioni di libreria per riferimento, evitando di sprecare memoria inutilmente per dati che la CPU (quando esegue le funzioni della libreria oggetto di questa tesi) legge e scrive solo nei trasferimenti verso la memoria del device.

In figura 3.3 è possibile vedere un digramma UML di sequenza che descrive i passaggi di dati da host a device durante il ciclo di esecuzione già analizzato in figura 2.11

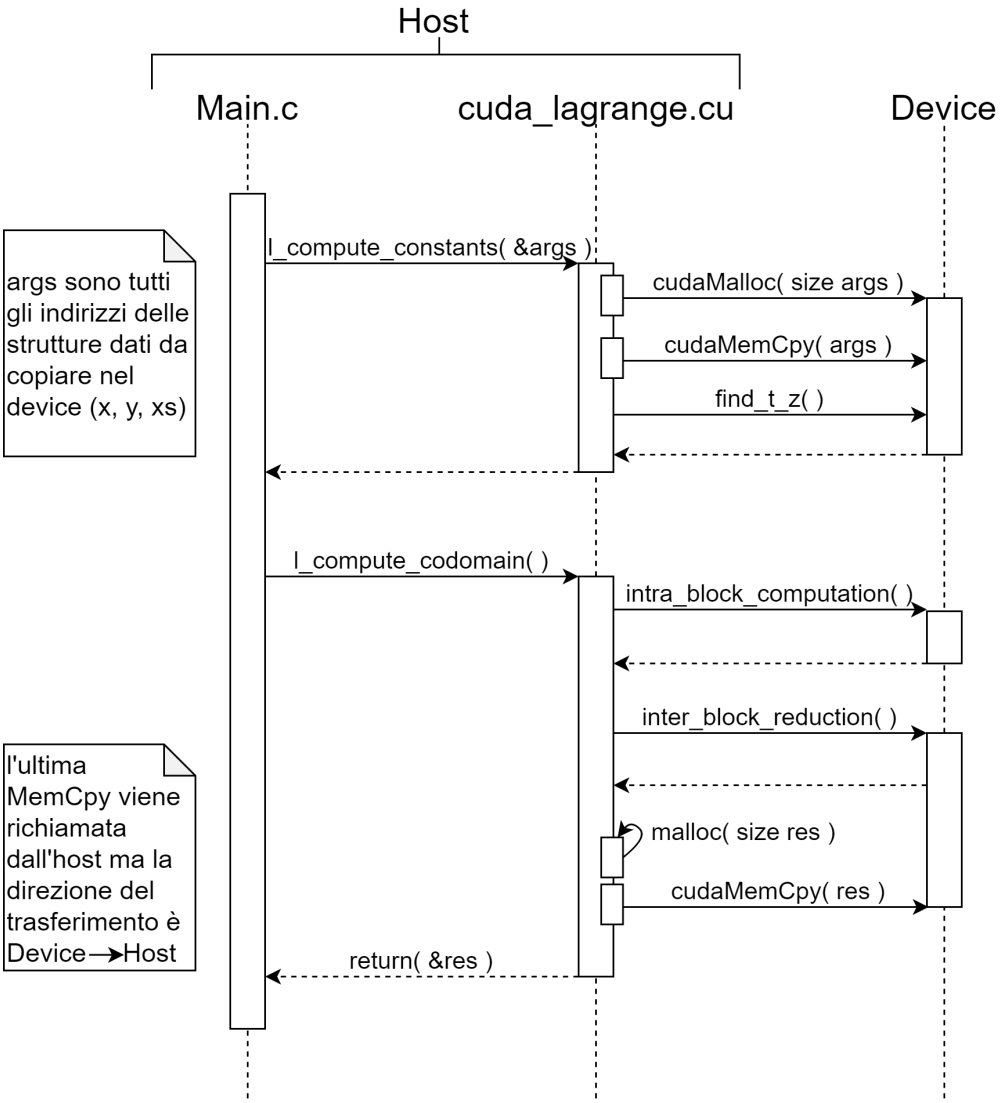

Figura 3.3: Diagramma UML di sequenza dell'I/O eseguito dall'host.

3.3 Strutture dati del device

La gestione dei dati in presenza di parallelismo inter-blocco e inter-thread richiede sia molteplici copie ridondanti, che un'indicizzazione estremamente precisa che assicuri i giusti riferimenti alle memorie condivise.

3.3.1 Utilizzo delle strutture dati durante un ciclo di esecuzione

Nella tabella 3.1 è possibile vedere quali strutture dati vengono lette e scritte da un thread (non necessariamente *lo stesso* thread) durante le varie fasi; questo ci fornisce una visione d'insieme da cui partire per analizzare la ripartizione in memoria e la quantità di copie ridondanti dei vari array.

	Lettura	**Scrittura**	**Kernel**
Fase 0 **Costanti**	array di x nodi	array di costanti t array di costanti z	find_t_z()
Fase 1 **Elem. atomici**	array di costanti t array di costanti z array di x'_s	array L intra-blocco	intra_block computation()
Fase 2 **Produttorie**	array L intra-blocco array di y nodi	array L intra-blocco	
Fase 3 **Somm. parziale**	array L intra-blocco	array L intra-blocco array R inter-blocco	
Fase 4 **Somm. totale**	array R inter-blocco	array di y'_s	inter_block reduction()

Tabella 3.1: Letture e scritture in memoria da parte dei thread durante un ciclo di esecuzione.

3.3.2 Ripartizione in memoria

Dalle caratteristiche delle strutture dati esposte nella tabella è possibile dedurre che:

- L'array di y'_s e gli array di x e y dei nodi possono essere salvati in **memoria globale** per via del loro scarso utilizzo durante la computazione.

- Gli array t, z e R devono necessariamente essere salvati nella **memoria globale** per via del fatto che, essendo utilizzati da più kernel, altrimenti il loro contenuto non verrebbe preservato.

- L'array L è la struttura dati in cui vengono materialmente salvati i valori γ_{ijs}. Il suo *scope* è il blocco, e essendo usato per le riduzioni delle fasi 2 e 3 viene letto e scritto un numero di volte considerevole. Questo comporta che ogni blocco abbia bisogno del "suo" array L, rendendo la **memoria shared** l'unica opzione di posizionamento sensata. In alternativa si sarebbe dovuto creare un array unico multidimensionale in memoria globale che, a causa delle numerose richieste concorrenti da blocchi diversi, avrebbe intasato il bus e dilatato sensibilmente i tempi di esecuzione.

- L'array di x' è inizialmente salvato nella **memoria globale** per via delle sue dimensioni. Sappiamo tuttavia che nella fase 1, per calcolare i γ_{ijs}, tutti i thread dell's-esimo blocco dovranno accedere all's-esimo elemento di quell'array, potenzialmente intasando il bus con un numero di richieste pari al numero di elementi γ da calcolare. Per evitare questo fenomeno a inizio computazione ogni thread con tupla $(j, i) = (0, 0)$ degli $S \cdot \Psi$ blocchi copia in **memoria shared** l's-esimo elemento dell'array, l'unico effettivamente usato dai thread di quel blocco.

