Implementazione in Open-MP e CUDA dell'algoritmo K-Means

Marco Calamai, *Elia Mercatanti*

Università degli Studi di Firenze
Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Anno Accademico 2019-2020

L'algoritmo K-means

1. Inizializzazione centroidi
2. Assegnamento dei punti ai clusters
3. Aggiornamento dei centroidi
4. Riperi 2 e 3 fino a quando gli assegnamenti non cambiano

Implementazione sequenziale K-Means

Strutture utilizzate

- Struct Point per i punti
  - int cluster_id
  - vector ⟨int⟩ dimensions
- Vector⟨Point⟩ dataset per il dataset
- Vector⟨Point⟩ centroids per i centroidi scelti random dal dataset comuni a tutte le versioni

Implementazione sequenziale K-Means - struttura

```cpp
std::tuple<std::vector<Point>, std::vector<Point>>
sequential_kmeans(std::vector<Point> dataset, const int num_clusters, std::vector<Point> centroids) {
    bool convergence = false;
    std::vector<Point> old_dataset = dataset;

    do {
        // Assignment phase, find the nearest centroid for all points, assign the point to that cluster.
        points_assignment( &dataset, num_clusters, centroids);

        // Update phase, calculate mean of all points assigned to that cluster, for all cluster.
        update_centroids(dataset, num_clusters, &centroids);

        // Check if the convergence criterion has been reached.
        if (check_convergence(dataset, old_dataset)) {
            convergence = true;
        } else {
            old_dataset = dataset;
        }
    } while (!convergence);

    return {dataset, centroids};
}
```

Implementazione sequenziale K-Means - assegnamento

```
void points_assignment(std::vector<Point> &dataset, const int num_clusters, const std::vector<Point> &centroids) {
    double min_distance, distance;
    int cluster_id;

    for (auto i = 0; i < dataset.size(); i++) {
        min_distance = std::numeric_limits<double>::max();
        for (auto j = 0; j < num_clusters; j++) {
            distance = calculate_distance(dataset[i].dimensions, centroids[j].dimensions);
            if (distance < min_distance) {
                min_distance = distance;
                cluster_id = centroids[j].cluster_id;
            }
        }
        dataset[i].cluster_id = cluster_id;
    }
}
```

Implementazione sequenziale K-Means - aggiornamento

```cpp
void update_centroids(const std::vector<Point> &dataset, const int num_clusters, std::vector<Point> &centroids) {
    const auto num_dimensions = dataset[0].dimensions.size();
    std::vector<int> num_points_clusters(num_clusters, _Val: 0);

    // Reset the centroids.
    for (auto i = 0; i < num_clusters; i++) {
        for (auto j = 0; j < num_dimensions; j++) {
            centroids[i].dimensions[j] = 0;
        }
    }

    // Calculate the sums and total number of points in each cluster.
    for (auto i = 0; i < dataset.size(); i++) {
        for (auto j = 0; j < num_dimensions; j++) {
            centroids[dataset[i].cluster_id].dimensions[j] += dataset[i].dimensions[j];
        }
        num_points_clusters[dataset[i].cluster_id]++;
    }

    // Calculate the new centroids, for all cluster.
    for (auto i = 0; i < num_clusters; i++) {
        for (auto j = 0; j < num_dimensions; j++) {
            centroids[i].dimensions[j] /= num_points_clusters[i];
        }
    }
}
```

