Esame di Programmazione su Architetture Parallele

Metodo del simplesso per la risoluzione della programmazione lineare

Belliato Riccardo (mat. 142652) Simone Tomada

2022-08-25

Abstract

In questa relazione si propone una implementazione del metodo del simplesso a due fasi in CUDA per la risoluzione dei problemi di programmazione lineare in forma canonica.

Dopo una breve descrizione dell'algoritmo, seguirà la discussione su alcune scelte implementative.

Infine verranno valutate performance e scalabilità della soluzione proposta confrontando i tempi di esecuzione dell'algoritmo su istanze a dimensione crescente generate casualmente.

Contents

Introduzione al metodo del simplesso **1**
- Problemi di ottimizzazione lineare . . . 1
- Forma canonica e forma standard . . . 2
- Metodo del simplesso a due fasi . . . 3

Scelte implementative e algoritmi utilizzati **5**
- Schema di parallelizzazione . . . 5
- Gestione della memoria . . . 5
- Ricerca del pivot e test di ottimalità . . . 7
- Eliminazione di Gauss . . . 7
- Aggiornamento della tabella . . . 9

Risultati sperimentali **9**

Introduzione al metodo del simplesso

Problemi di ottimizzazione lineare

Nell'ambito della Ricerca Operativa (Operations Research) uno dei principali argomenti è la cosiddetta **ottimizzazione lineare**, ossia lo studio di una classe di problemi del tipo:

$$\begin{gathered}\min / \max c^T x \\ \text{subject to} \\ Ax \lesseqgtr b\end{gathered}$$

con $x \in \mathbb{R}^n$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^{1 \times m}$.

In altre parole, si vogliono trovare dei valori per le componenti del vettore x (di dimensione n) tali da massimizzare (o minimizzare) il valore di una *funzione lineare* variabili (detta **funzione obiettivo**, mentre il vettore c è chiamato **vettore dei costi**), dati una serie di m vincoli espressi nel sistema di disequazioni lineari $Ax \lesseqgtr b$ (dove A è detta **matrice dei vincoli** e b **vettore dei termini noti**).

Questa classe di problemi permette di modellare un gran numero di situazioni reali in molteplici ambiti (ottimizzazione dei costi, creazione di orari, *etc.*), oltre che alcuni problemi NP-hard come il *Knapsack* o il *Vertex Cover*.

Dal punto di vista dell'algebra lineare, un problema in n varibili non è altro che uno spazio $\mathbb{R}^n$, la funzione obiettivo è una retta nello spazio, mentre i vincoli definiscono un *poliedro* nello spazio.

Utilizzando i teoremi e le tecniche dell'algebra lineare è stato possibile creare degli algoritmi per risolvere i problemi di ottimizzazione, come il **simplesso**, i quali si prestano molto bene ad essere parallelizzati (in quanto operano su matrici).

Problemi risolvibili, non risolvibili, illimitati

Dato un problema di ottimizzazione, questo può essere di tre tipi:

- ammissibile (*feasible*): ossia esistono uno o più vettori che moltiplicati per il vettore dei costi assegnano alla funzione obiettivo il valore massimo (o minimo possibile) e tutti i vincoli sono rispettati,
- non ammissibile (*infeasible*): se non esistono soluzioni
- illimitati (*unbounded*): se per una o più componenti della soluzione è possibile aumentarne (o ridurne) il valore all'infinito senza mai violare i vincoli
- degeneri (*degenerate*): sono problemi ammissibili in cui più soluzioni di base corrispondono allo stesso vertice del poliedro. In linea teorica possono far ciclare l'algoritmo (ossia l'algoritmo torna a controllare basi già viste in precedenza), ma all'atto pratico questo può essere evitato introducendo delle euristiche. Per semplicità la nostra implementazione una volta riconosciuto un problema come degenere lo segnala all'utente e termina senza cercare di risolverlo.