3.3.3 Indicizzazione

Come abbiamo visto nella parte introduttiva di questo capitolo, nell'ambito di un I/O efficiente in CUDA l'indicizzazione dei riferimenti alle strutture dati è importante tanto quanto la scelta delle memorie in cui memorizzarle. Nella lista seguente verranno illustrati **gli indici usati dai thread** per consultare le varie strutture dati nel corso di un ciclo di esecuzione.

- $\boldsymbol{j} = threadIdx.x + blockIdx.y \cdot blockDim.x$.

  Valore sull'asse orizzontale "globale" (livello di isolamento: Ψ blocchi).

- $\boldsymbol{i} = threadIdx.y$, poi varia secondo formula 2.5.

  Indice verticale di finestra scorrevole dei thread sull'asse y dei γ nella fase 1.

- $\boldsymbol{tid} = threadIdx.x + (threadIdx.y \cdot blockDim.x)$ $(= -1$ se thread inattivo).

  Identificativo del thread all'interno del blocco (livello di isolamento: blocco).

- $\boldsymbol{global\_tid} = j + (threadIdx.y \cdot (n+1))$ $(= -1$ se thread inattivo).

  Identificativo globale del thread tra i thread attivi negli Ψ blocchi a inizio computazione. I thread inattivi, come ad esempio quelli barrati nella figura 2.8 o quelli neri in figura 2.12 non vengono contati e questo permette di indicizzare array utilizzando indici derivati dalla posizione dei thread senza problemi, anche con n non multiplo di $blockDim.x$. Nella figura 3.4 si può trovare un esempio di questo tipo di indicizzazione: come visto in figura 2.10 gli Ψ blocchi (in quel caso 2) sono posizionati concettualmente sull'asse x, ma fisicamente sull'asse y. Questo si ripercuote anche sull'indicizzazione che tiene conto di j, risultando in "righe" lunghe $(n+1)$ thread concatenate tra i blocchi nel modo descritto in figura 3.4. Alcuni thread della griglia verranno inevitabilmente tenuti inattivi perché non sempre n sarà multiplo di

blockDim.x, e l'ultimo degli Ψ blocchi avrà alcune colonne di thread con un indice $j > n$ (come ad esempio in figura 2.8).

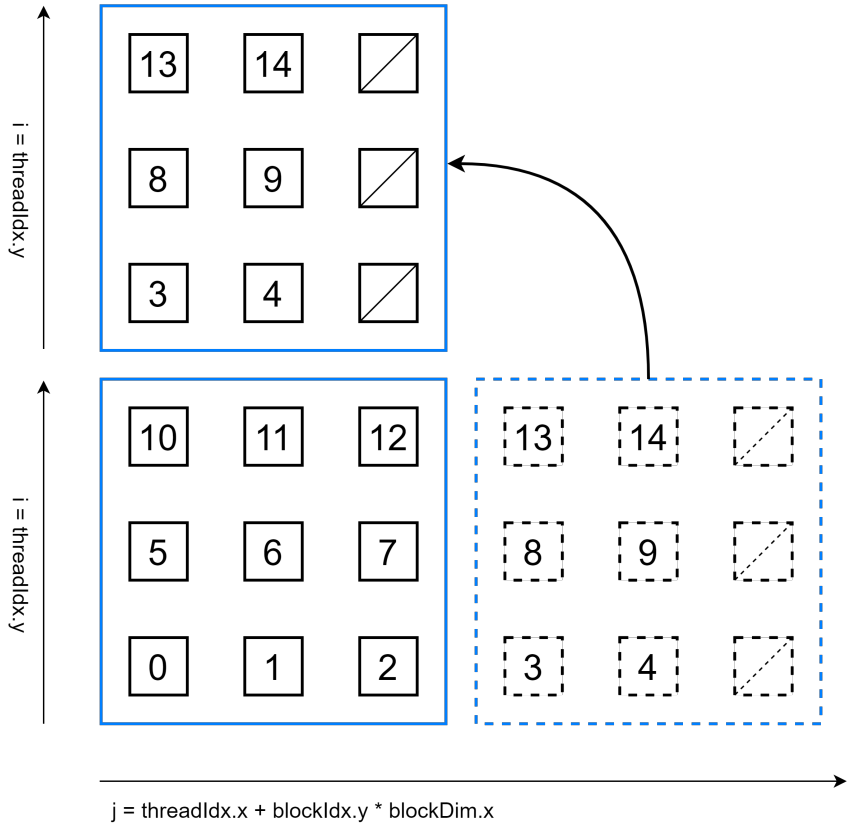

Figura 3.4: Criterio di indicizzazione di *global_tid*.

- ***index_2d*** $= j + (i \cdot (n + 1))$, aggiornato a ogni cambiamento di i.

  In CUDA l'implementazione di array dinamici multidimensionali nella forma `array[j][i]` risulta complessa da realizzare e scarsamente ottimizzata. Per questo motivo un array 2d viene "schiacciato" in una sola dimensione, e referenziato con un indice monodimensionale che è la traduzione della tupla (j, i). Questo indice in particolare viene usato per implementare la finestra scorrevole della fase 1. In figura 3.5 è possibile vedere come cambiano gli indici dei thread tra un'iterazione e l'altra della fase 1, descritta in figura 2.7.

