COMS4040A & COMS7045A Assignment 1

Hand-out date: 14:00, April 16, 2025
Due date: 23:59 pm, May 7, 2025

Objectives

1. Design and implement parallel algorithms using OpenMP and CUDA C/C++ with a focus on achieving scalable performance.
2. Evaluate the performance of a parallel algorithm implementation.
3. Writing a short scientific report.

Instructions

1. This is an individual assignment.
2. The due date is strictly enforced. There is no late penalty until **8:00 AM on May 8**. After that, **10% of your total mark will be deducted for each 24-hour period (or part thereof)**. No submissions will be accepted after **11:59 PM on May 9**, and failure to submit by then will result in a mark of **zero**.
3. Refer to the section **Hand-ins** for submission details.
4. The total mark for this assignment is **70**.

Problems

Problem 1. Parallelizing a Brute-Force K-Nearest Neighbors Classifier using OpenMP [28]

In this problem, we focus on implementing a K-Nearest Neighbors (KNN) classifier using a brute-force approach. You will parallelize the computationally intensive components of the KNN algorithm including distance computation and sorting. Specifically,

1. employing `for` construct for the distance computation, and
2. employing `sections/section` and `task` constructs, respectively, for sorting using quicksort.

The goal is to understand the impact of parallelization on performance and to compare the effectiveness of these OpenMP constructs.

Problem Description: Brute-Force KNN Classification

You will implement a KNN classifier to classify data points based on their proximity to training data. The brute-force KNN algorithm computes the Euclidean distance from a test point to all training points, sorts these distances, and selects the K nearest neighbors to determine the class via majority voting.

K-nearest neighbour (KNN) is a widely used algorithm for the tasks of classification and regression in pattern recognition, data mining, and machine learning. Let $\mathbf{P} = \{p_0, p_1, \ldots, p_{m-1}\}$ be a set of m reference points with values in $\mathbb{R}^d$, and $\mathbf{Q} = \{q_0, q_1, \ldots, q_{n-1}\}$ be a set of n query points in the same vector space. The KNN search problem refers to finding the k nearest neighbours of each query point $q_i \in \mathbf{Q}$ in the reference set $\mathbf{P}$ given a distance metric. Commonly used distance metrics include Euclidean or Manhattan distances.

For example, when the data points p_i or q_i represent complex objects, such as an image, the dimension d could be very high, as each d-dimensional vector represents a feature extracted from the object. Regarding the distance metric, Euclidean and Manhattan distances are simply the ℓ_2- and ℓ_1-norms, respectively. Efficient algorithms are proposed for KNN search problem, including the one using kd-trees to partition the search space which leads to $\mathrm{O}(\log \mathrm{m})$ query time. However, our focus here is a brute force algorithm.

The brute-force (BF) algorithm for solving the KNN problem consists of the following steps. For each query point q_i, $0 \leq i < n-1$,

1. compute all the distances between q_i and $p_j, 0 \leq j \leq m$;
2. sort the computed distances;
3. select the k reference points corresponding to the k smallest distances;
4. repeat steps 1 to 3 for q_{i+1}.

The complexity of this algorithm is $\mathrm{O}(nmd)$ for distance computation and $\mathrm{O}(m^2)$ for sorting (using quicksort). While the computational complexity is high, the algorithm presents ample opportunities for parallelization.

Tasks

1. Implement a sequential version of the BF algorithm, which consists of two compute intensive components - distance calculations and sorting. [**10/28**]
   - Compute Euclidean distances between each test point and all training points.
   - Implement quicksort to sort distances for each test point independently, and identify the K (**use K = 3, 5, and 7**) nearest neighbors based on the sorted distances.
   - Assign the class based on majority voting among the K neighbors.
   - Record execution time for distance calculation, sorting, and the total runtime, respectively.
2. Parallelize the two compute intensive components using OpenMP.
   - Employ `for` construct for the distance computation. Measure execution time for distance calculation phase. [**6/28**]
   - Parallelize the sorting of distances for each test point using OpenMP ‘sections/section‘ constructs. Measure execution time for the sorting phase. [**6/28**]
   - Parallelize the sorting of distances using OpenMP ‘task‘ constructs. Measure execution time for the sorting phase. [**6/28**]

- Measure the total runtime.

Requirements

1. Language: C/C++ with OpenMP.
2. **Dataset:** Use CIFAR10 dataset. Learned features (512D) for training and testing are provided.
3. Sorting: Implement your own quicksort algorithm (do not use STL 'std::sort').
4. Output: For each test point, output the predicted class and confidence (proportion of majority class among K neighbors).
5. Evaluate your solution in the following manner.
   - Benchmark the classification performance using accuracy.
   - Compare the performance (execution time and speedup) of the serial implementation with each of the parallel implementations, i.e., (1) parallel distance computation and parallel sorting using sections construct, and (2) parallel distance computation and parallel sorting using task construct.
   - Compare the effectiveness of the two parallel sorting approaches - OpenMP sections and task constructs - in terms of execution time and scalability.
   - By parallel performance, we consider mainly running time. The overall running time should be decomposed into distance calculation time and sorting time. The two running times (as percentage of the overall time), as well as the overall time should be recorded. (You don't need to include the file reading time in your overall time if you read your data from a file.)

