Triangolarizzazione di una matrice con scheda grafica

Emanuele Ricci

25 luglio 2021

1 Obiettivo del problema

Lo scopo è la risoluzione di sistemi di equazioni di primo grado servendosi di un algoritmo per il calcolo parallelo scritto per schede grafiche Nvidia. Il senso di un tale programma è dato dal fatto che per sistemi con tante equazioni è inevitabile che il tempo necessario per trovare la soluzione cresca in modo vertiginoso.
Dal momento che questo problema si presta alla parallelizzazione è possibile ridurre molto significativamente il tempo impiegato e da qui l'idea di usare una scheda grafica.

2 Formalizzazione matematica del problema

2.1 Matrice

Immaginando di avere una serie di equazioni con N incognite, è cosa nota che occorre che ci siano almeno N equazioni linearmente indipendenti per assicurarsi che esista una soluzione al problema.

Si potrebbe decidere di creare una matrice $N \times N{+}1$ in cui a ogni colonna corrispondono tutti i coefficienti della stessa incognita e l'ultima colonna è quella dei termini noti:

$$x + y + z = 5$$
$$2x + 3y + 5z = 8$$
$$4x + 4z = 2$$

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 5 \\ 2 & 3 & 5 & 8 \\ 4 & 0 & 5 & 2 \end{array}\right]$$

2.2 Il metodo di Gauss

Questa nuova forma consente di triangolarizzare la matrice con metodo di Gauss. Partendo dalla prima riga, tale metodo consiste nel sottrarre un multiplo della stessa a ogni riga sottostante volta per volta facendo sì che il primo elemento di ogni riga sia zero, dunque annullare la prima colonna tranne che per il primo elemento. Poi si va avanti con la seconda riga ed il secondo elemento annullando tutta la seconda colonna eccetto per il primo e secondo elemento, dunque sotto la diagonale, e così via. Riporto un esempio.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 5 \\ 2 & 3 & 5 & 8 \\ 4 & 0 & 5 & 2 \end{array}\right] \xrightarrow{R_1 - 2R_2} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 5 \\ 0 & 1 & 3 & -2 \\ 4 & 0 & 5 & 2 \end{array}\right] \xrightarrow{R_3 - 4R_1} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 5 \\ 0 & 1 & 3 & -2 \\ 0 & -4 & 1 & -18 \end{array}\right] \xrightarrow{R_3 + 4R_2} \left[\begin{array}{ccc|c} 1 & 1 & 1 & 5 \\ 0 & 1 & 3 & -2 \\ 0 & 0 & 13 & -26 \end{array}\right]$$

Il risultato finale è ottenere una matrice dove sotto la diagonale tutti gli elementi sono 0. L'utilità è evidente: ora è possibile risolvere il sistema partendo dal basso e sostituire progressivamente i risultati.

$$13z = -26 \quad \Rightarrow \quad z = -2 \quad \Rightarrow \quad y - 6 = -2 \quad \Rightarrow \quad y = 4 \quad \Rightarrow \quad x = 3$$

2.3 Pivoting

È stato scelto anche di usare la tecnica del pivoting. Questa consiste nel cercare ogni volta l'elemento massimo della colonna che si intende annullare col metodo di Gauss e scambiando la riga corrispondente con quella corrente per poi eseguire la triangolarizzazione. Chiaramente come prima occorre considerare solo gli elementi sotto la diagonale.

$$\left[\begin{array}{ccc|c} 1 & 1 & 1 & 5 \\ 2 & 3 & 5 & 8 \\ 4 & 0 & 5 & 2 \end{array}\right] \xrightarrow[\text{pivoting}]{R_1 \to R_3} \left[\begin{array}{ccc|c} 4 & 0 & 5 & 2 \\ 2 & 3 & 5 & 8 \\ 1 & 1 & 1 & 5 \end{array}\right] \xrightarrow[\text{prima colonna}]{\text{eliminazione}} \left[\begin{array}{ccc|c} 4 & 0 & 5 & 2 \\ 0 & 3 & \frac{5}{2} & 7 \\ 0 & 1 & \frac{-1}{4} & \frac{9}{2} \end{array}\right] \xrightarrow[\text{il piv. non serve}]{3 > 1} \left[\begin{array}{ccc|c} 4 & 0 & 5 & 2 \\ 0 & 3 & \frac{5}{2} & 7 \\ 0 & 1 & \frac{-1}{4} & \frac{9}{2} \end{array}\right]$$

A questo punto si va avanti eliminando la colonna 2 con Gauss per ottenere la matrice triangolarizzata. Per una matrice più grande si procederebbe di nuovo facendo prima il pivoting e poi Gauss come appena visto per ogni colonna/riga rimanente.
Questa tecnica risulta molto utile a livello computazionale poiché dividere sempre per il numero più grande gli elementi sottostanti contribuisce a stabilizzare i risultati e quindi ridurre gli errori operando su matrici molto grandi.

