Decomposizione LU

Maxim Nitsenko 132354

June 20, 2019

1 Descrizione dell'Algoritmo

1.1 Scopo

Lo scopo dell'algoritmo é fare la decomposizione LU di una matrice. La decomposizione LU é un metodo che a partire da una matrice $A \in \mathbb{R}^{n \times n}$ genera due matrici $L, U \in \mathbb{R}^{n \times n}$, con L triangolare inferiore a elementi diagonali unitari e U triangolare superiore, tali che $A = LU$.

1.2 Complessità

Il costo computazionale dell'algoritmo seriale (vedi 2.2) é $O(n^3/3)$

2 Implementazione Seriale

2.1 Verifica correttezza

La verifica del risultato avviene nel seguente modo: sia A una matrice da fattorizzare, la funzione ritorna una matrice contenente L nel triangolo inferiore e U in quello superiore. Vengono estratte tali matrici e allocate autonomamente. Sia A' la matrice ricostruita moltiplicando tra loro L e U. Il risultato é corretto se $A = A'$. L'uguaglianza è verificata se l'errore massimo assoluto calcolato tra elementi corrispondenti é minore di un certo valore di tolleranza ε.

2.2 Intensità computazionale

Considerando lo pseudo-codice della versione sequenziale:

```
1 FOR i = 0,...,n-2
2   FOR j = i+1,...,n-1
3     m = a(j,i)/a(i,i)
4     FOR k = i+1,...,n-1
5       a(j,k) -= m * a(i,k)
6     END FOR
7     a(j,i) = m
8   END FOR
9 END FOR
```

il calcolo dei FLOP avviene osservando che la riga 3 contiene 1 FLOP e che
se $i = 0$, la riga 3 viene eseguita $n - 1$ volte
se $i = 1$, la riga 3 viene eseguita $n - 2$ volte
...
se $i = n - 2$, la riga 3 viene eseguita 1 volta

Il numero totale dei FLOP alla riga 3 e'

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{1}{2}(n-1)n$$

Un conto simile viene fatto per la riga 5, la quale contiene 2 FLOP:
se $i = 0$ la riga 5 viene eseguita $n - 1$ nel ciclo j e $n - 1$ nel ciclo k per un totale di $2(n-1)(n-1)$ FLOP
se $i = 1$ alla riga 5 si eseguono $2(n-2)(n-2)$ FLOP
...
se $i = n - 2$ alla riga 5 si esegue 1 FLOP

Il numero totale dei FLOP alla riga 5 e'

$$\sum_{i=0}^{n-2} 2(n-1-i)(n-1-i) = \frac{1}{3}(n-1)n(2n-1)$$

Sommando i due risultati otteniamo che il numero totale di FLOP é

$$FLOP(n) = \frac{2}{3}n^3 - \frac{n^2}{2} - \frac{n}{6}$$

Un conto molto simile avviene per le read/write dove la riga 3 contiene 2

read, la riga 5 ha 2 read e 1 write, infine la riga 7 ha una write. Il numero totale di read/write é

$$BYTE(n) = 3\sum_{i=0}^{n-2} i + 3\sum_{i=0}^{n-2} i^2 = n^3 - n$$

Utilizzando la singola precisione vengono scritti/letti $4(n^3 - n)$ byte.

L'intensità computazionale e':

$$\lim_{n\to\infty} \frac{FLOP(n)}{BYTE(n)} = \frac{2}{3} = 0.66$$

2.3 Compute o memory bound?

Il codice esegue 6 accessi a memoria principale per ogni FLOP. Affinché il codice sfrutti a pieno le potenzialità dell'architettura sulla quale è eseguito, quest'ultima deve avere caratteristiche simili. Il rapporto FLOP a Byte di una CPU Intel Xeon E5-2630v3 è 10.17 contro 0.66 del codice seriale, questo significa che il codice è memory bound. Considerando l'intensità computazionale di 0.66, si possono calcolare i FLOPS massimi raggiungibili nel seguente modo:

$$59GB/s = 14.75 * 10^9 FL/s$$

$$14.75 * 0.66\, GFLOPS = 9.73\, GFLOPS$$

2.4 Performance seriale

La Figura 1 mostra il tempo esecuzione al crescere della dimensione.
Nella Figura 2 si vedono i FLOPS e bandwidth al variare di n.

