# Implementazione di un modulo per il calcolo del Determinante in una Libreria Ibrida Cpu-Gpu

Antonio Michele Miti

July 4, 2014

## 1 Introduzione

L'obiettivo di questo progetto è di dare le fondamenta di una libreria ibrida CUDA / C++ che permetta di risolvere in modo numerico alcuni problemi di algebra lineare sfruttando le peculiarità del calcolo parallelo su Gpu.

La libreria non è pensata per essere ottimizzata su classi di matrici particolari ma per lavorare su matrici ragionevolmente generali, nello specifico matrici dense, grandi e con valori uniformemente distribuiti tra un valore massimo e un minimo.

L'aspetto fontamentale di trasversalità host-device della classe è rappresentato dalla doppia copia dei valori della matrice, ovvero quando si definisce un oggetto di tipo *matrice* vengono automaticamente allocati 2 copie dell'array degli elementi, uno risiede nella memoria global della Gpu e l'altro nella memoria ram dell'host. In questo modo è possibile garantire una certa libertà di scelta su quale device far eseguire specifiche parti del calcolo.

Fra l'ampio ambito degli algoritmi numerici per la soluzione di problemi di algebra lineare ne è stato in particolare esaminato uno per mostrare le potenzialità dell'utilizzo di una Gpu per il calcolo numerico. Il progetto si è focalizzato quindi sul problema del calcolo del determinante per matrici grandi sfruttando un algoritmo di condensazione. Il resto dell'articolo è dedicato ad esporre gli algoritmi utilizzati per il calcolo determinante e la loro implementazione nella libreria ibrida host-device . L'algoritmo utilizzato è una forma modificata di quello presentato da Moreno e Haque in un loro articolo [3], riadattato in modo da essere ottimizzato per la classe generale di matrici su cui si intende lavorare.

## 2 Gli Algoritmi di Condensazione

Sono una classe di algoritmi nata specificatamente per il calcolo del determinante di matrici quadrate. L'idea chiave è la seguente: si identifica una trasformazione

$$\mathrm{T} : A \in \ \mathrm{Mat}(n,n) \longmapsto B \in \ \mathrm{Mat}(n-1,n-1)$$

tale per cui esista uno scalare $\alpha_A$, in generale dipendente dalla matrice di partenza, tale che:

$$\det(A) = \alpha_A \cdot \det(B)$$

Ammettendo che l'operatore T sia applicabile $n-1$ volte sulla matrice quadrata $A$ di ordine $n$ e definendo:

$$A_0 = A \qquad \mathrm{T}(A_k) = A_{k+1} \qquad \det\big(A_k\big) = \alpha_k \det(A_{k+1})$$

Si conclude banalmente che

$$\det\big(A\big) = \prod_{k=0}^{n-1} \alpha_k \tag{1}$$

### 2.1 La formula di Dodgson

Il primo algoritmo di questo tipo è stato introdotto da C. L. Dodgson [1], noto come Lewis Carroll, nel 1866. Gli elementi della matrice trasformata $B = \mathrm{T}_{\mathrm{D}}(A)$ sono definiti come segue:

$$b_{i,j} = \begin{vmatrix} a_{0,0} & a_{0,j+1} \\ a_{i+1,0} & a_{i+1,j+1} \end{vmatrix} \tag{2}$$

Praticamente tale metodo rimpiazza gli elementi della sottomatrice ottenuta privando $A$ della prima riga e della prima colonna con con il determinante della matrice 2x2 costituita dal primo elemento in alto a sinistra $a_{00}$ ,il primo valore nella colonna dell'elemento, il primo valore nella riga dell'elemento e il valore dell'elemento stesso da sostituire. Per esempio la riduzione di una matrice 3x3 risulterebbe:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \mapsto \begin{bmatrix} * & * & * \\ * & (a_{00}a_{11} - a_{01}a_{10}) & (a_{00}a_{12} - a_{02}a_{10}) \\ * & (a_{00}a_{21} - a_{01}a_{20}) & (a_{00}a_{22} - a_{02}a_{20}) \end{bmatrix} = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = B$$

Si dimostra per induzione che per questa trasformazione vale la seguente equazione di condensazione del determinante:

$$\det(A) = \frac{\det(B)}{a_{00}^{n-2}} \tag{3}$$

Prima di indagare ulteriormente sull'efficienza di questo metodo e' necessario osservare che questo algoritmo soffre di un importante limitazione algebrica: se il primo elemento della matrice e' nullo la condensazione non puo' avvenire.