3 Dalla matematica al calcolatore

3.1 Premessa

Per poter calcolare la correttezza dell'algoritmo è stato usato il linguaggio Mathematica che è un ambiente di calcolo che permette la manipolazione simbolica, oltre che il calcolo numerico, della matematica. Dunque è in grado di risolvere sistemi come quelli finora visti mantenendo le frazioni inalterate, ovvero senza svolgere calcoli e dunque trovando il valore esatto delle soluzioni. Come è facile intuire la pecca di un simile linguaggio è proprio il fattore tempo ma tale problema verrà affrontato in seguito.

3.2 Mathematica

Matematica è stato usato per generare matrici di dimensione a piacere generando per ogni elemento un numero casuale tra 0 e 100 (compreso il vettore dei termini noti). Successivamente il codice scritto prevedeva di risolvere il sistema e di stampare matrice e soluzioni in appositi file *.txt* che sono serviti da appoggio per gli altri programmi. Non mi dilungherò oltre a spiegare il funzionamento del codice scritto in mathematica perché effettivamente essendo un linguaggio ad alto livello non è necessario implementare nessuna tecnica per risolvere il sistema di equazioni: basta solo crearlo e usare il comando *Solve*.

3.3 CPU vs GPU

Prima di arrivare alla parallelizzazione è necessario spendere un po' di tempo per mettere in chiaro come viene eseguito un simile codice in modo sequenziale con un linguaggio come il C++.
In verità c'è poco da dire: il programma esegue passo per passo per ogni colonna prima il pivoting come visto prima e poi l'annullamento degli elementi sotto la diagonale per poi passare alla colonna successiva. Come si vedrà meglio in seguito questa parte dell'algoritmo è quella davvero parallelizzabile poiché la sottrazione di una riga dall'altra e lo scambio di due righe è ciò che può essere fatto su più fronti contemporaneamente. La sola CPU è costretta a eseguire questi calcoli rigorosamente uno alla volta.
Per quanto riguarda la risoluzione del sistema a triangolarizzazione avvenuta invece non è possibile fare niente: dal momento che chiaramente non si possono risolvere le equazioni senza conoscere tutti i valori delle incognite precedenti l'unica cosa che rimane è una risoluzione sequenziale.

4 CUDA e schede grafiche

4.1 Compute Unified Device Architecture

Per lavorare su schede grafiche, nvidia ha creato una estensione del linguaggio C, Cuda. Ciò permette di lanciare dal *main* di un programma C/C++ un kernel cuda, ovvero una funzione che viene eseguita direttamente sul device.

4.2 Blocks e threads

Bisogna capire ora come funziona e come "ragiona" una scheda grafica Nvidia.
I threads sono le parti in cui viene suddiviso il processo che porta alla risoluzione del problema, in questo caso i threads sono le unità fondamentali che vengono mandate in esecuzione contemporaneamente dalla scheda e che quindi eseguono il programma.
I blocchi, o blocchi di threads, sono delle astrazioni che rappresentano il modo in cui raggruppare i threads al momento del lancio della funzione, cioè quanti threads per blocco. Si possono avere anche threads e blocchi bidimensionali, in tal caso occorre specificare la quantità per ogni dimensione. A livello astratto il codice viene scritto pensando che i blocchi vengono eseguiti contemporaneamente, in realtà questo trascende le reali possibilità della scheda che sarà costretta a eseguirne il numero massimo in base alle reali capacità tecniche e all'ottimizzazione del codice.
Esiste una memoria globale in cuda, da cui threads di ogni blocco può attingere, una piccola memoria per ogni thread non utilizzabile da nessuno se non dal thread stesso e una per ogni blocco, detta "shared" che può essere usata da tutti i thread dello stesso blocco. Nota impostante: non esiste una memoria condivisa specificatamente per i blocchi, ovvero i thread di ogni blocco del kernel non possono comunicare. Dunque per far funzionare un kernel cuda è necessario specificare il numero di blocchi da usare, potenzialmente infiniti, e il numero di threads per blocco, che sono al massimo 1024 ogni blocco. A seconda delle dimensioni e della natura del problema sarà necessario ottimizzare queste 2 variabili per ottenere il massimo della prestazione.
Un esempio si trova in Figura 1, da notare come è stato organizzato un kernel bidimensionale dove è chiara la suddivisione in blocchi lungo l'asse x e y. Stessa cosa si ritrova dentro ogni blocco con i threads divisi anch'essi lungo l'asse x e y.