3 Implementazione Parallela

3.1 Linguaggi di programmazine

Nel progetto tutte le comunicazioni sono effettuate tramite la libreria Open-MPI, una particolare implementazione di MPI. MPI (Message Passing Interface) si occupa del modello di programmazione basato su scambio di messaggi per architetture parallele, nel quale i dati sono mossi dallo spazio di indirizzamento di un processo a quello di un processo di verso, attraverso operazioni cooperative svolte da ciascun processo.

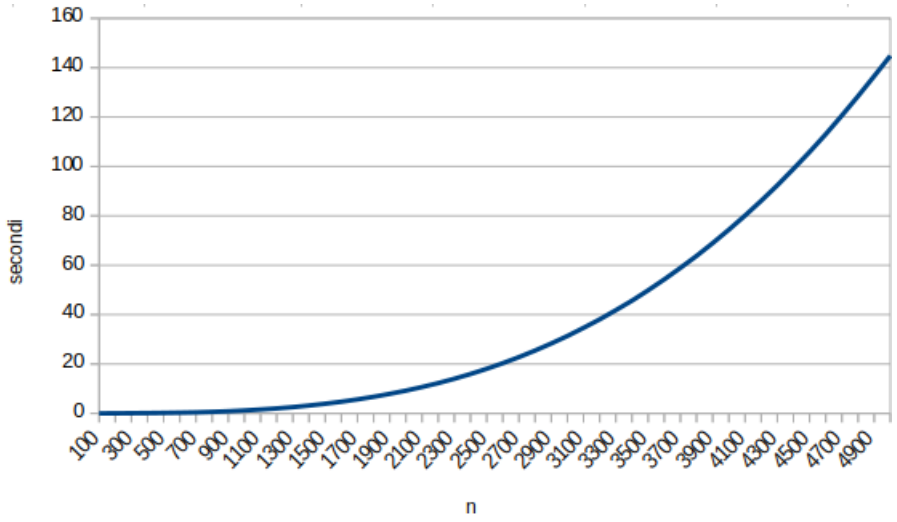

Figure 1

La computazione effettiva è stata eseguita tramite CUDA, un software-layer che permette accesso diretto a risorse hardware di GPU NVIDIA per esecuzione di kernel.

3.2 Algoritmo

Per bilanciare il carico di lavoro, é utile utilizzare una distribuzione ciclica delle righe, ovvero il processore P_j possiede le righe $\{a_i : i = j \, mod \, p\}$, dove p è il numero di processori.
Con questa distribuzione, l'algoritmo di decomposizione LU parallelo si puó schematizzare nel seguente modo:

1. Il processore P_0 comunica agli altri processori le righe della matrice da fattorizzare secondo la distribuzione ciclica. La riga i sará inviata al processore $i \, mod \, p$.

2. Consideriamo la riga i: il processore $i \, mod \, p$, ovvero il processore che detiene la riga i-esima, comunica con una broadcast la riga a tutti i processori.

3. La riga i ricevuta da P_0 al passo 2 corrisponde alla riga i della matrice finale e di conseguenza viene salvata opportunamente.

