GPU parallelized KNN Algorithm

Improving KNN with CUDA

Alex Garrett
Undergraduate in Computer Science
University of Victoria
Victoria, British Columbia, Canada
alexgarrett2468@gmail.com

QingZe Luo
Undergraduate in Computer Science
University of Victoria
Victoria, British Columbia, Canada
qluo@uvic.ca

Jingyi Lu
Graduate in Computer Science
University of Victoria
Victoria, British Columbia, Canada
jingyilu@uvic.ca

ABSTRACT

K-Nearest Neighbor (KNN) is a supervised machine learning algorithm widely employed in image classification tasks. It operates by storing training data as one-dimensional vectors and calculating the distances between the vector to be classified and all classified vectors. By identifying the first k minimum distance, known as the nearest neighbors, the algorithm determines the class that appears most frequently among these neighbors as the final classification. A small k value may lead to overfitting, while a larger k value can result in underfitting. KNN's advantage lies in its minimal training computation since it utilizes the training data in its original form. However, the computation of distances and the subsequent sorting of these values introduce higher time complexity during classification compared to feed-forward neural network alternatives, especially as the number and size of training vectors increase. This paper aims to examine and implement the KNN algorithm in CUDA for classifying images of handwritten digits from the MNIST labeled dataset. Specifically, it explores how we further improved our model from the previous report to create a more effective use of the GPU's parallelization capabilities.

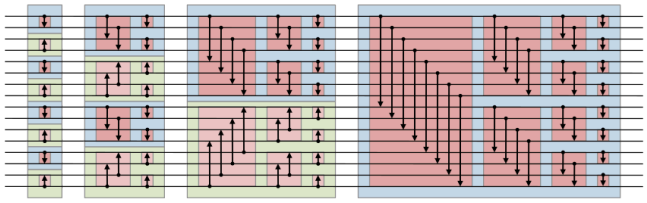
1 MNIST Classification

The MNIST database is a well-known benchmark in the field of machine learning, containing thousands of images of handwritten digits that have been manually classified by humans. Our model leverages this dataset as a straightforward demonstration of its capabilities and the enhancements achieved through GPU parallelization. Utilizing a Python script, each image, comprising 784 pixels arranged in a 28 x 28 grid, is downloaded and transformed into a one-dimensional vector. These vectors are then saved into two separate binary files along with their classification labels: one file for training data and another for testing data.

Our approach involves transferring the entire contents of both the training and testing binary files into the GPU's memory. By doing so, we enable the GPU to handle all computational tasks related to the KNN algorithm. Specifically, for each test image vector, the algorithm calculates its distance to every image vector in the training dataset and then sorts these values returning the K lowest values as the nearest neighbors.

2 Bitonic Sort

The sorting phase of KNN requires organizing distances to identify k nearest neighbors. We implemented a parallel bitonic sorting algorithm that is particularly suitable for GPU architectures due to its regular pattern and high degree of parallelism.



This image shows a visual representation of the iterative structure that takes place.

Bitonic sorting operates by repeatedly merging and sorting bitonic sequences into sorted subarrays through a series of compare-and-swap operations. In our implementation, the GPU kernel bitonicSortStep leverages this principle by dividing the workload among threads in a thread block, each responsible for processing a specific subset of distances. The algorithm exploits shared memory to store distances (s_distances) and their corresponding labels (s_labels) for each thread, minimizing the overhead of global memory access. Each thread compares and swaps pairs of elements based on their bitwise XOR (ixj) index, which determines the sorting order in a hierarchical manner

(ascending or descending) dictated by the current stage (j and k) of the bitonic sort.

Parallelism is achieved at multiple levels. At the block level, threads concurrently handle different data segments, effectively partitioning the computation. Additionally, the kernel is launched across multiple blocks, with each block processing independent portions of the dataset. This design ensures efficient utilization of the GPU's computational resources, as multiple elements are processed simultaneously across the CUDA grid. By utilizing the regular and predictable access patterns of bitonic sorting, we can minimize thread divergence, further optimizing parallel performance.
Our original implementation of bitonic sort outlined in the previous report was incorrect due to the quantity of image vectors not being a power of 2. As apparent in the image above, bitonic sort is effective if the array size can be divided into symmetrical ascending and descending pairs of values thus necessitating the size being a power of 2. Solving this issue is simply a matter of padding values in the array. Although this results in some redundant computation, the parallel speedup bitonic sort provided when running on the GPU makes up for this in contrast to sequential alternatives.

