16.04.2025
Maksymilian Katolik

Optymalizacja Kodu Na Różne Architektury

Zadanie 1

Parametry sprzętu

Parametr	Wartość
Producent	Intel
Model	11th Gen Intel(R) Core(TM) i5-11300H
Mikroarchitektura	Tiger Lake
Rdzenie	4
Wątki	8
Częstotliwość	3.10GHz
GFLOPS	198.4
GFLOPS/rdzeń	49.6

Informacje na temat GFLOPS znajdują się pod tym linkiem:
https://www.intel.com/content/www/us/en/content-details/841556/app-metrics-for-intel-microprocessors-intel-core-processor.html

Uwaga - wszystkie kompilacje wersji programu zostały wywołane poleceniem:
g++ normalizeX.cpp -pg -o normalizeX.exe
oraz drugi raz z opcją -O2
nie używałem tutaj:
-march=tigerlake ani *-march=native*
ponieważ zauważyłem znaczny wzrost czasu wywoływania programów

Plik wejściowy waży ~100KB i jest wypełniony znakami i wyrazami mającymi sprawdzić poprawne działanie programu.

Kod

Wszystkie kody oraz niektóre wyniki profiler'a można znaleźć pod tym linkiem:
https://github.com/Maksymilian-Katolik/OKNRA-Zadanie1

Problem

Celem zadania jest napisanie funkcji/programu który powinien:

- 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”).

Wersja bazowa

W wersji bazowej programu idea jest pros

- w main czytam plik wejściowy
- wywołuje funkcję NormalizeText (5000 powtórzeń)
- zapisuje wynik do pliku wyjsciowego
- wypisuję czas (mierzony od rozpoczęcia działania funkcji NormalizeText aż do jej zakończenia - odczyt i zapis do pliku nie są brane pod uwagę)

Sama funkcja NormalizeText wywołuje natomiast 3 funkcję:

- cleanChars - w funkcji tej iteruję znaku po znaku po pliku wejściowym i zapisuję do stringa pomocniczego (cleaned), białe znaki (zamienione na spacje), interpunkcyjne znaki (zamienione na przecinki), oraz znaki pomiędzy kodem 32 a 126 zamienione na małe litery

- removeWS - w tej funkcji usuwam nadmierne whitespace'y (czyli spacje bo funkcja "cleanChars" zamieniła wszystkie białe znaki na spacje)
- removeDuplicates - usuwam tu powtórzone słowa - słowo rozpoczyna się po spacji i kończy spacją (więc ciąg " hello hello, " są traktowane jak 2 różne wyrazy) - tutaj po kolei odczytuje słowa (patrząc gdzie są spacje) i zapamiętuje jedno ostatnie słowo, po tym jak kolejne słowo będzie takie samo to go nie zapisuje do pliku wyjściowego

Wyniki

Kompilacja bez -O2:
Time: 52.5351s

Profiler:Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
18.29	1.50	1.50	5000	0.30	0.40	removeDuplicates(std::__cxx11...
15.24	2.75	1.25				_init
14.76	3.96	1.21	5000	0.24	0.59	cleanChars(std::__cxx11::...)
12.13	4.96	0.99	948855000	0.00	0.00	bool __gnu_cxx::operator!=...
9.76	5.75	0.80	5000	0.16	0.40	removeWS(std::__cxx11::basic_string...
8.78	6.47	0.72	1897710000	0.00	0.00	__gnu_cxx...
7.99	7.13	0.66	948845000	0.00	0.00	__gnu_cxx::__...
3.90	7.45	0.32	355425000	0.00	0.00	isPunctuation(char)
3.11	7.71	0.26	948845000	0.00	0.00	__gnu_cxx::__normal_iterator<char ...
2.62	7.92	0.21	28675000	0.00	0.00	std::char_traits<char>::compare(...
1.83	8.07	0.15	61825000	0.00	0.00	bool std::operator==<char...
1.46	8.19	0.12	61825000	0.00	0.00	bool std::operator!=<char,...)
0.12	8.20	0.01	28675000	0.00	0.00	std::__is_constant_evaluated()
0.00	8.20	0.00	5000	0.00	1.39	normalizeText(std::__cxx11::basic_string...
0.00	8.20	0.00	2	0.00	0.00	dclock()

Widać tutaj, że najbardziej kosztowna czasowo jest funkcja removeDuplicates
tuż za nią