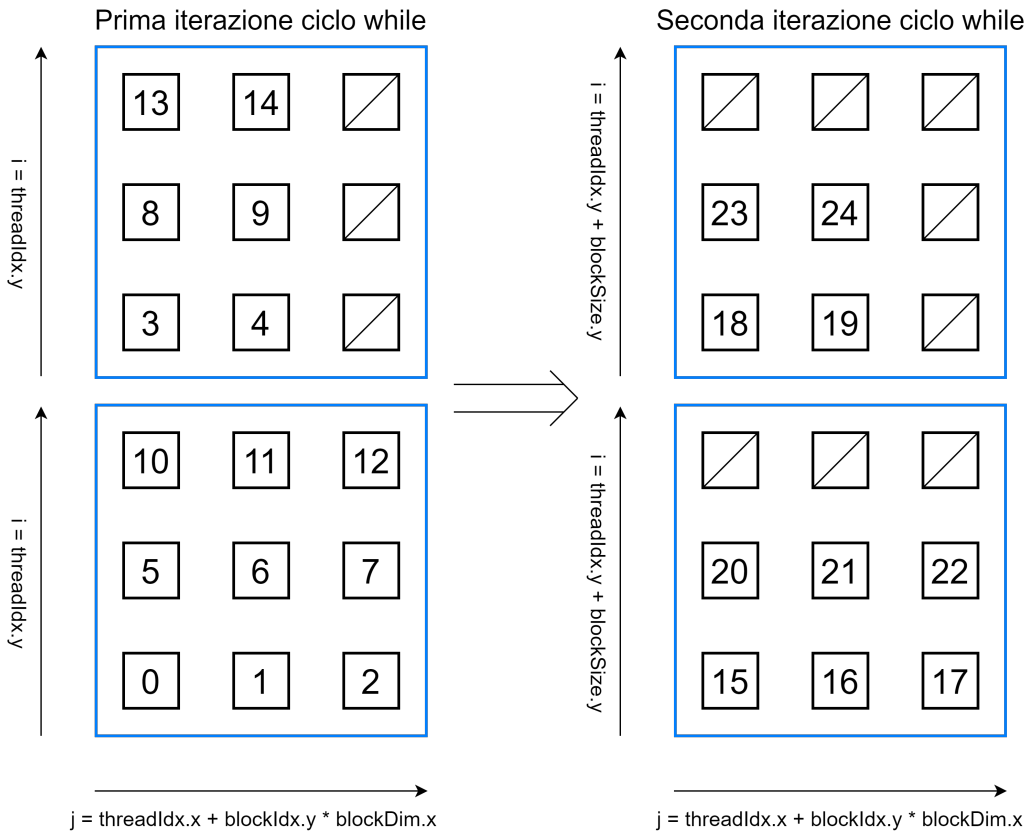

Figura 3.5: Criterio di indicizzazione di $index\_2d$ con $blockDim = 3$ e $n = 4$.

- ***block_global_tid*** $= blockIdx.x + blockIdx.y \cdot gridDim.x$.

  Indice globale del blocco nella griglia, viene usato al termine della fase 3 per salvare i risultati parziali del blocco nell'array R di supporto alla riduzione inter-blocco della fase 4.

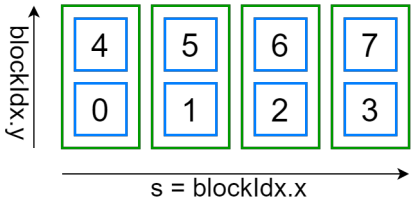

Figura 3.6: Criterio di indicizzazione dei blocchi con $block\_global\_tid$.

- $\boldsymbol{s} = blockIdx.x$.

  Indice che stabilisce quale tra gli S punti da interpolare sarà assegnato a quel blocco, basato sulla posizione orizzontale del blocco nella griglia.

- $\boldsymbol{blockIdx.y}$.

  A volte la posizione del blocco sull'asse delle y viene usata per determinare l'id del blocco all'interno degli Ψ blocchi, come da figura 2.10.

3.4 Accessi alla memoria globale allineati agli warp

A questo punto abbiamo tutte le nozioni necessarie per capire come le politiche di indicizzazione si combinino con la lista di strutture dati che risiedono in memoria globale per rispettare la buona pratica descritta nel capitolo 3.1. Le strutture dati critiche da ottimizzare sono tre:

- I due array di costanti t e z, scritti $(n+1)^2$ volte in fase 0 e letti $S(n+1)^2$ volte in fase 1.

- L'array di riduzione inter-blocco R, scritto e letto $S \cdot \Psi$ volte rispettivamente in fase 3 e in fase 4.

3.4.1 Accessi allineati: array di costanti

La figura 3.7 mostra la forma della griglia durante la fase 0 (in cui l'indice di ogni thread corrisponde alla sua posizione assoluta in tutta la griglia) riprendendo l'esempio in cui $n = 4$ e $blockDim = 3$. La figura è una semplificazione della vera griglia CUDA, che in blocchi quadrati ammette lati lunghi fino a 32 thread (dato che $32^2 = 1024$). Ne consegue che **inizializzando sempre blocchi di 32×32 thread la dimensione del lato corrisponda all'ampiezza degli warp**.

Gli indici usati per accedere agli array dai thread sono sequenziali all'interno di una riga, e quindi all'interno dello warp: questo rende gli accessi alla memoria globale allineati, in quanto **tutti i thread dello warp accedono a elementi vicini in memoria** (Casi A e B della figura 3.1).

In caso di $n < 32$ o di blocchi "dispari" (come quello in alto a destra in figura 3.7), e quindi di righe di thread attivi più piccole dello warp, questo allineamento sarà preservato. Questo perché thread vicini accederanno ancora a elementi vicini, con l'unica differenza che consiste in una più bassa percentuale di utilizzo del bus: ci troviamo ancora nei casi A e B della figura 3.1, ma con meno richieste rispetto al numero totale di thread in uno warp.

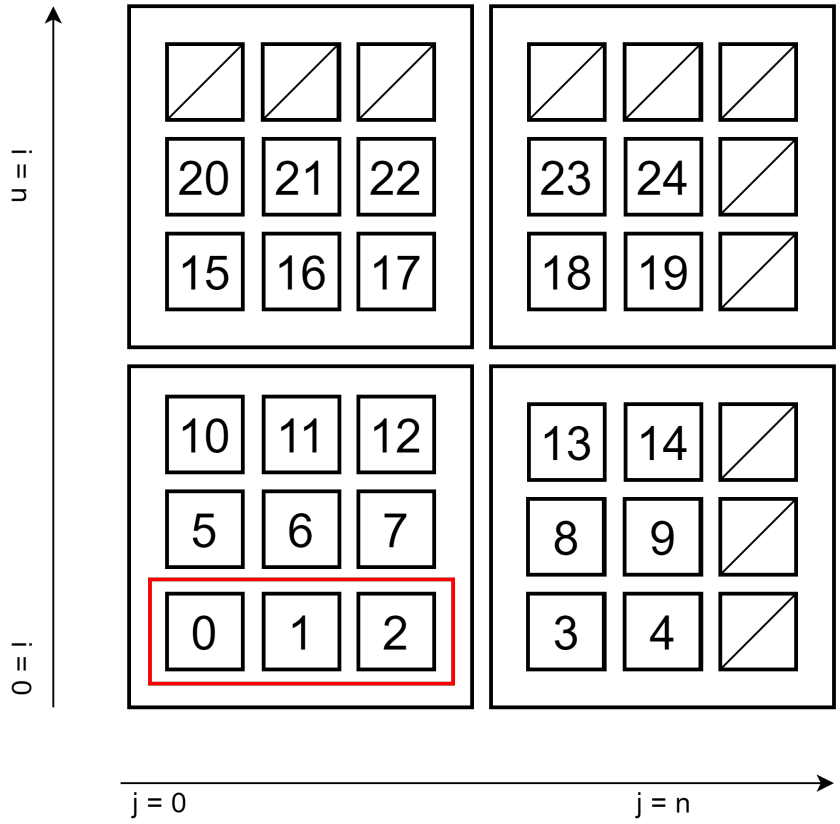

Figura 3.7: Gli indici dei thread basati sulla posizione assoluta sono sequenziali all'interno della riga e dello warp (in rosso).

