Final Report

Parallel k-means clustering

이름 : 김민수, Student id : 2025-22681

1. The problem statement and motivation

   K-Means 클러스터링 알고리즘은 간결하고 효율적인 알고리즘이지만, Assignment step 에서 모든 데이터 포인트와 모든 중심점 간의 유클리드 거리를 반복적으로 계산해야 하므로, 데이터의 크기 N이 증가하거나 차원 M이 증가할 경우 계산 비용이 크게 증가합니다. 다만 이 알고리즘은 많은 부분이 독립적으로 연산되기에 data parallelism을 적용하면 가속화하기 좋고 수업시간에 배운 다양한 기법들을 적용하기 좋을 것으로 생각되었습니다.

2. Brief summary of existing work in the literature, with citations

   A. Warp centric 병렬화 관련 연구들.

      기존 GPU 기반 K-means 구현은 주로 thread-centric 방식으로, 각 스레드가 하나의 데이터 포인트에 대해 모든 클러스터 중심과의 거리를 계산한다. 이러한 방식은 고차원 데이터에서 메모리 접근 비효율과 warp divergence로 인해 성능 저하가 발생한다. 이를 해결하기 위해 *Parallel K-means on GPU using Warp-Centric Strategies* 에서는 은 warp-centric 병렬화 전략을 제안하며, warp를 기본 연산 단위로 사용한다. 하나의 warp가 하나의 데이터 포인트를 공동으로 처리하고, 각 스레드는 feature 차원의 일부를 담당하여 거리 계산을 수행한다. 이후 warp-level reduction을 통해 부분 결과를 효율적으로 합산함으로써 동기화 오버헤드를 줄인다. 실험 결과, 제안 기법은 특히 고차원 데이터셋에서 기존 thread-centric GPU K-means 대비 높은 성능 향상을 보인다.

   B. 희소 선형대수 연산을 활용한 연구들.

      Kernel K-means와 같이 계산 및 메모리 비용이 큰 알고리즘을 대상으로, 연산을 선형대수 형태로 변환하여 가속하는 접근들이 제안되고 있습니다. 이러한 연구 흐름 중 하나로 Popcorn: Accelerating Kernel K-means on GPUs through Sparse Linear Algebra에서는 Kernel K-means를 희소 선형대수 연산 중심으로 표현하는 프레임워크를 제안합니다. 이를 통해 GPU에서 효율적으로 처리 가능한 연산 패턴을 유도하고, 대규모 데이터셋에서도 확장성을 확보하고자 합니다. Popcorn은 알고리즘 수준에서의 재구성이 GPU 성능 최적화에 효과적임을 보여주는 대표적인 사례입니다.

3. Overview of your parallel algorithm design

A. Detailed design and implementation description with pseudo-code

i. Constant Memory 사용 (할당 단계 - Assignment step)

K-Means 할당 단계에서 사용되는 Centroids 데이터는 매 Iteration 내내 Read-Only 속성을 가집니다 각 반복마다 Centroids 데이터를 GPU의 **Constant Memory**(__constant__)에 할당하여 사용합니다. 이를 이용해 반복적인 메모리 접근을 최대한 빠르게 합니다.

```
// Constant Memory 선언 (Global Scope)
__constant__ float c_centroids[N_CLUSTERS * MAX_FEATURES]; // Centroids 데이터 저장용
__constant__ float c_centroid_norms[N_CLUSTERS]; // Centroid의 Norm 저장용
...
cudaMemcpyToSymbol(c_centroids, h_curr_centroids.data(), centroid_size_float);
cudaMemcpyToSymbol(c_centroid_norms, h_curr_norms.data(), N_CLUSTERS * sizeof(float));
...
```

ii. Shared Memory Reduction (할당 단계 - Compute New Centroids step)

각 point가 속하는 cluster에 관한 정보를 블록 내 Shared Memory에 우선 누적 후 한번에 add해서 Contention을 최소화합니다.

