K-Means: Strategie di Ottimizzazione e Parallelizzazione

Mattia Pandolfi 2087310
Università di Roma *"La Sapienza"*

Introduzione

L'obiettivo dell'implementazione è ottimizzare il codice sequenziale dell'algoritmo K-Means utilizzando MPI, OpenMP e Cuda.
Vengono proposte due soluzioni, una soluzione ibrida unificando MPI+OpenMP, per sfruttare la memoria distribuita per spartire i dati tra i vari processi (o nodi) e la memoria condivisa per migliorare il parallelismo intra-processo, e una soluzione cuda, per servirsi della potenza di calcolo delle GPU Nvidia.

Analisi del codice

L'implementazione sequenziale K-Means segue questi passi:

1. Si generano K centroidi che rappresentano i cluster.
2. Assegnazione di ciascun punto al cluster più vicino.
3. Ricalcolo dei centroidi basato sui punti assegnati.
4. Ripetizione fino alla convergenza.

L'algoritmo necessita di alcuni parametri, come il numero di cluster, il numero minimo di cambiamenti e una threshold, utili per stabilire in quali casi l'algoritmo deve convergere.
Entreremo nel dettaglio su come sono stati realizzati i passi dell'algoritmo.
L'immagine sottostante mostra l'assegnazione di ciascun punto al cluster più vicino.

```
//1. Calculate the distance from each point to the centroid
//Assign each point to the nearest centroid.
changes = 0;
for(i=0; i<lines; i++)
{
        class=1;
        minDist=FLT_MAX;
        for(j=0; j<K; j++)
        {
                dist=euclideanDistance(&data[i*samples], &centroids[j*samples], samples);

                if(dist < minDist)
                {
                        minDist=dist;
                        class=j+1;
                }
        }
        if(classMap[i]!=class)
        {
                changes++;
        }
        classMap[i]=class;
}
```

Qui possiamo vedere che per ogni punto viene calcolata la distanza da ogni cluster e viene individuato il cluster più vicino ad esso.

Con un numero molto elevato di punti, il tempo di esecuzione può diventare estremamente lungo, rendendo necessario l'uso di parallelizzazione.

In questo secondo estratto di codice, possiamo vedere come l'algoritmo ricalcola i centroidi.
Anche qui è presente un ciclo che itera sui punti.

```
for(i=0; i<lines; i++)
{
        class=classMap[i];
        pointsPerClass[class-1] = pointsPerClass[class-1] +1;
        for(j=0; j<samples; j++){
                auxCentroids[(class-1)*samples+j] += data[i*samples+j];
        }
}
for(i=0; i<K; i++)
{
        for(j=0; j<samples; j++){
                auxCentroids[i*samples+j] /= pointsPerClass[i]; // atten
        }
}
```

Infine per ogni centroide, calcola la distanza euclidea tra il centroide attuale e il centroide ricalcolato e aggiorna maxDist che tiene traccia della distanza massima tra i centroidi delle due iterazioni, questa parte è necessaria per la convergenza dell'algoritmo.
I nuovi centroidi vengono copiati e pronti per la prossima iterazione.

```
maxDist=FLT_MIN;
for(i=0; i<K; i++){
       distCentroids[i]=euclideanDistance(&centroids[i*samples], &auxCentroids[i*samples], samples);
       if(distCentroids[i]>maxDist) {
              maxDist=distCentroids[i];
       }
}
memcpy(centroids, auxCentroids, (K*samples*sizeof(float)));
```

Sezioni ad alto costo computazionale

Dall'analisi del codice sequenziale mostrato, emergono chiaramente le due sezioni principali che sono considerate le più onerose a livello computazionale e che quindi richiedono un ottimizzazione mirata:

- ❖ L'assegnazione dei punti al cluster più vicino.
- ❖ Ricalcolo dei centroidi.

Infine, la verifica della convergenza rappresenta un passaggio leggermente meno oneroso ma comunque importante per il controllo del ciclo.