Wywołanie z opcją -O2:
Time: 5.79162s

Widać tutaj zdecydowaną poprawę czasu wykonywania
Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
35.09	0.93	0.93	5000	186.00	186.00	removeDuplicates(std::__…)
27.17	1.65	0.72	5000	144.00	144.00	cleanChars(std::__cxx11::basi…
23.40	2.27	0.62				_init
14.34	2.65	0.38	5000	76.00	76.00	removeWS(std::__cxx11::ba…
0.00	2.65	0.00	5000	0.00	406.00	normalizeText(std::__cxx11::basic_string<…
0.00	2.65	0.00	2	0.00	0.00	dclock()

znów przeważa funkcja removeDuplicates (trend pozostał ten sam - najgorsza czasowo jest removeDuplicates potem cleanChars i na końcu removeWS)
widać też brak wyników wywołań prostszych funkcji niz w stosunku do profilera bez opcji -O2.

Prealokacja pamięci

Idea jest prosta za każdym razem gdy potrzebuje jakiejś zmiennej string to prealokuje na nią pamięć.
Przykład:
```
 cleaned.reserve(input.size());
```

Zawsze prealokuję miejsca tyle ma plik wejściowy (bo w najgorszym wypadku plik wyjściowy będzie miał tyle znaków - gdyby nie miał powtórzeń słów, białych znaków oraz gdyby miał tylko dozwolone znaki)

Wyniki

Bez -O2:
Time: 55.8822s

Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
18.12	1.73	1.73	5000	0.35	0.45	removeDuplicates(std::__cxx11::basic_s...
13.61	3.03	1.30				_init
13.40	4.31	1.28	5000	0.26	0.65	cleanChars(std::__cxx11::b...
12.98	5.55	1.24	5000	0.25	0.55	removeWS(std::__cxx11::basic_stri...
11.73	6.67	1.12	1897710000	0.00	0.00	__gnu_cxx::__normal_itera...
10.68	7.69	1.02	948855000	0.00	0.00	bool __gnu_cxx::operato...
9.27	8.57	0.89	948845000	0.00	0.00	__gnu_cxx::__normal_iterator...
2.30	8.79	0.22	355425000	0.00	0.00	isPunctuation(char)
2.25	9.01	0.21	948845000	0.00	0.00	__gnu_cxx::__normal_iterator...
2.20	9.22	0.21	28675000	0.00	0.00	std::char_traits<char>::compare...
1.88	9.40	0.18	61825000	0.00	0.00	bool std::operator!=<char...
1.26	9.52	0.12	61825000	0.00	0.00	bool std::operator==<char...
0.16	9.54	0.01	28675000	0.00	0.00	std::__is_constant_evaluated()
0.10	9.54	0.01	5000	0.00	1.65	normalizeText(std::__cxx11::basi...
0.05	9.55	0.01				main
0.00	9.55	0.00	2	0.00	0.00	dclock()

Trend czasu wykonywania funkcji jest ten sam ale widac mniejszą różnicę między cleanChars i removeWS

Z opcją -O2:
Time: 5.45468s

Widać niewielką poprawę co do wersji bazowej z -O2
Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
30.00	0.75	0.75				_init
28.40	1.46	0.71	5000	142.00	142.00	removeDuplicates(std::__cxx11::b…
27.20	2.14	0.68	5000	136.00	136.00	cleanChars(std::__cxx11::basic_strin…
14.40	2.50	0.36	5000	72.00	72.00	removeWS(std::__cxx11::basic_str…
0.00	2.50	0.00	5000	0.00	350.00	normalizeText(std::__cxx11::basic…
0.00	2.50	0.00	2	0.00	0.00	dclock()

Tu natomiast funkcję removeDuplicates i CleanCHars są bliżej pod względem zajętego przez nie czasu

In-place transformation

3 najważniejsze funkcje programu (cleanChars, removeWS, removeDuplicates) zostały zamienione na wersje gdzie wszystkich tych operacji dokonuje “w miejscu” czyli w samym pliku - nie używam tak dużo dodatkowych zmiennych typu string

Wyniki

Bez opcji -O2:
Time: 15.7501s

Zdecydowana poprawa jeśli chodzi o poprzednie wersje.
Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name

25.56 1.93 1.93 5000 0.39 0.43
removeDuplicatesInPlace(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >&)
24.37 3.77 1.84 _init
23.97 5.58 1.81 5000 0.36 0.44 cleanCharsInPlace(std::__cxx11::...
18.15 6.95 1.37 5000 0.27 0.27 removeWSInPlace(std::__cxx11::bas...
4.90 7.32 0.37 355425000 0.00 0.00 isPunctuation(char)
1.06 7.40 0.08 61825000 0.00 0.00 bool std::operator==<...
0.86 7.46 0.07 28675000 0.00 0.00 std::char_traits<char>::...
0.66 7.51 0.05 61825000 0.00 0.00 bool std::operator!=<ch...
0.40 7.54 0.03 28675000 0.00 0.00 std::__is_constant_evaluated()
0.07 7.55 0.01 std::char_traits<char>::lt(char const&, char const&)
0.00 7.55 0.00 5000 0.00 1.14 normalizeText(std::__cxx11::ba...
0.00 7.55 0.00 2 0.00 0.00 dclock()