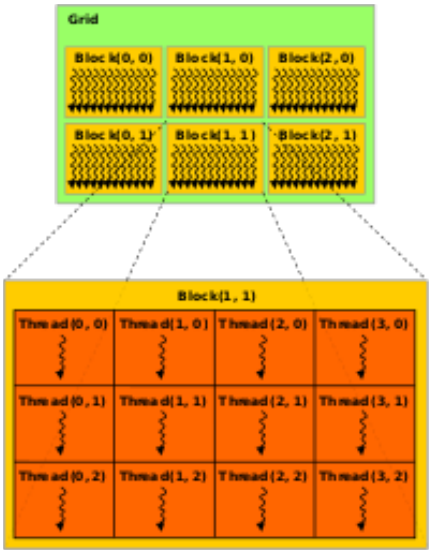


Figura 1: Esempio di suddivisione di blocchi e threads

4.3 Hardware

Fino ad ora si è parlato ad un livello astratto, ora è doveroso fare un piccolo approfondimento hardware sulle reali possibilità e limiti della scheda, in modo da capire come sfruttarne appieno le potenzialità. La particolarità della scheda è quella di avere molti più core della CPU, è questo ciò rende possibile parallelizzare l'algoritmo, mandando simultaneamente più core. Nella pratica i threads sono raccolti in gruppi di 32, detti warp e sono quelli che fisicamente vengono eseguiti in contemporanea. Il numero massimo di warp che possono essere eseguiti simultaneamente dipende dalle potenzialità della scheda. Ogni warp è fatto per avere 32 threads che devono essere presi esclusivamente dal singolo blocco a cui è stato assegnato. Nel caso il numero di threads per blocco fosse non multiplo di 32 l'ultimo warp si ritroverebbe a essere parzialmente inutilizzato poiché come detto prima non è possibile comunicare tra blocchi e il resto della divisione per 32 è il numero di threads sprecati. Quindi ciò che se ne ottiene è un calo della prestazione, di seguito un esempio per chiarire meglio il concetto.
Si immagini di aver lanciato 100 blocchi con 1000 threads ciascuno, in tutto 100000 threads. Per ogni blocco sono assegnati 32 warp poiché è il multiplo di 32 più vicino a 1000, $32 * 32 = 1024$. Questo crea uno spreco di 24 threads a blocco, in tutto 2400 threads. Usando 1024 threads i risultati migliorano: il multiplo di 1024 più vicino a 100000 è 100352, quindi $100352/1024 = 98$. Dichiarando un kernel cuda con 98 blocchi e 1024 threads per blocco il risultato è 352 threads inutilizzati. Nettamente meglio di prima.
Da notare anche un'altra cosa: un warp è capace di eseguire i suoi 32 threads contemporaneamente solo se tutti hanno le stesse istruzioni, in caso contrario è costretto a dividere e eseguire i threads in gruppi in base ai compiti uguali e ciò provoca un ulteriore rallentamento. Nel primo caso un thread per blocco si ritrovava in questa situazione, nel secondo invece solo uno in tutto. Più avanti si vedrà anche come ovviare, quando possibile, a questo problema.

5 Il codice

Finalmente si arriva a ciò che costituisce il nucleo del lavoro svolto. Concettualmente è possibile dividere il codice a seconda di ciò che gira sulla CPU o GPU.

5.1 Main

Il main è in mano alla CPU ed è da lì che è possibile invocare i kernel cuda. Il codice per prima cosa prende i file *.txt* della matrice, termini noti e soluzioni precedentemente creati con mathematica e li salva in memoria. Di seguito la parte di main interessata.

```
1  dato soluzionimath[N];
2  dato *matrice;
3  ifstream GetMatrix;
4  ifstream GetTerm;
5  ifstream GetSol;
6
7  matrice=new dato [N*(N+1)];
8
9  GetMatrix.open("matrix.txt");
10 if(GetMatrix.fail()){
11    cout<< endl << "Problema apertura file di ingresso dati! Esco!";
12    return 0;
13 }
14
15 GetTerm.open("term.txt");
16 if(GetTerm.fail()){
17    cout<< endl << "Problema apertura file di ingresso dati! Esco!";
18    return 0;
19 }
20
21 for(int i=0; i<N; i++){
22   GetTerm>>matrice[i*(N+1)+N];
23   for(int j=0; j<N; j++){
24     GetMatrix>>matrice[i*(N+1)+j];
25   }
26 }
27
```

```
  GetSol.open("solutions.txt");
  if(GetSol.fail()){
     cout<< endl << "Problema apertura file di ingresso dati! Esco!";
     return 0;
  }


  for(int i=0; i<N; i++){
  GetSol>> soluzionimath[i];
  }


  GetMatrix.close();
  GetTerm.close();
  GetSol.close();
```

