Esame di Programmazione su Architetture Parallele

Metodo del simplesso per la risoluzione della programmazione lineare

Belliato Riccardo (mat. 142652) Simone Tomada

2022-10-05

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

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*

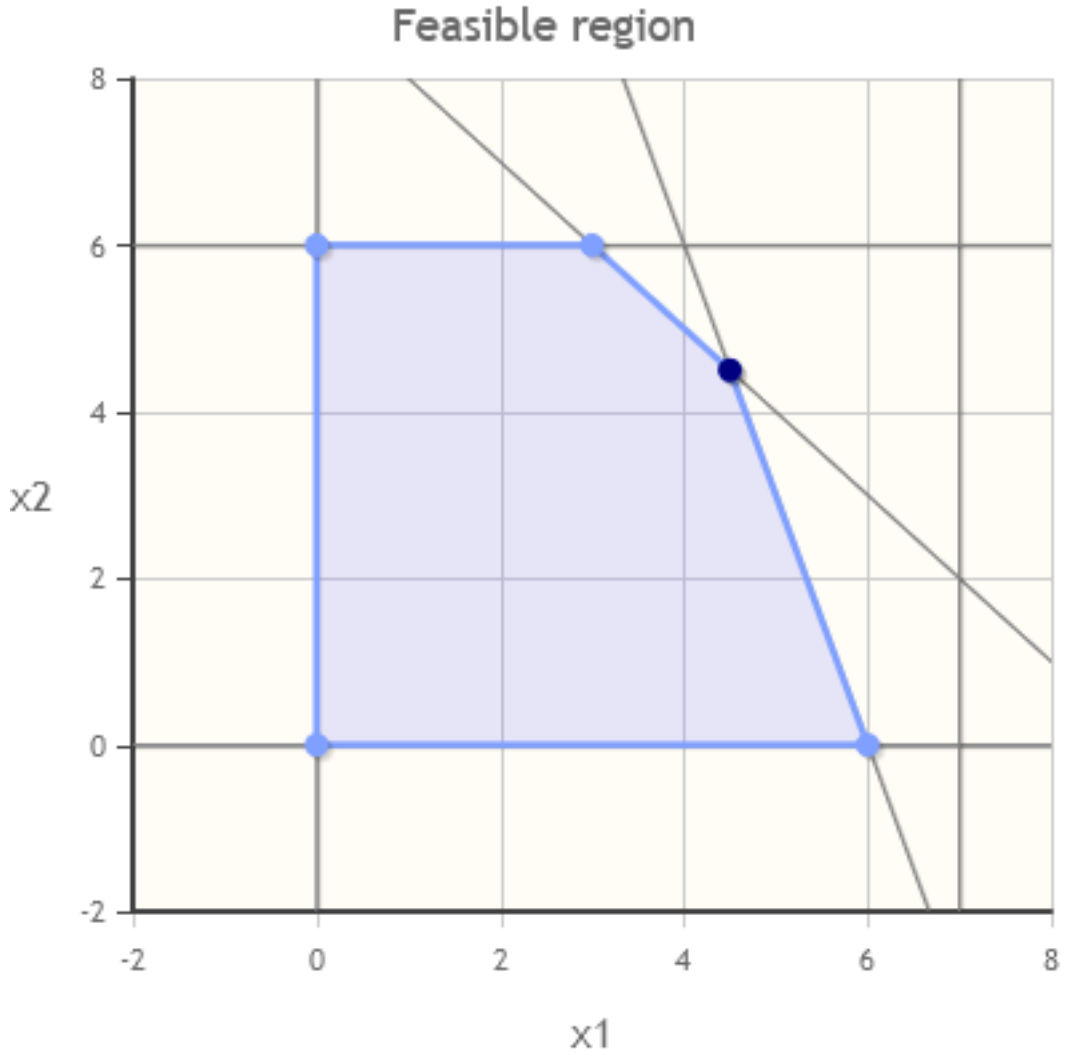
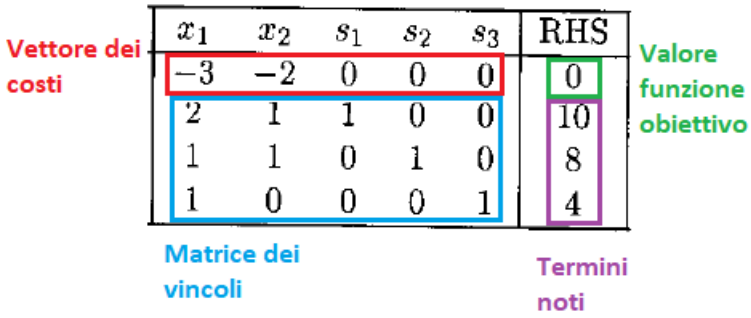


Figure 1: Esempio di problema con due variabili sul piano cartesiano. La regione evidenziata è il poliedro costruito sui vincoli. Il vertice evidenziato in blu scuro è la soluzione.

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 `(compresi) */`
$b \leftarrow T[1..m][n]$;
$c \leftarrow T[0][0..n-1]$;
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**
        $T[i][j] \leftarrow \frac{1}{p} T[i][j]$;
      end
      else
        $T[i][j] \leftarrow T[i][j] - \frac{C_p[i]}{p} R_p[j]$;
      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. Per lo pseudocodice si veda l'algoritmo 2.

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. Per lo pseudocodice si veda l'algoritmo 3.

Simplesso a due fasi

Si veda l'algoritmo 4.

Scelte implementative

Gestione della 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.
- 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):

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**
    $T[0][j] \leftarrow T[0][j] - T[0][\mathcal{B}[i]] * T[i][j]$
  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

```
1 T[0][0..n − 1] ← −c;
2 T[0][n..n + m] ← 0;
  /* esprimo la funzione obiettivo solo in funzione delle variabili non di base */
3 for i ← 1 to m do
4 |   for j ← 0 to n + m do
5 |   |   T[0][j] ← T[0][j] − T[0][B[i]] ∗ T[i][j]
6 |   end
7 end
8 return Simplex(T, 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