Jak widać trend się utrzymuje

Z -O2:
Time: 4.4008s

Profiler:
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls us/call us/call name
48.12 1.15 1.15 5000 230.00 230.00 removeDuplicatesInPlace(std::__cxx11:_..
23.01 1.70 0.55 _init
20.50 2.19 0.49 5000 98.00 98.00 cleanCharsInPlace(std::__…
8.37 2.39 0.20 5000 40.00 40.00 removeWSInPlace(std::_…
0.00 2.39 0.00 2 0.00 0.00 dclock()

Widać znaczną przewagę funckji removeDuplicatesInPlace.
Warto zauważyć, że funkcja removeWSInPlace jest o wiele wydajniejsza.

Iteratory i algorytmy standardowe

Idea jest taka, że zastępuje pętle for czy while - lepszymi rozwiązaniami jak iterator czy funkcja for_each

Wyniki

Bez -O2:

Time: 145.892s

Zdecydowane pogorszenie
Profiler:
(wysłany na githubie bo dużo tam tekstu)
Widać znów przewagę czasową funkcji removeAdjacentDuplicatesSTL

Z O2:
Time: 8.38241s

Znów czas gorszy niż dla innych wariantów z opcją -O2
Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
40.26	1.26	1.26	5000	252.00	252.00	cleanCharsSTL(std::__cx…
23.96	2.01	0.75				_init
19.49	2.62	0.61	5000	122.00	122.00	removeAdjacentDuplicatesSTL(std::__c…
16.29	3.13	0.51	5000	102.00	102.00	removeExtraWhitespaceSTL(std::_…
0.00	3.13	0.00	5000	0.00	476.00	normalizeText(std::__cxx11::basic…
0.00	3.13	0.00	2	0.00	0.00	dclock()

Tutaj widzimy zmianę - funkcja cleanCharsSTL zajmuje zdecydowanie większą część czasu niż reszta funkcji

Wersja bare-metal - wersja 1

Tutaj wykorzystam w moim programie surowe tablice i wskaźniki.
Zamiast operować na string'ach, operuję na wskaźnikach do tablic.

Wyniki

Bez -O2:
Time: 13.9303s
Dotychczas najlepszy - prześcigając wydajnościowo wersję In-place transformation

Profiler:
(znów wysłany na githubie - zajmował by tutaj dużo miejsca)
najważniejszy fragment:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name

28.35	1.10	1.10	5000	220.00	264.00	cleanChars(char const*, u…
20.10	1.88	0.78	5000	156.00	278.00	removeDuplicates[abi:cxx11](ch…
18.81	2.61	0.73				_init
11.08	3.04	0.43	5000	86.00	86.00	removeWS(char const*, unsigne…

Znów obserwujemy że czyszczenie znaków jest najbardziej złożone czasowo

Z -O2:
Time: 4.13362s

Czas podobny do transformacji w miejscu
Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
35.29	0.60	0.60				_init
29.41	1.10	0.50	5000	100.00	100.00	cleanChars(char const*, ..
23.53	1.50	0.40	5000	80.00	80.00	removeDuplicates[abi:cx…
11.76	1.70	0.20	5000	40.00	40.00	removeWS(char const*, …
0.00	1.70	0.00	5000	0.00	220.00	normalizeText(std::__cxx11::basic_string…
0.00	1.70	0.00	2	0.00	0.00	dclock()

Najwięcej czasu zajmuje sam init, ale tuż za nim cleanChars - znowu bardziej pochłania czas niż usuwanie duplikatów

Wersja bare-metal - wersja 2

Główną różnicą jest użycie memcpy - używane do kopiowania bajtów w funkcji removeDuplicates by zmaksymalizować wydajność:

Wyniki

Bez -O2:
Time: 6.84841s
Znaczna poprawa w stosunku do wersji bez memcpy

Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
33.20	0.86	0.86	5000	172.00	228.00	cleanChars(char con..
23.17	1.46	0.60	5000	120.00	120.00	removeDuplicates(char cons…
19.69	1.97	0.51				_init