A questo punto viene mandata una funzione, sempre CPU, che prende in ingresso la matrice e che è responsabile dell'invocazione dei kernel cuda. Il compito di ogni elemento verrà spiegato a tempo debito, per ora occorre concentrarsi solo sulle cose fondamentali.
Dato è un typdef di tipo float o double, THREADS e THREADS1 sono variabili globali poste rispettivamente a 32 e 1024. Dunque tutti gli nblock hanno come valore la divisione intera $\frac{N+1}{THREADS}$ arrotondata per eccesso. La funzione cudaMalloc ha il compito di allocare uno spazio di memoria sulla scheda grafica, è importante perché le due memorie, CPU e GPU, non comunicano. Prendendo in esame *matricedev* si nota che non ha le dimensioni della matrice del main $(N \times (N+1))$ ma il più piccolo multiplo intero di 32 maggiore di $N+1$ per il discorso di prima.
Il cudaMemcpy2d copia una certà quantità di dati dalla memoria del processore alla memoria globale della scheda, nel caso specifico nello spazio di memoria allocato precedentemente. Si nota che dunque una parte di memoria rimane inutilizzata. Il ciclo *for* in cui uno dopo l'altro vengono invocati i kernel responsabili del pivoting - findmax, getmax, pivoting - e quella dell'annullamento della i-esima colonna - triangolo. Infine a matrice triangolarizzata il solveCuda risolve il sistema, poi le soluzioni vengono copiate indietro sul main con un altro cudaMemcpy2d. Tutte le funzioni *cudaEvent* servono per misurare i tempi di esecuzione.

```
dato* triangolizza( dato *matrice ){

 dato* matrice_dev;
 dato *pivot_dev;
 dato* solcuda;
 dato *solcpu;
 int *Max;
 int nblock1; // y, colonne (j)
 int nblock2 = (N+1)/THREADS + 1; // x, righe (i)
 int nblock3 = (N+1)/(THREADS1) + 1;
 nblock1=nblock2;
 int width1= nblock1*THREADS;
 int width2= nblock2*THREADS;

 solcpu = new dato [N];
 cudaMalloc((void**)&matrice_dev,width2*width1*sizeof(dato));
 cudaMalloc((void**)&pivot_dev,nblock3*sizeof(dato));
 cudaMalloc((void**)&Max,nblock3*sizeof(int));
 cudaMalloc((void**)&solcuda,N*sizeof(dato));

 cudaEvent_t start,stop;
 cudaEventCreate(&start);
 cudaEventCreate(&stop);
 cudaEventRecord(start,0);

 cout<< "numero blocchi kernel 2D: "<<nblock2<<" x "<< nblock1 <<", width: "<<width2
   <<" x "<<width1<<endl;

 cudaMemcpy2D(matrice_dev,width2*sizeof(dato),matrice,(N+1)*sizeof(dato),(N+1)*sizeof
   (dato),N,cudaMemcpyHostToDevice);


 dim3 c_threads(THREADS,THREADS);
 dim3 c_blocks(nblock2,nblock1);

```

```
  completa_matrice<<<c_blocks,c_threads>>>(matrice_dev);

  cout<< "numero blocchi kernel 1D: "<<nblock3<<", width: "<<THREADS1*nblock3<<endl;

  for( int i=0; i<N-1; i++){
    findmax<<<nblock3,THREADS1>>>(matrice_dev, pivot_dev, i, Max);
    getmax<<<1,1>>>(pivot_dev, Max, nblock3);
    pivoting<<<nblock3,THREADS1>>>(matrice_dev, pivot_dev, i, Max);
    triangolo<<<c_blocks,c_threads>>>(matrice_dev,i);
  }

  solveCuda<<<1,1>>>(matrice_dev, solcuda, nblock1);

  cudaMemcpy(solcpu, solcuda, N*sizeof(dato), cudaMemcpyDeviceToHost);

  cudaEventRecord(stop,0);
  cudaEventSynchronize(stop);
  float elapsed;
  cudaEventElapsedTime(&elapsed,start,stop);
  cout<<"tempo: " << elapsed/1000. << endl;

  //dealloco
  cudaEventDestroy(start);
  cudaEventDestroy(stop);
  cudaFree(solcuda);
  cudaFree(pivot_dev);
  cudaFree(matrice_dev);
  return solcpu;
  }
```