Forma canonica e forma standard

I problemi di massimizzazione possono essere convertiti in problemi di minimizzazione (e viceversa), così come è possibile manipolare le singole disequazioni. Queste operazioni servono a riportare i problemi in una forma che permetta di utilizzarli da parte dei solver. In particolare vengono utilizzate la forma *canonica* e la forma *standard*.

Un problema (di massimizzazione) è in forma canonica se è nella forma

$$\begin{gathered} \max c^T x \\ \text{subject to} \\ Ax \leq b \\ x \geq 0 \end{gathered}$$

Considerando che qualsiasi disequazione nella forma $\alpha x \leq y$ può essere convertita in una equazione equivalente $\alpha x + \delta = b$ definiamo la forma standard di un problema in forma canonica

$$\begin{gathered} \max (c|0)^T x \\ \text{subject to} \\ (A|I)x = b \\ x \geq 0 \end{gathered}$$

con $(A|I) \in \mathbb{R}^{m \times (n+m)}$, $(c|0) \in \mathbb{R}^{n+m}$ e I la matrice di identità di dimensione m.

In altre parole aggiungiamo una nuova variabile al problema (detta variabile *slack*) per ogni disequazione.

La forma standard è quella che viene utilizzata dagli algoritmi di soluzione.

Metodo del simplesso a due fasi

Il metodo del simplesso è un'algoritmo per risolvere i problemi di ottimizzazione lineari.

Si basa sul teorema secondo il quale le soluzioni (dette *soluzioni di base*) di un qualsiasi problema in forma standard sono i *vertici* del poliedro costruito sui vincoli.

Ogni vertice è individuato da una o più *soluzioni di base ammissibili*, cioè tutti i valori delle variabili in base sono maggiori o uguali a 0, mentre il valore delle altre variabili è 0. Se in base sono presenti uno o più valori uguali a 0, si dice che la base è *degenere*.

Esistono diverse implementazioni del metodo del simplesso, per questo progetto verrà implementato il cosiddetto *simplesso a due fasi con il metodo del tableau*

