Wydział Matematyki
i Nauk Informacyjnych
POLITECHNIKA WARSZAWSKA

Procesory Graficzne w Zastosowaniach Obliczeniowych

Odległość Levenshteina

Dokumentacja projektu

Autor: Adam Grącikowski
Prowadzący: dr inż. Jan Bródka

Oświadczam, że niniejsza praca, stanowiąca podstawę do uznania osiągnięcia efektów uczenia się z przedmiotu Procesory Graficzne w Zastosowaniach Obliczeniowych została wykonana przeze mnie samodzielnie.

Warszawa, 1 stycznia 2025

Spis treści

1 Wstęp

1.1 Cel projektu

Celem projektu jest implementacja algorytmu Levenshteina do wyznaczania odległości edycyjnej napisów (skończonych ciągów znaków) oraz porównanie dwóch różnych implementacji - dynamicznej wykonywanej na CPU oraz równoległej wykonywanej na GPU.

1.2 Motywacja

Algorytm Levenshteina znajduje szerokie zastosowanie w wielu dziedzinach informatyki, takich jak przetwarzanie języka naturalnego, bioinformatyka czy systemy wyszukiwania i autokorekty tekstu.

Wraz z rozwojem technologii i rosnącą ilością danych tekstowych, istnieje potrzeba optymalizacji metod obliczeniowych. Zastosowanie równoległego przetwarzania na GPU może znacząco przyspieszyć obliczenia, szczególnie dla dużych zbiorów danych, co czyni projekt wartościowym zarówno z perspektywy naukowej, jak i praktycznej.

Porównanie tradycyjnego podejścia z implementacją równoległą pozwala lepiej zrozumieć potencjał i ograniczenia nowoczesnych architektur obliczeniowych.

2 Definicja odległości edycyjnej

Odległość edycyjna pomiędzy dwoma napisami definiowana jest jako minimalna liczba operacji koniecznych do przekształcenia jednego napisu w drugi.

Do podstawowych operacji zalicza się:

- **podmianę** jednego znaku na inny (ang. *substitution*),
- **wstawienie** nowego znaku (ang *insertion*),
- **usunięcie** istniejącego znaku (ang. *deletion*).

2.1 Przykłady

- Odległość edycyjna pomiędzy napisami identycznymi jest zerowa, ponieważ potrzeba zero działań, aby jeden z nich przeprowadzić na drugi.
- Odległość edycyjna pomiędzy napisami *kawa* oraz *kara* wynosi 1, ponieważ do przeprowadzenia pierwszego na drugi wystarcza jedno działanie - zamiana litery w na r.
- Odległość edycyjna pomiędzy napisami *marka* oraz *ariada* wynosi 4, ponieważ potrzeba co najmniej czterech działań - na przykład: usunięcia litery m, zamiany k na i oraz dodania d i a.

3 Opis programu

Oznaczmy przez m długość napisu źródłowego oraz przez n długość napisu docelowego. Program zakłada, że oba napisy składają się z liter należących do przedziału `[a-z]`. W przypadku pojawienia się dużych liter, zostaną one potraktowane jak małe litery.

Ze względu na efektywność, zamiana wielkości liter jest jedyną walidacją znaków, którą wykonuje program.

3.1 Format danych wejściowych

W programie zaimplementowano dwa formaty danych wejściowych.

3.1.1 Format tekstowy

- Pierwsza linia pliku zawiera dwie liczby naturalne, rozdzielone spacją. Liczby te interpretowane są jako m i n.
- W drugiej linii pliku znajduje się ciąg znaków długości m, interpretowany jako napis źródłowy.
- W trzeciej linii pliku znajduje się ciąg znaków długości n, interpretowany jako słowo docelowe.

3.1.2 Format binarny

- Format ten jest w pełni analogiczny do formatu tekstowego.
- Na początku pliku znajdują się parametry m i n zapisane na pierwszych 8 bajtach jako liczby typu `int`.
- Kolejne $m + n$ bajtów zawiera napis źródłowy oraz docelowy.

3.2 Uruchomienie programu

Program można uruchomić podając 4 wymagane parametry pozycyjne:

- `data_format` - określa format danych wejściowych (`txt` lub `bin`),
- `computation_method` - określa zastosowany algorytm (`cpu` lub `gpu`),
- `input_file` - określa ścieżkę do pliku wejściowego w zadeklarowanym formacie,
- `output_file` - określa ścieżkę do pliku wyjściowego.