Implementazione MPI + OpenMP

La prima implementazione parallela proposta per l'algoritmo K-Means combina MPI e OpenMP, sfruttando i vantaggi sia della memoria distribuita sia di quella condivisa. La soluzione andrà ad ottimizzare le sezioni critiche individuate in precedenza.
È stato scelto il livello di supporto *MPI_THREAD_FUNNELED*, in cui solo il thread principale è autorizzato a effettuare chiamate MPI, mentre gli altri thread (gestiti da OpenMP) operano esclusivamente su dati locali.

Per prima cosa, vengono distribuiti i punti nel modo più equo possibile tra i vari processi MPI.

```
int baseNumPoints = lines / size;
int extraPoints = lines % size;
int startIndex = rank * baseNumPoints + (rank < extraPoints ? rank : extraPoints);
int endIndex = startIndex + baseNumPoints + (rank < extraPoints ? 1 : 0);
int localLines = endIndex - startIndex;
```

Questo frammento di codice rappresenta la **fase di assegnazione dei punti al cluster più vicino**, uno dei passaggi più costosi dell'algoritmo K-Means dal punto di vista computazionale.

```
changes = 0;
#pragma omp parallel for private(i,j,class,dist,minDist) reduction(+:changes) schedule(dynamic)
for(i=0; i<localLines; i++) {
    class=1;
    minDist=FLT_MAX;
    for(j=0; j<K; j++){
         dist=0.0;
         float *point = &data[(startIndex + i)*samples];
         float *center = &centroids[j*samples];
         for(int d=0; d<samples; d++) {
              dist+= (point[d]-center[d])*(point[d]-center[d]);
         }
         dist = sqrt(dist);
         if(dist < minDist){
              minDist=dist;
               class=j+1;
         }
    }
    if (localClassMap[i]!=class) {
          changes++;
    }
    localClassMap[i]=class;
}
MPI_Allreduce(MPI_IN_PLACE, &changes, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

In questa versione parallela viene utilizzata una direttiva OpenMP, il *#pragma omp parallel for* ,per parallelizzare il ciclo principale che itera su tutti i punti locali del processo MPI.

Il numero di punti vengono divisi tra i threads disponibili, ogni thread calcola la distanza euclidea tra il punto corrente e ciascun centroide, il punto viene assegnato al cluster il cui centroide è più vicino, se il cluster del punto cambia rispetto all'iterazione precedente, si incrementa la variabile *changes* , che tiene traccia del numero di cambiamenti avvenuti nell'assegnazione dei punti. Notiamo subito che changes rappresenta un problema, perché questa variabile è condivisa tra i vari threads e potrebbe essere aggiornata contemporaneamente da uno o più threads generando una possibile e probabile race condition. *La direttiva reduction(+:changes)* garantisce che il conteggio delle modifiche venga sommato correttamente tra tutti i thread.

Infine con la chiamata *MPI_Allreduce*, i cambiamenti locali vengono sommati tra tutti i processi MPI per ottenere il valore globale di changes, necessario per il controllo della convergenza a livello distribuito.

Confrontando il codice parallelo con quello sequenziale osserviamo che è stata rimossa la funzione per il calcola della distanza euclidea e il calcolo è stato implementato direttamente all'interno del ciclo principale. Questa scelta è motivata dalla necessità di ridurre l'overhead associato alle chiamate di funzione, che, in presenza di un elevato numero di punti da elaborare, può avere un impatto significativo sulle prestazioni complessive del programma. Poiché l'algoritmo esegue il calcolo della distanza migliaia (o milioni) di volte, evitare la chiamata a una funzione esterna consente di risparmiare tempo ed eseguire il codice in modo più efficiente.

Da notare l'utilizzo della clausola *schedule(dynamic),* le iterazioni del ciclo non vengono assegnate in blocco fisso ai thread all'inizio, ma vengono distribuite dinamicamente a runtime, il che permette un migliore bilanciamento del carico di lavoro, evitando che alcuni thread restino inattivi mentre altri lavorano.

Questo frammento di codice rappresenta la **fase di ricalcolo dei centroidi basato sui punti assegnati e verifica della convergenza.**