E' evidente che nel procedimento di Dogdson l'elemento $a_{00}$ riveste il ruolo di "Pivot" (nel senso di elemento cardine per il passo di condensazione) pertanto e' necessario introdurre qualche forma di prescrizione (o "pivoting") per scegliere un diverso elemento su cui basare la condensazione.

## 2.2 La formula di Salem-Said

Rappresenta il più' naturale miglioramento del metodo precedente: come pivot non viene preso il primo elemento della prima riga ma il primo elemento non nullo. In questo modo la successione degli step di condensazione puo' proseguire senza intoppi a meno che la prima riga della matrice non sia completamente nulla, quindi singolare.
Sia $a_{0l}$ il pivot, in questo caso la matrice $\mathrm{T}_{\mathrm{ss}}(B) = (b_{ij})$ prodotta dallo step di condensazione risulterebbe:

$$b_{i,j} = \begin{cases} \begin{vmatrix} a_{0,l} & a_{0,j+1} \\ a_{i+1,l} & a_{i+1,j+1} \end{vmatrix} & \text{se } j \geq l \\ \\ \begin{vmatrix} a_{0,j} & a_{0,l} \\ a_{i+1,j} & a_{i+1,l} \end{vmatrix} = -a_{0,l}a_{i+1,j}, & \text{se } j < l \end{cases} \tag{4}$$

Nel loro articolo [2] gli autori hanno dimostrato che in questo caso vale un'equazione di condensazione simile a quella precedente:

$$\det(A) = \frac{\det(B)}{a_{0,l}^{n-2}} \tag{5}$$

In sostanza il calcolo del determinante con il metodo di Salem e Said si riduce al calcolo del prodotto di potenze degli inversi dei pivot.

### 2.2.1 Complessità Algebrica

Il numero di operazioni aritmetiche necessarie per completare l'esecuzione dell'algoritmo precedente e facilmente stimabile:

- *Pivoting*: per ogni step di condensazione richiede al più $n$ letture per trovare il primo elemento non nullo nella prima linea
- *Singolo step di Condensazione*: per calcolare ogni elemento della matrice $B$ è necessario il calcolo di 2 prodotti e di 1 sottrazione. Quindi sono richieste $3(n-1)^2$ operazioni.
- *Condensazione*: per calcolare il determinante sono necessari $(n-2)$ step di pivoting e condensazione.

In conclusione il costo complessivo per calcolare il determinante con un algoritmo di condensazione risulta dell'ordine di $O(n^3)$. Tali algoritmi risultano molto più efficienti rispetto agli algoritmi di Laplace e Liebniz, basati sulla definizione esplicita di determinante, che richiedono un numero di operazioni dell'ordine di $O(n!)$ .

### 2.2.2 Stabilità numerica

L'implementazione computazionale di un algoritmo di questo tipo presenta comunque un'importante limitazione di stabilità numerica, più precisamente dopo pochi passi di condensazioni c'è un elevato rischio di underflow o overflow, ovvero il tipo di struttura dati scelta per immagazzinare in memoria il valore degli elementi della matrice condensata potrebbe avere delle limitazioni troppo restrittive riguardo al più piccolo e più grande numero rappresentabile . Il problema può essere evidenziato come segue: sia $S =< A >$ la "dimensione caratteristica" o "taglia" della matrice $A$ da condensare, non esiste un modo formale per definire questo valore ma lo si può ad esempio prendere come la media dei valori assoluti di tutti gli elementi della matrice. E' evidente che dopo uno step di condensazione la taglia della matrice $B$ risulterà essere:

$$< B >=< (a_* a_* - a_* a_*) > \simeq S^2 \tag{6}$$

Pertanto dopo $n$ condensazioni la taglia crescerà ( o convergerà verso lo 0 in caso di $S < 1$) molto rapidamente, tale andamento si può stimare essere:

$$< B_n >= \big(S\big)^{2n} \tag{7}$$

La cosa in realtà non sorprende, considerando la definizione di determinante (come sommatoria di numerose permutazioni) è chiaro che l'ordine di grandezza di $\det(A)$ cresca come la potenza n-sima della taglia. In questo caso però Il problema si palesa molto prima di essersi avvicinati alla valore del determinante, il rischio di under/over - flow è presente già nei primi dieci passi di condensazione.

