Progetto Programmazione parallela su architetture GPU
Algoritmo della discesa del gradiente

1-Obiettivo
L'obbiettivo che si pone il progetto è la parallelizzazione dell'algoritmo della discesa del gradiente, molto utile nel campo del machine learning dove la potenza di calcolo di una GPU può essere molto utile.

2- Strategia di risoluzione
Per parallelizzare questo algoritmo si è pensato di utilizzare la risoluzione dei sistemi lineari. Quindi la risoluzione di un sistema del tipo:

$$A\mathbf{x} = \mathbf{b},$$

che quindi equivale alla minimizzazione della forma quadratica:

$$Q(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}.$$

che tramite alcuni passaggi matematici, su cui non ci dilungheremo, viene tirato fuori l'algoritmo:

$k = 0$
while $\mathbf{r}_k \neq \mathbf{0}$
    calcolare la direzione di discesa $\mathbf{p}_k := \mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$
    calcolare il passo di discesa $\alpha_k := \dfrac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T A\mathbf{p}_k}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $k = k + 1$
end.

in cui, finché non si arriva a una direzione di discesa nulla(o più piccolo di una quantità da noi decisa), viene calcolata la direzione di discesa tramite la differenza tra b e il prodotto tra la matrice A e il punto in cui ci troviamo(x_k) poi viene calcolato il passo di discesa e infine viene aggiornato x_k quindi ci spostiamo verso la direzione e il passo calcolati in precedenza.

3- Versione0
La prima versione del programma, denominata versione0.c, viene eseguita solamente dalla cpu quindi non è stata parallelizzata. È servita solo come linea guida per la successiva parallelizzazione del programma e per verifica del programma dopo la parallelizzazione.

4-Versione1

La seconda versione, la prima versione realmente parallelizzata, denominata versione1.cu, ha alcune limitazione per quanto riguarda il numero di elementi in quanto l'idea è quella di avviare il kernel con dimensioni del blockSize e del numBlock uguali alla matrice di uscita di nostro interesse, questo limita l'uso di matrici grandi in quanto la gpu usata ha un numero massimo di threads per block che è pari a 1024. Andiamo a veder adesso i kernel con le loro rispettive prestazioni.

4.1-Multi_vec

Il primo kernel che andiamo a studiare riguarda la prima parte della moltiplicazione riga per colonna delle matrici. Il suo compito sarà quello di moltiplicare ogni elemento per il corrispettivo della seconda matrice.

```
__global__
void multi_vec(int n_row1,int n_col1,int n_row2,int n_col2,float* __restrict__ res_vec,float* __restrict__ d_vec1,float* __restrict__ d_vec2)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int c= blockIdx.x*n_row1 + (threadIdx.x)%n_col1;
  int j= ((int)(threadIdx.x/n_row2) + (threadIdx.x%n_row2)*n_col2);
  res_vec[i]=d_vec1[c]*d_vec2[j];

}
```

Vediamo come vengono creati due indici uno per la prima matrice mentre il secondo per la seconda matrice.
Primo indice c= blockIdx.x*n_row1 + (threadIdx.x)%n_col1 recupera quindi la riga in cui siamo tenendo conto del blockIdx.x andando ad aggiungere lo sfasamento della colonna.
Il secondo indice è un po' più complesso in quanto dobbiamo tener conto di muoverci lungo la colonna j= ((int)(threadIdx.x/n_row2) + (threadIdx.x %n_row2)*n_col2); quindi andiamo a prendere la parte intera della divisione tra il thread id e il numero di righe della seconda matrice che ci indicherà lo sfasamento della riga. Mentre il il secondo elemento(threadIdx.x %n_row2)*n_col2) ci indica in quale colonna andare. Per le prestazioni di questa versione si indica per una grandezza della matrice di 32x32 quindi un massimo di 1024 il limite dato dal maxThread per block.
prodotto: runtime 0.008448ms, 0.1212 GE/s, 0.4848 GB/s
Cosi pochi elementi naturalmente non riescono a riempire la banda e quindi abbiamo prestazioni molto basse.