```
#pragma omp parallel
{
   zeroIntArray(pointsPerClass,K);
   zeroIntArray(localpointsPerClass,K);
   zeroFloatMatriz(auxCentroids,K,samples);
   zeroFloatMatriz(localauxCentroids,K,samples);

   #pragma omp barrier

   #pragma omp for reduction(+: localpointsPerClass[0:K]) reduction(+: localauxCentroids[0:K*samples])
   for (int i = 0; i < localLines; i++) {
       int cls = localClassMap[i];
       localpointsPerClass[cls - 1]++;
       for (int j = 0; j < samples; j++) {
           localauxCentroids[(cls - 1) * samples + j] += data[(i + startIndex) * samples + j];
       }
   }
}

MPI_Allreduce(localpointsPerClass, pointsPerClass, K, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(localauxCentroids, auxCentroids, K*samples, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
#pragma omp parallel shared(maxDist)
{

   #pragma omp for private(i,j) schedule(static, 16)
   for(int t=0; t<K*samples; t++){
       i = t / samples;
       j = t - i*samples;
       auxCentroids[i*samples+j] /= pointsPerClass[i];
   }
   #pragma omp single
   {
       maxDist=FLT_MIN;
   }
   #pragma omp for reduction(max:maxDist)
   for(i=0; i<K; i++){
      distCentroids[i]=euclideanDistance(&centroids[i*samples], &auxCentroids[i*samples], samples);
      if(distCentroids[i]>maxDist) {
           maxDist=distCentroids[i];
      }
   }
   cpyarray(centroids, auxCentroids, K*samples);
}
```

Le funzioni per azzerare i vettori e le matrici (che contengono i conteggi e le somme delle coordinate per ciascun cluster) e il primo ciclo for vengono aggiunti in una regione parallela comune, questo approccio consente di evitare la continua attivazione e terminazione dei thread. Nel for ogni thread elabora una porzione dei punti locali. Per ciascun punto viene incrementato il conteggio di punti per il cluster a cui appartiene e vengono sommate le coordinate del punto ai valori del centroide corrispondente.
Le operazioni di somma sono protette tramite direttive reduction, evitando race condition.

Dopo la computazione locale, le somme e i conteggi vengono aggregati tra tutti i processi MPI tramite *MPI_Allreduce*, così ogni processo ottiene il numero totale di punti per ciascun cluster e la somma totale delle coordinate.

Dopo aver combinato i risultati parziali troviamo subito una nuova regione parallela. Qui si calcolano i nuovi centroidi dividendo la somma totale delle coordinate per il numero di punti in ciascun cluster.
Per permettere una migliore suddivisione del carico di lavoro tra i threads, i due cicli for annidati vengono collassati in uno solo.

```
for(i=0; i<K; i++){
   for(j=0; j<samples; j++){
      auxCentroids[i*samples+j] /= pointsPerClass[i];
   }
}
```