## 2.3 Una possibile modifica all'algoritmo di Salem-Said

Considerando la specifica classe di matrici su cui si vuole implementare questo algoritmo (grandi, dense, con valori degli elementi distribuiti uniformemente tra un valore minimo e un massimo) l'algoritmo di Salem-Said non è direttamente applicabile per via del problema di stabilità numerica evidenziato in precedenza.

Per poter proporre un eventuale modifica degli algoritmi precedenti è necessario evidenziare le operazioni che vengono ripetute per ognuno degli $n-1$ step che costituiscono un ciclo di condensazione:

1. Viene scorsa la prima riga , o al limite l'intera matrice, per determinare l'elemento di Pivot.
2. Viene effettuata una Trasformazione della matrice $A \mapsto A'$, in una seconda matrice dello stesso ordine ma in qualche modo più efficiente da condensare.
3. Viene Condensata la matrice trasformata $A'$.

Un ciclo di questo tipo completo fornisce una lista di $n$ valori, ottenuti a partire dai $n-1$ pivot più il singolo elemento che si ottiene dopo tutte le condensazioni, il cui prodotto coincide con il determinante.

Per ovviare al problema dell'overflow, o meglio per tenerlo sotto controllo limitando questo rischio solo al calcolo della produttoria finale, viene proposta una modifica ai primi 2 punti del ciclo.

1. **Pivoting**: viene scelto come pivot l'elemento in modulo più grande dell'ultima riga della matrice e viene salvato il valore $p$ del pivot.
2. **Trasformazione**: viene divisa l'ultima riga per il valore di pivot e ,se necessario, viene scambiata la colonna in cui si trova l'elemento di pivot con l'ultima colonna della matrice e cambiato il segno del valore $p$ salvato in modo da tener conto che la matrice iniziale è stata moltiplicata per una di permutazione.
3. **Condensazione**: siccome a questo punto l'elemento di pivot della matrice da condensare si trova nell'angolo in basso a destra, la formula di condensazione assume un'espressione particolarmente semplice:

$$b_{i,j} = \begin{vmatrix} a'_{i,j} & a'_{i,n-1} \\ a'_{n-1,j} & a'_{n-1,n-1} \end{vmatrix} \qquad \forall i,j \in [0, n-2] \tag{8}$$

dove $a'_{n-1,n-1}$ coincide con il valore dell'elemento di pivot trovato nel punto 1).

Alla fine di questo processo il valore del determinante si ottiene dalla produttoria di tutti i pivot trovati con la routine precedente. Questo deriva dalla semplice identità algebrica

$$\det\big(A\big) = \left| \begin{bmatrix} 1 & 0 & \cdots \\ 0 & \ddots & \\ \vdots & & p \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots \\ 0 & \ddots & \\ \vdots & & \frac{1}{p} \end{bmatrix} A \right| = p \det\big(A'\big)$$

e dall'equazione di condensazione di Salem Said 2.2 per cui il determinante della matrice condensata coincide con quello della matrice di partenza nel caso che il pivot sia uguale ad 1 ovvero:

$$\det(A') = \det(B)$$

Il vantaggio principale di questo tipo di pivoting deriva dal fatto che la taglia della matrice condensata rimane dello stesso ordine di grandezza di quello di partenza, infatti considerando l'equazione 8 e che il pivot di $A'$ è a seguito della divisione per il pivot $a'_{n-1,n-1} = 1$ risulta che:

$$< B >=< (a'_* 1 - \alpha a'_*) > \simeq S - \alpha S \simeq S$$

dove $\alpha$ e il rapporto tra un elemento della riga del pivot e il valore del pivot, ovvero $-1 < \alpha < 1$.

Il motivo dello swap di colonne e della scelta dell'ultima riga della matrice come riga di pivot ha invece una motivazione puramente computazionale che verrà discussa nei seguenti paragrafi.