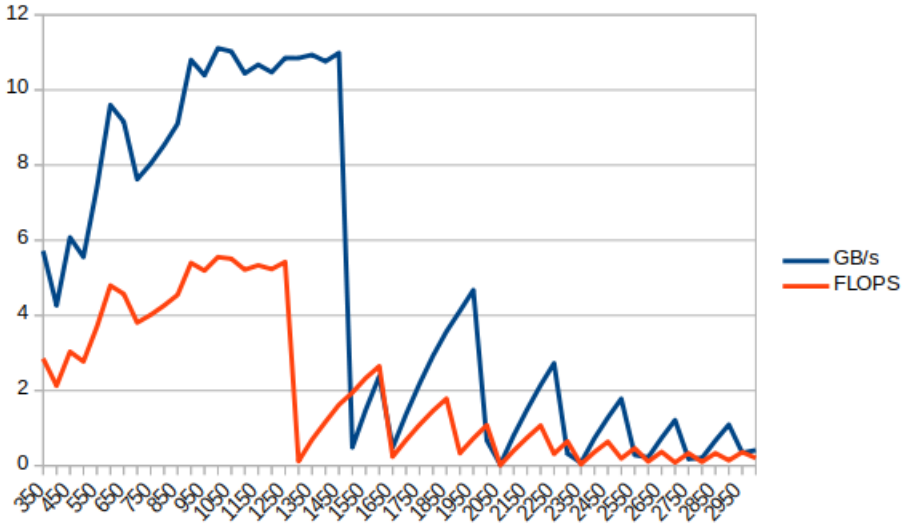

Figure 2

4. Sia $R_q = \{q + kp : k \in \mathbb{N},\ q + kp < n\}$ l'insieme delle righe possedute dal processore q-esimo. Per ciascuna riga in R_q, con indice maggiore di i, annulla l'i-esimo elemento, sfruttando la riga ricevuta al passo 2.

5. Ripetere dal passo 2, per $i = 0, ..., n - 2$.

3.3 Misurazione dei Tempi

Il tempo totale impiegato dall'algoritmo T_{tot} è composto da due parti T_c e T_e tale che $T_{tot} = T_e + T_c$.
T_c si riferisce al tempo occupato dalle direttive MPI di comunicazione/sincronizzazione e T_e al tempo di esecuzione sulla GPU.
Per calcolare T_c si è usata la funzione *MPI_Wtime()* prima e dopo ogni funzione MPI e per trovare T_e si sono usati gli eventi CUDA insieme alla funzione *cudaEventElapsedTime()*.

3.4 Ottimizzazioni

Per ridurre al minimo i trasferimenti tra CPU e GPU si fa uso di code memorizzate direttamente su GPU e di una versione di MPI CUDA-aware che permette diretto trasferimento tra due GPU.

Le funzioni *enqueue()* e *dequeue()* ritornano il puntatore dell'ultima riga

e della prima riga nella coda. In questo modo si lavora direttamente sulla memoria evitando numerose memcpy.

Al passo 3, la copia da parte di P_0 della riga ricevuta nella matrice finale viene fatta in modo asincrono su uno stream dedicato.

Al passo 4 viene invocato un kernel CUDA. Si è tentato con diversi approcci, come assegnare un blocco per riga oppure assegnare un thread per riga. Il risultato migliore si è ottenuto dedicando un thread per ogni elemento.

Nella versione attuale, al passo 4 viene lanciato un kernel ed immediatamente il controllo ritorna alla CPU, che rimane in attesa sul passo 2, fino al completamento del passo 4. Dal momento che il passo 2 ha bisogno solo della riga i-esima, cioè la prima riga del processore $i \, mod \, p$, al passo 4 si potrebbe tentare di invocare un ulteriore kernel che azzera l'i-esima riga, sicché il passo 2 possa essere eseguito senza attesa.

3.5 Sincronizzazioni

Non sono necessarie sincronizzazioni esplicite (*MPI_Barrier()*), in quanto le sincronizzazioni avvengono in maniera implicita tramite le chiamate bloccanti a *MPI_Bcast()* e *MPI_Recieve()*.

4 Performance parallela

Di seguito sono riportati i fondamentali grafici in riguardo alla performance parallela.
Figura 3: Strong Scaling
Figura 4: Weak Scaling
Figura 5: Speedup relativo
Figura 6: Efficienza
Figura 7: Funzione di Kuck
La Figura 8 mostra il tempo di esecuzione su una GPU al variare della dimensione del blocco. La dimensione del blocco con prestazioni migliori è 128.