```
#pragma omp for private(i,j) schedule(static)
for(int t=0; t<K*samples; t++){
   i = t / samples;
   j = t - i*samples;
   auxCentroids[i*samples+j] /= pointsPerClass[i];
}
```

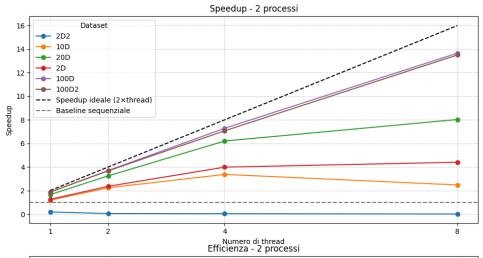
Giunti alla parte finale, la verifica della convergenza avviene nel seguente modo, per ogni cluster si calcola la distanza euclidea tra il vecchio e il nuovo centroide e maxDist viene aggiornata con il valore massimo tra tutte le distanze trovate usando *reduction(max:maxDist).*
Infine, i nuovi centroidi vengono copiati nei centroidi ufficiali per l'iterazione successiva.
Da notare che *maxDist=FLT_MIN;* viene eseguita da un solo thread (il primo che arriva all'istruzione) perché è una variabile condivisa tra tutti i thread e non necessita che ogni thread vada a modificarlo.

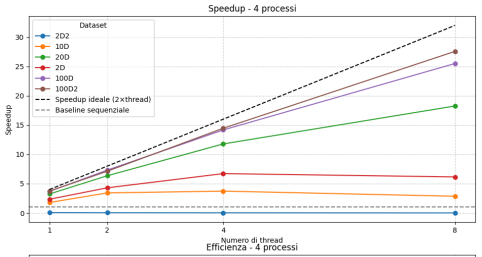
Nella soluzione parallela sono presenti diverse sincronizzazioni tra i thread e tra i processi, sia implicite che esplicite, al fine di garantire la correttezza e la robustezza del codice durante l'esecuzione concorrente.

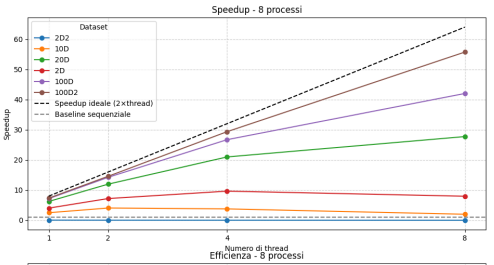
Il ciclo *while* termina non appena una delle condizioni di uscita non è più soddisfatta. In tal caso avviene la fase di *post-while* dove ci rimane di raccogliere tutti i risultati finali dei vari processi nel processo 0 (processo master).

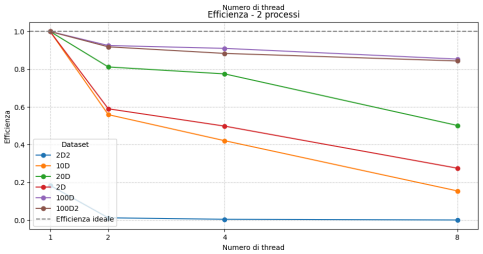
```
int recvcounts[size];  // Numero di elementi per ogni processo
int displs[size];      // Offset di inizio per ogni processo
// Calcola recvcounts e displs da usare in MPI_Allgatherv
displs[0] = 0;
for (int i = 0; i < size; i++) {
     recvcounts[i] = (lines / size) + (i < (lines % size) ? 1 : 0);
}
// Eseguire la raccolta dei dati locali in classMap globale di rank 0 da salvare sul file
MPI_Gatherv(localClassMap, localLines, MPI_INT, classMap, recvcounts, displs, MPI_INT, 0 ,MPI_COMM_WORLD);
```

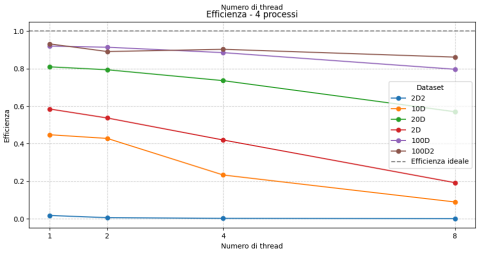
SpeedUp & Efficiency

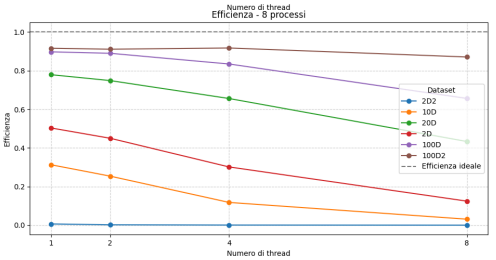




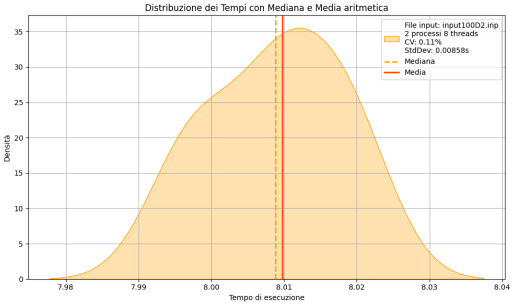


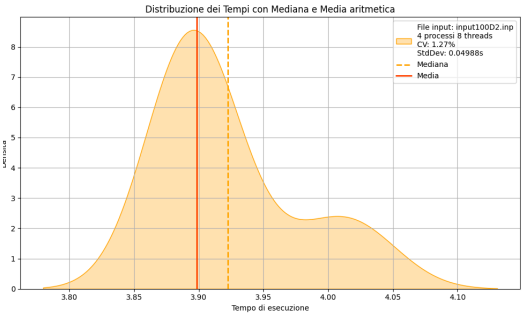


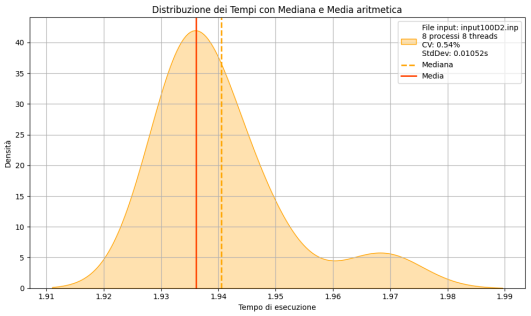




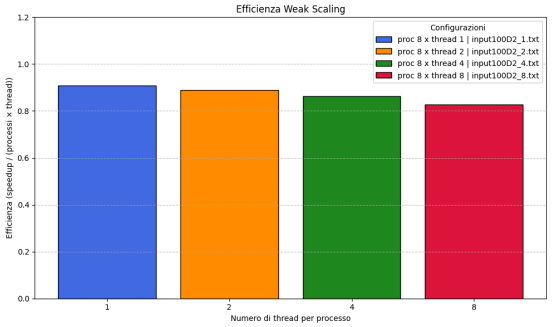
Time Distribution

