Lab 2: KMeans with CUDA:
Performance analysis and detailed approach

Qiying Wu

October 7, 2024

Abstract

This report presents the implementation and performance analysis of the KMeans clustering algorithm using both CPU and GPU parallelism. The goal of this lab is to explore the benefits of GPU programming through the use of CUDA and Thrust libraries. The KMeans algorithm, commonly used for unsupervised machine learning, was implemented in four variations: a sequential CPU version, a basic CUDA version, an optimized CUDA version utilizing shared memory, and a high-level parallel implementation using the Thrust library.

I have conducted performance evaluations across datasets of varying sizes and dimensions, comparing the execution times and speedups of each implementation. The results show significant improvements in the CUDA-based versions, with the shared memory implementation providing the fastest runtime, surpassing both the sequential and Thrust implementations. We also analyze the effect of data transfer overhead between the CPU and GPU and examine the impact of non-deterministic behavior introduced by atomic operations in the CUDA implementations. The analysis highlights the trade-offs between ease of implementation and performance optimization, with the CUDA shared memory version achieving the best speedup compared to the sequential approach.

Introduction

KMeans is one of the most widely used clustering algorithms in machine learning, particularly for unsupervised tasks where the goal is to group data points into cohesive clusters based on feature similarity. The algorithm iteratively assigns data points to clusters and updates the cluster centroids until convergence. Despite its simplicity, KMeans is computationally expensive, especially for large datasets and high-dimensional data, making it an ideal candidate for parallelization.

The purpose of this lab is to explore the performance benefits of using GPUs for parallel computation through the implementation of KMeans using CUDA and the Thrust library. GPU programming enables significant speedups by exploiting data parallelism, allowing thousands of threads to run concurrently, which can greatly reduce the execution time of computationally intensive tasks like KMeans.

In this lab, we implement and compare four different versions of the KMeans algorithm:

- **Sequential CPU Implementation:** A basic, single-threaded version of KMeans executed on the CPU.
- **CUDA Basic Implementation:** A parallel GPU version of KMeans using CUDA for computation.
- **CUDA with Shared Memory:** An optimized GPU version that utilizes CUDA's shared memory to reduce global memory access latency and improve performance.
- **Thrust Implementation:** A high-level parallel implementation using NVIDIA's Thrust library, which abstracts thread management and allows for efficient use of parallel primitives.

This report presents the design and implementation of these different approaches, along with performance analyses based on their execution times for various datasets. We also discuss the challenges encountered during the optimization process, including memory management and non-deterministic behavior caused by atomic operations in CUDA. By comparing the performance of these implementations, we aim to highlight the trade-offs between different levels of abstraction in GPU programming and the impact of hardware-level optimizations on the performance of parallel algorithms like KMeans.

1 Hardware and Software Specifications

GPU Hardware Details:

- GPU: [Include your GPU details here]
- CUDA cores: [Number of CUDA cores]

CPU Hardware Details:

- CPU: [Include your CPU details here]
- Cores: [Number of CPU cores]

OS Details:

- Operating System: [Include your OS version]
- CUDA Version: [CUDA Version]
- Compiler: [GCC/NVCC version used]

2 Methodology

In this lab, the starter code includes spin_barrier and I implemented a counter-based barrier for prefix_sum calculation. And using Blelloch algorithm (Work-Efficient Parallel Prefix Scan) with two sweeps to calculate our prefix_sum parallely.

3 Algorithm and Implementations

3.1 Sequential CPU Implementation

The sequential CPU implementation was designed to iteratively assign data points to the nearest centroid and then update the centroids by calculating the mean of the assigned points. We ensured the algorithm converges when the centroids stop moving significantly or after a fixed number of iterations.

3.2 CUDA Basic Implementation

In the CUDA implementation, each data point and centroid computation was parallelized using CUDA threads. Kernels were designed to compute the distances between points and centroids, assign labels, and update centroids. The implementation includes memory transfers between host (CPU) and device (GPU), which contributes to the overall runtime.

3.3 CUDA with Shared Memory

The shared memory version of CUDA optimized memory access by reducing global memory accesses and using shared memory to store centroids, thus reducing latency and improving performance.

3.4 Parallel GPU Implementation using Thrust

The Thrust-based implementation used high-level parallel primitives such as `thrust::for_each`, `thrust::reduce_by_key`, and `thrust::transform` to implement the KMeans algorithm. This method abstracts CUDA thread management but may not fully utilize GPU capabilities compared to direct CUDA kernel implementation.

4 Performance Analysis

4.1 Execution Times

4.2 Fastest Implementation

The fastest implementation was the [**CUDA Shared Memory/Basic**] version, which outperformed other implementations due to optimized memory access and parallelism. This matched expectations, as the shared memory version is designed to reduce latency by minimizing global memory accesses.

4.3 Slowest Implementation

The slowest implementation was [**Thrust/Sequential**], as expected. The Thrust implementation abstracts memory management, which can introduce overhead, while the sequential implementation is inherently limited by the lack of parallelism.

4.4 Data Transfer Overhead

In the CUDA implementations, a significant portion of the runtime was spent transferring data between the CPU and GPU. For larger input sizes, this transfer overhead became more pronounced, accounting for approximately [**fraction**] of the total runtime.

4.5 Speedup Comparison

The expected speedup was estimated based on the number of CUDA cores and threads. The best-case speedup was [**calculated speedup**] times faster than the sequential implementation, while the observed speedup for the CUDA Shared Memory implementation was [**observed speedup**] times faster.

4.6 Convergence Behavior

For all implementations, convergence was achieved within [**number**] iterations on average. The convergence time was consistent across implementations but varied based on the use of atomic operations in the CUDA versions, which introduced some non-determinism in the iteration count.

5 Time Spent on the Lab

I spent approximately [**number of hours**] hours working on this lab, including coding, testing, and performance analysis.

Conclusions

The CUDA implementations demonstrated significant speedups compared to the sequential version, with the shared memory implementation being the fastest. The trade-off between abstraction and performance was evident in the Thrust implementation, which, while easier to write, did not achieve the same performance as the more optimized CUDA kernel implementations.

6. References

Technical Sources:

- Understanding Implementation of Work-Efficient Parallel Prefix Scan
- Lab 1: Prefix Scan and Barriers
- Guy E. Blelloch: Prefix Sums and Their Applications
- CS380P Lecture: Conditions and Barriers

- C++ Atomic Memory Order Reference
- Spin-Lock in Modern C++ with Atomics, Memory Barriers, and Exponential Back-Off
- Spinlocks in C++ with CoffeeBeforeArch
- Chapter 39. Parallel Prefix Sum (Scan) with CUDA