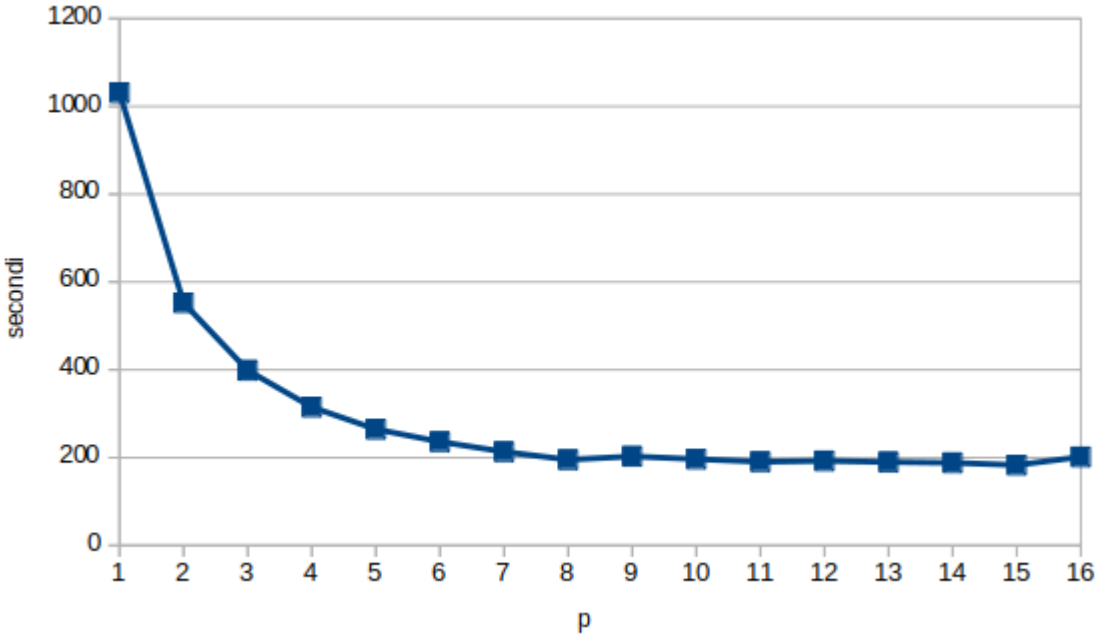

Figure 3: Strong scaling con $n = 30000$

5 Risultati e Conclusioni

Dal grafico dello speedup si conclude che il numero di GPU che conduce a risultati migliori è 15. Se il numero di GPU è una risorsa limitata e preziosa, in tal caso è più adeguato usare 8 GPU, ed ottenere così un buon compromesso tra efficienza e speedup. Come ci si doveva aspettare, lo speedup ha raggiunto un plateau a causa delle eccessive comunicazioni e sincronizzazioni tra elementi. La Figura 9 ci rivela che già con 13 GPU $T_c > T_e$.
Nel complesso il lavoro di parallelizzazione ha avuto successo. Usando una GPU lo speedup rispetto alla versione seriale è stato 32.45.