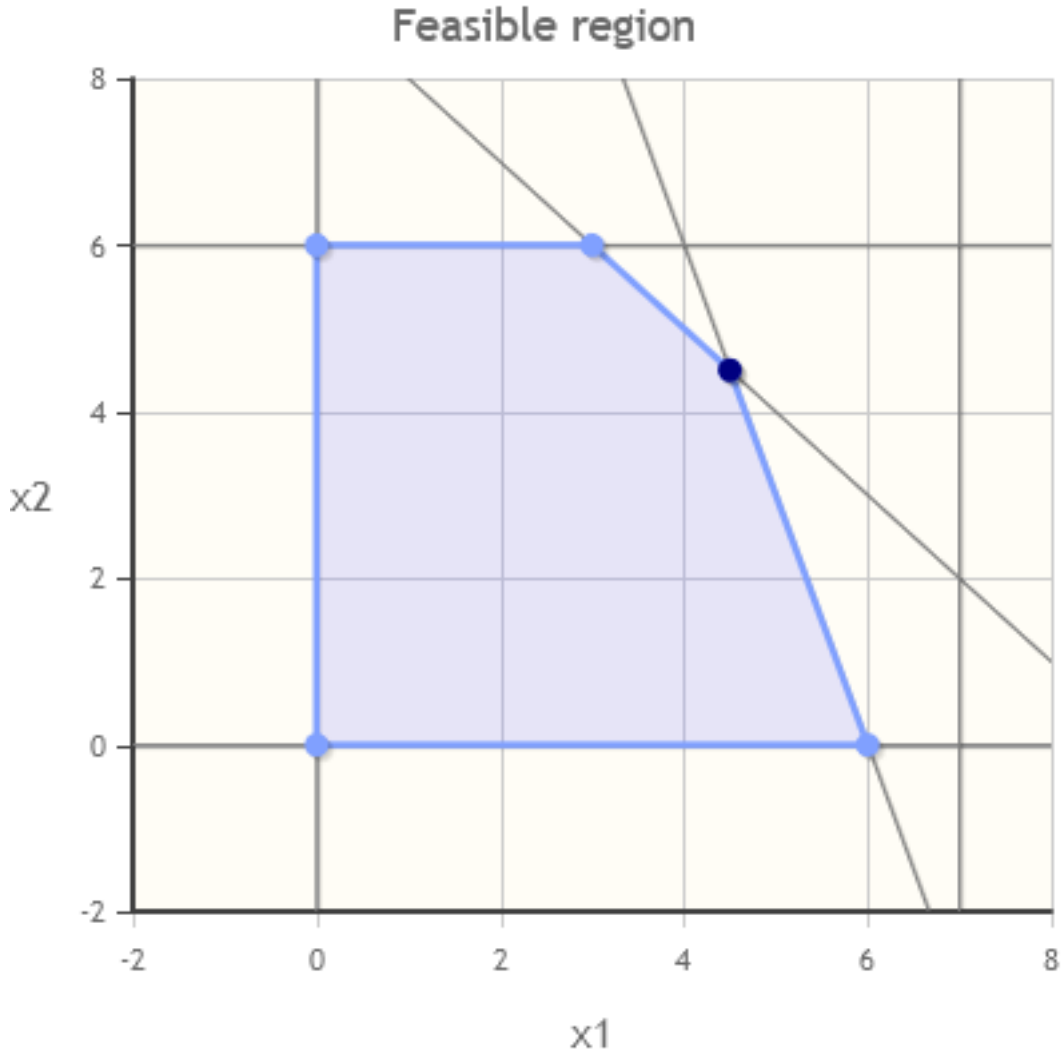
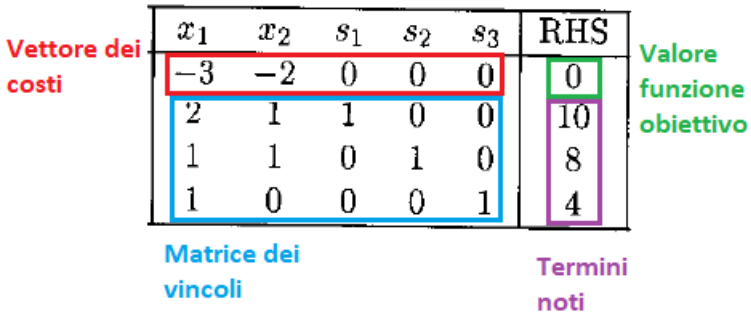


Figure 1: Esempio di problema con due variabili. Il vertice evidenziato in blu è la soluzione

Cos'è il tableau

Il tableau di un simplesso è la struttura dati su cui viene eseguito l'algoritmo del simplesso.

Si tratta di una matrice composta nel seguente modo:



x_1	x_2	s_1	s_2	s_3	RHS
-3	-2	0	0	0	0
2	1	1	0	0	10
1	1	0	1	0	8
1	0	0	0	1	4

Nella prima riga della matrice (R_0) è presente il vettore dei costi. Inizialmente il vettore dei costi contiene i coefficienti della funzione obiettivo con segno invertito. La base viene memorizzata in un vettore a parte.

Algoritmo di soluzione

Algorithm 1: Simplex

Data: $T \in \mathbb{R}^{(m+1)\times(n+1)}, \mathcal{B} \in \mathbb{N}^m$

Output: x come soluzione, $\mathcal{B}$ base della soluzione e $T[0][n]$ valore ottimo della funzione obiettivo
OPPURE problema UNBOUNDED

$A \leftarrow T[1..m][0..n-1]$; `/* La notazione` $[a..b]$ `indica una porzione di array dall'indice` a `all'indice` b `*/`

$b \leftarrow T[1..m][n]$;