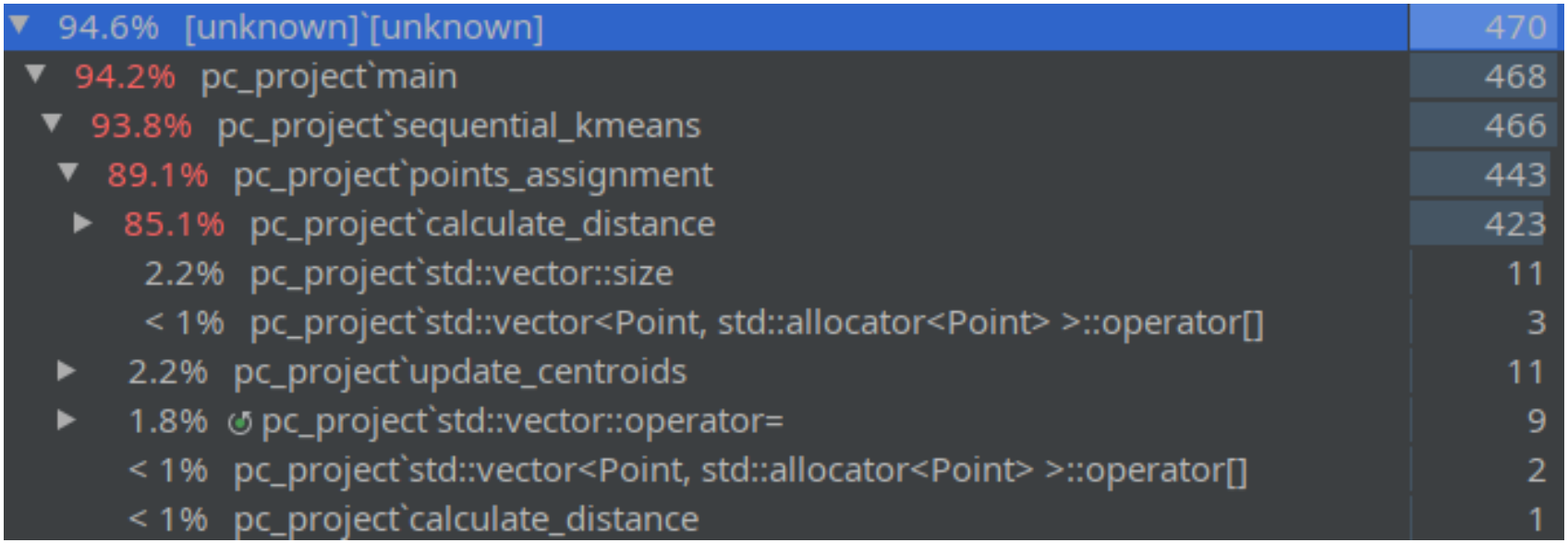
Implementazione sequenziale K-Means - criterio d'arresto

```cpp
bool check_convergence(const std::vector<Point> &dataset, const std::vector<Point> &old_dataset) {
    for (auto i = 0; i < dataset.size(); i++) {
        if (dataset[i].cluster_id != old_dataset[i].cluster_id) {
            return false;
        }
    }
    return true;
}
```

Implementazione parallela K-Means - benefici

- Fase di assegnamento, operazioni indipendenti
  - Calcolo distanze
  - Determinazione centroide più vicino
- Fase di aggiornamento
  - Eseguendo le somme in modo atomico

Analisi algoritmo K-Means tramite profiler



K-Means in Open-Mp

```
#pragma omp parallel default(none) shared(dataset, num_clusters, centroids, num_points_clusters)
```

- Sezione parallela che ingloba i due step del k-means
  - variabili shared: dataset, num clusters, centroids, num points clusters come
  - variabili private: min distance, distance e cluster id

K-Means in Open-Mp - assegnamento

```
void
points_assignment_openmp(std::vector<Point> &dataset, const int num_clusters, const std::vector<Point> &centroids) {
    double min_distance, distance;
    int cluster_id;

#pragma omp for
    for (auto i = 0; i < dataset.size(); i++) {
        min_distance = std::numeric_limits<double>::max();
        for (auto j = 0; j < num_clusters; j++) {
            distance = compute_distance_openmp(dataset[i].dimensions, centroids[j].dimensions);
            if (distance < min_distance) {
                min_distance = distance;
                cluster_id = centroids[j].cluster_id;
            }
        }
        dataset[i].cluster_id = cluster_id;
    }
}
```