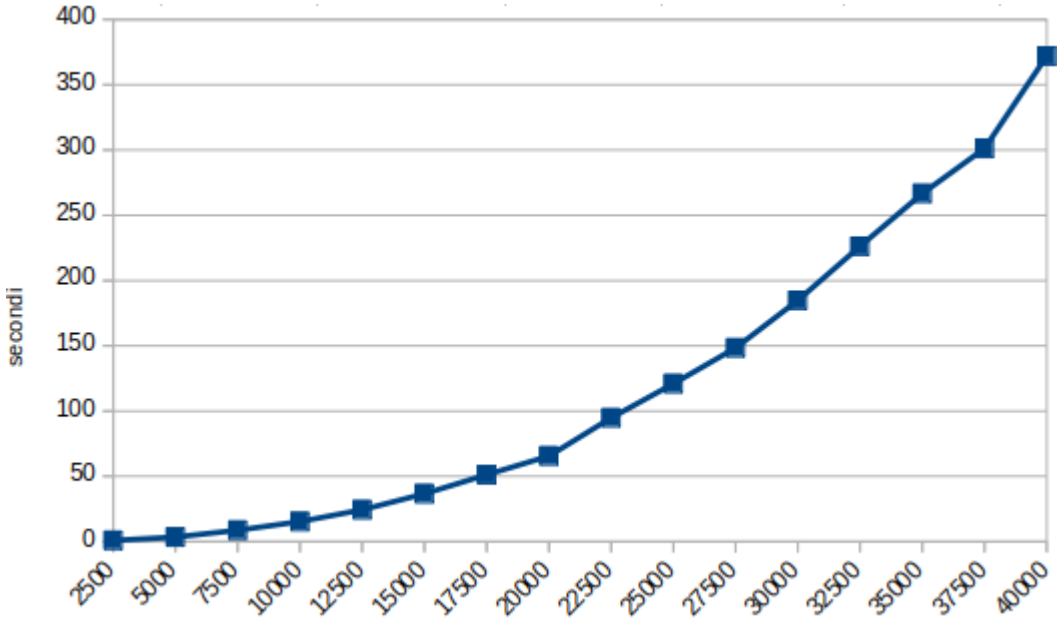

Figure 4: Weak scaling aumentando n di 2500 per ogni GPU

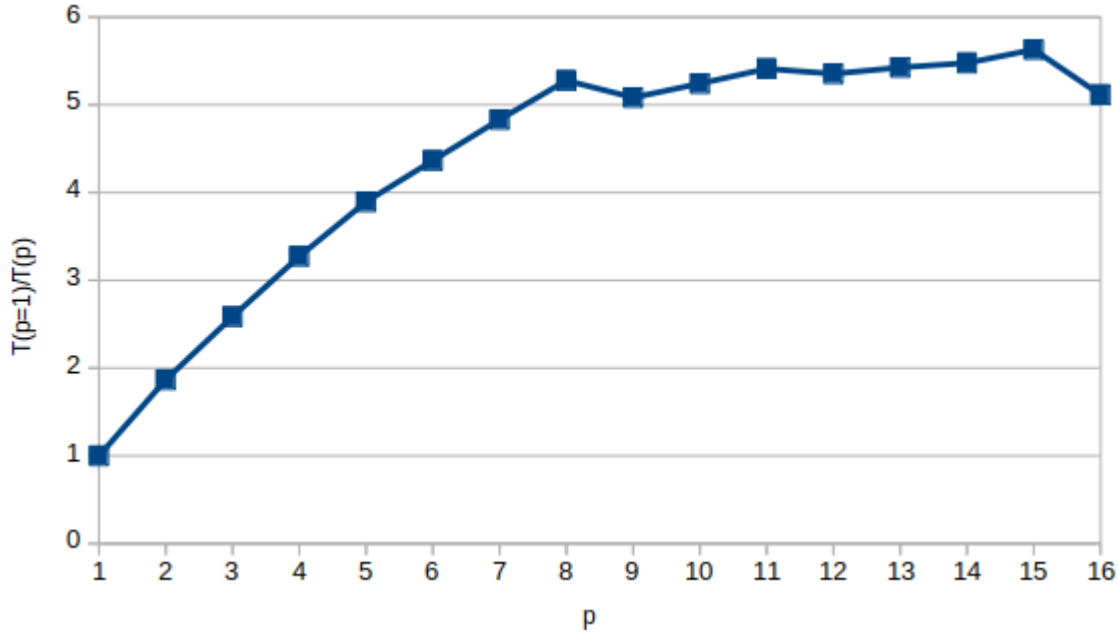

Figure 5: Speedup relativo

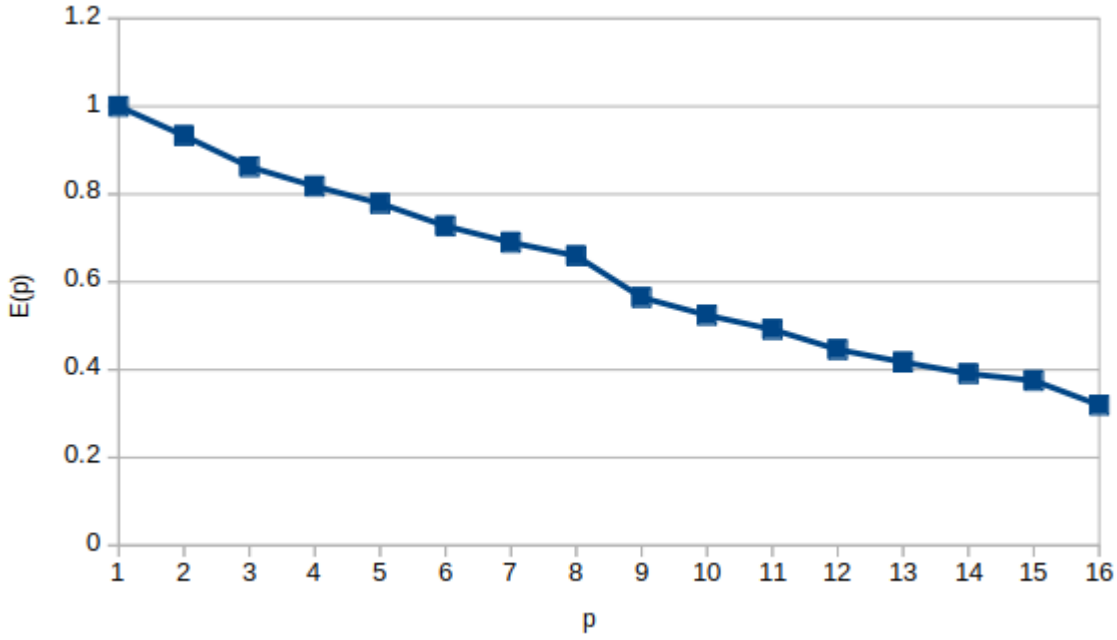