Tornando sul main si vede l'invocazione della funzione appena descritta che avendo come return dato*, viene assegnata a un puntatore.

```
dato* solfromcuda;
solfromcuda = triangolizza(matrice);
confrontaSol(solfromcuda, soluzionimath);
```

La *confrontaSol* è importante perché mette a paragone le 2 soluzioni, quella esatta di mathematica e quella approssimata di cuda.

```
void confrontaSol(dato *sol,dato *solmath){
  cout<<"Soluzioni con differenza > 0.00001 rispetto al risultato esatto: "<<endl;
  for(int i=0; i<N; i++){
  if(abs(sol[i]/solmath[i]-1)>0.00001)
  cout<<i<<" "<< abs(sol[i]/solmath[i]-1)<<endl;
  }
}
```

5.2 Kernel cuda

Ora rimane da analizzare con chiarezza tutte le funzioni cuda una per una.

5.2.1 Completa matrice

Andando in ordine questa è la prima. Il kernel cuda con cui è invocata è bidimensionale, *dim*3 serve proprio a questa dichiarazione. Si capisce al volo che si tratta di un kernel cuda per la presenza di $<<<, >>>$ che servono per specificare a sinistra la quantità e la dimensione dei blocchi (1d, 2d, 3d) e stessa cosa per i threads per blocco a destra.

```
dim3 c_threads(THREADS,THREADS);
dim3 c_blocks(nblock2,nblock1);

completa_matrice<<<c_blocks,c_threads>>>(matrice_dev);
```

```
__global__ void completa_matrice(dato *a) {

  int i = blockIdx.y*blockDim.y + threadIdx.y;
  int j = blockIdx.x*blockDim.x + threadIdx.x;
  int offset = gridDim.x*blockDim.x*i + j;
```

```
6
7   if(i >= N || j >= N+1)
8     a[offset] = 0.;
9 };
```

La scritta *global* significa che è possibile lanciare tale funzione dalla CPU. Come visto prima dato che la memoria allocata è maggiore di quella della matrice un certo numero di elementi rimane non inizializzato, ed è questo problema che si intende risolvere. Cuda genera un griglia bidimensionale con un certo numero di blocchi determinato da $nblock1$ e $nblock2$ con 32 threads per dimensione, dunque 1024 threads per blocco. Le indicizzazioni $threadIdx.x$ e $threadIdx.y$ servono per specificare le "coordinate" del singolo thread all'interno del blocco. Per $blockIdx.x$ e $blockIdx.y$ è la stessa cosa ma con i blocchi dentro la griglia (che però e una sola). Invece $blockDim.x$ e $blockDim.y$ servono rispettivamente a indicare la dimensione dei blocchi lungo l'asse x e y, ovvero la quantità di thread che contengono; $gridDim.x$, $gridDim.y$ indicano la dimensione della griglia, cioè quanti blocchi contiene sempre lungo x e y. Riguardare la Figura 1 per capire meglio.
Arrivando a *completamatrice*, quello che fa è indicizzare delle variabili i e j, proprie di ogni thread come fossero la coordinata x e y di un elemento della matrice allocata, a questo punto controlla se è maggiore di N o $N+1$, le dimensione "vere", e in caso affermativo inizializza l'elemento (i,j) a 0. Ora è possibile operare sulla matrice allocata prima.