4.2 Reduction_row

La seconda parte del prodotto righe per colonne è una riduzione che deve ridurre gli elementi precedentemente moltiplicati della stessa riga della matrice uno e della stessa colonna della matrice due. In questa versione bisogna modificare gli elementi della riduzione manualmente. Naturalmente non è una versione molto scalabile di questo kernel e della versione in generale.

```
__global__
void reduction_row(int N,float* __restrict__ res_vec,float* __restrict__ d_vec1)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int idx=(int)(i/N);
  float c =res_vec[idx];
  float d =d_vec1[i];
  if(i%N==31){
    res_vec[idx]=d_vec1[i-31]+d_vec1[i-30]+d_vec1[i-29]+d_vec1[i-28]+d_vec1[i-27]+d_vec1[i-26]+d_vec1[i-25]+d_vec1[i-24]+
                 d_vec1[i-23]+d_vec1[i-22]+d_vec1[i-21]+d_vec1[i-20]+d_vec1[i-19]+d_vec1[i-18]+d_vec1[i-17]+d_vec1[i-16]+
                 d_vec1[i-15]+d_vec1[i-14]+d_vec1[i-13]+d_vec1[i-12]+d_vec1[i-11]+d_vec1[i-10]+d_vec1[i-9]+d_vec1[i-8]+
                 d_vec1[i-7]+d_vec1[i-6]+d_vec1[i-5]+d_vec1[i-4]+d_vec1[i-3]+d_vec1[i-2]+d_vec1[i-1]+d_vec1[i];
  }
}
```

Anche in questo caso le prestazioni non sono molto alte anche per la quantità di elementi che non riesce a saturare la banda.
reduction: runtime 0.007648ms, 0.1339 GE/s, 0.5356 GB/s

4.3 Transpose

Per calcolare il passo di discesa abbiamo bisogno della trasposta della matrice p_k quindi andiamo a calcolarcela con questo kernel:

```
__global__
void traspose(int nrow,int ncols, float* __restrict__ res_vec, float* __restrict__ d_vec1)
{
  int c = threadIdx.x;
  int r=blockIdx.x;
  int l_in = r*ncols + c;
  int l_out = c * nrow + r;
  res_vec[l_out] = d_vec1[l_in];
}
```

Anche in questo caso vediamo la dipendenza con i la grandezza del kernel lanciato. Le sue prestazioni sono:
transpose: runtime 0.006656ms, 0.004808 GE/s, 0.01923 GB/s
considerando che dovrà trasporre semplicemente 32 elementi al massimo.

4.4 Scalare matrice

Dopo aver calcolato il passo di discesa si deve moltiplicare per la direzione di discesa, quindi, essendo il passo di discesa uno scalare, avremo uno scalare per la matrice. Quindi il kernel è cosi sviluppato:

```
__global__
void scalareMatrice(float* __restrict__ res_vec,float scalar,float* __restrict__ d_vec)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  res_vec[i]=d_vec[i]*scalar;
}
```

E le sue prestazioni considerando 32 elementi sono:
transpose: runtime 0.008ms, 0.004 GE/s, 0.016 GB/s

4.5 Vecsum e vecdif

Infine andiamo a guardare due kernel utili per la prima operazione di calcolo della direzione e per il calcolo di x_{k+1} quindi la somma e la differenza vediamo come sono stati sviluppati.

```
__global__
void vecsum(int nels, float* __restrict__ res_vec, float* __restrict__ d_vec1, float* __restrict__ d_vec2)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  res_vec[i] = d_vec1[i]+d_vec2[i];
}

__global__
void vecdif(int nels, float* __restrict__ res_vec, float* __restrict__ d_vec1, float* __restrict__ d_vec2)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  res_vec[i] = d_vec1[i]-d_vec2[i];
}
```

Molto semplicemente vado a recuperare l’indice linearizzato e vado a sommare gli elementi agli indici corrispondenti.
Prestazioni:
vecdif: runtime 0.00672ms, 0.004762 GE/s, 0.01905 GB/s
vecsum: runtime 0.006592ms, 0.004854 GE/s, 0.01942 GB/s