```
1 (T, B) ← Simplex − Phase1(A, b);
2 return Simplex − Phase2(T, B, c)
```

- la matrice del tableau è memorizzata linearizzata per **colonne**. In altre parole l'implementazione utilizza il tableau trasposto, ossia le colonne del tableau sono i vincoli del problema, mentre le righe sono le variabili. L'algoritmo, infatti, tende di più ad accedere ai vettori delle singole variabili che a quelli dei singoli vincoli. Per copiare singole colonne della matrice in vettori in global memory si è visto che l'approccio migliore (dal punto di vista del tempo necessario ad eseguire l'operazione) è quello di utilizzare comunque gli accessi strided, invece di altri pattern di accesso più efficienti dal punto di vista della bandwith utilizzata (Es. copiare un tile della matrice in shared memory e poi copiare la colonna desiderata dalla shared memory in global memory);
- 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 senza dover intervenire sull'allocazione della memoria;
- 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 (analogalmente a quanto fatto con il vettore dei termini noti al punto precedente).

Schema di parallelizzazione

Per introdurre parallelismo si è deciso di parallelizzare le singole operazioni svolte sui vettori e sulle matrici, mantenendo intatta la struttura dell'algoritmo.

Così facendo non si prevede una riduzione della complessità teorica dell'algoritmo, ma ottenere un miglioramento empirico sul tempo necessario a risolvere un'istanza rispetto a un'algoritmo puramente seriale.

Creazione tableau (Algoritmo 2, righe 1-16)

Il tableau viene creato a partire dai dati del problema (matrice dei vincoli già linearizzata per colonne, vettore dei termini noti e dei coefficienti della funzione obiettivo). Per prima cosa si istanziano in global memory:

- il vettore che memorizza la matrice dei vincoli e il vettore dei termini noti con *cudaMallocPitch* di dimensione $m \times (n + 2m + 1)$. La larghezza reale della matrice (in byte) in memoria viene salvata nel campo *pitch* di *tabular_t*. Per comodità si salvano in due campi distinti di *tabular_t* i puntatori al

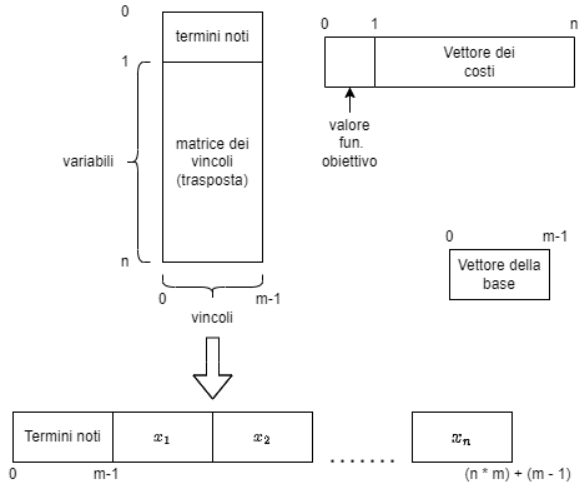


Figure 2: Schema di memorizzazione in memoria globale

vettore dei termini noti (lo stesso della matrice) e quello alla matrice dei vincoli (la seconda riga della matrice);

- il vettore dei costi di dimensione $n + 2m + 1$ con *cudaMalloc*
- il vettore della base di dimensione m utilizzando *cudaHostAlloc* con il flag *cudaHostAllocMapped*

Successivamente si procede a riempire questi vettori. Per fare ciò si utilizzano sei cudaStream differenti:

1. il primo stream imposta a 0 i primi $n + m + 1$ valori del vettore dei costi a 0 con *cudaMemset*,
2. il secondo stream imposta a 1 i restanti m valori della funzione dei costi (utilizzando un kernel apposito),