Figure 6: Efficienza

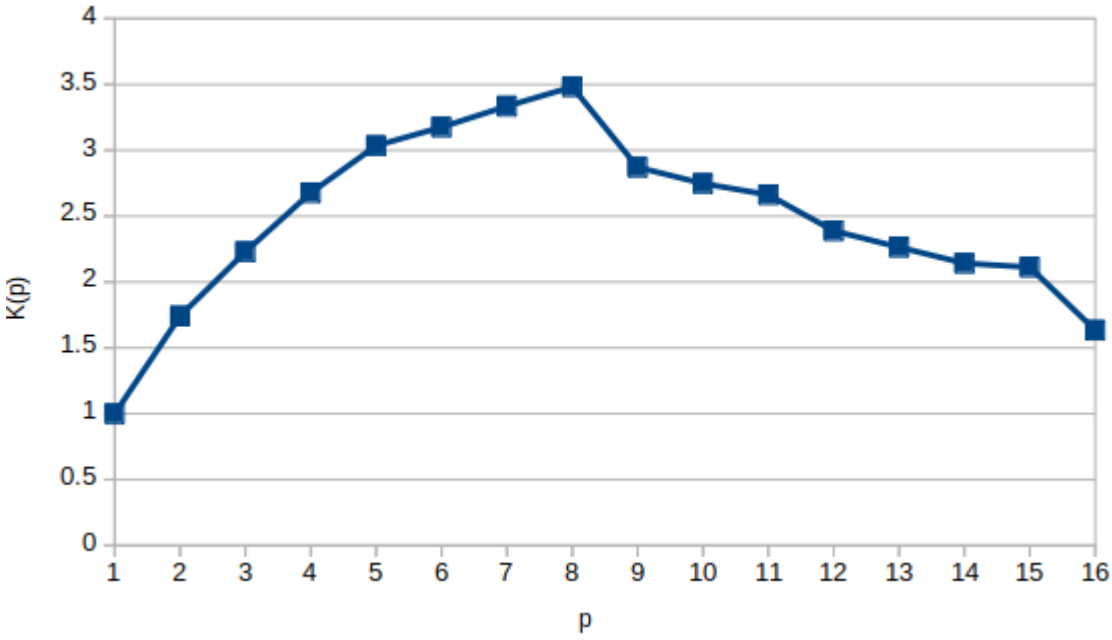

Figure 7: Funzione di Kuck

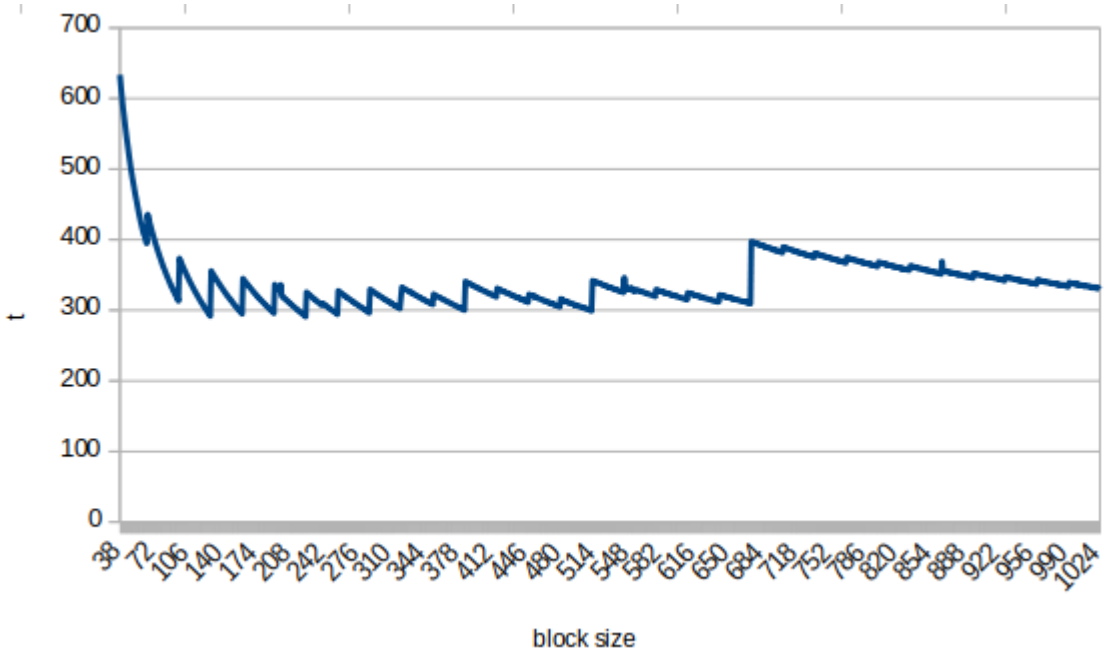

Figure 8: Tempo al crescere della dimensione