5-Versione2

Denominata versione2.cu porta notevoli miglioramenti soprattutto per la quantità di elementi che possiamo andare ad analizzare. Per prima cosa è stato introdotto un blockSize fisso e numBlocks che varia in maniera consequenziale al numero di elementi di cui abbiamo bisogno.
const int blockSize = 1024;
int numBlocks = (nels + blockSize – 1)/blockSize;
Vediamo come sono cambiati i kernel per seguire questa esigenza, quindi non dovranno più fare affidamento sul blockID e sul threadID ma soltanto sull’indice linearizzato.

5.1Multi_vec

Nel caso del prodotto si sono rivisti gli argomenti da passare e sono stati introdotte le dimensioni delle matrici così da andare a calcolare precisamente quali sono gli indici degli elementi da moltiplicare. Naturalmente è stato aggiunto anche un if per il controllo degli elementi, se siamo ancora all’interno delle dimensioni della matrice di uscita.

```
__global__
void multi_vec(int nels,int n_row1,int n_col1,int n_row2,int n_col2,float* __restrict__ res_vec,
               float* __restrict__ d_vec1,float* __restrict__ d_vec2)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int r_res,c_res;
  r_res=n_row1;
  c_res=n_row2*n_col2;
  if(i<(r_res*c_res)){
    int c= ((int)(i/c_res))*n_row1 + ((int)(i%n_col1))%n_col1;
    int j= ((int)(((int)(i%c_res))/n_row2) + (((int)(i%c_res))%n_row2)*n_col2);
    res_vec[i]=d_vec1[c]*d_vec2[j];
  }
}
```

Anche i due indici linearizzati ora non sono più dipendenti dal threadIdx e da blockIdx ma soltanto da ‘i’ e dalle dimensione della prima matrice e della seconda.

5.2 Reduction_row

Arriviamo adesso alla vera rivoluzione per questa versione ovvero la riduzione degli elementi dopo il prodotto. In questa versione è stato sviluppato un kernel più flessibile per la gestione della riduzione che quindi può ora pensare di accettare piu elementi l’unica restrizione è che siano potenze del 4,8,12,16. Vediamo come è stato sviluppato e come viene chiamata per capire il perché di queste restrizioni:

```
__global__
void reduction_row2(int nels,int l_elem,float* res_vec, float*  d_vec1)
{
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  const float4 noels = make_float4(0.0, 0.0, 0.0, 0.0);
  const int nquarts = nels*4;
  const int elem=nels/l_elem;
  int i=idx*l_elem;
  int i0 = i;
  int i1 = i +   4;
  int i2 = i + 2*4;
  int i3 = i + 3*4;
     __syncthreads();
  float4 r0;
  if(l_elem >= 4){
    r0.x=d_vec1[i];
    r0.y=d_vec1[i+1];
    r0.z=d_vec1[i+2];
    r0.w=d_vec1[i+3];
  }
  else r0= noels;
  float4 r1;
  if(l_elem >= 8){
    r1.x=d_vec1[i1];
    r1.y=d_vec1[i1+1];
    r1.z=d_vec1[i1+2];
    r1.w=d_vec1[i1+3];
  }
  else r1= noels;
  float4 r2;
  if(l_elem >= 12){
    r2.x=d_vec1[i2];
    r2.y=d_vec1[i2+1];
    r2.z=d_vec1[i2+2];
    r2.w=d_vec1[i2+3];
  }
  else r2= noels;
  float4 r3;
  if(l_elem >= 16){
    r3.x=d_vec1[i3];
    r3.y=d_vec1[i3+1];
    r3.z=d_vec1[i3+2];
    r3.w=d_vec1[i3+3];
  }
  else r3= noels;

  float4 v = (r0 + r1) + (r2 + r3);

  if (idx < nels)
    res_vec[idx] = (v.x + v.y) + (v.z + v.w);

}
```