3. il terzo stream copia la matrice dei vincoli dalla memoria host alla matrice in global memory con *cudaMemcpy2D*,
4. il quarto stream costruisce in coda alla matrice dei vincoli le due matrici di identità, una per le variabili slack e una per le variabili artificiali, utilizzando un kernel apposito,
5. il quinto stream copia il vettore degli indicatori dalla memoria host alla global memory con *cudaMemcpy*,
6. il sesto stream inizializza il vettore della base con numeri progressivi da $n + m$ a $n + 2m - 1$ con un kernel apposito.

Una volta terminata questa prima fase si procede a scandire il vettore dei termini noti per verificare che tutti i valori siano maggiori o uguali a 0 (in modo che la base iniziale sia ammissibile). Se un elemento è negativo viene lanciato un nuovo kernel (tramite parallelismo dinamico) che inverte i segni a tutti gli elementi della colonna corrispondente. Questo equivale a moltiplicare l'equazione della colonna per -1.

Aggiornamento funzione obiettivo (Algoritmo 2, righe 17-21 e Algoritmo 3, righe 3-7)

Lo scopo di questa operazione è quello di azzerare tutti gli elementi del vettore dei costi in corrispondenza delle variabili in base (una variabile è in base se nel suo vettore è presente un solo elemento a 1 e il resto è a 0).

Per fare ciò si effettuano una serie di eliminazioni di Gauss sottraendo al vettore dei costi tutte le righe della matrice opportunatamente moltiplicate per il valore della i-esima variabile di base nel vettore dei costi.

Per fare questa operazione sono stati provati due approcci:

- un approccio "naive" che sfrutta le atomicAdd native e che funziona banalmente lasciando ad ogni thread del tile il caricamento, moltiplicazione ed aggiornamento nella funzione obiettivo,

- un approccio più ragionato che sfrutta la riduzione, dove l'aggiornamento della funzione obiettivo viene fatta solo da un sottoinsieme di kernel sfruttando a tal fine la riduzione intra-warp.

Il motivo per sono state pensate due soluzioni è dovuto al fatto che le somme atomiche per i double sono disponibili solo a partire dalla compute capability 6.0, il che richiede di implementare manualmente le somme atomiche attraverso l'operazione atomica CAS se si vuole garantire il funzionamento del kernel anche per schede video con C.C. inferiore alla 6.0.

Nella seguente tabella sono riportati i tempi ottenuti per i due kernel (con configurazione di lancio ottimale ed ottimizzazioni implementate).

Table 1: Risultati sperimentali dei diversi kernel, con la configurazione di lancio ottimale (blocco: 32*32, grid: (1, roof(24.588/32))) ed ottenuti con una scheda Geforce MX250

Naive (cc $<$ 6.0)	Riduzione (cc $<$ 6.0)	Naive (cc $>$ 6.0)	Riduzione (cc $>$ 6.0)
209.23ms	52.142ms	38.720ms	50.610ms

IL risultato peggiore ottenuto risulta essere quello con l'approccio naive e con cc $<$ 6.0, che impiega circa quattro volte tanto gli altri casi. Per tale motivo in tale situazione conviene utilizzare l'approccio con riduzione.