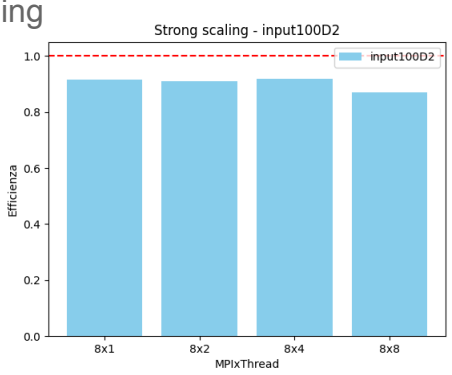






Weak & Strong Scaling





Nell’immagine a destra, per un input abbastanza grande notiamo che l’efficienza rimane molto alta e costante quindi possiamo dire che è strong scaling. Invece nell’immagine di sinistra possiamo vedere che l’efficienza varia leggermente e diminuisce all’aumentare della grandezza dell’input e del numero di thread.

Considerazioni tecniche e Problemi incontrati

Ho provato a utilizzare l'istruzione FMA nel ciclo *for* per la distanza euclidea, poiché la sua struttura - una somma di prodotti quadratici *(a*b+c)* - è compatibile con questa ottimizzazione. Ci si aspettava una maggiore precisione e maggiore velocità. Tuttavia, nonostante la potenziale maggiore precisione numerica, ho riscontrato un aumento nei tempi di esecuzione. Di conseguenza, ho deciso di rinunciare all'uso di FMA, privilegiando le prestazioni complessive rispetto all’accuratezza.

Nell’implementazione originale veniva usato *outputMsg = strcat(outputMsg, line);* per concatenare l’andamento del codice ad ogni iterazione. Però, quando il numero di punti era molto grande, il programma andava in crash. Probabilmente succedeva perché la memoria allocata per *outputMsg* non era sufficiente a contenere la nuova stringa concatenata, causando un buffer overflow o un errore di memoria.

Il livello di supporto al threading scelto, come detto in precedenza, è *MPI_THREAD_FUNNELED.* Inizialmente non avendo specificato il livello da adottare nel programma, MPI ha usato il livello di default *MPI_THREAD_SINGLE*, che non garantisce la thread safety. Di conseguenza il comportamento diventava indefinito e poteva causare crash o errori difficili da diagnosticare.

Implementazione Cuda

L’implementazione CUDA dell’algoritmo K-means utilizza una configurazione di thread block e grid per massimizzare il parallelismo e sfruttare al meglio l’architettura GPU. In particolare viene scelto un blocco di thread con dimensione fissa di 64 thread per blocco.
Per il calcolo del numero di blocchi per il kernel CUDA, vengono utilizzate tre diverse configurazioni di griglia, corrispondenti a tre diversi step di elaborazione nell’algoritmo