Vediamo come al kernel passiamo l_elem che è la lunghezza degli elementi da ridurre ma in realtà è un thread_load cioè quanti elementi deve ridurre ogni work item. Ogni elemento poi viene aggiunto a un float4 e quindi tutti gli elementi dentro ogni float4 vengono sommati ed esce un unico numero risultante.
Vediamo però come per fare questo viene invocato il kernel e come viene scelto il thread_load:

```
c=N*N;
while (c>N) {
  c/=THREAD_LOAD;
  numBlocks = (c + blockSize - 1)/blockSize;
  reduction_row2<<<blockSize, numBlocks>>>(c,THREAD_LOAD,res2,res);
  err = cudaMemcpy(res, res2, c*sizeof(float), cudaMemcpyDeviceToDevice);
  cuda_check(err, "cpy");
  printf("%d\n",c );
}
```

Quindi vediamo che finché la matrice non avrà la lunghezza da noi desiderata lanciamo il
kernel e riduciamo i nostri elementi.
Il thread_load viene invece deciso tramite delle divisioni successive di N:

```
int THREAD_LOAD=0;
float n = N;
while (n > 1) {
  n/=4;
  if(n==1){
    THREAD_LOAD=4;
  }
}
n = N;
while (n > 1) {

  n/=8;
  if(n==1){
    THREAD_LOAD=8;
  }
}
n=N;
while (n > 1) {
  n/=12;
  if(n==1){
    THREAD_LOAD=12;
  }
}
while (n > 1) {
  n/=16;
  if(n==1){
    THREAD_LOAD=16;
  }
}
if(THREAD_LOAD==0){
  printf("Errore N deve essere una potenza di 4,8,12,16");
  exit(0);
```

5.3 Transpose

Nel kernel della trasposta è stato cambiato il riferimento agli indici linearizzati che non fanno riferimento al blockIdx e al threadIdx ma soltanto all'indice linearizzato principale cioè 'i'

```
__global__
void transpose(int nrow,int ncols, float* __restrict__ res_vec, float* __restrict__ d_vec1)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int c =i%ncols;
  int r=i/ncols;
  int l_in = r*ncols + c;
  int l_out = c * nrow + r;
  res_vec[l_out] = d_vec1[l_in];
}
```

5.4 Scalare matrice vecdif e vecsum

Non presentano cambiamenti in quanto dipendono soltanto dall'indice generale.

5.5 Confronto delle prestazioni

Possiamo adesso andare a confrontare le prestazioni dei vari kernel al variare del numero del blokSize. E per quanto riguarda la riduzione anche al variare del thread load. Block size 32:

	N=16	N=64
Init	0.01875ms, 0.01365 GE/s, 0.05461 GB/s (256 elementi)	0.0217ms, 0.1888 GE/s, 0.7552 GB/s (4096 elementi)
Prodotto	0.00832ms, 0.03077 GE/s, 0.1231 GB/s (256 elementi)	0.00912ms, 0.4491 GE/s, 1.796 GB/s (4096 elementi)
reduction	0.01523ms, 0.005252 GE/s, 0.02101 GB/s (80 elementi)	0.01635ms, 0.03523 GE/s, 0.1409 GB/s (576 elementi)
transpose	0.006688ms, 0.002392 GE/s, 0.009569 GB/s (16 elementi)	0.007584ms, 0.008439 GE/s, 0.03376 GB/s (64 elementi)
scalareMatrice	0.008672ms, 0.001845 GE/s, 0.00738 GB/s (16 elementi)	0.009152ms, 0.006993 GE/s, 0.02797 GB/s(64 elementi)
vecdif	0.006912ms, 0.002315 GE/s, 0.009259 GB/s (16 elementi)	0.006624ms, 0.009662 GE/s, 0.03865 GB/s (64 elementi)
vecsum	0.006816ms, 0.002347 GE/s, 0.00939 GB/s(16 elementi)	0.00688ms, 0.009302 GE/s, 0.03721 GB/s(64 elementi)