$c \leftarrow T[0][0..n-1]$;

while $\exists i : c[i] < 0$ **do**

  Pick $h \in \{0,\dots,n-1\} : c[h] = \min_{i=0,\dots,n-1} c[i]$;

  if $\forall i \in \{0,\dots,m-1\} \Rightarrow A[0..m-1][h] \leq 0$ **then**

    return *problem UNBOUNDED*;

  end

  Pick $k \in \{0,\dots,m-1\} : c[k] = \min_{i=0,\dots,m-1} \left\{ \frac{b[i]}{A[i][h]} \wedge A[i][h] > 0 \right\}$;

  $p \leftarrow A[k][h]$;

  $\mathcal{B}[k] \leftarrow h$;

  $R_p \leftarrow copy(T[k][0..n])$;

  $C_p \leftarrow copy(T[0..m][h+1])$;

  for $i \leftarrow 0$ *to* m **do**

    for $j \leftarrow 0$ *to* n **do**

      if $i = k$ **then**

        $T[i][j] = \frac{1}{p} T[i][j]$;

      end

      else

        $T[i][j] = T[i][j] - \frac{C_p[i]}{p} R_p[j]$;

      end

    end

  end

end

$x[0..n-1] \leftarrow 0$;

for $i \leftarrow 0$ *to* $m-1$ **do**

  $x[\mathcal{B}[i]] \leftarrow b[i]$;

end

return $(x, \mathcal{B}, T[0][n])$; `/*` $T[0][n]$ `è il valore della funzione obiettivo */`

Complessità Considerando che l'algoritmo non fa altro che esplorare le soluzioni di base ammissibili, l'algoritmo esplora al massimo $O\binom{n+m}{m}$ (con n numero di variabili e m il numero di vincoli).

Nei casi peggiori il simplesso può raggiungere complessità esponenziali. Tuttavia nel corso degli anni si è visto EMPIRICAMENTE come questa complessità venga raggiunta in pochissimi casi specifici, anzi l'algoritmo si è rivelato molto efficiente su problemi "reali", dove in media raggiunge una complessità nell'ordine di $O(mn)$, anche a confronto con altri algoritmi di soluzione (polinomiali per costruzione) come il metodo dei punti interni.

Da queste considerazioni si può concludere anche come la complessità del simplesso dipenda molto dall'istanza del problema fornito in input.

Fase 1

In questa fase si costruisce il tableau e si verifica se il problema di partenza è ammissibile o meno. Per fare ciò si risolve il cosiddetto **problema ausiliario**

$$\begin{gathered}\max(0|-1)^T x \\ \text{subject to} \\ (A|I|I)x = b \\ x \geq 0\end{gathered}$$

In pratica aggiungo al problema originario m nuove variabili (dette **variabili artificiali**) e cerco di azzerarne la somma negata. Se ciò avviene il problema di partenza è ammissibile e la soluzione di base ottenuta può essere utilizzata come base di partenza per la fase 2 dell'algoritmo, altrimenti il problema originale non è ammissibile. Se nella base finale una delle variabili artificiali è rimasta in base il problema è degenere.

Fase 2

Terminata la prima fase si eliminano le variabili artificiali dal tableau e si sostituisce la funzione obiettivo con quella del problema originale, in seguito si procede a risolvere il problema originale.

Simplesso a due fasi

Scelte implementative e algoritmi utilizzati

Schema di parallelizzazione

Per introdurre parallelismo si è deciso di parallelizzare i singoli passi che lo compongono, mantenendo invece sequenziale la struttura dell'algoritmo in sè.

Così facendo non si prevede una riduzione della complessità teorica dell'algoritmo, ma ottenere un miglioramento empirico sui tempi necessari a risolvere modelli anche molto complessi (in quanto si va a ridurre la complessità delle singole operazioni sui vettori e sulle matrici).