4 Metody implementacji

4.1 Implementacja dynamiczna `cpu`

Implementacja algorytmu Levenshteina dla procesora (`cpu`) oparta jest na dynamicznym programowaniu. W tej metodzie wykorzystuje się dwie macierze: jedną do przechowywania obliczonych wartości odległości edycyjnych (`distances`), a drugą do przechowywania operacji transformacji (`transformations`).

Proces składa się z trzech głównych etapów: inicjalizacji, wypełniania macierzy oraz odzyskiwania sekwencji transformacji.

Na początku macierz odległości jest wypełniana wartościami odpowiadającymi operacjom usunięcia dla pierwszej kolumny oraz operacjom wstawienia dla pierwszego wiersza.

Algorytm przetwarza kolejne komórki macierzy, porównując litery dwóch napisów. Obliczana jest minimalna liczba operacji spośród trzech możliwych: podmiany (`substitution`), wstawienia (`insertion`) oraz usunięcia (`deletion`). Wybrana operacja jest zapisywana w macierzy transformacji.

Ostatnim krokiem jest odtworzenie sekwencji transformacji na podstawie macierzy zawierającej operacje transformacji. Proces zaczyna się od prawego dolnego rogu macierzy i podąża w kierunku lewego górnego rogu, zależnie od wybranej operacji.

Główna funkcja obliczająca odległość Levenshteina inicjuje macierze, uruchamia wypełnianie dynamiczne oraz odzyskiwanie sekwencji transformacji. Dodatkowo, na potrzeby analizy wydajności, mierzy czas wykonania poszczególnych etapów za pomocą modułu `TimerManager`.

4.2 Implementacja równoległa `gpu`

Implementacja równoległa algorytmu obliczania odległości Levenshteina opiera się na podejściu opisanym w pracy naukowej dostępnej pod adresem:

https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0186251.

Algorytm ten zoptymalizowano do działania na procesorach graficznych (GPU) z wykorzystaniem modelu programowania CUDA, co pozwoliło na znaczące przyspieszenie obliczeń w porównaniu do implementacji na CPU.

Implementacja rozpoczyna się od alokacji buforów pamięci na urządzeniu (GPU), które są niezbędne do przechowywania znaków wejściowych, macierzy odległości, macierzy transformacji, macierzy X oraz innych pomocniczych struktur danych.

Kluczowym elementem algorytmu jest wspomniana macierz X, która przechowuje informacje o występowaniu poszczególnych znaków alfabetu w słowie docelowym.

Macierz X jest obliczana równolegle na GPU w kernelu `PopulateDeviceX`. Każdy wątek przetwarza jeden znak alfabetu, aktualizując jego pozycję w słowie docelowym. Dzięki zastosowaniu pamięci współdzielonej (`__shared__`) operacje są szybkie i efektywne.

Macierz odległości jest obliczana w kernelu `PopulateDeviceDistances`. Mechanizm wymiany danych między wątkami w warpach przyspiesza synchronizację obliczeń bez kosztownych barier.

Preobliczona macierz X umożliwia szybki dostęp do pozycji liter, minimalizując złożoność porównywania znaków.

Po zakończeniu obliczeń macierzy odległości, dane są przesyłane z GPU na host (CPU). Rekonstrukcja transformacji odbywa się poprzez backtracking w macierzy transformacji. Proces ten umożliwia odtworzenie minimalnego ciągu operacji przekształcających jedno słowo w drugie.

5 Struktura projektu

Implementacje algorytmów `cpu` oraz `gpu` znajdują się odpowiednio w folderach `CPU` oraz `GPU`.

Klasy `CPU::LevenshteinDistance` oraz `GPU::LevenshteinDistance` dzedziczą po abstrakcyjnej klasie bazowej `LevenshteinDistanceBase`, która definiuje jedną publiczną metodę odpowiedzialną za obliczenia. Metoda ta nazywa się `CalculateLevenshteinDistance`.

W folderze `Timers` znajdują się klasy odpowiedzialne za pomiar czasu wykonania poszczególnych części algorymów.