Nel caso di cc >= 6.0 la situazione è diversa e quello più efficiente risulta essere quello che sfrutta l'approccio naive, grazie all'ottimizzazione dell'atomicAdd

A partire da queste considerazioni si è deciso di sfruttare l'approccio naive mantenendo quello ragionato solamente nel caso il programma venga compilato per essere utilizzato su schede video con cc $<$ 6.0.

Per fare ciò si utilizza la macro ___CUDA_ARCH___ con la direttiva di pre-processore #if.

In ogni caso nella pratica il tempo necessario per svolgere l'eliminazione di gauss è trascurabile, anche nel caso peggiore (naive, cc<6.0 e privo di ottimizzazioni che impiega un tempo dell'ordine delle decine di secondi con gpu MX250), come si vedrà durante la discussione dei risultati sperimentali.

Verifica di degenerazione (Algoritmo 2, riga 26)

Per verificare se il problema è degenere si scandisce il vettore della base e si verifica se l'elemento in questione sia compreso tra $n+m$ e $n+2m$ (non compreso). Se sì, si incrementa di 1 (con *atomicAdd*) un contatore in global memory e una volta terminata la scansione si controlla se il valore di questo contatore sia maggiore di 0.

Viene utilizzato lo schema di parallelizzazione visto a lezione per le riduzioni intra-warp:

- 512 thread per blocco,
- il numero di blocchi calcolato con la formula $\min((N+512-1)/512, 1024)$, con N dimensione del vettore,
- ogni thread controlla più elementi del vettore utillizzando l'indirizzamento sequenziale.

Sostituzione vettore dei costi (Algoritmo 3, righe 1-2)

All'inizio della fase 2 il vettore dei costi utilizzato nella fase 1 va sostituito nei primi n elementi dal vettore della funzione obiettivo del problema originale, mentre il resto del vettore (variabili slack) va impostato a 0.

Anche qui vengono utilizzati due CUDA Stream: il primo esegue *cudaMemsetAsync* sugli ultimi m elementi del vettore dei costi per impostare i valori a 0, il secondo copia il vettore della funzione obiettivo dalla memoria host ai primi n elementi del vettore dei costi con *cudaMemcpyAsync* e nega il segno degli elementi copiati (utilizzando lo schema già visto nella sezione precedente).

Ricerca dell'elemento minimo nel vettore dei costi e in quello degli indicatori (Algoritmo 1, righe 4-5 e 9)

Questa fase prevede di trovare il minimo in due vettori di numeri comunicando allo stesso tempo l'**indice** dell'elemento nel vettore.

Per fare ciò è stata implementata una variante della riduzione intra-warp a due fasi vista a lezione che lavora su due vettori: il primo contiene i valori candidati, il secondo gli indici di questi valori nel vettore. Quest'ultimo inizialmente è vuoto e viene popolato alla fine della prima fase con gli indici dei valori candidati per ogni blocco, per essere poi utilizzati nella seconda fase per ottenere il minimo globale.