Gestione della memoria

Implementando gli algoritmi in CUDA, merita particolare risalto il modo in cui viene organizzata la memoria:

- i dati del problema originale sono memorizzati in memoria host, mentre il tableau viene costruito e memorizzato nella memoria globale della scheda video
- i dati del problema originale vengono memorizzati utilizzando memoria *page-locked*, questo per permettere i trasferimenti di memoria in parallelo con i CUDA Stream (e quindi ottimizzare la fase di costruzione del tableau in global memory)
- il vettore della base è memorizzato come memoria *mapped page-locked*, in modo da potervi accedere sia lato host che lato GPU in qualsiasi momento senza dover gestire i trasferimenti

Algorithm 2: Simplex-Phase1

Data: $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$ in forma standard (senza var. slack)
Output: $T \in \mathbb{R}^{(n+m+1)\times(m+1)}$ tableau per la fase 2 e $\mathcal{B} \in \mathbb{N}^m$ base ammissibile OPPURE problema INFEASIBLE o DEGENERE

```
/* riempimento tableau                                                    */
```
$T \in \mathbb{R}^{(m+1)\times(n+2m+1)}$;
$T[0][0..n+m-1] \leftarrow 0$;
$T[0][n+m..n+2m-1] \leftarrow 1$;
$T[0][n+2m] \leftarrow 0$;
$T[1..m][n+2m] \leftarrow b$;
$T[1..m][0..n-1] \leftarrow A$;
$T[1..m][n..n+m-1] \leftarrow I$; /* variabili slack */
$T[1..m][n+m..n+2m-1] \leftarrow I$; /* variabili artificiali */
```
/* Nego eventuali disequazioni con termine noto negativo (in modo da rendere la base
   iniziale ammissibile)                                                  */
```
for $i \leftarrow 1$ *to* m **do**
  if $T[i][n+2m] < 0$ **then**
    for $j \leftarrow 0$ *to* $n+2m$ **do**
      $T[i][j] = -T[i][j]$;
    end
  end
end
$\mathcal{B} \leftarrow [n+m, \ldots, n+2m-1]$; /* base iniziale con le variabili artificiali */
```
/* esprimo la funzione obiettivo solo in funzione delle variabili non di base */
```
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 0$ *to* $n+2m$ **do**
    $T[0][j] \leftarrow T[0][j] - \frac{T[0][\mathcal{B}[i]]}{T[i][\mathcal{B}[i]]} T[i][j]$
  end
end
$(x, \mathcal{B}, f) \leftarrow Simplex(T, \mathcal{B})$;
if $f < 0$ **then**
  return *problem INFEASIBLE*;
end
if $\exists i : n+m \leq \mathcal{B}[i] < n+2m$ **then**
  return *problem DEGENERE*;
end
```
/* Elimino le variabili artificiali                                       */
```
$T[0..m][n+m] \leftarrow copy(T[0..m][n+2m])$;
$drop(T[0..m][n+m+1..n+2m])$;
return $(T, \mathcal{B})$

Algorithm 3: Simplex-Phase2

Data: $T \in \mathbb{R}^{(m+1)\times(n+m+1)}, \mathcal{B} \in \mathbb{N}^m, c \in \mathbb{R}^n$
Output: soluzione al problema iniziale OPPURE problema UNBOUNDED
$T[0][0..n-1] \leftarrow -c$;
$T[0][n..n+m] \leftarrow 0$;
`/* esprimo la funzione obiettivo solo in funzione delle variabili non di base */`
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 0$ *to* $n+m$ **do**
    $T[0][j] \leftarrow T[0][j] - \frac{T[0][\mathcal{B}[i]]}{T[i][\mathcal{B}[i]]} T[i][j]$
  end
end
return $Simplex(T, \mathcal{B})$

Algorithm 4: 2Phases-Simplex

Data: $A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$
Output: soluzione al problema OPPURE problema INFEASIBLE o UNBOUNDED o DEGENERE
$(T, \mathcal{B}) \leftarrow Simplex - Phase1(A, b)$;
return $Simplex - Phase2(T, \mathcal{B}, c)$