Shared Memory 선언

```
__global__ void computeNewCentroidsKernel_Shared(...) {
    extern __shared__ float s_block_centroids[];
...
```

블록 내 누적

```
if (gid < n_samples) {
        int cluster_id = d_labels[gid];
        atomicAdd(&s_block_counts[cluster_id], 1);
        for (int f = 0; f < n_features; ++f) {
            float val = d_data_soa[f * n_samples + gid];
            atomicAdd(&s_block_centroids[cluster_id * n_features + f], val);
...
```

Write-back - Global Memory로 합계 최종 누적 (Double Atomic 사용)

```
for (int i = tid; i < n_clusters; i += blockDim.x) {
        if (s_block_counts[i] > 0) atomicAdd(&d_cluster_counts[i], s_block_counts[i]);
    }
    for (int i = tid; i < total_elements; i += blockDim.x) {
        float sum = s_block_centroids[i];
        if (abs(sum) > 1e-6)
            atomicAdd(&d_new_centroids[i], (double)sum);
...
```

iii. AoS -> SoA - Assignment step

각 Thread가 하나의 포인트를 담당하고 x->y->z와 같이 순차적으로 centroid와의 거리를 계산하는데, AoS 방식으로 데이터를 저장하면 Thread들이 읽는 메모리 주소가 띄엄띄엄 떨어져 있습니다. 반면 SoA방식은 x값들끼리, y값들끼리 모아 두어 Thread들이 데이터를 읽을 때 Coalescing하게 메모리를 읽을 수 있습

니다.

[Host] AoS(Row-Major) -> SoA(Col-Major) 변환

```
for (int i = 0; i < n_samples; ++i) { // i: Sample Index
    for (int j = 0; j < n_features; ++j) { // j: Feature Index
        h_data_soa[j * n_samples + i] = h_data_aos[i * n_features + j];
    }
}
```

[Device] assignClusterKernel_SoA -AoS일때 float val = d_data_aos[gid * n_features + f];에서 변경

```
for (int f = 0; f < n_features; ++f) {
    float val = d_data_soa[f * n_samples + gid];
    // ... 거리 계산 ...
}
```

iv. CSR 적용

Data가 sparse하면 csr format으로 변경 후 연산하도록 변경했습니다. 이렇게 하고 거리계산을 **||x||^2 + ||c||^2 - 2(x·c)**의 형태로 변경하면 거리 계산시 0인 값들에 대한 연산횟수 감소 및 효율적 연산 가능합니다.

But Dense의 경우 메모리 접근을 비효율적으로 하기 때문에 더 느려지므로 sparse 여부를 미리 판단하여 sparse하지 않으면 csr format을 적용하지 않습니다. sparse 여부 판단은 직접 host에서 count합니다.

[Host] : sparsity 여부 판단

```
float calculate_sparsity(const std::vector<float>& data) {
    long long zero_count = 0;
    for (float val : data) {//전체 순회
        if (std::abs(val) < 1e-6) zero_count++;// 3. 값이 0인지 확인 (부동소수점 오차 고려)
    }
    return (float)zero_count / data.size();
}
```

[Device Side] : CSR 기반 할당 커널 (Assignment Step)

```
// 0이 아닌 값에 대해서만 Dot Product 계산
for (int i = start_idx; i < end_idx; ++i) {
  int col = csr_col_ind[i];    float val = csr_values[i];
  // Constant Memory에 있는 Centroid (c_centroids)와 곱셈
  dot_product += val * c_centroids[c * MAX_FEATURES + col]; //(x·c)
}
// 유클리드 거리 공식 D(x,c) = ||x||^2 + ||c||^2 - 2(x·c)
float dist = my_norm + c_centroid_norms[c] - (2.0f * dot_product);
```

B. Short examples that show how your code works in different scenarios. Choose the example carefully to highlight the major technical capabilities and limitation.

1. Dense한 data만 있거나 sparse한 경우만 있는경우 : sparsity를 kernel launch전에 단하기 때문에 두가지 케이스 모두 문제가 없습니다.

2. Feature가 너무 많은 경우 : 현재는 eature가 2000개인 경우 10*2000*4 = 80kb로 Constant memory 초과하면 실행이 불가합니다. 따라서 이에 대한 예외처리가 필요합니다. (File uses too much global constant data (0xc3500 bytes, 0x10000 max) 발생)

4. Evaluation

A. Evaluation methodology (system configurations, input dataset, compilation/runtime parameters,etc.)

i. System configuration

GSDS server b7
Device : NVIDIA GeForce RTX 3090 1
Total Memory : 24576 MiB (24 GB) 2
CUDA Version : 12.4 3
Driver Version : 550.54.15 4

ii. input dataset

- sklearn.datasets의 make_blobs 기반으로 생성된 합성 데이터셋을 사용했습니다.
- 샘플 수 N = 10M, 차원 M = 100, 클러스터 수 = 10
- 총 30개의 파일(0번~29번)로 구성되며, Dense (Sparsity 0~20%)와 Sparse (Sparsity 80~95%) 파일이 약 50% 확률로 랜덤 구성됩니다.