L'indice utilizzato per l'accesso a t e a z nella fase 1 è **index_2d**, che come possiamo vedere dalla figura 3.8 riesce a preservare lo stesso allineamento tra accessi alla memoria e warp della fase 0: anche in questo caso thread vicini consultano elementi dell'array vicini. **Questo allineamento viene mantenuto per ogni iterazione del ciclo while** (figura 3.5).

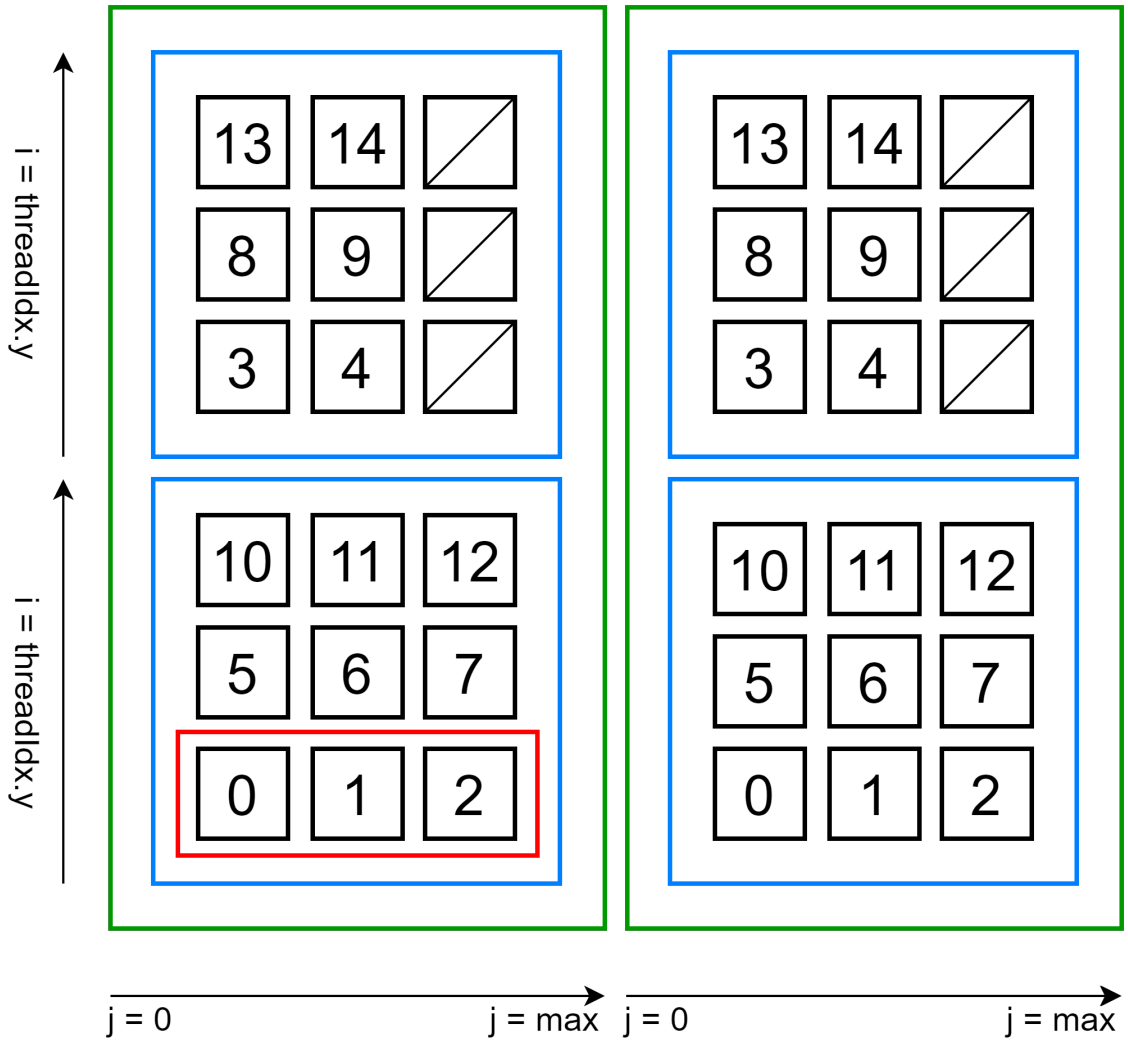

Figura 3.8: Gli indici dei thread basati sulla finestra scorrevole sono sequenziali all'interno della riga e dello warp (in rosso).

3.4.2 Accessi allineati: array R di riduzione inter-blocco

L'implementazione della fase 4 prevede una griglia di $(S \times 1)$ thread in cui **ogni thread somma tra loro i risultati delle sommatorie parziali** utilizzando un *ciclo for* simile al seguente pseudocodice:

```
1 s ← threadIdx.x + blockIdx.x · blockDim.x;
2 parziale ← 0;
3 for k ← 0 to Ψ do
4 |   parziale ← parziale + R[s + (k · S)];
5 end for
6 y'[s] ← parziale;
```

Dato che $threadIdx.x$ riparte da 0 a ogni blocco, è necessario far derivare l'indice s del thread dalla sua posizione assoluta all'interno della griglia (e non solo da quella nel blocco), permettendoci di lavorare anche con $S >= 1024$.

La figura 3.9 fornisce una rappresentazione grafica di questo ciclo mostrando come l'indicizzazione dei blocchi nella fase 3 permetta di avere accessi allineati da parte degli warp nella fase 4. La griglia nella parte superiore è la stessa della figura 3.6 e rappresenta il momento in cui ogni thread con coordinate $(j, i) = (0, 0)$ salva in memoria globale il risultato della computazione intra-blocco. Le frecce nella parte inferiore rappresentano le letture dell'array durante la fase 4 e sono colorate in base all'iterazione in cui avvengono (nero per la prima e rosso per la seconda).

