05.06.2025
Maksymilian Katolik

Optymalizacja Kodu Na Różne Architektury

Zadanie 2

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++ gemmX.cpp -O2 -march=native -o normalizeX.exe

Kod

Wszystkie kod i wyniki wraz z pomiarami czasu i GFLOPS:
https://github.com/Maksymilian-Katolik/OKNRA-Zadanie2

Cel Zadania

Celem zadania jest implementacja i optymalizacja algorytmu eliminacji Gaussa, z uwzględnieniem mikroarchitektury posiadanego procesora oraz praktycznego wykorzystania technik niskopoziomowej optymalizacji kodu.

Jako sposób sprawdzania działania kolejnych optymalizacji będę mierzył czas wykonywania algorytmu oraz szacowane GFLOPS - przyjmuje że FLOP dla tego algorytmu to około $\frac{2}{3}n^3$ - gdzie n to wielkość macierzy

Wersja bazowa

Algorytm eliminacji Gaussa służy do rozwiązywania układów równań liniowych postaci:

Ax=b

gdzie:

- Ato macierz współczynników (rozmiaru n×n),
- x to wektor niewiadomych,
- b to wektor wyrazów wolnych.

W mojej implementacji ten algorytm składa się z 2 części:
a) Eliminacja (górna trójkątność):

Dla każdego wiersza k od 0 do n−1::

- Dzielenie przez element główny (pivot) A[k][k]
- Dla każdego wiersza poniżej (i>k):
    - Obliczamy współczynnik eliminacji:
      $factor = \frac{A[i][k]}{A[k][k]}$
    - Odejmujemy odpowiednio przeskalowany wiersz k od wiersza i:
      $A[i][j] = factor \cdot A[k][j]$
    - $b[i] = factor \cdot b[k]$

Po tym etapie macierz A ma postać górnej trójkątnej.

b) Podstawianie wsteczne (back-substitution):

Rozwiązujemy układ równań od końca:

$$x[i] = \frac{b[i] - \sum_{j=i+1}^{n-1} A[i][j] \cdot x[j]}{A[i][i]}$$

Zastosowana wersja algorytmu **nie zawiera pivotingu**:

- Nie zamieniam wierszy w celu unikania dzielenia przez małe liczby
- Prostsze i szybsze, ale mniej stabilne numerycznie

Zastosowane optymalizacje

Spłaszczenie macierzy z 2D na 1D

Oryginalnie macierz przechowywałem jako `vector<vector<double>>`, co wiąże się z kosztami alokacji i złym rozkładem danych w pamięci. Zamiana na jednowymiarowy `vector<double>` pozwala na:

- Lepszą lokalność pamięci – dane są przechowywane w jednym ciągłym bloku

- Łatwiejszą optymalizację przy użyciu SIMD i cache blocking
- Mniejszy narzut związany z dereferencją wskaźników

SIMD z AVX2

Zamiast wykonywać operacje na pojedynczych elementach, używam wektorów 256-bitowych (8 `double` na raz) i instrukcji AVX2, co:

- przyspiesza przetwarzanie pętli eliminacji przez równoległe wykonywanie obliczeń
- zmniejsza liczbę iteracji pętli
- efektywniej wykorzystuje jednostki obliczeniowe CPU

Muszę tutaj obsłużyć “ogonek” – elementy, które nie mieszczą się w pełnym wektorze

OpenMP

Tutaj wykorzystuje wielu rdzeni CPU.

Dyrektywa `#pragma omp parallel for`, pozwala równolegle przetwarzać wiersze i przy każdym kroku eliminacji.

Daje to wzrost wydajności dzięki równoległemu przetwarzaniu wielu wierszy na raz.

Usunięcie nieefektywnych operacji

Minimalna zmiana w algorytmie eliminacji - zamiana:
double pivot = get(A, k, k, n);
…
double factor = get(A, i, k, n) / pivot;

na:
double pivot = get(A, k, k, n);
double inv_pivot = 1.0 / pivot;
…
double factor = get(A, i, k, n) * inv_pivot;

to w teorii powinna przyspieszyć działanie bo używanie działania mnożenia “*” jest szybsze niż dzielenie “/”

Cache blocking

Tutaj dzielę macierz na bloki o ustalonym rozmiarze (najlepszy wynik dostałem dla rozmiaru 16x16)
Operacje wykonywane są wewnątrz bloków, co poprawia trafienia do cache L1/L2 i powinno poprawić wydajność przy dużych rozmiarach macierzy.