	N=144
Init	0.01974ms, 1.05 GE/s, 4.201 GB/s
Prodotto	0.02205ms, 0.9405 GE/s, 3.762 GB/s (256 elementi)
reduction	0.01994ms, 0.0939 GE/s, 0.3756 GB/s (80 elementi)
transpose	0.01405ms, 0.01025 GE/s, 0.041 GB/s(16 elementi)
scalareMatrice	0.008512ms, 0.01692 GE/s, 0.06767 GB/s (16 elementi)
vecdif	0.006752ms, 0.02133 GE/s, 0.08531 GB/s (16 elementi)
vecsum	0.009536ms, 0.0151 GE/s, 0.0604 GB/s(16 elementi)

Vediamo come il trend principale sia quello di aumento delle prestazioni all'aumentare degli elementi, in quanto riusciamo a saturare meglio la banda. Riusciamo ad non avere piu altri elementi in quanto dobbiamo procedere per multipli di 4 8 12 16 e andando avanti con un block size cosi piccolo superiamo presto il limite imposto dalla GPU(max threads per block=1024).

Proviamo adesso Block size 128:

	N=64	N=256
Init	0.02067ms, 0.1981 GE/s, 0.7926 GB/s (4096 elementi)	0.02339ms, 2.802 GE/s, 11.21 GB/s(65536 elementi)
Prodotto	0.009792ms, 0.4183 GE/s, 1.673 GB/s (4096 elementi)	0.0529ms, 1.239 GE/s, 4.956 GB/s(65536 elementi)
reduction	0.01734ms, 0.03321 GE/s, 0.1328 GB/s (576 elementi)	0.04221ms, 0.5155 GE/s, 2.062 GB/s(21760 elementi)
transpose	0.0088ms, 0.007273 GE/s, 0.02909 GB/s (64 elementi)	0.02714ms, 0.009434 GE/s, 0.03774 GB/s(256 elementi)
scalareMatrice	0.00992ms, 0.006452 GE/s, 0.02581 GB/s (64 elementi)	0.01104ms, 0.02319 GE/s, 0.09275 GB/s(256 elementi)
vecdif	0.007744ms, 0.008264 GE/s, 0.03306 GB/s (64 elementi)	0.0104ms, 0.02462 GE/s, 0.09846 GB/s(256 elementi)
vecsum	0.007648ms, 0.008368 GE/s, 0.03347 GB/s (64 elementi)	0.02141ms, 0.01196 GE/s, 0.04783 GB/s(256 elementi)

Vediamo come per 256 elementi il trend è migliorato notevolmente soprattutto per i primi kernel che sono quelli a cui passiamo più elementi.
Infine andiamo a vedere cosa succede con block size di 512:

	N=256	N=512
Init	0.02294ms, 2.856 GE/s, 11.43 GB/s(65536 elementi)	0.0511ms, 5.13 GE/s, 20.52 GB/s (262144 elementi)
Prodotto	0.05059ms, 1.295 GE/s, 5.182 GB/s(65536 elementi)	0.1892ms, 1.386 GE/s, 5.543 GB/s(262144 elementi)
reduction	0.05955ms, 0.3654 GE/s, 1.462 GB/s(21760 elementi)	0.1241ms, 0.3012 GE/s, 1.205 GB/s(37376 elelementi)
transpose	0.0265ms, 0.009662 GE/s, 0.03865 GB/s(256 elementi)	0.09363ms, 0.005468 GE/s, 0.02187 GB/s(512 elementi)
scalareMatrice	0.01325ms, 0.01932 GE/s, 0.07729 GB/s(256 elementi)	0.01354ms, 0.03783 GE/s, 0.1513 GB/s(512 elementi)
vecdif	0.01386ms, 0.01848 GE/s, 0.0739 GB/s(256 elementi)	0.01302ms, 0.03931 GE/s, 0.1572 GB/s(512 elementi)
vecsum	0.0217ms, 0.0118 GE/s, 0.0472 GB/s(256 elementi)	0.064ms, 0.008 GE/s, 0.032 GB/s(512 elementi)

Vediamo come il miglioramento continua per quanto riguarda l’init ma il trend per il resto dei kernel resta lo stesso o addirittura peggiora.