- siccome l'algoritmo opera spesso su singole righe o colonne è fondamentale memorizzare i dati in memoria globale in modo da minimizzare il numero di accessi strided o disallineati. Per questo motivi si è deciso il seguente schema di memorizzazione (visibile in figura 2):
  - la matrice del tableau viene allocata con *cudaMallocPitch* ed è memorizzata in maniera *lineare* per **colonne**. In altre parole l'implementazione lavorerà sul tableau trasposto. Questo perchè l'algoritmo tende ad accedere più alle singole colonne che alle singole righe. Infatti così facendo sarà necessario implementare solo un kernel che accede alla memoria in maniera strided (laddove la classica memorizzazione per righe avrebbe richiesti tre),
  - il vettore dei termini noti non viene memorizzato in fondo al tableau, ma nella prima riga della matrice. In questo modo nel passaggio dalla fase 1 alla fase 2 non è necessario spostare la colonna nella matrice, ma è sufficiente troncare il numero di righe della matrice da una variabile,
  - il vettore dei costi non viene memorizzato nella matrice del tableau, ma in un vettore a parte e il valore della funzione obiettivo è nel primo elemento di questo vettore.

Ricerca del pivot e test di ottimalità

Eliminazione di Gauss

Come accennato nell'introduzione teorica all'algoritmo del simplesso in due fasi un passo fondamentale per raggiungere la soluzione consiste nell'esprimere la funzione dei costi in funzione delle variabili non in base, per fare ciò si utilizzano una serie di eliminazionei di gauss.

Ignorando i dettagli del problema, ciò che si vuole fare consiste in:

Dati:

- Una matrice A_{n*m}
- Un vettore di moltiplicatori M di dimensione m
- Il vettore dei costi C di dimensione n

Si vuole calcolare C' come:

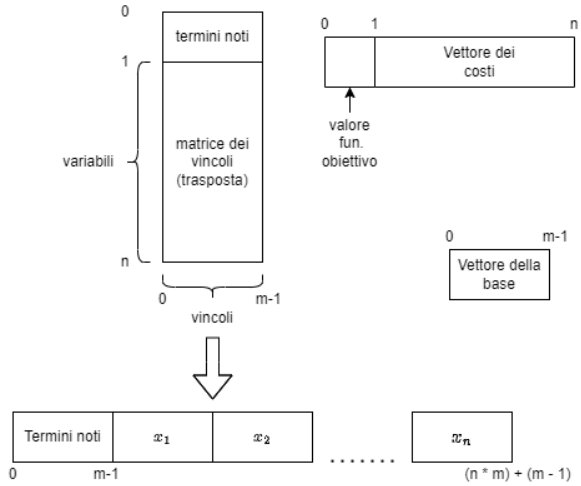


Figure 2: Schema di memorizzazione in memoria globale

$$c'_i = c_i - \sum_{k=0}^{m}(a_{k,i} * m_k) \qquad 0 < i < n$$

L'approccio seriale è banale e consiste nello scorrimento di tutta la matrice e per ogni elemento $a_{a,b}$ si calcola il rispettivo c'_b.

Risulta possibile ottimizzare l'efficienza di questo algoritmo attraverso il parallelismo.

Approccio naive

Per discutere gli approcci paralleli è bene ricordare come i dati su cui l'eliminazione di gauss deve agire sono rappresentati in memoria, ovvero:

- Il vettore dei costi C è rappresentato in un vettore a se di dimensione m, con il primo elemento il valore della funzione obiettivo
- il vettore dei moltiplicatori M è rappresentato da un vettore a se di dimensione naive
- La matrice dei A_{m*n} vincoli è linearizzata per righe, con la prima colonna contenente i termini noti

Dato che la matrice è linearizzata per colonne è più semplice effettuare i ragionamenti sulla trasposta A^{t}.

L'approccio più semplice per parallelizzare questo algoritmo consiste nell'utilizzare delle tiles di thread (blocchi bidimensionali) dove ciascuno di essi si occupa di eseguire la stessa operazione della versione seriale, dunque ogni thread in base alla sua posizione globale (posizione nella grid) aggiorna il valore della funzione obiettivo.