Per distinguere le due fasi senza dover riscrivere due volte la stessa funzione sono stati utilizzate le funzioni template di C++ con un parametro booleano.

Verifica di unbounding (Algoritmo 1, riga 6)

Per verificare se il problema è unbounded si deve verificare che tutti i valori del vettore della variabile che sta per entrare in base siano minori o uguali a 0.

Per fare ciò è stata implementata una reduction (sempre intra-warp a due fasi perchè non esiste *atomicMax* per i valori double) per ottenere il valore massimo di un vettore. Se il valore massimo di un vettore è ≤ 0, allora devono per forza esserlo anche tutti gli altri.

Le procedure di reduction utilizzano copie dei vettori passati come parametri.

Aggiornamento tableau (Algoritmo 1, righe 12-24)

Una volta ottenuto il pivot si procede ad aggiornare il tableau. Per prima cosa si copiano in due nuovi vettori in global memory la riga (con *cudaMemcpy*) e la colonna (con un kernel apposito) del pivot, dopodiché viene lanciato il kernel per l'aggiornamento.

Per lo schema di parallelizzazione è stato utilizzato lo schema visto a lezione per l'esercizio dell'istogramma per l'accesso sequenziale (che a sua volta estende quello visto per le reduction).

Per quanto riguarda la grid si è visto che per una matrice di 8192×8192 elementi e dati dei blocchi da 32×32 thread, dal punto di vista delle prestazioni la grid migliore è risultata essere quella da 16×16 blocchi.

Prendendo in considerazione che per una matrice da 32×32 elementi il pattern migliore è quello che ad ogni thread assegna un valore, per calcolare il numero di blocchi necessari data una dimensione della matrice è stata calcolata una retta tra i punti $(1, 32)$ e $(8192, 16)$ ottenendo così l'equazione

$$y = \left\lceil \frac{x + 512}{544} \right\rceil$$

Considerazioni sulle operazioni in virgola mobile Considerato che l'algoritmo del simplesso utilizza numeri in virgola mobile, particolare enfasi va messa sull'arrotondamento dei numeri reali e sul fare meno operazioni possibili per evitare di propagare gli errori.

Inoltre per fare i confronti tra numeri non è possibile utilizzare le classiche espressioni logiche (in virgola mobile due numeri sono considerati uguali se la loro differenza in valore assoluto è minore di un valore ϵ "piccolo"), questa fattispecie è stata implementata dalla funzione *compare* di macro.h, utilizzando come valore di $\epsilon = 10^{-9}$.

Per quanto riguarda l'aggiornamento del tableau, per effettuare il calcolo $T[i][j] - \frac{C_p[i]}{p} R_p[j]$ viene utilizzata la funzione *fma* della libreria math di CUDA[1]. Il calcolo viene così ridotto a due operazioni in virgola mobile (divisione e fma) invece che tre.

[1] Questa funzione riduce calcoli del tipo $A * B + C$ in una sola operazione in virgola mobile

Estrazione della soluzione (Algoritmo 1, righe 25-29)