```
dim3 blockDim(64);
dim3 gridDim1((lines + blockDim.x -1) / blockDim.x);
dim3 gridDim2((K * samples + blockDim.x - 1) / blockDim.x);
dim3 gridDim3((K + blockDim.x - 1) / blockDim.x);
```

Il primo Kernel CUDA sviluppato *KernelClusterAssignment* implementa la fase di **assegnazione dei punti al cluster più vicino** e una parte del ricalcolo dei centroidi dell'algoritmo K-means. Ogni thread CUDA gestisce l’elaborazione di un singolo punto dati, identificato dall’indice globale.

```
int id = blockIdx.x * blockDim.x + threadIdx.x;
```

All’inizio viene dichiarata una porzione di shared memory dinamica come array di unsigned char. Successivamente, si reinterpretano porzioni di questa memoria per usarla con tipi diversi, prima si alloca spazio per un intero, poi si riversa spazio per due array di float.

```
extern __shared__ unsigned char sharedMemory[];
int* sharedChanges = (int*)&sharedMemory[0];
float* sharedCentroids = (float*)&sharedMemory[sizeof(int)];
float* localLines = (float*)&sharedMemory[sizeof(int) + sizeof(float) * d_K * d_samples];
```

Successivamente, ogni elemento dell'array d_centroids viene copiato in sharedCentroids (memoria condivisa), e i dati di d_data vengono copiati in localLines, anch'essa allocata nella memoria condivisa. Lo scopo di questo trasferimento è sfruttare la maggiore velocità di accesso della shared memory rispetto alla global memory.

Ogni thread si occupa di un singolo punto dati e ne calcola la distanza da tutti i centroidi. I dati dei punti e dei centroidi sono stati precedentemente copiati in memoria condivisa, che è molto più veloce. Questo migliora significativamente le prestazioni.

E’ stato necessario l’utilizzo di atomicAdd su sharedChanges per permetterci di contare in modo sicuro quante modifiche di cluster sono avvenute dentro il blocco, successivamente questa istruzione verrà riutilizzata per avere la somma totale tra tutti i blocchi.

```
if (id < d_lines) {
   float minDist = FLT_MAX;
   int cluster = 1;
   for (int j = 0; j < d_K; j++) {
       float dist = CudaEuclideanDistance(&localLines[tid * d_samples], &sharedCentroids[j * d_samples], d_samples);
       if (dist < minDist) {
           minDist = dist;
           cluster = j + 1;
       }
   }
   if (d_classMap[id] != cluster) {
       atomicAdd(sharedChanges, 1);
   }
   d_classMap[id] = cluster;
}
__syncthreads();
```

Poiché una parte del ricalcolo dei centroidi richiede l'accesso all'array data[] (cioè ai punti), ho deciso di integrare questa fase direttamente nel kernel appena sviluppato, sfruttando il fatto che i punti sono già disponibili in memoria condivisa tramite localLines[]. In questo modo, si evita un ulteriore accesso alla global memory, migliorando l'efficienza complessiva del kernel. Anche è stato necessario l'uso di atomicAdd per evitare che due thread modifichino contemporaneamente.

