Optymalizacja Kodu Na Różne Architektury
Zadanie 1

1 Ogólne zasady zaliczenia

- Zadanie oceniane jest w skali od 0-50 punktów.
- Warunkiem koniecznym jest indywidualne oddanie zadania.
- Ocenie podlega m.in. stopień zrozumienia mechanizmów odpowiedzialnych za obserowanane zmiany w wydajności.
- Weryfikacja posiadanego modelu procesora i **dopasowanie do jego specyfikacji zooptymalizowanego kodu** (cache i jednostki wektorowe, w tym ich typ i generacja, oraz inne) jest konieczne do uzyskania pozytywnej oceny.
- Zadania umieszczone w systemie Moodle/UPEL po terminie będą bezwględnie oceniane na maksymalnie 25 punktów.

2 Cel zadania

Celem tego zadania jest implementacja funkcji, która przetwarza duże ciągi tekstowe (np. zawartość dużego pliku) wykonując szereg operacji normalizacyjnych oraz optymalizacyjnych. Funkcja powinna:

- Usuwać znaki spoza zakresu drukowalnych (np. wszystkie znaki o kodzie poniżej 32 lub powyżej 126).
- Zamieniać sekwencje białych znaków (spacje, tabulatory, nowe linie) na pojedynczą spację.
- Konwertować wszystkie litery do małych liter.

- Konwertować interpunkcję na przecinki.
- Eliminować duplikaty wyrazów występujących jeden po drugim (np. ”hello hello world” → ”hello world”).

3 Wymagania implementacyjne

1. **Wersja bazowa:** Napisz funkcję realizującą powyższe operacje przy użyciu standardowych metod `std::string` i algorytmów C++. Kod powinien być czytelny, ale nie musi być zoptymalizowany pod kątem wydajności.
2. **Profilowanie:** Uruchom profilowanie funkcji (np. za pomocą `gprof` lub `perf`) i zidentyfikuj krytyczne fragmenty kodu.
3. **Optymalizacje:** Na bazie wersji bazowej wykonaj serię modyfikacji:
   - **Prealokacja pamięci:** Upewnij się, że wynikowy `std::string` ma zarezerwowaną odpowiednią ilość pamięci przed rozpoczęciem przetwarzania.
   - **In-place transformation:** Zminimalizuj liczbę alokacji poprzez modyfikację ciągu w miejscu, jeśli to możliwe.
   - **Iteratory i algorytmy standardowe:** Wykorzystaj iteratory oraz funkcje z biblioteki standardowej (`std::transform`, `std::remove_if`, itp.).
   - **Wersja bare-metal:** Zaimplementuj wersję operującą na surowych tablicach znaków z użyciem wskaźników oraz funkcji takich jak `memcpy()`.
   - **Wersja równoległa:** Rozważ przetwarzanie tekstu w wielu wątkach (`std::thread`, `OpenMP`), dzieląc dane na fragmenty, a następnie scalając wyniki.
4. **Testy i analiza:** Dla każdej wersji:
   - Zmierz czas wykonania.
   - Wykonaj profilowanie, aby ocenić wykorzystanie CPU i pamięci.
   - Porównaj uzyskane wyniki.
   - Inspirując się poniższą instrukcją wykonaj testy dla różnych rozmiarów problemów, oraz różnych proporcji występowanych modyfikacji tekstu.

https://github.com/flame/how-to-optimize-gemm

4 Etapy rozwoju zadania

W miarę postępów należy wykonywać kopie kolejnych wersji kodu (np. `normalize1.cpp`, `normalize2.cpp`, ...) i wprowadzać kolejne optymalizacje:

1. **Wersja 1:** Implementacja bazowa bez optymalizacji.
2. **Wersja 2:** Optymalizacja prealokacji pamięci...
3. **Wersja ...:** Kolejne optymalizacje.

5 Sprawozdanie

W sprawozdaniu opisz:

- Szczegółowy opis implementacji każdej wersji.
- Wyniki pomiarów czasowych oraz analizy profilera.
- Wnioski dotyczące efektywności poszczególnych optymalizacji oraz wyjaśnienie zastosowanych mechanizmów (np. minimalizacja alokacji, lokalność pamięci, wykorzystanie równoległości, itp.).
- Problemy napotkane przy implementacji i sposób ich rozwiązania.