Problem 2. Image Convolution Using CUDA C [32]

Introduction

Convolution is an array operation in which each output element is computed as a weighted sum of neighboring input elements. The weights used in the weighted sum calculation are defined by an input mask array, referred to as the convolution mask here. The same convolution mask is typically used for all elements of the array. Convolution is commonly used in various forms in signal processing, digital recording, image processing, video processing, and computer vision. In these application areas, convolution is often performed as a filter that transforms signals and pixels into more desirable values.

Convolution typically involves a significant number of arithmetic operations on each data element. For large data sets such as high-definition images and videos, the amount of computation can be very large. Each output element can be calculated independently of each other, making it well-suited for parallel computing. On the other hand, there is a substantial amount of input data sharing for computing neighboring output data elements with non-trivial boundary handling. This makes convolution an important use case of sophisticated tiling methods and input data staging methods.

When applied in image processing tasks, convolution leads to results such as noise removal, edge detection and sharpening of details. If an image is represented as a 2D discrete signal $Y \in \mathbb{Z}^{M \times N}$, we can perform the

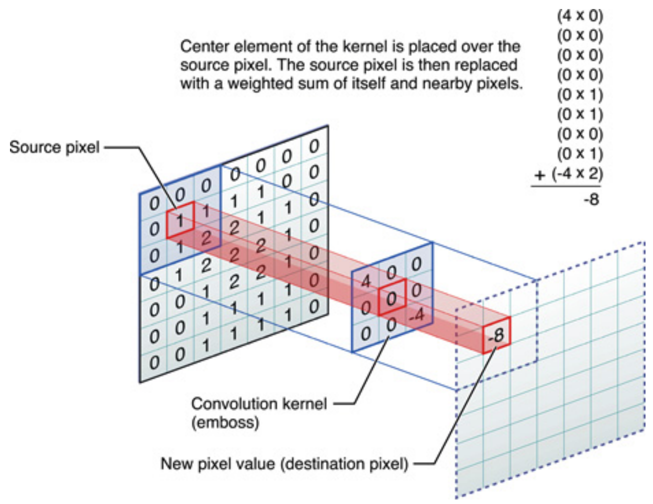
Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)

Figure 1: Performing convolution operation

(discrete) convolution in 2-dimension using a kernel $F \in \mathbb{Z}^{L\times P}$ as

$$(Y * F)[i,j] = \sum_{l=-a}^{a} \sum_{p=-b}^{b} Y[i+l, j+p]F[l+a, p+b], \quad 0 \le i < M,\ 0 \le j < N, \tag{1}$$

where $a = \lfloor L/2 \rfloor, b = \lfloor P/2 \rfloor$ for a kernel size of $L \times P$.

The convolution operation in Equation (1) is actually a scalar product between the corresponding filter weights and the pixel values within a window centered at each pixel location, where the window size is determined by the dimensions of the filter. Figure 1 illustrates the convolution using a small 3×3 kernel. The design of the convolution filter requires careful selection of the weights to achieve desired results.

Implementation Considerations

Image convolution can be efficiently implemented on massively parallel hardware, since the same operator gets executed independently for each image pixel. A naive implementation would simply execute the convolution for each pixel by one CUDA thread, read all values inside the filter area from the global memory, and write the result. (Note that this approach is inefficient.)

In this assignment, you are going to implement image convolution using the following methods.

1. Serial computation [10/34]
2. CUDA implementation using global memory [12/34]
3. CUDA implementation using shared memory [12/34]

In your implementation:

1. Use the filters in Table 1 for testing the convolution result.

2. A pgm test image is given. To load a pgm image or write a pgm image, you may use the relevant CUDA SDK functions. A CUDA SDK sample program is provided.
3. The pixel values outside the image boundaries should be treated as 0 values.
4. Your code should be written in a way that the size of convolution mask is arbitrary in the case of **averaging** filter, that is, you should test it on different sizes of convolution masks, e.g., 3 by 3, 5 by 5, and 7 by 7.
5. When applicable, describe any further relevant implementation design choices in your report.

Sharpen	Emboss	Average
$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 \end{bmatrix}$	$\frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$

Table 1: Convolution masks

Hand-Ins

1. Source code (well-commented, including sequential and both parallel versions). [4]
2. Script or instructions to compile and run the code. [4]
3. Output data file.
4. Report with performance analysis. A combined report for Problems 1 & 2 named as `<yourStudentNo>_report.pdf`: You may organize the contents in two sections, one for each problem. In each section: [62[1]]
   - Description of your implementation and experiments
   - Performance results (tables/graphs of classification accuracies, execution times and speedup).
   - Discussion and analysis of results.
   - Challenges faced and solutions.
   - Cite the sources where applicable.
   - **Note:** Do not put your results in a table or graph without giving any discussions or explanations of those in the text.
   - A conclusion for each section.
5. Submit the report and source codes separately via the respective submission link provided.
6. Submit your report to Turnitin to produce a similarity check report. [Negative marks[2]]
7. The source code submission should be a single compressed file named as `<yourStudentNo>_assignment1.tar.gz`. The compressed file should be extracted to a folder named `<yourStudentNo>_assignment1`. In this folder, organize the codes for different problems in separate folders.
8. For each problem, include `Makefiles` to build your code, run script(s), such as `run.sh`, to run your programs, and readme file with instructions to compile and run your code.

[1]This mark is a combination of the marks indicated in Problems 1 & 2.

[2]Depending on the similarity score, varying negative marks will be given. A similarity score below 15% is not penalized.

9. The report should be written using latex. Template tex files are provided in this regard. You just need to edit `assignment1_report.tex` file and compile `template.tex` using latex.