# 3 Implentazione dell' Algoritmo su Cpu

L'implementazione dell' algoritmo precedente come una routine sequenziale sulla cpu è piuttosto immediato. Dopo aver trovato il pivot e aver scambiato la sua colonna con l'ultima, viene eseguito un ciclo for che scorre tutta la matrice da sinistra a destra e dall'alto verso il basso. Questo schema di lettura è suggerito dal criterio che ha la classe *Matrice* di immagazzinare in memoria la matrice come un array 1D.
Ad ogni passo vengono letti i 2 valori uno sulla riga e l'altro sulla colonna del pivot, il valore da modificare viene letto e il risultato viene riscritto immediatamente nella posizione corretta sull'array originale Per esempio per una matrice 3x3 i risultati delle 2 condensazioni successivi verrebbero riscritte nel seguente modo (notare che in memoria la matrice viene immagazzinata come un array):

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \mapsto \begin{bmatrix} b_{00} & b_{01} & b_{10} \\ b_{11} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \mapsto \begin{bmatrix} D & b_{01} & b_{10} \\ b_{11} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

Ad ogni passo del ciclo pivoting - trasformazione - condensazione viene salvato in memoria il pivot.
Come già detto per il calcolo del determinante è necessario calcolare la produttoria di questi valori, per matrici grandi è necessario immagazzinare questi valori in una struct apposta che memorizzi separatamente la mantissa e l'esponente del numero in esame.

L'aver posto il pivot nell'angolo in basso a destra dà la garanzia che ogni volta che si tenterà di aggiornare un sito della matrice al valore condensato tutti i dati necessari siano disponibili e non sovrascritti. Grazie a questo accorgimento è possibile allocare un solo array in memoria e non è necessario disporre di un secondo array su cui salvare il risultato della condensazione. E' da notare che alla fine di questo processo le informazioni sulle componenti della matrice iniziale sono state perse ma è possibile sfruttare la peculiarità della classe ibrida che conserva sulla memoria global della Gpu una copia della matrice in esame.
Il codice sorgente dei metodi che realizzano la condensazione è disponibile nel file `/CudaMatrixClass/Src/Condensation.cuh`.

# 4 Implementazione dell'Algoritmo su Gpu

Come prima, la routine viene sostanzialmente eseguita in 3 passi, ad ognuno di essi corrisponderà uno specifico kernel CUDA che agirà esclusivamente sulla copia Gpu della matrice considerata. Sono da notare alcune differenze:

La prima risiede nello spiccato utilizzo della struttura ibrida della classe. Questo interviene, in primo luogo, nel calcolo della produttoria finale, che viene eseguita esclusivamente dall'host. Il motivo di questa scelta è la superiore efficienza della cpu nel calcolo di questo tipo di operazioni su array non troppo grandi ( il numero di pivot coincide con l'ordine della matrice) e nella maggiore affidabilità nell'utilizzare strutture dati ( Classi c++) personalizzate , in questo caso la classe per i numeri grandi. Affinchè l'host possa eseguire questo calcolo è necessario che tutta la lista dei pivot venga copiata dalla memoria global al host.
Invece di copiare la lista dei pivot in una sola volta alla fine di tutto il calcolo viene passato un singolo valore ogni volta che viene eseguito il pivoting. La presenza sulla memoria host di questo dato permette di trarne vantaggio anche per l'operazione di divisione dell'ultima riga per il pivot. Infatti dividere gli elementi di un array presente sulla memoria

global della scheda per il valore di una specifica entry soffrirebbe di un elevato traffico in lettura, dovuto al fatto che tutti i thread dovrebbero leggere almeno una volta sulla cella di memoria contenente il dividendo. Siccome questa operazione non può essere eseguita in parallello i thread saranno forzati ad agire in sequenza determinando un evidente perdita di parallelismo. Ci sono 2 possibili soluzioni per ovviare a questo problema, il primo è passare il dividendo per valore come argomento del kernel, la seconda è sfruttare la memoria costant fornita da CUDA che assicura una lettura molto veloce da parte dei thread del kernel ma che in scrittura è accessibile solo dall'host.