del blocco. Esperimento effettuato con $n = 2000$

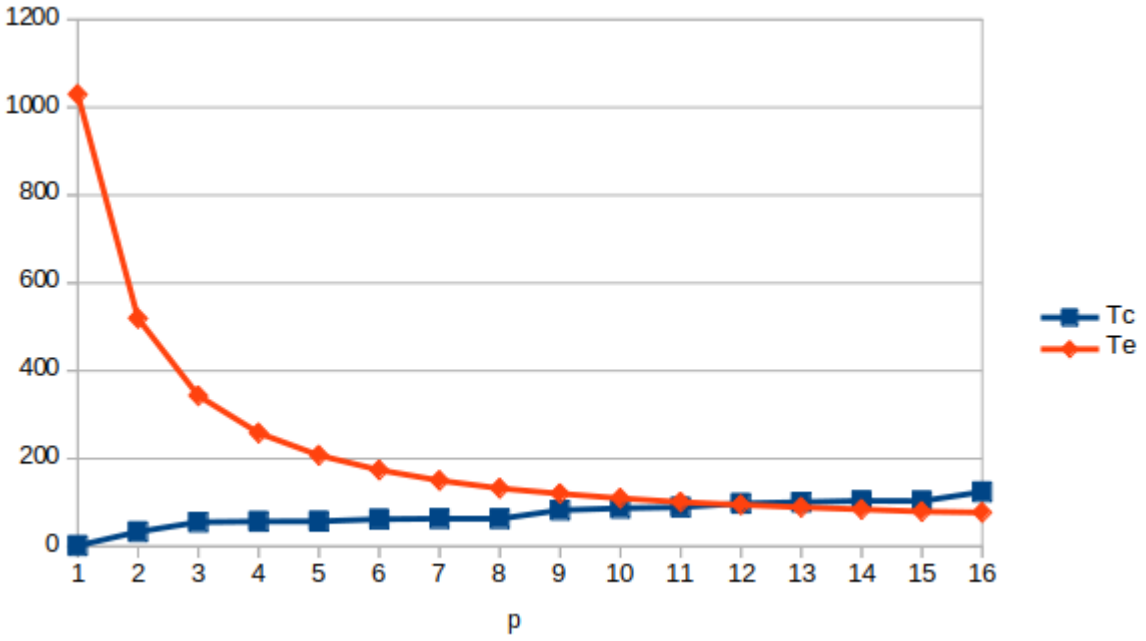

Figure 9: Tempi di sincronizzazione/comunicazione e tempi di esecuzione al variare del numero di GPU