Per ottenere la soluzione al problema si procede nel seguente modo:

1. Si copia il primo valore del vettore dei costi del tableu per ottenere il valore ottimo della funzione obiettivo,
2. Si istanzia un vettore in mapped memory con gli elementi impostati a 0,
3. Si scansiona il vettore della soluzione di base ottenuta (con lo schema di parallelizzazione già visto in precedenza). Se la variabile considerata non è una variabile slack si legge il valore della variabile dal vettore dei termini noti (ossia la prima riga del tableau).

Risultati sperimentali

Al fine di analizzare le prestazioni del solver è stato predisposta una funzione di benchmark con dati generati da un generatore casuale di istanze.

Il benchmark consiste nella risoluzione di una serie di problemi con dimensione crescente a partire da 256 variabili per 256 vincoli a 8192 variabili per 8192 vincoli. Il passo tra un problema e l'altro è esponenziale sulle potenze di due.

Per i benchmark il generatore utilizza come seed la dimensione del problema, per cui si andranno a generare ogni volta gli stessi problemi. In questo modo è possibile rieseguire il benchmark su dispositivi diversi e poter confrontare le prestazioni.

Misurazione dei tempi

Per misurare i tempi sono stati utilizzati gli eventi di CUDA così come visto a lezione e sulla guida ufficiale di NVIDIA. Nonostante il metodo *cudaEventElapsedTime* misuri i tempi in ms, per evitare numeri eccessivamente piccoli (soprattutto sulle istanze più piccole) i tempi dei benchmark sono espressi in μs (ottenuti moltiplicando il valore della misurazione per 1000).

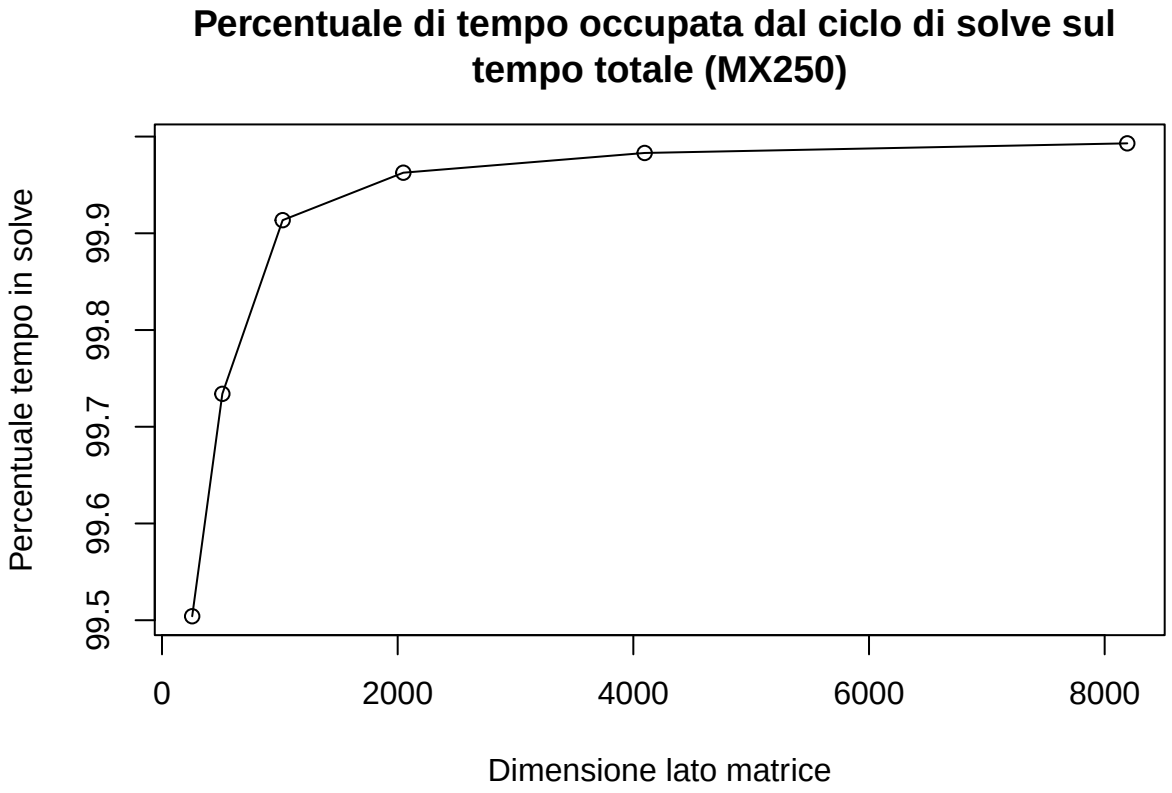
Come già discusso nel paragrafo 29 sulla complessità del simplesso, non ha molto senso misurare il tempo globale dell'algoritmo, per cui si è deciso di misurare i tempi delle singole operazioni che lo compongono, in particolare il tempo necessario a svolgere una singola iterazione del ciclo ricerca del pivot-aggiornamento del tableau (algoritmo 1 da riga 4 a riga 24), che d'ora in poi chiameremo **ciclo di solve** per semplicità.

La possibilità di misurare i tempi è attivabile compilando il progetto con la macro TIMER.

Tutte le misurazioni vengono salvate su file csv per essere analizzate utilizzando il software R.

Occupazione ciclo di solve

Come accennato durante la spiegazione degli approcci utilizzati per la parallelizzazione dell'algoritmo il collo di bottiglia dell'algoritmo risulta essere il **ciclo di solve** come mostrato dal seguente grafico.



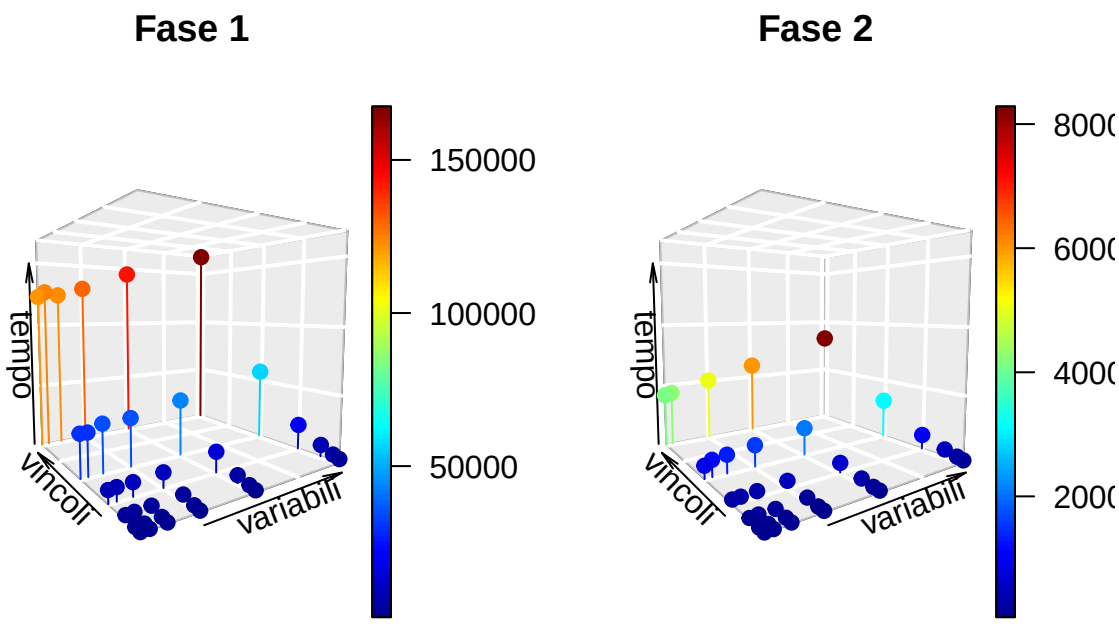
Come possibile notare dal grafico, per l'esecuzione delle istanze di benchmarking si ha un tempo speso nella esecuzione del ciclo di solving superiore al 99% del tempo totale già per problemi di dimensione 256×256, per cui ad una riduzione della durata di una singola iterazione di tale ciclo corrisponde una riduzione nel tempo di esecuzione e rispettiva risoluzione del problema di partenza.