La seconda è più sostanziale e riguarda il procedimento di scrittura in memoria degli elementi della matrice condensata. In questo caso non è possibile sfruttare il pattern di lettura - scrittura utilizzato per l'algoritmo Cpu perchè il non perfetto parallelismo tra i thread non assicura che al momento di scrivere sulla matrice originale gli elementi da leggere non siano già stati modificati da altri thread, La soluzione più semplice per superare questo problema è quello di allocare sulla memoria global della Gpu un array adatto a contenere i dati della matrice risultato della condensazione. Alla fine dello step sarà sufficiente disallocare la memoria relativa alla matrice vecchia e sostituire il valore del puntatore alla copia Gpu, presente tra gli attributi della classe, con l'indirizzo della matrice trasformata.

Detto questo l'operazione di scambio della colonna del pivot con la colonna più esterna potrebbe sembrare superflua. Invece questo accorgimento si rivela utile per preservare la coelescenza degli indirizzi. In questo modo se tutti i thread di un blocco scriveranno elementi contigui di una riga della matrice $B$ è garantito che leggeranno celle contigue anche nella matrice di partenza $A$.
Anche in questo caso il codice sorgente è disponibile nel file `/CudaMatrixClass/Src/Condensation.cuh`.

## 4.1 Kernel di condensazione

Del Kernel che esegue lo step di condensazione, su una matrice precedentemente arrangiata con il termine di pivot in basso a destra, sono state realizzate cinque versioni, il codice è presente in forma commentata in `ProgettoFinale/KernelTest/condensation_tentative.cuh`.

Tutti questi kernel sono stati pensati per essere lanciati con una griglia 2d di blocchi 1d, il numero di thread ottimale deve essere tale che la griglia sia sufficientemente larga da coprire almeno una volta l'intera larghezza della matrice. Questa disposizione fa emergere immediatamente il problema che tutti i thread che dovranno di calcolare simultaneamente la riga $i$ della matrice condensata $B$ tenteranno di leggere contemporaneamente il valore $a_{i,n-1}$ nella colonna del pivot determinando un collo di bottiglia simile a quello che si presenta quando si cerca di dividere una riga per il pivot.

Visto che l'obbiettivo è trattare matrici molto grandi, qualsiasi via che sfrutti la memoria constant o il passaggio dei valori per argomento non è più percorribile in questo caso. L'approccio che si è dimostrato migliore è stato quello di sfruttare la memoria texture dell'architettura CUDA. Se prima di ogni passaggio di condensazione si procede ad un*bind* dell'array degli elementi di matrice come un texture monodimensionale si garantisce una notevole velocizzazione nella lettura, da parte di ogni thread, dei valori necessari al calcolo del singolo elemento di $B$ oltre che scongiurare il congestionamento dovuto all'elemento di maggiore traffico.

Con questo accorgimento e con la disposizione della griglia e dei blocchi detta prima si garantisce, oltre ad un efficiente pattern di lettura nella memoria, anche la coalescenza di indirizzi visto che tutti i thread nel blocco eseguono le stesse quattro operazioni su celle contigue:

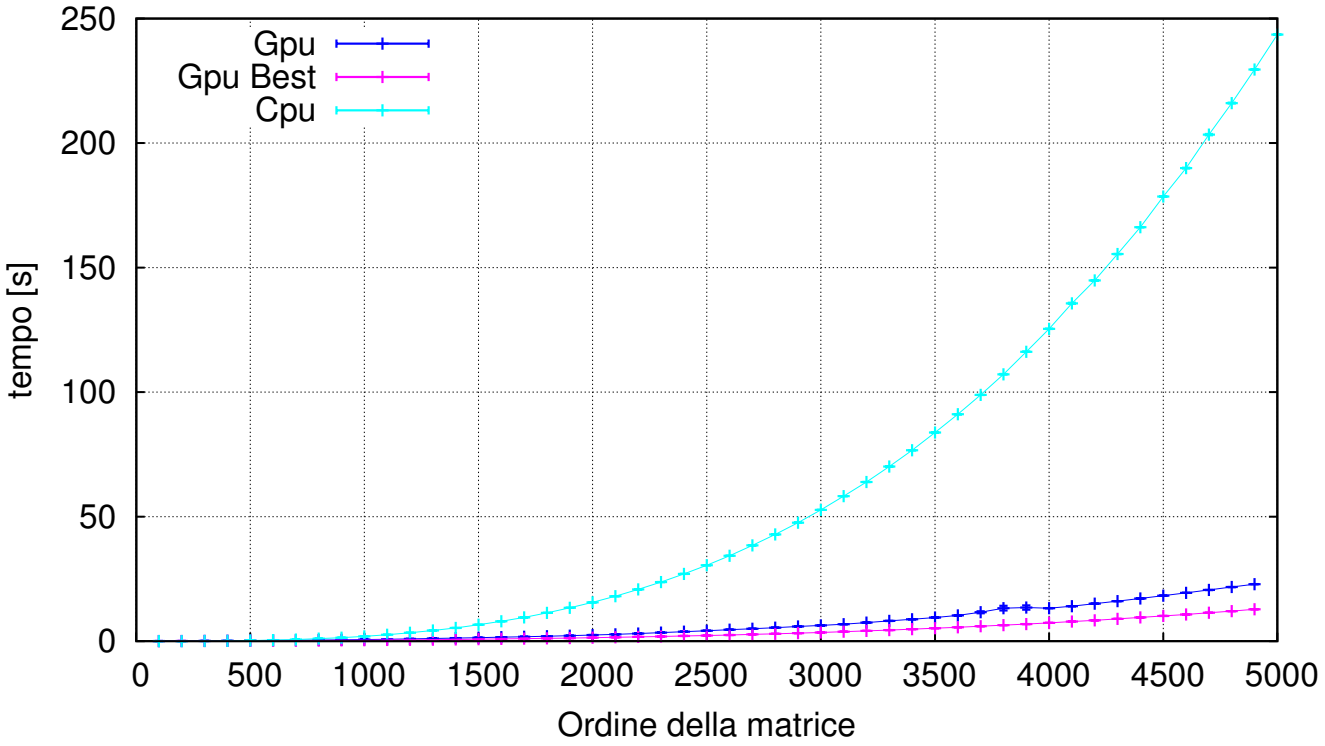
1. I thread contigui del blocco leggono celle contigue sulla memoria texture lineare contenente $A$
2. I thread del blocco leggono tutti lo stesso elemento sulla colonna del pivot
3. i thread contigui del blocco leggono celle contigue sulla riga contenente il pivot (anche questa si trova sulla memoria texture contenente $A$)
4. ogni thread calcola l'elemento di $B$ e lo scrive su celle di memoria contigue.

E' da notare però che la dimensione massima della memoria *bindabile* come texture non è illimitata e dipende delle specifiche di ogni dispositivo Nvidia.
Per questo è stato implementato anche un kernel che fa un utilizzo inferiore della memoria texture. In questo specifico caso viene fatta una copia della colonna contenente il pivot su un secondo array (quindi messi anch'essi in linea) che viene subito dopo *bindato* come texture per garantire una lettura più efficiente e sulla stessa cella di memoria. In questo caso la memoria texture occuperà solo $n$ celle invece che $n^2$.
Nella cartella `/ProgettoFinale/KernelTest` sono presente dei codici sorgente ( denominati `condensation_test*.cu`) pensati per confrontare le funzionalità dei kernel di condensazione.

# 5 Conclusioni

Oltre ai programmi necessari a testare il corretto funzionamento degli algoritmi sono state effettuate delle prove di performance per mettere a confronto il tempo necessario al calcolo della lista dei pivot impiegato dall'algoritmo cpu e dai due algoritmi Gpu discussi nel precedente paragrafo ( come "best" si intende il metodo che sfrutta il bind a texture dell'intera matrice). Dal grafico 1 è evidente senza alcun dubbio come l'algoritmo parallelizzato con CUDA sia