5.2.2 Findmax

La funzione findmax ha il compito di prendere per ogni blocco il valore massimo del valore assoluto e la sua posizione all'interno della colonna corrente della matrice. Questo kernel monodimensionale è mandato con 1024 thread e chiaramente i necessari blocchi per coprire tutta la matrice. L'intero *col* è l'indice del ciclo *for* che rappresenta la colonna da annullare. Ai fini della ricerca risulta utile usare la memoria *shared* in cui vengono caricati gli elementi della colonna della matrice indicizzandoli come visto prima. A questo punto viene fatto quello che si potrebbe definire un "confronto parallelo": ogni elemento nel thread della prima metà del blocco viene confrontato con quello corrispondente della seconda metà e nel caso quest'ultimo sia maggiore il valore viene sostituito. Si procede poi riducendo ogni volta della metà il numero di thread da usare fino a quando non rimarrà il thread 0 con il massimo del suo blocco, e questo per ogni blocco. La memoria shared ha il compito fondamentale di mettere in comunicazione i thread del blocco che altrimenti non potrebbero fare i confronti con i loro elementi. Ogni volta viene sempre scambiato anche il valore della posizione dell'elemento più grande, oltre che il valore stesso. Alla fine l'elemento più grande del blocco viene caricato nel vettore $pivot_dev$, caricato in precedenza e che sta nella memoria globale con il corrispondente indice nel vettore Max. Per un esempio guardare la Figura 2: una volta selezionato l'elemento maggiore viene ridotto della metà il numero di thread che operano.

```
1  __global__ void findmax (dato* matrice_dev, dato* pivot_dev, int col, int *Max){
2
3    int row = blockIdx.x*blockDim.x + threadIdx.x;
4    int len = int((N+1)/THREADS+1)*THREADS;
5
6    __shared__ dato cachecontrol [THREADS1];
7    __shared__ int indices [THREADS1];
8
9    indices[threadIdx.x]=row;
10
11   if(row>=col && row < len){
12   cachecontrol[threadIdx.x] = abs(matrice_dev[(row)*(len)+col]);
13   } else {
14     cachecontrol[threadIdx.x] = 0;
15   }
16
17   __syncthreads();
18
19   int i=blockDim.x/2;
20
21   while (i!=0){
22     if(threadIdx.x<i){
23     if(cachecontrol[threadIdx.x]<cachecontrol[threadIdx.x+i]){
24       cachecontrol[threadIdx.x] = cachecontrol[threadIdx.x+i];
25       indices[threadIdx.x]=indices[threadIdx.x+i];
```

```
    }
    }

    i/=2;
    __syncthreads();

  }

  if(threadIdx.x==0){
    pivot_dev[blockIdx.x]=cachecontrol[0];
    Max[blockIdx.x]=indices[0];
  }

  __syncthreads();
}
```

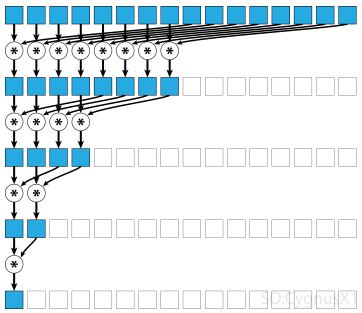


Figura 2: Un esempio di confronto parallelo

5.2.3 Getmax

Finita la $findmax$ non è ancora stato trovato il massimo della colonna, bensì l'elemento massimo di ogni blocco. Ora è necessario trovare il massimo effetivo, con la sua posizione. La funzione è molto semplice, dichiara un kernel con 1 blocco e 1 thread ed esegue in modo sequenziale un confronto fra tutti gli elementi del vettore e inserisce in testa a $pivot_dev$ e Max l'elemento più grande e la sua posizione.

```
getmax<<<1,1>>>(pivot_dev, Max, nblock3);
```

```
__global__ void getmax(dato * pivot_dev, int * Max, int nblock3){

    for( int i=0; i<nblock3; i++){
      if(pivot_dev[0]<pivot_dev[i]){
        pivot_dev[0]=pivot_dev[i];
        Max[0]=Max[i];
      }
    }
  __syncthreads();
}
```

5.2.4 Pivoting

La funzione pivoting è essenzialmente lo scambio delle due righe interessate della matrice: quella attuale e quella in cui è stato trovato il massimo. In vantaggio come al solito è che non è stato fatto un passaggio alla volta. Dopo la solita indicizzazione in base al threadIdx.x e al blockIdx.x è possibile scambiare velocemente le righe

```
pivoting<<<nblock3,THREADS1>>>(matrice_dev, pivot_dev, i, Max);
```

```
__global__ void pivoting (dato* matrice_dev, dato* pivot_dev, int col, int* Max ){

  if(pivot_dev[0]==0){
    return;
```

```
  }

  int lenrow = int((N+1)/THREADS+1)*THREADS;
  int row = blockIdx.x*blockDim.x + threadIdx.x;

  if(row < int((N+1)/THREADS+1)*THREADS){
    dato appo = matrice_dev[lenrow*Max[0]+row];
    matrice_dev[lenrow*Max[0]+row] = matrice_dev[lenrow*col+row];
    matrice_dev[lenrow*col+row]=appo;
    //Max[blockIdx.x]=0;
    //pivot_dev[blockIdx.x]=0;
  }
  __syncthreads();
}
```