iii. compilation/runtime parameters

1. cuml Baseline 테스트시

   A. cuml 설치 환경 필요

   B. (dataset start #, dataset end #+1) 파라미터로 전달 필요.

      ex) python3 ./cuml_KMeans.py 0 5

      (메모리 이동이 연산시간에 포함되지 않도록 cupy 라이브러리를 사용으로 인해 data가 n개 이상 넘어가면 memory 부족 현상이 발생 가능)

2. double atomicAdd 사용으로 nvcc 컴파일시 **-arch=sm_XX** 옵션

   ex) nvcc -o kmeans mCSRKmeans.cu **-arch=sm_86**

   ex) nvcc -o kmeans mSoAKmeans.cu **-arch=sm_86**

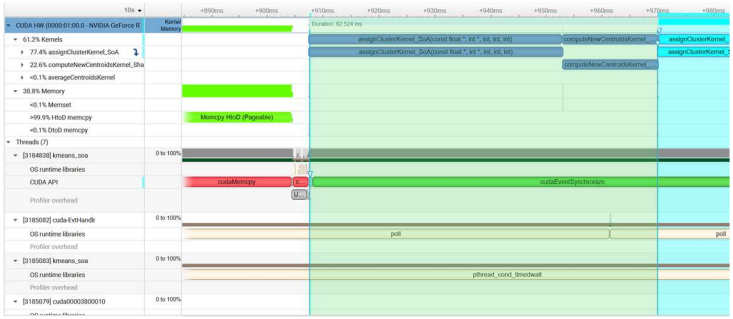
   ex) nvcc -o kmeans mNaiveKmeans.cu **-arch=sm_86**

B. Experimental results: Compare the performance of your algorithm with a baseline sequential version and/or other parallel versions

i. CUDA 구현 커널 1회 수행시간

1. Naïve 구현시 커널 1회 수행시간

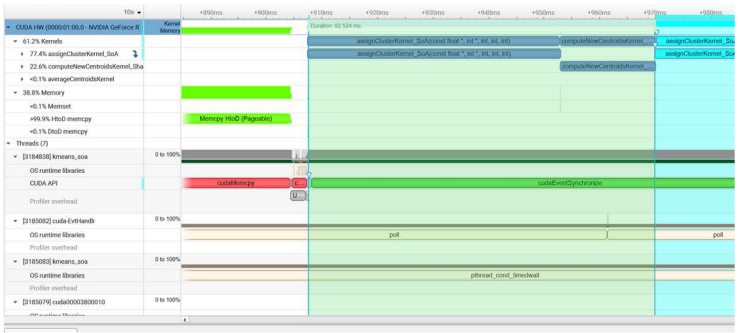
assignClusterKernel 1회 수행시간 : **191.5 ms**
computeNewCentroidsKernel 1회 수행시간 : **323.3 ms-**



ii. SoA 적용시 커널 1회 수행시간

assignClusterKernel 1회 수행시간 : **45.3 ms**

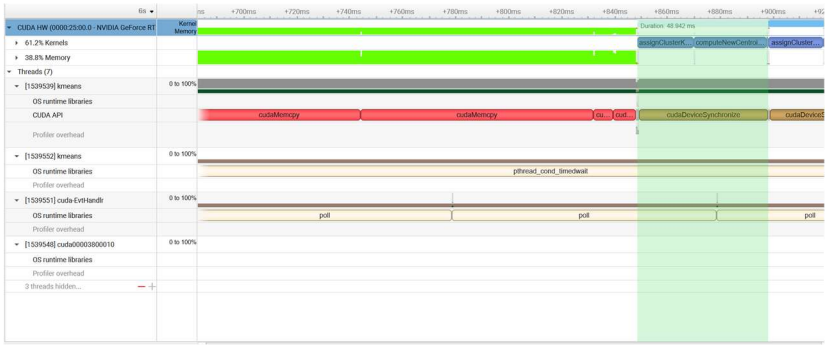
computeNewCentroidsKernel 1회 수행시간 : **17.1ms**



iii. CSR 적용시 커널 1회 수행시간

assignClusterKernel 1회 수행시간 : **21ms**

computeNewCentroidsKernel 1회 수행시간 : **28ms**



iv. 최종 성능 비교

1. 평균 수행 시간

Implementation	Average Execution Time (ms)	Speedup vs CPU
CSR Format	**462.45 ms**	**53.10x**
cuML	993.25 ms	24.72x
Optimezed with SoA	1679.87 ms	14.62x
naive cuda	3650.78 ms	6.73x
sklearn(cpu)	24556.43 ms	1.00x