Per come è rappresentata la matrice abbiamo che thread consecutivi accedo ad elementi consecutivi sia della matrice sia del vettore dei moltiplicatori, questo permette di sfruttare in modo naturale la coalescenza evitando accessi stride se si scieglie in maniera appropriata la dimensione sull'asse x dei blocchi.

Il problema di questa soluzione che la rende non ottimale è la concorrenza durante la scrittura sul vettore dei costi, infatti thread con lo stesso threadIdx.x nello stesso blocco accederanno in scrittura sulla stessa locazione di memoria dato che dovranno aggiornare lo stesso elemento della funzione obiettivo. Anche blocchi differenti possono accedere alla stessa locazione di memoria in scrittura, causando concorrenza, questo accade quando thread di blocchi diversi lavorano sulla stessa riga della matrice, aggiornando di conseguanza lo stesso elemento della funzione obiettivo.

A causa della concorrenza è necessario utilizzare l'operazione di somma atomica per evitare inconsistenze, ma questo porta ad una perdita di efficienza dato che scritture atomiche sulla stessa locazione di memoria vengono serializzate.

A seconda della compute capability della scheda video utilizzata il discorso varia:

-Nel caso in cui la scheda grafica disponga di compute capability superiore o uguale alla 6.0 potrebbe essere sufficiente l'approccio naive, con opportuni accorgimenti, dato che l'effiecienza delle operazioni atomiche permette di avere risultati paragonabili se non migliori alla migliore soluzione trovata che eviti o limiti l'utilizzo delle operazioni atomiche.

-Nel caso in cui la compute capability sia inferiore alla 6.0 non è disponibile l'operazione atomica di somma di double, per tale motivo è necessario utilizzare un implementazione simulata con costi non triviali sulle performance, per tale motivo utilizzare l'approccio che limita le atomicadd risulta migliore dell'approccio naive.

Approccio ragionato

L'approccio ragionato utilizza l'idea della riduzione per evitare somme atomiche concorrenti permettendo tempi ottimali anche nel caso di compute capability inferiori alla 6.0.

Pensando al problema da risolvere quello che si vuole effettivamente andare a fare è ridurre delle righe opportunamente moltiplicate per un moltiplicatore e sottrarre il risultato di tale riduzione all'elemento della funzione obiettivo corrispondente a tale riga.

La soluzione migliore ottenuta è la seguente:

- Si utilizzano dei blocchi bidimensionali di dimensione (32,k) ed una grid di dimensione (1, roof(righeTrasposta/k))
- Per come funziona la suddivisione in warp di un blocco si avranno warp di 32 thread consecutivi che lavorano su 32 elementi consecutivi della stessa riga della matrice
- Ogni thread di un warp carica il proprio elemento della matrice ed il rispettivo moltiplicatore, dato che i thread del warp accedono ad elementi consecutivi della riga della trapsosta si avranno accessi coalescenti sia per i dati della matrice sia per i moltiplicatori.
- Si utilizza una riduzione attraverso le primitive di comunicazione tra thread di un warp, ovvero si utilizza una riduzione intra-warp.
- Un thread per warp aggiorna la funzione obiettivo nella locazione relativa alla riga su cui ha lavorato.
- Si utilizza un grid stride per scorrere tutta la riga della matrice

Tale soluzione permette di evitare completamente l'utilizzo delle operazioni di somma atomica.

Un ulteriore ottimizzazione consiste nell'eseguire una somma prima di passare alla riduzione: ogni thread esegue già una somma quando carica il proprio valore della matrice, caricandone due invece che 1.

Rimane un problema di questa soluzione: quando il risultato di una riduzione viene sottratto nella funzione obiettivo si ha una divergenza nel warp, quello che invece sarebbe più efficiente sarebbe di fare eseguire l'aggiornamento da un unico warp (invece che tanti quanti l'altezza del blocco) ma qualsiasi soluzione provata aggiunge un overhead superiore al tempo impiegato lasciando l'aggiornamento a warp diversi.

Aggiornamento della tabella

Risultati sperimentali