13.13	2.31	0.34	5000	68.00	68.00	removeWS(char const*, char…
10.81	2.59	0.28	355425000	0.00	0.00	isPunctuation(char)
0.00	2.59	0.00	5000	0.00	416.00	normalizeTextBareMetal(std::__c…
0.00	2.59	0.00	5000	0.00	0.00	std::__new_allocator<char>::...
0.00	2.59	0.00	2	0.00	0.00	dclock()

Znów przeważa czyszczenie znaków

z -O2:
Time: 3.47481s
Znów widać poprawę (w porównaniu do wersji bez memcpy)

Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
37.31	0.50	0.50				_init
29.10	0.89	0.39	5000	78.00	78.00	cleanChars(char const*, ch…
26.87	1.25	0.36	5000	72.00	72.00	removeDuplicates(char const*, cha…
6.72	1.34	0.09	5000	18.00	168.00	normalizeTextBareMetal(std::__cxx11::...
0.00	1.34	0.00	2	0.00	0.00	dclock(

Funkcje cleanChars i removeDuplicates mają podobny czas, ale zastanawiający jest brak informacji o funkcji removeWS - być może program wykonuje się na tyle szybko, że czas zajęty przez removeWS nie jest zapisywany? Być może jest to własność opcji optymalizacji -O2.

Wersja równoległa

Tutaj funkcję cleanChars, removeWS i removeDuplicates są takie same jak w wersji z prealokacją pamięci. Natomiats w fukncji NormalizeTextParallel dzielę tekst wejściowy na 8 części i daję każdą część do przemielenia innemu wątkowi (jest ich również 8)

Wyniki

Wersja bez -O2:
Time: 97.4513s

Ciekawe - jest gorsze od samej wersji z prealokacją bez dzielenia na wątki
Profiler:
(w całości na githubie)
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
14.19	1.49	1.49	5000	298.00	409.00	removeDuplicates(std::__c…
12.48	4.14	1.31	4349	301.22	924.03	removeWS(std::__…
9.57	7.50	1.00	8089	124.24	348.79	cleanChars(std::__cxx11::...

W zwykłej wersji z prealokacją pamięci (bez dzielenia na wątki) cleanChars i removeWS miały podobne czasy jednak tutaj cleanChars jest zdecydowanie szybsze

Z O2:
Time: 5.38048s

Bardzo podobnie jak wersja 2 programu
Profiler:
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
32.65	0.95	0.95				_init
28.52	1.78	0.83	5000	166.00	166.00	removeDuplicates(std::__cxx11::bas…
21.65	2.41	0.63	39370	16.00	16.00	cleanChars(std::__cxx11…
16.84	2.90	0.49	39878	12.29	12.29	removeWS(std::__cxx11::bas…
0.34	2.91	0.01				std::thread::_State_impl<s…
0.00	2.91	0.00	39279	0.00	28.51	processChunk(std::__cxx11::ba…
0.00	2.91	0.00	5000	0.00	166.00	normalizeTextParallel(std::__cx….
0.00	2.91	0.00	2	0.00	0.00	dclock()

Bardzo podobny rozkład jak w wersji 2 programu z opcją -O2 - czego też się spodziewałem

Wyniki zbiorczo:

Bez opcji O2:

- wersja bazowa - Time: 52.5351s
- prealokacja pamięci - Time: 55.8822s
- in-place transformation - Time: 15.7501s
- iteratory i algorytmy standardowe - Time: 145.892s
- wersja bare-metal (bez memcpy) - Time: 13.9303s
- wersja bare-metal (z memcpy) - Time: 6.84841s
- wersja równoległa - Time: 97.4513s

Najlepszą optymalizacją okazał się sposób baremetal z memcpy .
Dużym, negatywnym, zaskoczeniem są wyniki iteratorów i wersji równoległej - spodziewałem się tam lepszych wyników - nie gorszych od wersji bazowej

Z opcją O2:

- wersja bazowa - Time: 5.79162s
- prealokacja pamięci - Time: 5.45468s
- in-place transformation - Time: 4.4008s
- iteratory i algorytmy standardowe - Time: 8.38241s
- wersja bare-metal (bez memcpy) - Time: 4.13362s
- wersja bare-metal (z memcpy) - Time: 3.47481s
- wersja równoległa - Time: 5.38048s

Najlepszą optymalizacją znów okazał się sposób baremetal z memcpy .
Widać jednak, że optymalizacją opcją -O2 poprawiła negatywne efkety iteratorów i wersji równoległej - pierwsza dalej jest gorsza od bazowej, ale druga - wersja równoległa- jest już trochę lepsza od bazowej