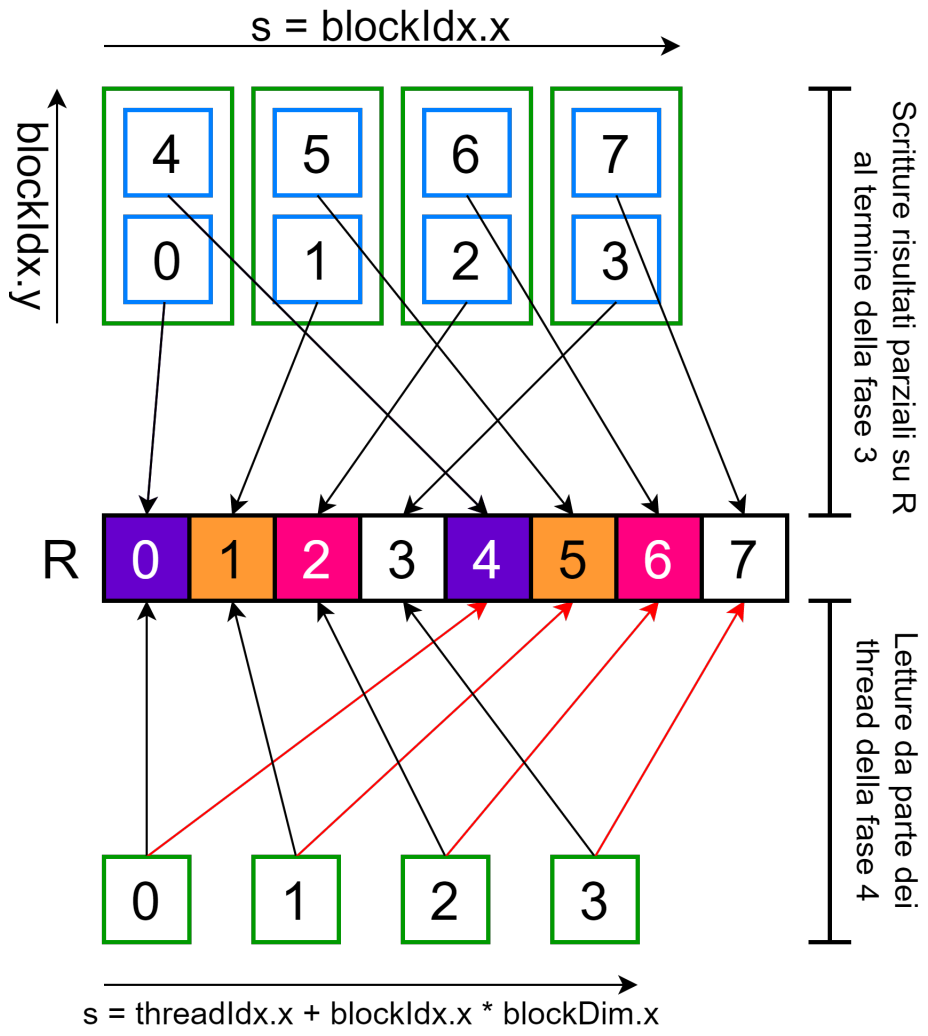

Figura 3.9: Scritture e letture da R durante la fasi 3 e 4 con $\Psi = 2$ e $S = 4$.

Capitolo 4

Conclusione

4.1 Test di correttezza

Il primo aspetto da valutare in una libreria matematica è la **correttezza dei risultati**. Il grafico superiore della figura 4.1 mostra la sovrapposizione tra due polinomi interpolanti costruiti sugli stessi nodi: il primo è stato calcolato dalla libreria CUDA oggetto di questa tesi, mentre il secondo è stato ottenuto tramite un'implementazione MATLAB dell'algoritmo di Lagrange [1]. Il grafico inferiore invece mostra la sovrapposizione tra l'errore assoluto

$$|\Phi(x') - f(x')|$$

(y interpolata - y originale) del primo e del secondo polinomio.

La figura 4.2 mostra un grafico che quantifica in termini esatti la differenza tra i due errori:

$$\Delta = ||(\Phi_{MATLAB}(x') - f(x'))| - |(\Phi_{CUDA}(x') - f(x'))|| \tag{4.1}$$

[1] Carlo Castoldi (2021). (https://www.mathworks.com/matlabcentral/fileexchange/899-lagrange-polynomial-interpolation), MATLAB Central File Exchange.

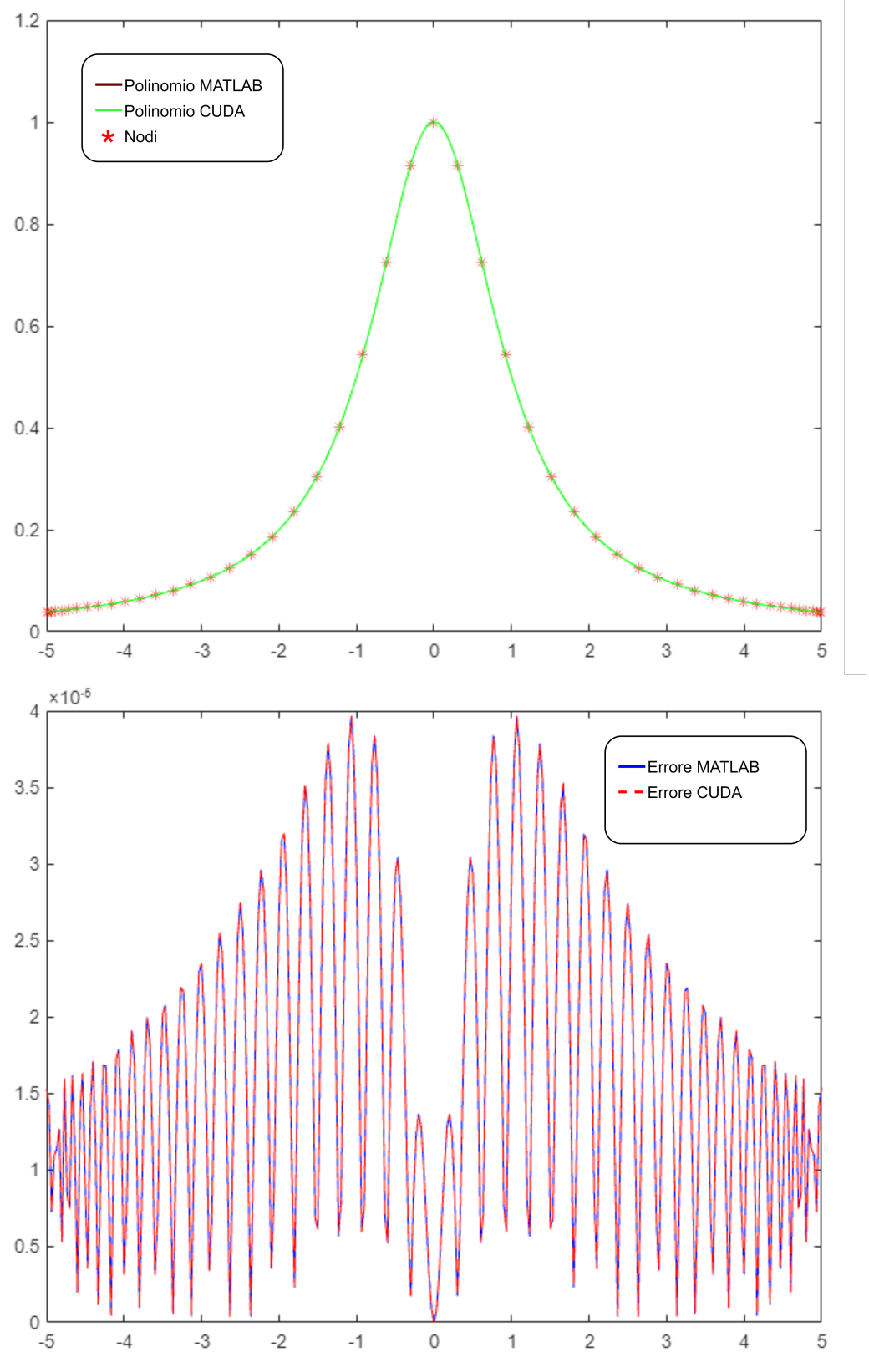

Figura 4.1: Confronto tra polinomi interpolati ($n = 50$) costruiti con MATLAB e con CUDA.