K-Means in Open-Mp - aggiornamento

```cpp
void update_centroids_openmp(const std::vector<Point> &dataset, const int num_clusters, std::vector<Point> &centroids,
                             std::vector<int> &num_points_clusters) {
    const auto num_dimensions = dataset[0].dimensions.size();

    // Reset shared num_points_clusters.
#pragma omp for
    for (auto i = 0; i < num_clusters; i++) {
        num_points_clusters[i] = 0;
    }

    // Reset the centroids.
#pragma omp for collapse(2)
    for (auto i = 0; i < num_clusters; i++) {
        for (auto j = 0; j < num_dimensions; j++) {
            centroids[i].dimensions[j] = 0;
        }
    }

    // Calculate the sums and total number of points in each cluster.
#pragma omp for
    for (auto i = 0; i < dataset.size(); i++) {
        for (auto j = 0; j < num_dimensions; j++) {
#pragma omp atomic
            centroids[dataset[i].cluster_id].dimensions[j] += dataset[i].dimensions[j];
        }
#pragma omp atomic
        num_points_clusters[dataset[i].cluster_id]++;
    }

    // Calculate the new centroids, for all cluster.
#pragma omp for collapse(2)
    for (auto i = 0; i < num_clusters; i++) {
        for (auto j = 0; j < num_dimensions; j++) {
            centroids[i].dimensions[j] /= num_points_clusters[i];
        }
    }
}
```

K-Means in CUDA - struttura

Strutture dati

- Dataset e centroidi array bidimensionali
- Assegnamenti in array

Due kernel per la fase di assegnamento e tre kernel per l'aggiornamento

- fase di sincronizzazione tra ognuno

K-Means in CUDA - assegnamento - compute distances

```
__global__ void
compute_distances(const double *device_dataset, const double *device_centroids, double *device_distances) {
    unsigned int col = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int row = blockIdx.y * blockDim.y + threadIdx.y;
    double distance = 0;

    if (row < const_num_points && col < const_num_clusters) {
        for (auto i = 0; i < const_num_dimensions; i++) {
            distance += pow(
                    base: device_dataset[row * const_num_dimensions + i] - device_centroids[col * const_num_dimensions + i],
                    iexp: 2);
        }
        device_distances[row * const_num_clusters + col] = sqrt(distance);
    }
}
```

K-Means in CUDA - assegnamento - points assignment

```
__global__ void points_assignment(const double *device_distances, short *device_assignments) {
    int unsigned thread_id = blockDim.x * blockIdx.x + threadIdx.x;
    auto min_distance = INFINITY;
    double distance;
    short cluster_id;

    if (thread_id < const_num_points) {
        for (auto i = 0; i < const_num_clusters; i++) {
            distance = device_distances[thread_id * const_num_clusters + i];
            if (distance < min_distance) {
                min_distance = distance;
                cluster_id = i;
            }
        }
        device_assignments[thread_id] = cluster_id;
    }
}
```

K-Means in CUDA - aggiornamento - initialize centroids

```
__global__ void initialize_centroids(double *device_centroids) {
    int unsigned col = blockDim.x * blockIdx.x + threadIdx.x;
    int unsigned row = blockDim.y * blockIdx.y + threadIdx.y;

    if (row < const_num_clusters && col < const_num_dimensions) {
        device_centroids[row * const_num_dimensions + col] = 0;
    }
}
```

K-Means in CUDA - aggiornamento - compute sums

```
__global__ void compute_sums(double *device_centroids, const double *device_dataset, const short *device_assignments,
                             int *device_num_points_clusters) {
    int unsigned col = blockDim.x * blockIdx.x + threadIdx.x;
    int unsigned row = blockDim.y * blockIdx.y + threadIdx.y;

    if (row < const_num_points && col < const_num_dimensions) {
        short cluster_id = device_assignments[row];
        doubleAtomicAdd( address: &(device_centroids[cluster_id * const_num_dimensions + col]),
                         val: device_dataset[row * const_num_dimensions + col]);
        atomicAdd(&(device_num_points_clusters[cluster_id]), val: 1);
    }
}
```

K-Means in CUDA - aggiornamento - update centroids

```
__global__ void update_centroids(double *device_centroids, const int *device_num_points_clusters) {
    int unsigned col = blockDim.x * blockIdx.x + threadIdx.x;
    int unsigned row = blockDim.y * blockIdx.y + threadIdx.y;

    if (row < const_num_clusters && col < const_num_dimensions) {
        device_centroids[row * const_num_dimensions + col] =
                device_centroids[row * const_num_dimensions + col] /
                (double(device_num_points_clusters[row]) / const_num_dimensions);
    }
}
```