When performing this modification without any other of the improvements, outlined later in this paper, we were able to increase accuracy from 90% to 96%. This demonstrates how the lack of padding resulted in partially incorrect sorting.

```
Final Results:
Total test samples: 10000
Correct predictions: 9665
Accuracy: 96.65%
```

3 Parallelization Improvements

A number of key memory and kernel use optimizations permitted a more effective use of the GPU's resources to tackle this problem. This section describes each of these techniques and how they improved our original implementation.

3.1 Pinned Memory Optimization

Pinned host memory enables higher bandwidth between the host and device compared to pageable memory. Allocating pinned memory for frequently transferred data avoids unnecessary page faults and synchronization overhead, significantly speeding up data transfers.

Original

```
float*h_train_images= new float[num_trainsamples
* IMAGESIZE];
int* h_train_labels = new int[num_trainsamples];
```

Optimized

```
float* h_train_images;
int* h_train_labels;
CUDA_CHECK(cudaMallocHost(&h_train_images,
num_trainsamples * IMAGESIZE * sizeof(float)));
CUDA_CHECK(cudaMallocHost(&h_train_labels,
num_trainsamples * sizeof(int)));
```

3.2 Vectorized Memory Access

By restructuring the memory access from scalar operations to vectorized operations, we aligned memory access with the GPU's hardware capabilities. Using the float4 data type allowed for fetching four floating-point values in a single operation, improving memory bandwidth utilization and decreasing instructions.

Original

```
float diff = d_images[idx * IMAGESIZE + i] -
d_testImage[i];
sum += diff * diff;
```

Optimized

```
float4* train_vec = (float4*)(&d_images[idx *
IMAGESIZE]); float4* test_vec =
(float4*)d_testImage;
# pragma unroll 8
for (int i = 0; i < IMAGESIZE/4; i++) {
float4 diff;
float4 img = img_vec[i];
float4 test = test_vec[i];
// ...
}
```

3.3 Memory Alignment and Management

We implemented proper memory alignment using cudaMallocPitch and ensured efficient management by pre-allocating GPU memory and reusing it across iterations.

Original

```
cudaMalloc(&d_train_images, num_trainsamples *
IMAGESIZE * sizeof(float));
```

Optimized

```
size_t pitch;
CUDA_CHECK(cudaMallocPitch((void**)&d_train_im
ages, &pitch, IMAGESIZE * sizeof(float),
num_trainsamples));
```

3.4 Shared Memory Optimization

We utilized shared memory to cache frequently accessed data, particularly the test image that is used by all threads in a block. This optimization reduces global memory access latency since each thread in a block can access the test image from the fast shared memory instead of repeatedly accessing global memory.

Original

```
for (int i = 0; i < IMAGESIZE; i++) { float
diff = d_images[idx * IMAGESIZE + i] -
d_testImage[i];
sum += diff * diff;
}
```

Optimized

```
extern __shared__ float shared_mem[]; float*
shared_test = shared_mem; // Cooperatively
load test image into shared memory
for (int i = tid; i < IMAGESIZE; i +=
blockDim.x) { shared_test[i] = d_testImage[i];
} __syncthreads(); // Use shared memory for
distance calculation
for (int i = 0; i < IMAGESIZE; i++) { float
diff = train_img[i] - shared_test[i]; sum +=
diff * diff; }
```

3.5 Loop Unrolling

Loop unrolling reduces the overhead of loop control instructions and can help the compiler generate more efficient code. By explicitly unrolling loops, we let the compiler schedule instructions better and hide latency.