Table 1

2. 각 케이스 수행시간

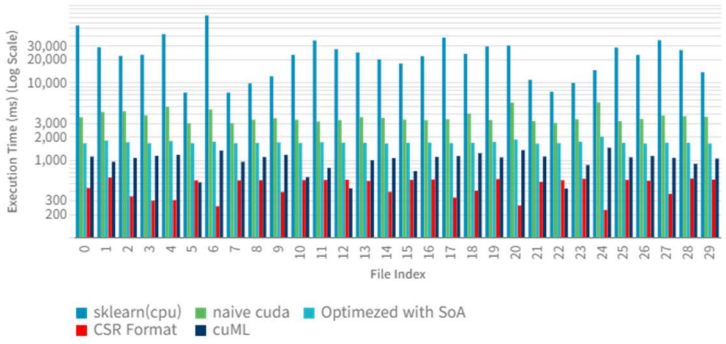


Figure 1

3. Summary

Table 1과 Figure1에서 확인할 수 있는 것 처럼 CSR 포맷을 적용한 최종 알고리즘이 cuML을 포함한 Baseline 대비 우수한 성능을 달성했습니다. 또한 Naïve 구현 대비 SOA와 CSR등의 기법을 적용할수록 성능이 점진적으로 개선되는 것을 알 수 있었습니다.

cuML보다 더 잘나온 이유를 확인한 결과 cuML의 경우도 CSR을 지원하지만 CSR을 직접 판단해서 CSR 최적화 커널이 동작하지 않고 입력 데이터 타입에 따라 다르게 동작합니다. 다만 수행된 실험의 경우 이를 따로 명시해주는 상황이 아니었기 때문에 dense한 방식으로만 연산을 수행합니다. 반면 제가 구현한 방식은 입력의 sparsity를 직접 측정하여 가변적으로 동작하기 때문에 cuML보다 좋은 결과가 나온 것으로 생각됩니다.

다만 sparsity 판단 시간은 평균 346ms으로 연산시간 462.45ms을 더하더라도 808ms로 근소하게 앞서며, 사용성 측면에서도 data에 따라 최적의 연산을 adaptive하게 수행할 수 있다는 점에서 연구의 의미가 있습니다.

5. References

[1] M. B. Cordeiro and W. N. Zola, "Parallel K-means on GPU using Warp-Centric Strategies," in *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)*, Belgrade, Serbia, Oct. 2024, pp.

[2] J. Bellavita, T. Pasquali, L. Del Rio Martin, F. Vella, and G. Guidi, "Popcorn: Accelerating Kernel K-means on GPUs through Sparse Linear Algebra," in *PPoPP '25: Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, Feb. 2025, pp. 426–440.

6. Appendix: a description of where to find your source code, executable, and input programs, and how to run the executable for example inputs

A. Source code

https://github.com/Hulruru/Computing2_Final

B. Executable program

```
//각 파일은 dataset 폴더와 cpu_results 폴더가 있는 폴더에서 실행.
//python 파일은 별도 제공 x
https://github.com/Hulruru/Computing2_Final/tree/master/executable
```

C. How to run

i. 데이터 생성

```
python3 data_generator.py (실행시 dataset 폴더에 data 30개 생성됨.)
```

ii. Sklearn 실행

```
//실행시 cpu 버전 Kmeans 결과(accuracy 측정용)가 cpu_results 폴더에 생성.
python3 sklearn_KMeans.py
```

iii. Custom 버전 빌드

```
//CSR 예시
nvcc -o kmeans mCSRKmeans.cu -arch=sm_86
./mCSRKmeans
```

iv. Cuml 빌드

```
//Rapid환경 설정필요
conda create -n rapids_env -c rapidsai -c conda-forge -c nvidia cuml=24.02
python=3.11 cuda-version=12.4 numpy scikit-learn
conda activate rapids_env
```

```
// parameter 0 1 은 실행 조건에 맞게 변경
python3 cuml_KMeans.py 0 1
```

v. Shell을 통한 실행

```
1. iv.에 해당하는 conda 환경을 설정
2. i.에 해당하는 data_generation 수행
3. ii.~iv.의 프로그램 실행은
./run_kmeans.sh
을 이용하여 수행 (sklearn->naïve->SoA->CSR->cuML 순으로 실행)
4. 결과로 result 폴더가 생성되면 실행 결과가 txt 파일로 저장됨.
```