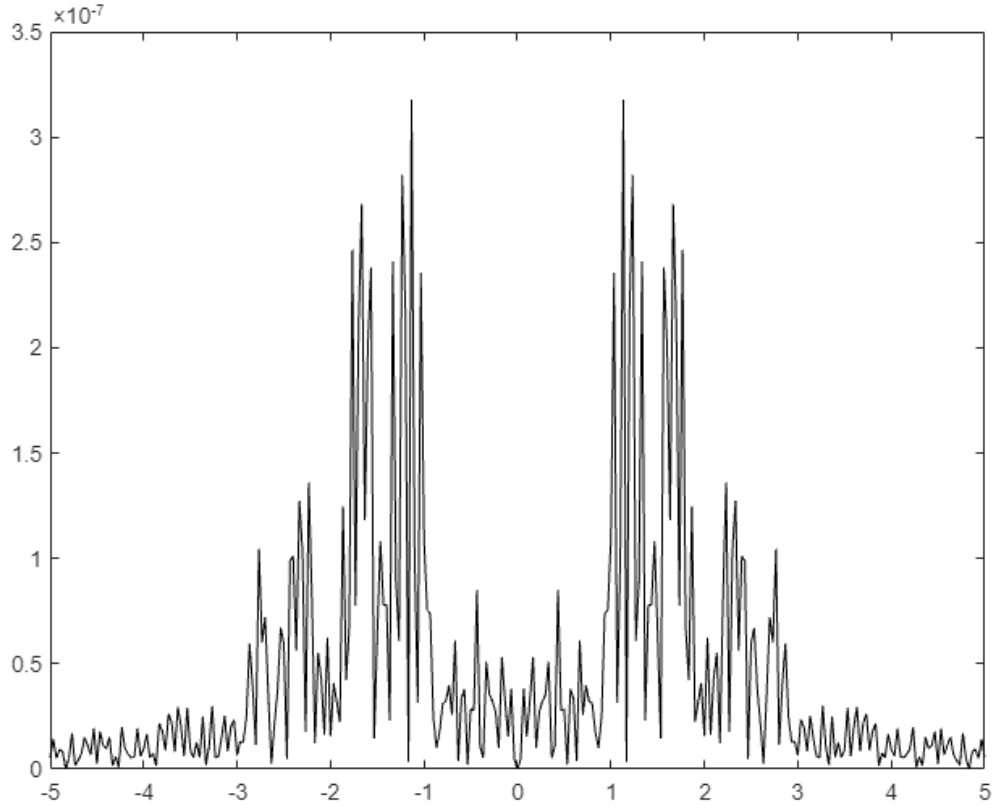

Figura 4.2: Grafico del valore assoluto della differenza tra i due errori (equazione 4.1).

Questa differenza è impercettibile a occhio nudo e in linea con le aspettative: essendo Δ inferiore di diversi ordini di grandezza rispetto alla differenza tra una delle due interpolazioni e la funzione di Runge originale (10^{-7} contro 10^{-5}) possiamo dedurre che **l'aver parallelizzato il calcolo del polinomio interpolante non ha incrementato l'instabilità numerica**.

Le figure 4.3 e 4.4 mostrano che lo stesso discorso può essere fatto anche per funzioni "difficili" da interpolare [**5**] come $y = sign(x)$. In questo caso la differenza tra i polinomi interpolanti è nell'ordine di 10^{-6}, mentre quella tra i polinomi e $f(x)$ raggiunge numeri nell'ordine di 10^{0}.