```
if (id < d_lines) {
   int cluster = d_classMap[id];
   atomicAdd(&d_pointsPerClass[cluster - 1], 1);
   for (int j = 0; j < d_samples; j++) {
       atomicAdd(&d_auxCentroids[(cluster - 1) * d_samples + j], localLines[tid * d_samples + j]);
   }
}
__syncthreads();
```

La dimensione della shared memory viene calcolata in questo modo:

```
int sharedMemSize_1 = sizeof(int) + (K * samples * sizeof(float)) + (blockDim.x * samples * sizeof(float));
```

La dimensione della memoria condivisa non è illimitata, nella GPU utilizzata si ha 48KB per blocco che non è molto, quindi bisogna cercare di utilizzarla saggiamente senza superare i limiti disponibili.

In questo caso osservando sharedMemSize_1 possiamo intuire che scegliere K, Samples o il numero di thread per blocco troppo grande porterà inevitabilmente a superare il limite. Durante lo sviluppo sono state prese le seguenti considerazioni: K molto piccolo (es. 30 o 40) e Samples non più grande di 100.

Successivamente viene creato un secondo Kernel. Questo ciclo aggiorna i centroidi calcolando la media delle coordinate dei punti assegnati a ciascun cluster.

```
__global__ void KenelUpdateCentroids(float* auxCentroids, int* pointsPerClass, int K, int samples){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < K*samples) {
        int i = idx / samples;
        int j = idx - i * samples;
        auxCentroids[i*samples+j] /= pointsPerClass[i];
    }
}
```

Da notare, anche qui i due cicli *for* nel codice originale sono stati collassati in uno solo per migliorare la parallelizzazione.

Il kernel KernelMaxDistance serve a calcolare la massima distanza tra i nuovi centroidi (auxCentroids) e i vecchi centroidi (centroids) per ciascun cluster, ed è utile ad esempio come criterio di convergenza nel K-means (cioè per capire se i centroidi stanno ancora cambiando significativamente).

```
__global__ void KernelMaxDistance(float* centroids, float* auxCentroids, int K, int samples, float *maxDist){
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    int tid = threadIdx.x;
    __shared__ float localMax;
    float dist;
    if(tid == 0) {
        localMax = 0;
    }
    __syncthreads();
    if(id < K){
        dist = CudaEuclideanDistance(&auxCentroids[id * samples], &centroids[id * samples], samples);
        atomicMax((int*)&localMax, __float_as_int(dist));
    }
    __syncthreads();
    if(tid == 0) {
         atomicMax((int*)maxDist, __float_as_int(localMax)); a
    }