Analisi in funzione della dimensione del problema

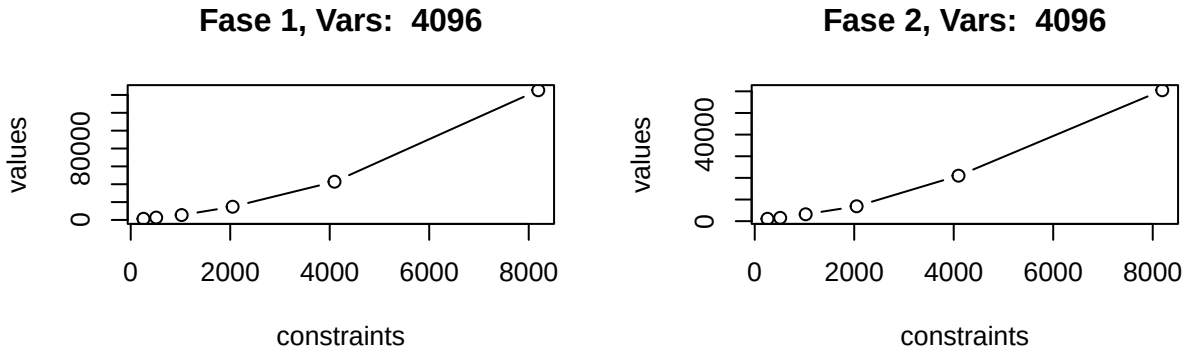
Dato che il tempo richiesto dall'algoritmo dipende quasi esclusivamente dal tempo totale del ***ciclo di solve***. Il centro dell'attenzione in questa analisi viene posto proprio sul tempo medio richiesto per un singolo singolo ciclo.

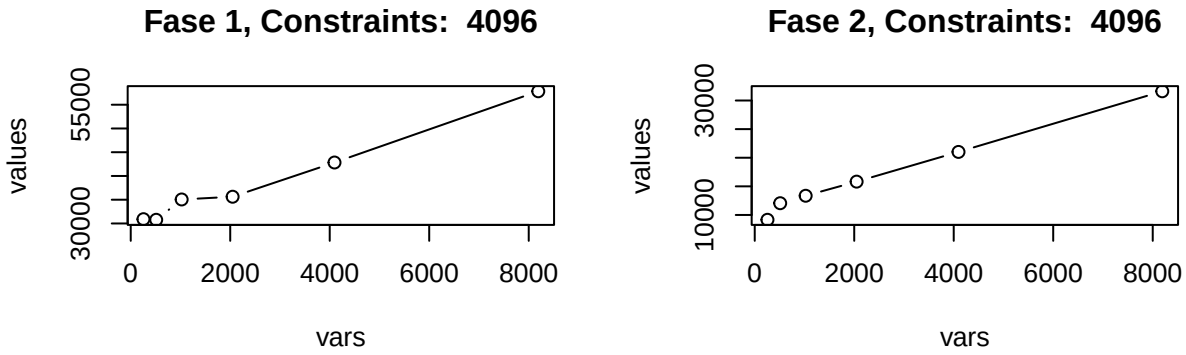
La durata del ciclo dipenderà chiaramente dalla dimensione del problema, che a sua volta dipende dal numero di variabili e dal numero di vincoli.

Il tempo medio richiesto da un singolo ciclo di solve non cresce nello stesso modo all'aumentare delle due dimensioni, come possibile notare dai seguenti grafici tridimensionali:



La differenza nella variazione risulta essere più chiara se si fissa una delle due dimensioni e si analizza il tempo medio per ***ciclo di solve*** al variare dell'altra.

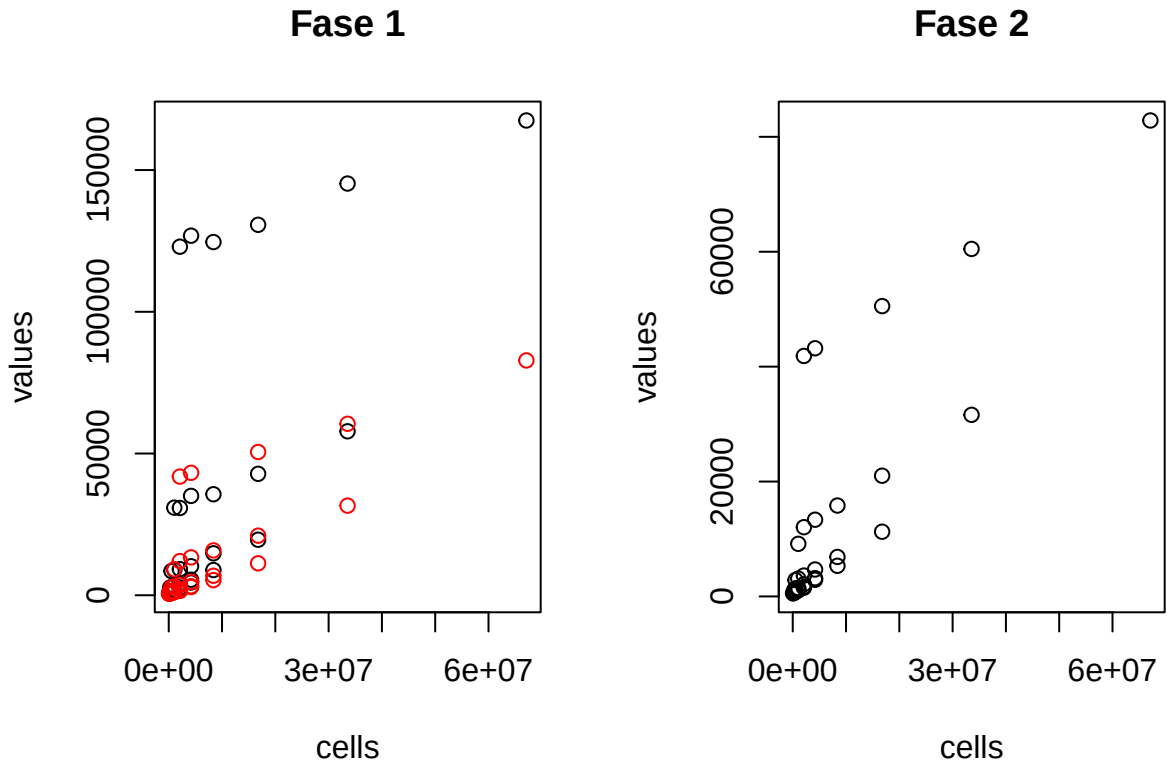




La spiegazione di questa caratteristica è da ricercarsi nel funzionamento pratico dell'algoritmo: durante il ***ciclo di solve*** un passo fondamentale dell'algoritmo è l'aggiornamento del tableu, tale tableu non ha una dimensione pari a $n°variabili \times n°vincoli$, bensì pari a $(n°variabili + 2n°vincoli) \times n°vincoli$ nel caso della fase 1, mentre $(n°variabili + n°vincoli) \times n°vincoli$ nel caso della fase 2.

Si ha di conseguenza un incremento quadratico nel numero di aggiornamenti da fare all'aumentare dei vincoli ma solo lineare all'aumentare delle variabili.

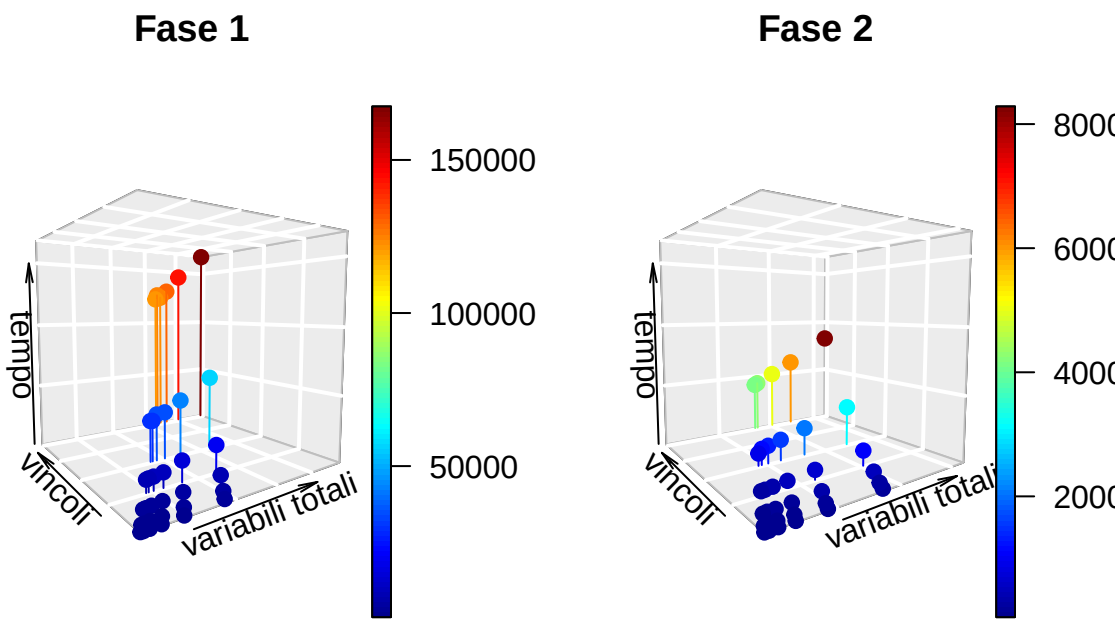
Un ulteriore grafico interessante che mette ancora di più in evidenza questa caratteristica è quello che mette in relazione il tempo medio per ciclo di solve alla dimensione del problema, quest'ultima calcolata come $n°variabili \times n°vincoli$.



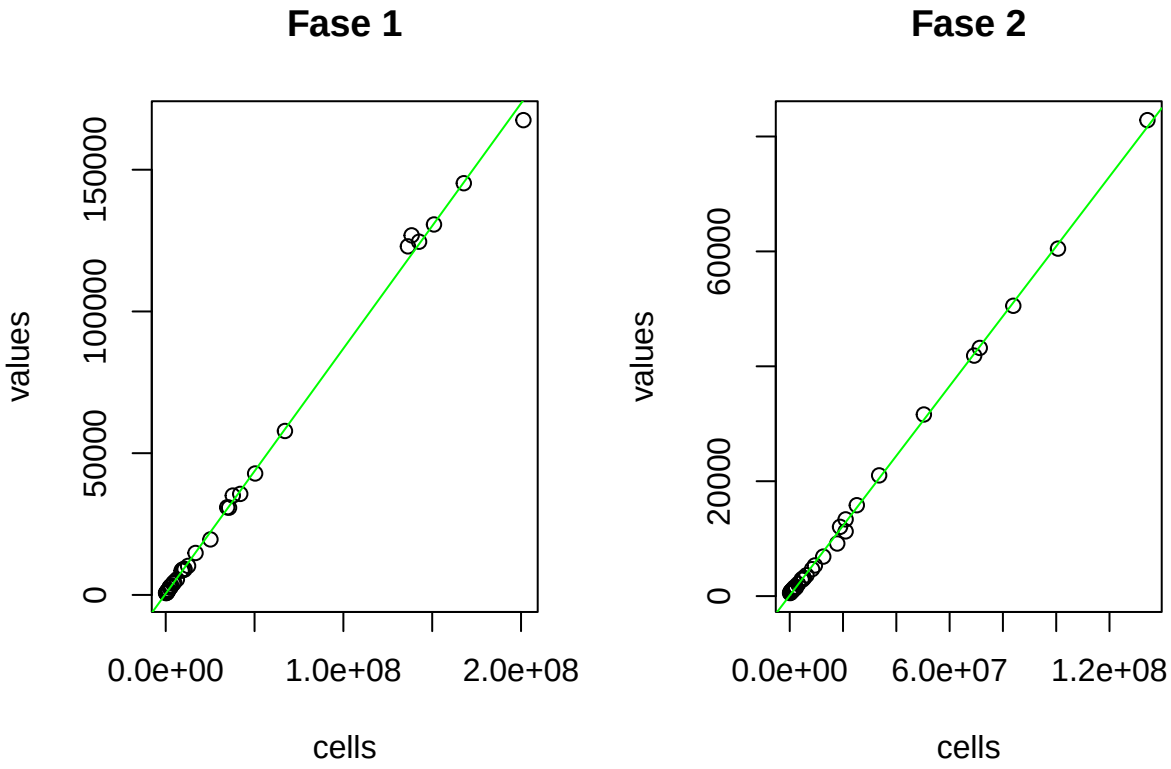
Dai grafici è possibile notare come a stessa dimensione corrispondono tempi medi molto diversi tra di loro, confermando di fatto che il tempo necessario per un singolo ciclo di solve scala diversamente per le due dimensioni, dato che se così non fosse non si dovrebbero notare eccessive differenze di tempo medio a parità di dimensione del problema.

Analisi in funzione della dimensione reale

Partendo dalle considerazioni effettuate precedentemente è possibile procedere ad analizzare il tempo medio per ciclo di solve in base alla dimensione del tableu utilizzato per la risoluzione.



Se si mette in relazione il tempo medio per ***ciclo di solve*** con il numero di celle totali si ottengono i seguenti grafici



Si nota come il tempo medio necessario per ciclo di solve sembra cresce linearmente all'aumentare della dimensione effettiva del tableu.

Confronti tra diverse gpu

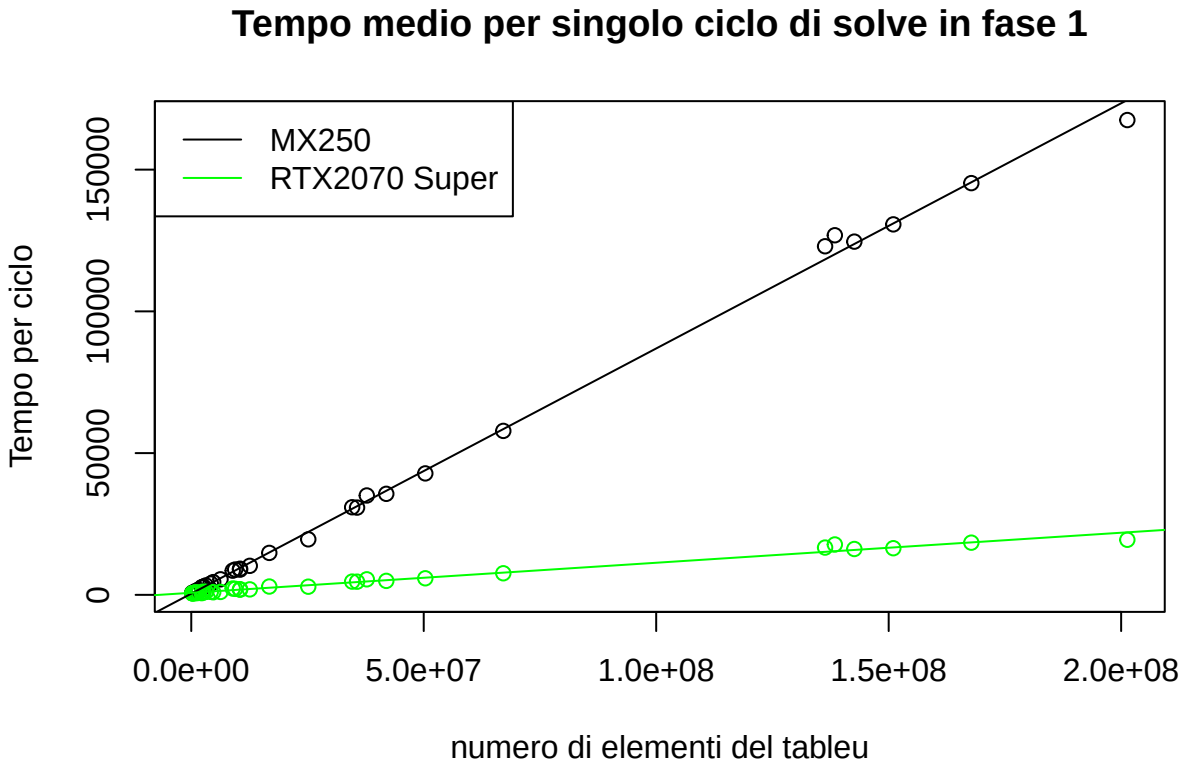
Questa sezione si analizzerà come l'algoritmo scala utilizzando 2 gpu con diverse con diverse caratteristiche e performance:

- Geforce MX250
- Geforce RTX 2070 Super

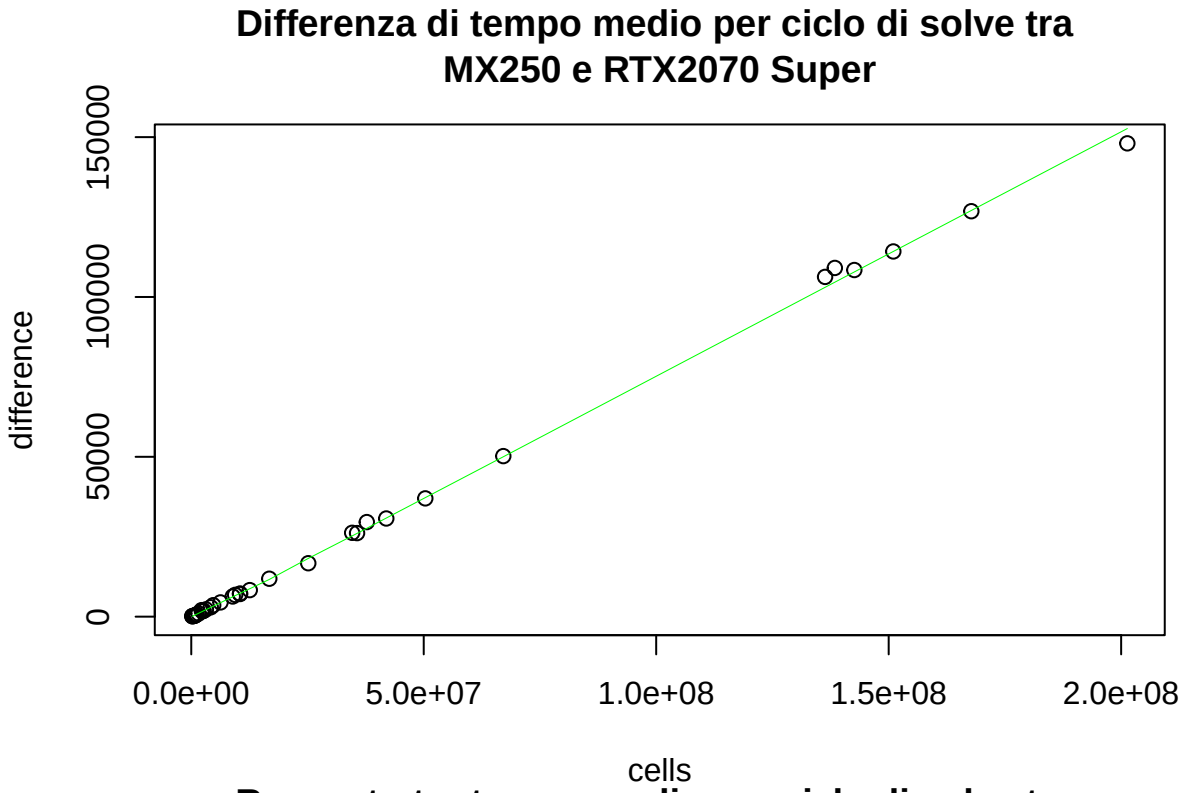
Confronto risultati

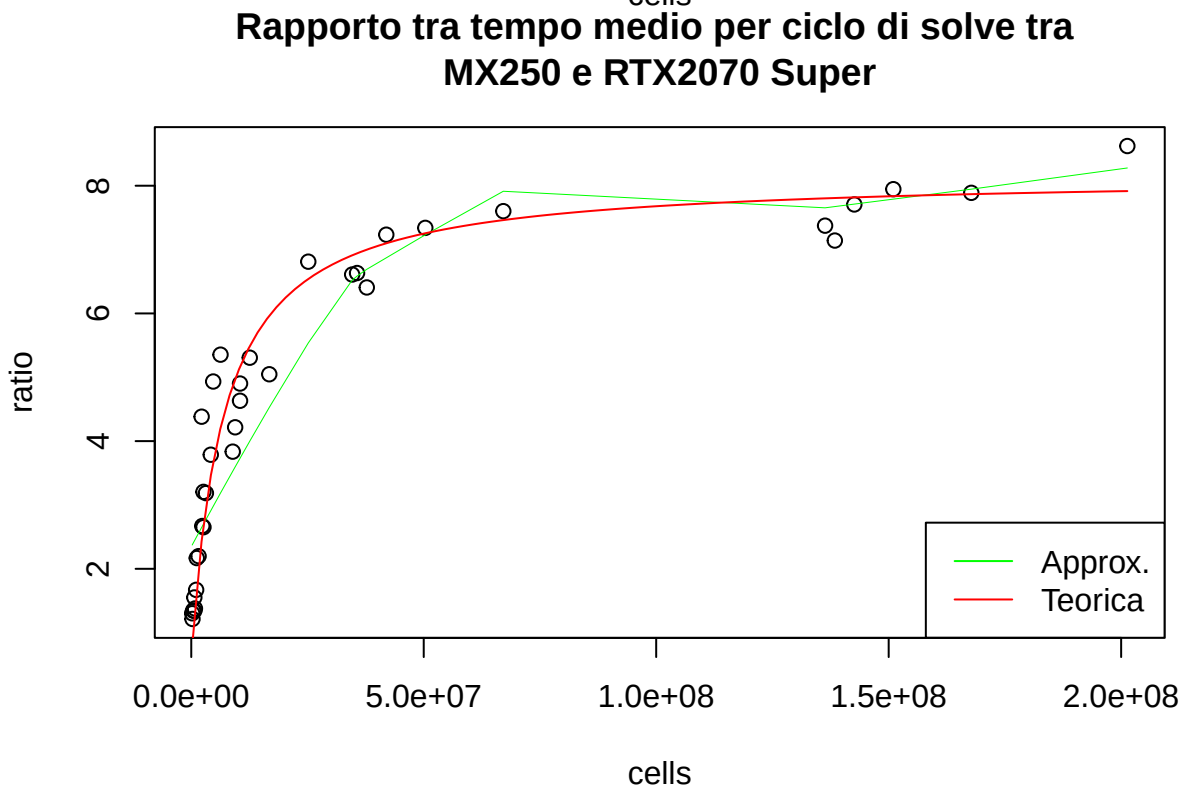
Dato che il tempo medio per ciclo di solve dipende principalmente dalla dimensione reale del tableu i confronti tra schede diverse verranno effettuati in base a tale parametro.

Visto inoltre che questa varia in base alla fase in cui ci si trova solo una delle due verrà presa in considerazione, in particolare si è scelta la fase 1 per via del maggiore numero di cicli eseguiti che permettono di ottenere dei tempi medi più attendibili.



Dal grafico è chiaro come il tempo necessario per ***ciclo di solve*** cresca diversamente per le due schede. Si può anche notare come la differenza di tempo speso per ***ciclo di solve*** aumenti all'aumentare della dimensione reale del problema.



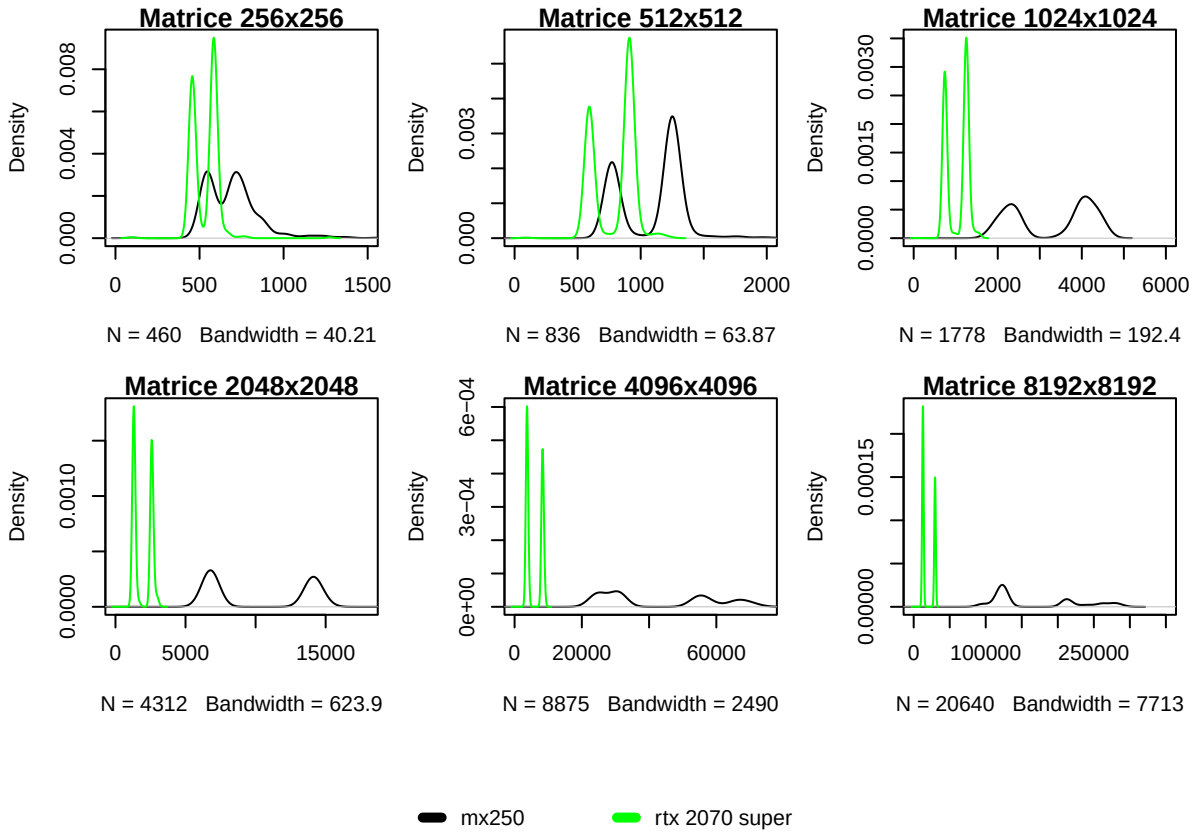


Dal primo grafico è possibile notare come la differenza di tempo medio tra la MX250 e la RTX2070 Super cresca limearmente all'aumentare della dimensione effettiva del problema, ovvero all'aumentare della dimensione di celle effettive del tableu, passando da 0.16ms nel caso di problemi di dimensione 256x256 a 148.06ms per quelli di dimensione 8192x8192.

Si può notare dal secondo grafico come il rapporto tra tempo medio tra le due schede tende ad aumentare seguendo una andamento iperbolico (in rosso l'iperbole ottenuta rapportando la retta di regressione per i tempi delle due schede video), per problemi piccoli si ha infatti che il rapporto tra il tempo della rtx 2070 e della mx250 risulta essere molto basso, ovvero nel caso del problema di dimensione 256x256 di 1.3 (basso perchè sembrerebbe che la 2070 super sia solo 1.3x migliore della mx 250), cresce però molto velocemente per poi stabilizzarsi intorno a 8.62 per problemi più grandi

Si può concludere che la nostra soluzione permetta di sfruttare al meglio le schede con caratteristiche hardware migliori solo per problemi sufficientemente grandi, per problemi piccoli invece i dati mostrano un miglioramento non eccessivamente elevato.

Tempo medio solve a confronto



Attraverso questi grafici si può notare come il tempo per la il ***ciclo di solve*** sia distribuito per le due schede video, si nota ovviamente una distribuzione di tempi inferiori per la 2070 super rispetto alla mx250.

Si può notare anche un comportamento anomalo nella distribuzione dei tempi, infatti invece che avere una distribuzione gaussiana attorno ad un unico valore medio si ottengono due picchi, questo avviene indipendentemente dalla dimensione del problema utilizzato e dalla scheda utilizzata.

La causa sembra essere dovuta ad una variazione del tempo di esecuzione del kernel di aggiornamento del tableu ma il motivo di tale variazione non è stato possibile da individuare.

Conclusioni