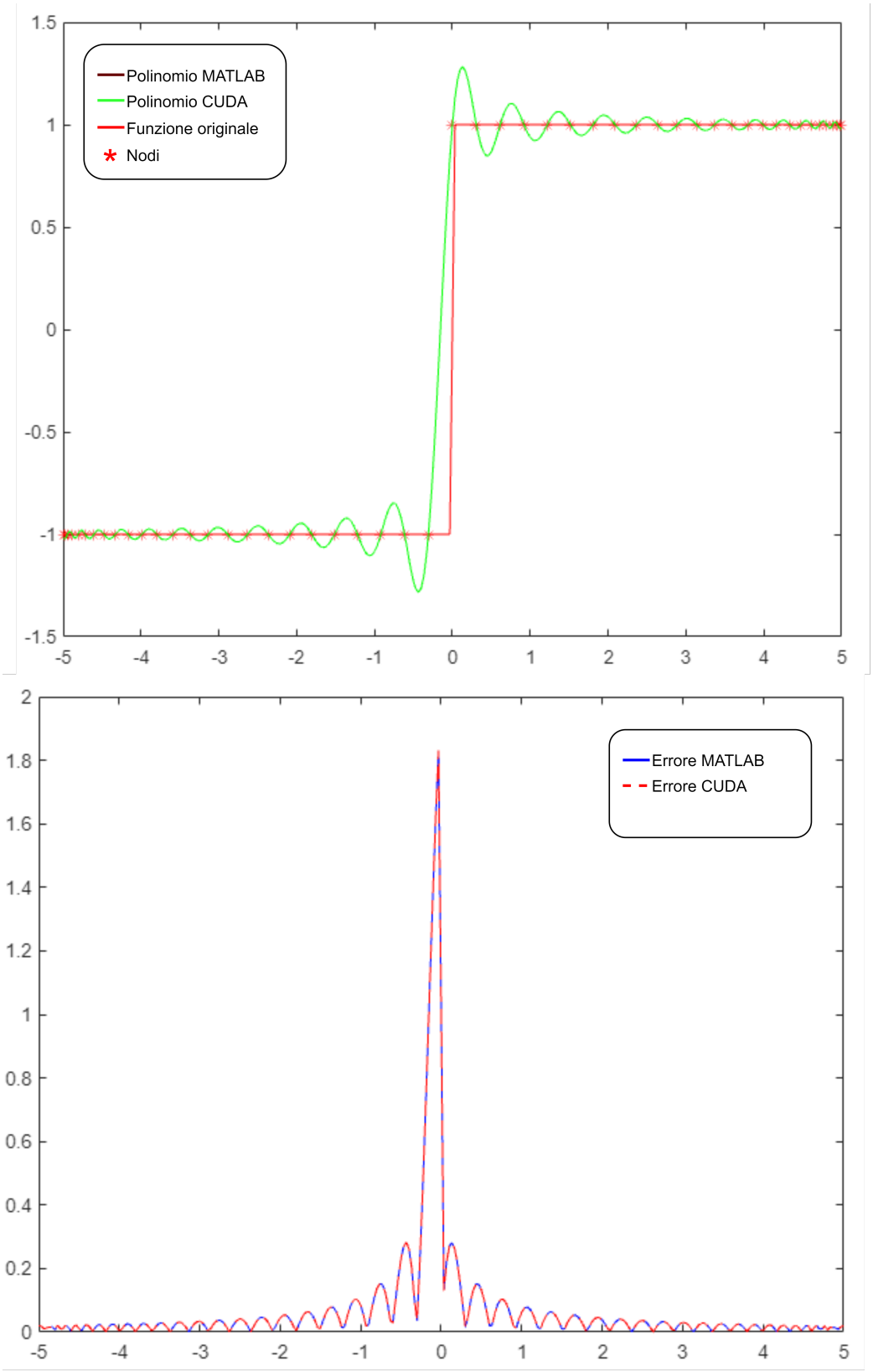

Figura 4.3: Sovrapposizione di polinomi interpolanti della funzione $sign(x)$.

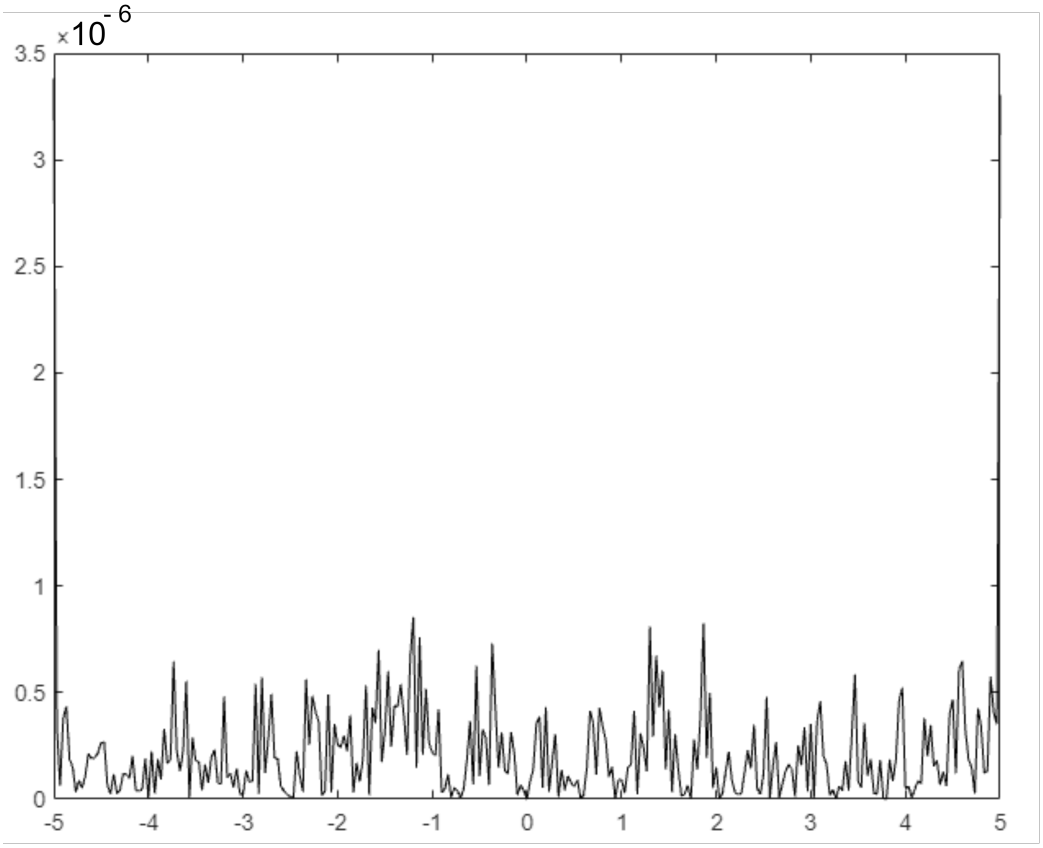

Figura 4.4: Grafico del valore assoluto della differenza tra i due errori (equazione 4.1).

4.2 Test di scalabilità

Un altro importante aspetto da tenere in conto in un programma CUDA è l'efficacia della parallelizzazione. Questa può essere constatata attraverso la misurazione dei tempi di esecuzione all'aumentare dell'input. Nel caso dell'interpolazione di Lagrange abbiamo un input dato dal prodotto $S(n+1)$ dove n è il grado del polinomio interpolante e S è la quantità di punti da interpolare.

La figura 4.5 mostra la variazione dei tempi di un ciclo di esecuzione completo del programma mantenendo n costante e aumentando esponenzialmente

S. Possiamo notare come i tempi rimangano costanti fino a $S = 10^4$, per poi cominciare ad aumentare, probabilmente a causa delle grosse quantità di memoria da allocare e spostare tra Host e Device.