}
```

Per garantire il corretto funzionamento ho utilizzato atomicMax per trovare la distanza maggiore. Tuttavia, poiché atomixMax è pensata per lavorare con valori interi, ho dovuto apportare un cambiamento. Il valore float viene rappresentato come intero con __float_as_int() perché atomicMax accetta solo interi. La conversione bitwise è sicura qui perché le distanze sono sempre positive.

Anche qui è stato utilizzato l'uso della shared memory per i valori locali maxDist.

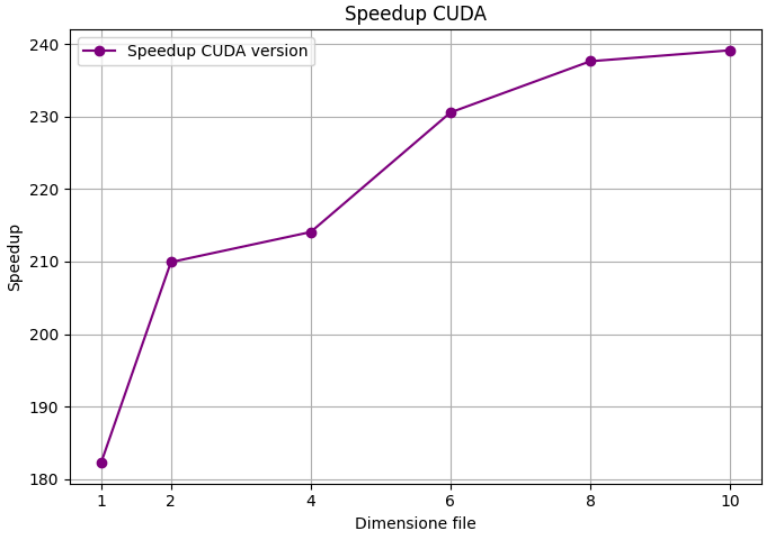
In questa implementazione CUDA è stata posta particolare attenzione a minimizzare il numero di trasferimenti dati tra host e device, perché il trasferimento da GPU a CPU avviene su PCI Express che ha latenza alta per trasferimenti piccoli e banda limitata.

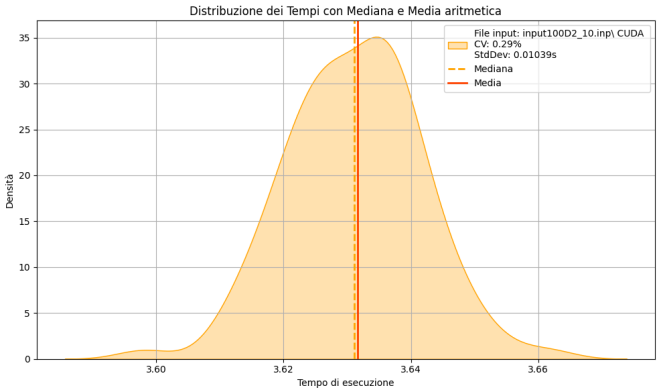
```
CHECK_CUDA_CALL(cudaMemcpy(&changes, d_changes, sizeof(int), cudaMemcpyDeviceToHost));
```

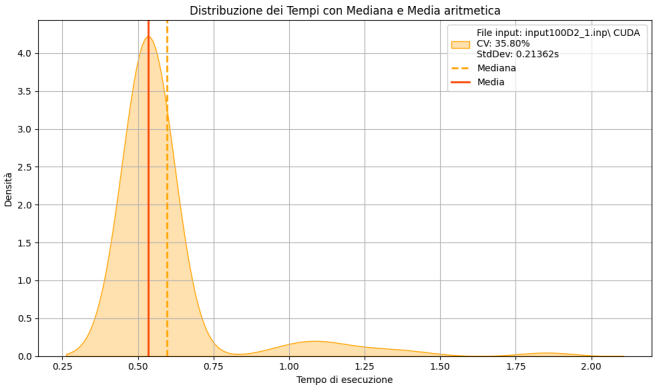
```
CHECK_CUDA_CALL(cudaMemcpy(&maxDist, d_maxDist, sizeof(float), cudaMemcpyDeviceToHost));
```

Performance

Nell'immagine sottostante possiamo vedere un grafico che ci mostra lo speed up della versione cuda di kmeans. Tutti i test e le misurazioni delle prestazioni sono stati condotti su una GPU NVIDIA RTX 6000. Il grafico mostra un miglioramento dello speed up quando si hanno molti punti.







Dai grafici di distribuzione dei tempi, nel primo è stato usato un file con 1,000,000 di punti mentre nel secondo 100,000 punti, possiamo vedere che nel primo la media e la mediana sono più vicini rispetto al secondo.

Considerazioni tecniche e Problemi incontrati

Inizialmente ho implementato l'operazione *atomicMax* direttamente su variabili di tipo *float*, senza applicare il trucco di reinterpretare i bit come interi. Tuttavia, CUDA non supporta nativamente *atomicMax* su *float*, quindi questa soluzione non garantiva risultati corretti.

Per garantire la corretta comparazione dei risultati in output con la versione sequenziale/MPI+OMP, ho disattivato esplicitamente l'ottimizzazione FMA (fused multiply-add) durante la compilazione.

Inizialmente esisteva un quarto Kernel Cuda per il ricalcolo dei centroidi, però si è notato che l'array data[] era già in shared memory.

Un possibile miglioramento futuro, per provare ad evitare di esaurire la shared memory nel Kernel KernelClusterAssignment e possibile applicare il tiling su sharedCentroids. L'idea è di suddividere il problema in sottoinsiemi più piccoli di centroidi, caricandone solo una parte alla volta nella shared memory e processandoli sequenzialmente. Questo approccio permette di ridurre la quantità di memoria condivisa necessaria per ciascun blocco, evitando di esaurirla e consentendo una maggiore scalabilità del kernel.