Test e Risultati

- Test eseguiti su CPU quad core i7 3770K e GPU Nvidia GTX 1080.
- Centroidi iniziali selezionati in modo random dal dataset e riutilizzati per l'esecuzione di ogni versione del K-Means.
- Per ogni versione è stato calcolato il tempo di esecuzione e lo speed-up ottenuto rispetto alla versione sequenziale.
- I test sono stati eseguiti sia su datasets random che su datasets reali.
- Datasets random generati variando numero di punti, numero di dimensioni e numero di cluster da ricercare tra 10, 100 e 1000.

Analisi performance su dataset random

DATASET

# CLUSTERS		Random 1000x10	Random 1000x100	Random 1000x1000	Random 10000x10	Random 10000x100	Random 10000x1000	Random 100000x10	Random 100000x100	Random 100000x1000	Random 1000000X10
10	**Sequenziale**	0.0995483	0.638251	2.03217	10.2006	52.0753	134.221	162.108	2622.35	14810.8	3873.36
	Open-MP	0.0426349	0.173283	0.547636	3.46449	14.405	38.6007	55.2159	710.503	4081.3	1320.14
	Speedup Open-MP	2.33	3.68	3.71	2.94	3.62	3.48	2.94	3.69	3.63	2.93
	CUDA	0.00684048	0.0262636	0.0918535	0.823457	0.988186	2.76661	24.9337	45.3077	289.722	584.845
	Speedup CUDA	14.55	24.30	22.12	12.39	52.70	48.51	6.50	57.88	51.12	6.62
	# ITERAZIONI	22	16	5	228	130	33	363	639	366	861
100	**Sequenziale**	0.390759	2.33382	7.84521	29.5799	150.701	705.307	940.892	-	-	-
	Open-MP	0.102776	0.595576	1.99878	7.78527	38.7765	180.048	248.872	-	-	-
	Speedup Open-MP	3.80	3.92	3.92	3.80	3.89	3.92	3.78			
	CUDA	0.00773597	0.0408883	0.13143	0.489796	2.22139	10.0715	14.3992	-	-	-
	Speedup CUDA	50.51	57.08	59.69	60.39	67.84	70.03	65.34			
	# ITERAZIONI	10	6	2	76	39	18	242	-	-	-
1000	**Sequenziale**	0.786591	7.8893	78.2989	65.8951	733.294	1555.92	3840.99	-	-	-
	Open-MP	0.198559	1.98681	19.8394	16.776	189.338	401.735	995.712	-	-	-
	Speedup Open-MP	3.96	3.97	3.95	3.93	3.87	3.87	3.86			
	CUDA	0.0147833	0.134762	1.15328	1.00158	10.5292	21.8604	57.8113	-	-	-
	Speedup CUDA	53.21	58.54	67.89	65.79	69.64	71.18	66.44			
	# ITERAZIONI	2	2	2	17	19	4	99	-	-	-

Analisi performance su dataset reali

DATASET

With K clusters

	IRIS K=3	S1-Set K=15	Birch1-set K=100	Letter K=26	worms_2d K=35	worms_64d K=35
Sequenziale	0.000429641	0.113001	82.724	36.3694	35.5275	579.928
Open-MP	0.000396073	0.0519949	24.2833	10.2238	12.6263	152.624
Speedup Open-MP	1.08	2.17	3.41	3.56	2.81	3.80
CUDA	0.000340643	0.0223294	2.29935	0.774311	3.70815	37.303
Speedup CUDA	1.26	5.06	35.98	46.97	9.58	15.55
# ITERAZIONI	3	14	99	109	106	63

DATASETS DESCRIPTIONS

DATESET	# PUNTI	# DIMENSIONI	#CLUSTERS
IRIS	150	4	3
S1-SET	5000	2	15
BIRCH1-SET	100000	2	100
LETTER	20000	16	26
WORMS_2D	105600	2	35
WORMS_64D	105600	64	35