5.2.5 Triangolo

Di nuovo tornano i kernel bidimensionali. L'indicizzazione è sempre la stessa e come detto prima i threads e i blocchi sono organizzati in modo che a ognuno corrisponda un elemento della matrice. Ogni thread prende come coefficiente il prodotto tra l'inverso dell'elemento della diagonale nella colonna a cui è arrivato il ciclo for (*index*) e l'elemento sotto quest'ultimo sulla stessa riga del thread, che quindi è univocamente determinato. Dopodiché all'elemento identificato dal thread viene sottratto l'elemento sulla stessa colonna ma sulla riga *index* moltiplicato per il coefficiente trovato prima. Il processo è del tutto analogo a quello spiegato nella sezione di formalizzazione matematica ma, come al solito, qui la colonna viene annullata tutta insieme. Guardare l'esempio sotto.

```
triangolo<<<c_blocks,c_threads>>>(matrice_dev,i);
```

```
__global__ void triangolo(dato* matric_dev, int index){

  int i = blockIdx.y*blockDim.y + threadIdx.y;
  int j = blockIdx.x*blockDim.x + threadIdx.x;
  int offset = gridDim.x*blockDim.x*i + j;

  int row = index*gridDim.x*blockDim.x;
  int ind = row+j;

  if(i>index && j>index){
    if(matric_dev[row+index]!=0){
      dato coef = matric_dev[i*gridDim.x*blockDim.x+index]/matric_dev[row+index];

      matric_dev[offset] -= matric_dev[ind]*coef;
    }
  }
  __syncthreads();
}
```

$$\begin{bmatrix} th_{00} & th_{01} & \dots & th_{0K} \\ th_{10} & th_{11} & \dots & th_{1K} \\ \vdots & \vdots & \ddots & \vdots \\ th_{K0} & th_{K1} & \dots & th_{KK} \end{bmatrix}$$

Supponendo di essere appena partiti e che *index* sia 0, ogni th_{ij} memorizza il coefficiente $\frac{th_{i0}}{th_{00}}$. Poi sottrarrà a th_{ij}, $coef \times th_{0j}$. Questo chiaramente causerà l'annullamento della colonna 0: $t_{i0} - \frac{th_{i0}}{th_{00}} \times th_{00} = 0$.

5.2.6 Solve Cuda

Una volta che la matrice è stata triangolarizzata si può procedere alla risoluzione del sistema. Di nuovo un kernel 1,1 perché il problema non è parallelizzabile. Quindi la funzione qui presente è del tutto analoga, a meno di minime correzioni, a quella da scrivere sul C++. Ci si potrebbe chiedere perché non risolvere il sistema nel main, dopotutto il singolo core del main è più veloce del singolo core cuda. La ragione è che prima occorrerebbe copiare tutta la matrice triangolarizzata indietro sul main con grande dispendio di tempo. Anche se per matrici di piccole dimensioni questo sistema potrebbe

risultare più efficiente, è sicuro che a lungo andare diventa scomodo. Pertanto è consigliabile risolvere il sistema sul device e poi copiare indietro solo il vettore delle soluzioni.

```
solveCuda<<<1,1>>>(matrice_dev, solcuda, nblock1);
```

```
__global__ void solveCuda(dato*m, dato *sol, int block1){

    int len = block1*THREADS;
    sol[N-1]=m[(N-1)*(len)+N]/m[(N-1)*(len)+N-1];

    for(int i=N-2; i>=0; i--){
      for(int j=N-1; j>i; j--){
  m[i*(len)+N] -= m[i*(len)+j]*sol[j];
      }
      if(m[i*(len)+i]!=0){
        sol[i]=m[i*(len)+N]/m[i*(len)+i];
      }
      else{
        sol[i]=0;
      }
   }
}
```

6 Esecuzione e risultati

È arrivato il momento di guardare i benefici di un algoritmo parallelizzato.