Original

```
for (int i = 0; i < IMAGESIZE; i++) {
float diff = train_img[i] - shared_test[i];
sum += diff * diff;
}
```

Optimized

```
#pragma unroll 4
for (int i = 0; i < IMAGESIZE; i++) {
float diff = train_img[i] - shared_test[i];
sum += diff * diff;
}
```

4 Results

4.1 Output

The Previous Version:

```
Final Results:
Total test samples: 10000
Correct predictions: 9665
Accuracy: 96.65%
Total execution time: 37.21 seconds
```

Our Final Version:

```
Final Results:
Total test samples: 10000
Correct predictions: 9688
Accuracy: 96.88%

Kernel Timing Breakdown:
Average Data Transfer Time: 0.0105649 ms
Average Distance Calculation Time: 1.95 ms
Average Sorting Time: 0.383331 ms
Total Data Transfer Time: 105.649 ms
Total Distance Calculation Time: 19500 ms
Total Sorting Time: 3833.31 ms

Total execution time: 24.316 seconds
```

These results demonstrate that our optimizations were highly effective, achieving both high accuracy and efficient

performance. The 96.88% accuracy shows that our implementation successfully maintains the classification quality while leveraging GPU parallelization.

5 Comparison to State of the Art

Preprocessing techniques play an important role in improving the performance of traditional k-NN classifiers on MNIST by transforming the data into a more consistent, noise-free, or lower-dimensional representation. Methods such as normalization, dimensionality reduction (e.g., PCA), skew correction, and feature extraction (e.g., HOG) are widely used to improve classification accuracy and computational efficiency. For example, PCA reduces the high dimensionality of 784-pixel vectors, making distance calculations faster and less likely to overfit. Similarly, normalization ensures that all features contribute equally to the distance metric, while skew correction corrects for mismatched orientations in handwritten digits. By combining these techniques, a preprocessing-based k-NN classifier can achieve 96-98% accuracy on MNIST. Execution time varies depending on the complexity of the preprocessing, but is typically 10 to 30 seconds for a dataset containing 10,000 test samples on a modern CPU.

Compared to previous methods, our GPU-parallel k-NN classifier achieved significant computational efficiency improvements through four key performance optimization techniques: Pinned Memory Optimization, Vectorized Memory Access, Memory Alignment and Management, and Shared Memory Optimization. These innovative strategies not only accelerated raw distance computation and sorting processes but also ensured the stability of classification accuracy. Through our carefully designed CUDA acceleration scheme, we reduced the processing time for 10,000 test samples to 24.36 seconds while maintaining a classification accuracy of 96.88%.

Compared to unoptimized CPU implementations, our approach demonstrates the immense potential of GPU computing. Pinned Memory Optimization reduced memory transfer overhead, Vectorized Memory Access improved data throughput, Memory Alignment and Management ensured more efficient data access patterns, and Shared Memory Optimization minimized global memory access latency. The synergistic effect of these techniques not only enhanced computational performance but also provided valuable technical insights for GPU acceleration in complex machine learning tasks.

By continuously optimizing preprocessing methods, such as introducing regularization and Principal Component Analysis (PCA), our GPU-optimized framework promises to further improve classification efficiency and accuracy on large-scale datasets in the future.

6 Future Modification

While we have achieved high accuracy (96.88%) with our current implementation, there are still opportunities for further optimization. Future improvements could focus on:

5.1 Hypothetical New Type of KNN

Imagining each of these image vectors mapped to a 3 dimensional plane indicates a misclassification likely occurs when a cluster of outliers exists away from the majority of members of that classification. Since KNN recognizes only the distance to these vectors, it is not uncommon for it to make a mistake on this basis. Future implementations of KNN could calculate some coefficient of clustering to scale the original distance value, biasing it either positively or negatively depending on how far it is from its classification group. This could be done by running KNN during training on every possible pair of vectors within the same class. The distances of these K nearest neighbors would indicate how much of an outlier each node was. This quantity would then be modified by some normalization and either added or multiplied to the original distance calculation. This process could even be applied multiple times in a recursive manner for each distance calculation performed by this additional KNN.

One way of implementing this would be to set a high k for this preliminary KNN and then calculate the standard deviation of distances of the k nearest neighbors within the same class. Low standard deviations would mean a high level of clustering indication some feature in the image is common and relevant at that point therefore that node should be considered closer to any future classifications. Similarly low standard deviation would be that particular node has low clustering and is likely an outlier and should therefore not be considered as intensely.

Although this new method presents a reasonable chance for higher classification accuracy, it would drastically increase the time complexity of training the model. Future iterations of this project could attempt to answer if this method is more accurate and determine if the trade off in training efficiency is worth it.