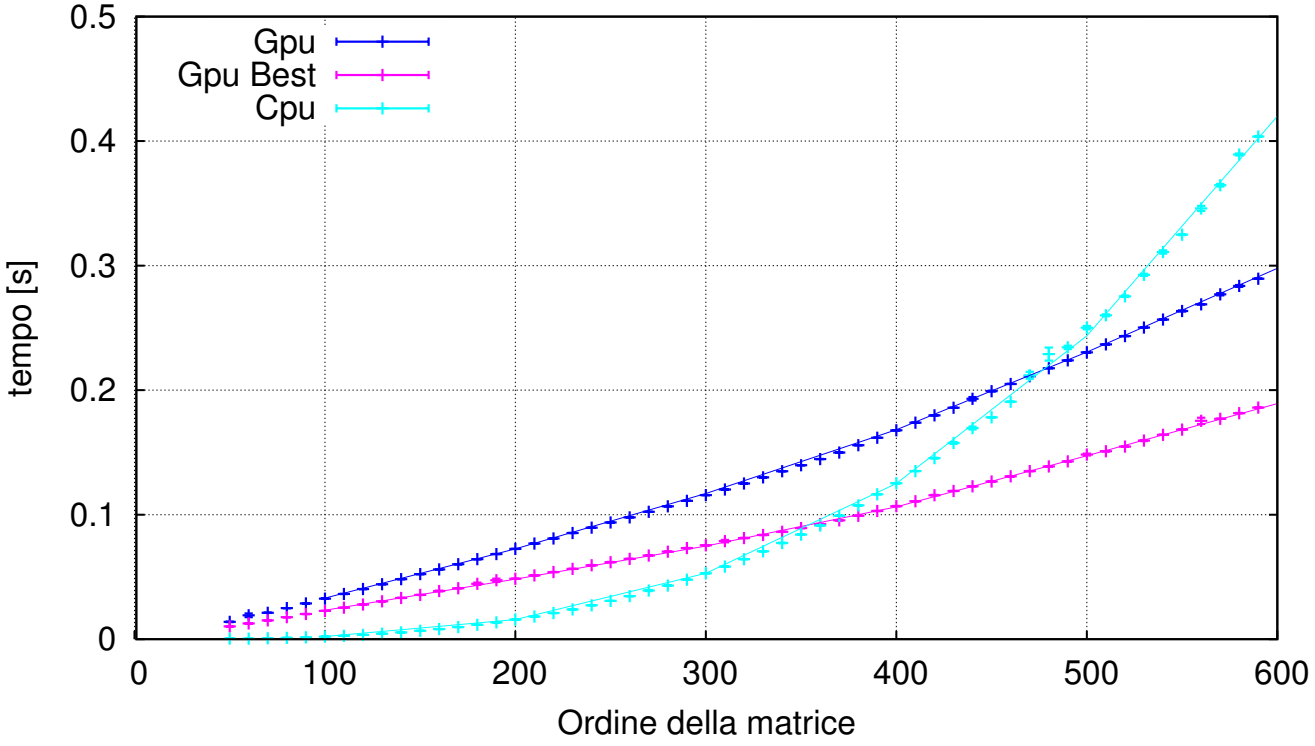
Figure 1: Confronto tempo d'esecuzione vs Taglia della matrice




estremamente più veloce rispetto alla Cpu già a partire dalle matrici di ordine 1000. E' necessario notare però che le API di CUDA non permettono di allocare array di celle contigue nella memoria global oltre una certa dimensione, questo determina un limite alla dimensione massima delle matrici considerabili. In questi test non si è andato oltre i $\tilde{1}0^8$ elementi di tipo *float*. Osservando invece il grafico 2 si nota che la situazioni è invertita quando si considerano matrici più piccole, per matrici fino all'ordine $n = 350$ l'algoritmo classico si dimostra più perfomante. Questo non è sorprendente, preso singolarmente il singolo thread dell'architettura CUDA non raggiunge la velocità del singolo thread della cpu. La scheda video riesce ad eccellere nel momento in cui vengono realizzati dei kernel in grado di distribuire il calcolo uniformemente tra un eleveta numero di thread.

Il computer usato per il test è stato il nodo "Sonic" di LCM, le specifiche salienti sono processore *Intel(R) Core(TM) i5-3550 CPU @ 3.30GHz* con 8GB di ram per quanto riguarda l'host e scheda video CUDA *Geforce GTX480*,

Figure 2: Confronto tempo d'esecuzione vs Taglia della matrice (zoom)




## References


[1] *Dodgson .C L.* Condensation of Determinants *Proceedings of the R. Society London 1866*

[2] *Abdelmalek Salem, Kouachi Said.* Condensation of Determinants `http://arxiv.org/abs/0712.0822`

[3] *Sardar Anisul Haque, Marc Moreno Maza.* Determinant Computation on the Gpu using the Condensation Method `http://hgpu.org/?p=7012`