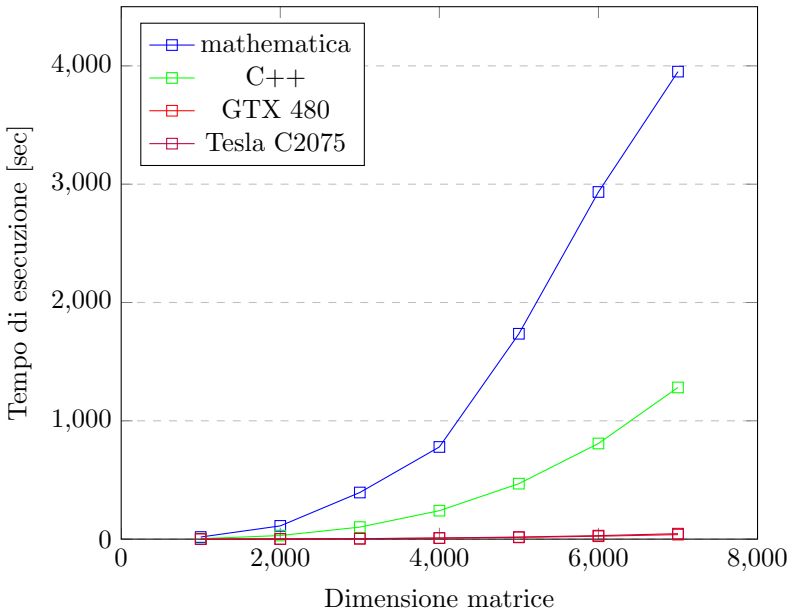
6.1 Macchine usate

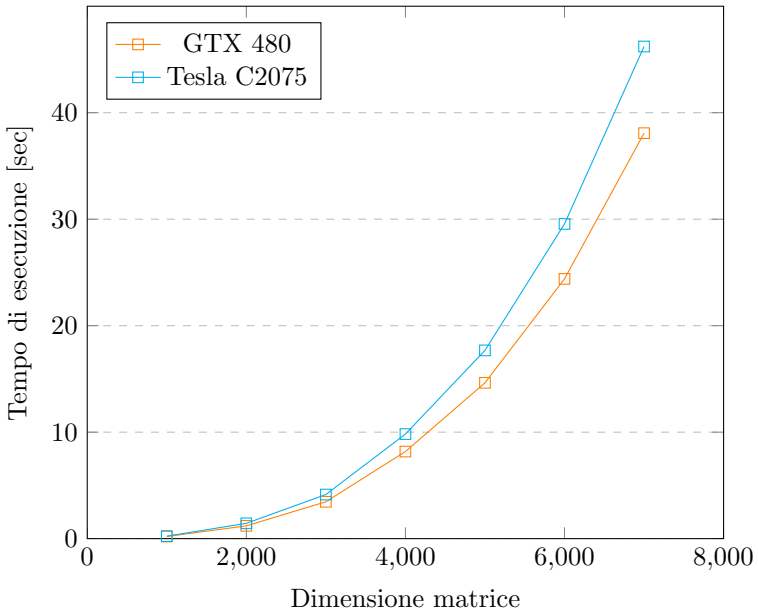
In questo lavoro sono stati usati i computer del cluster LCM (Laboratorio di Calcolo e Multimedia) dell'Università Statale di Milano, dipartimento di Fisica. In particolare per il codice mathematica è stata usata la macchina *king*, mentre per il codice C++/CUDA la macchina *jacobi*. Le schede grafiche su *jacobi* sono la GeForce GTX 480 e la Tesla C2075.

6.2 Tempi ed errori

In entrambi i programmi C++ e Cuda è stato usato il tipo double. Come spiegato prima gli errori sono stati calcolati con la $confrontasol$.

	mathe	C++		GTX 480	Tesla C2075	GPU
N	Tempo [sec]	Tempo [sec]	errore	Tempo [sec]	Tempo [sec]	errore
1000	17.688	3.83	0.00001	0.199706	0.235094	0.00001
2000	111.808	30.29	0.00001	1.20059	1.43763	0.00001
3000	393.968	101.61	0.00001	3.46228	4.14105	0.00001
4000	778.868	240.43	0.00001	8.17753	9.81975	0.00001
5000	1735.6	468.76	0.00001	14.6341	17.6783	0.00001
6000	2934.02	808.25	0.00001	24.3941	29.5559	0.00001
7000	3951.39	1281.98	0.00001	38.0796	46.2182	0.00001





6.3 Conclusioni

Emerge chiaramente il vantaggio spaventoso di usare una scheda grafica in termini di tempistiche: la crescita dei tempi è evidentemente esponenziale, tuttavia la curva delle schede risulta comunque avere una curva molto più lenta. Come ci si aspetterebbe mathematica è il linguaggio che risente di tempi più lunghi, questo è chiaramente da imputare al fatto di essere un linguaggio ad alto livello, quindi comodo da usare ma difficilmente ottimizzabile. Il C++ dà risutati migliori per lo stesso motivo: non gode delle funzioni built-in di mathematica ma è possibile scriverne di proprie con tempi di esecuzione minori. La GTX 480 risulta essere leggermente più performante della Tesla C2075, poiché l'architettura montata è la stessa ciò è da imputare alla differenza nelle specifiche tecniche.
Grazie all'uso dei double e alla tecnica del pivoting gli errori risultano minimi, sicuramente usando i float le prestazioni ne avrebbero guadagnato a discapito della precisione ma date le potenzialità della scheda non c'è stato bisogno di ricorrere a questo. In conclusione nonostante la difficoltà computazione della parallelizzazione dell'algoritmo si può affermare che il lavoro svolto ha dato i risultati sperati e che quindi ne valga la pena.