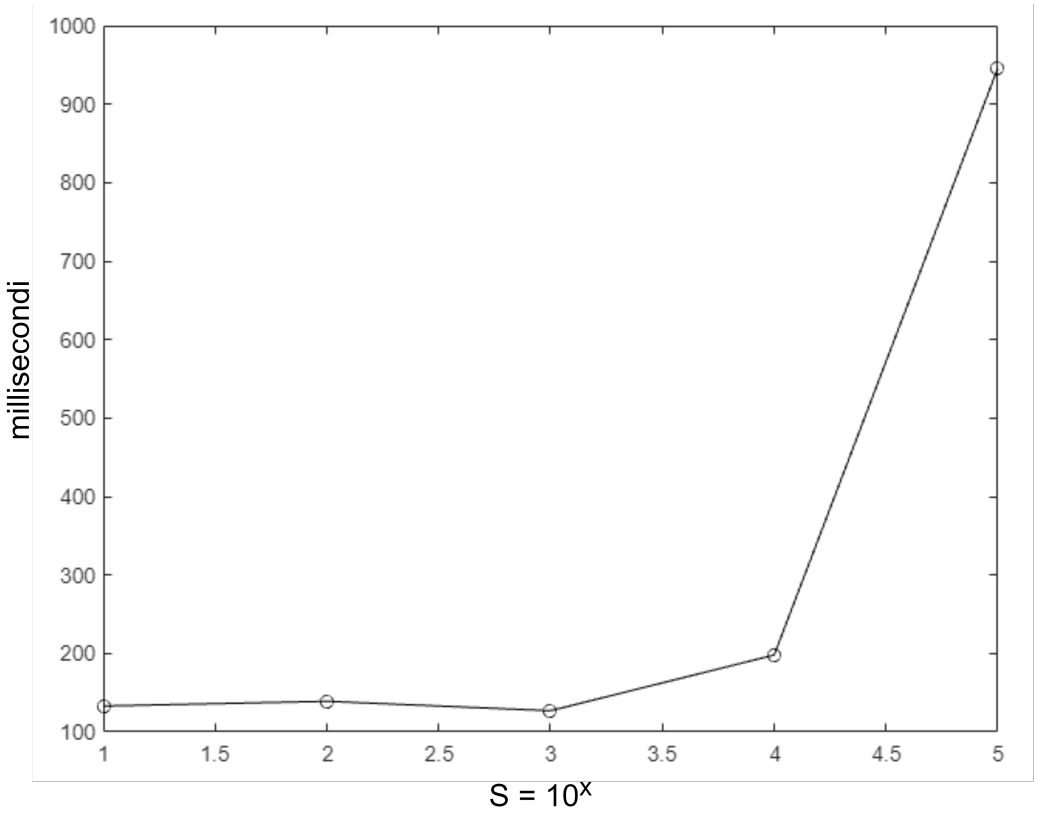

Figura 4.5: Tempi di esecuzione in funzione di S.

La figura 4.6 mostra invece la variazione di tempi di esecuzione all'aumentare del grado n del polinomio da interpolare (mantenendo S costante). I valori di n nel grafico sono scelti in modo da valutare l'impatto della riduzione inter-blocco, essendo il numero di blocchi per punto da interpolare $\Psi = \frac{n+1}{32}$. L'asse x quindi rappresenta contemporaneamente sia Ψ che n (secondo la formula in figura).

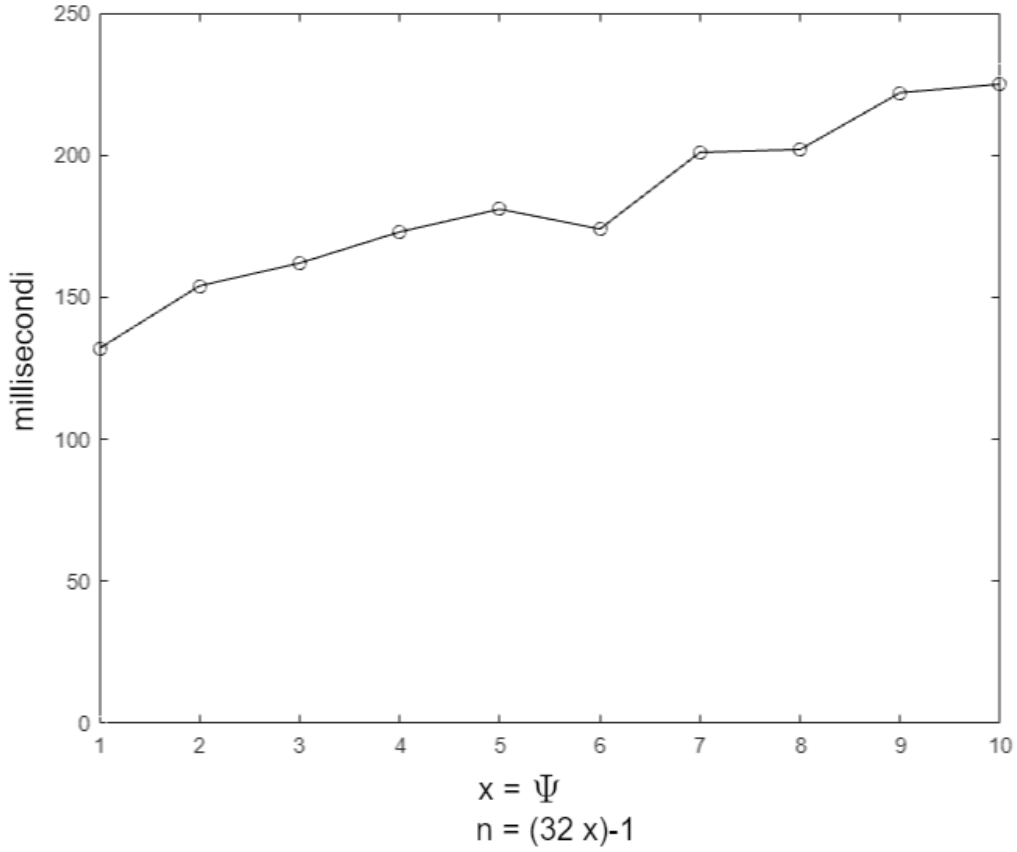

Figura 4.6: Tempi di esecuzione in funzione di n (o Ψ).

Nell'osservare questo grafico dobbiamo tenere presente che in un'implementazione seriale dovremmo aspettarci tempi di esecuzione quadratici rispetto a n: la complessità computazionale dell'algoritmo interpolatorio delle basi di Lagrange è infatti $O(n^2)$. Assistiamo invece a un incremento lineare dei tempi di esecuzione, tanto che da $n = 31$ a $n = 319$ (difficilmente i casi d'uso prevederanno n di ordini di grandezza superiori) i millisecondi passano da 132 a 225.

Ne risulta che la libreria CUDA oggetto di questa tesi si **molto scalabile per n e leggermente meno scalabile per S**: questo limite può essere risolto implementando meccanismi di I/O più efficienti rispetto alla copia in blocco dei dati da CPU a GPU che dividano S in sottogruppi più gestibili.

4.3 Considerazioni finali

L'essere riusciti a implementare l'algoritmo delle Basi di Lagrange in una libreria che aggira il problema rappresentato dal suo costo computazionale dimostra che, con un'attenta analisi dei pattern di programmazione parallela e dei limiti dell'hardware, **è possibile lavorare con algoritmi costosi mantenendo tempi di esecuzione bassi**.

Questo apre a numerose possibilità di applicazione nel campo della data-science e della robotica fornendo uno strumento in più nelle mani di chi, attraverso queste discipline, affronterà i problemi di questo secolo.

Bibliografia

[1] Blaise Barney, "Introduction to Parallel Computing", Lawrence Livermore National Laboratory.

[2] Michael McCool, Arch D. Robinson, James Reinders, Structured Parallel Programming, Morgan Kaufmann, 2012.

[3] NVIDIA Corporation, "CUDA Programming Guide", NVIDIA Corporation, 2021.

[4] Alfio Quateroni,Riccardo Sacco,Fausto Saleri, Paola Gervasio, "Matematica Numerica", Springler, 2014.

[5] Carlo Sau, Giuseppe Rodriguez, "Interpolazione e approssimazione di funzioni", Università degli studi di Cagliari (dipartimento di Ingegneria elettrica ed elettronica), 2011.

[6] NVIDIA Corporation, "CUDA C++ Best Practices Guide", NVIDIA Corporation, 2021.