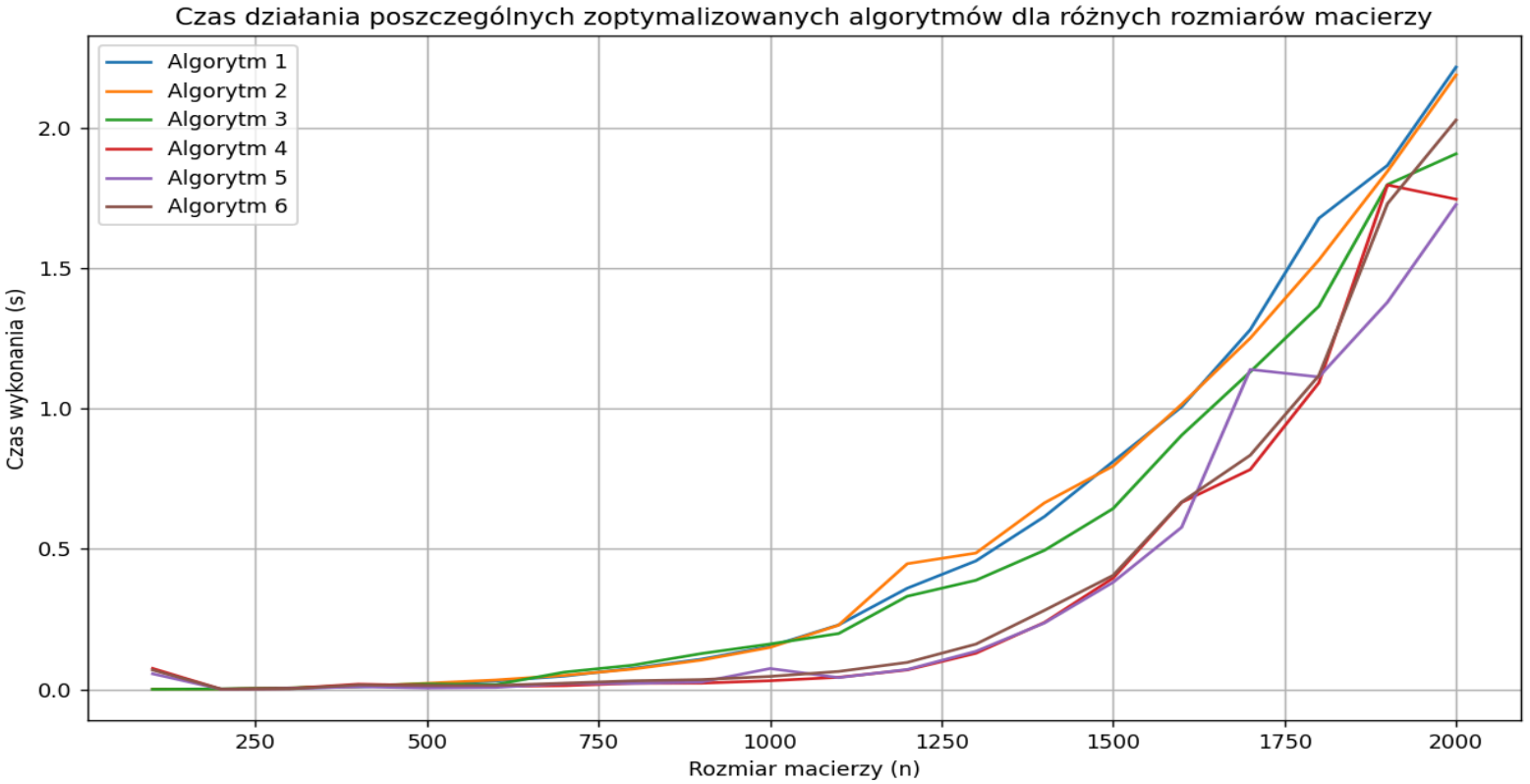
Wyniki i pomiary

Dla każdej z wersji programu przeprowadzałem testy czasu i GFLOPS (by sprawdzić w jakim stopniu używają dostępnych zasobów)

Czas

Dla każdej wersji algorytmu mierzyłem czas wykonania funkcji eliminate_gauss oraz back

Z tych pomiarów powstał taki oto wykres:



Widać tutaj wzrost czasu o charakterze wykładniczym.
Zauważyć spadek tej krzywej dla algorytmów 4, 5, 6 - czyli począwszy od użycia OpenMP do rozłożenia działań na wszystkie dostępne wątki na sprzęcie.

Inne statystyki:
Średnia czasów dla każdego z algorytmów:
Średni czas dla Algorytmu 1 (bazowa wersja): 0.548458 s
Średni czas dla Algorytmu 2 (spłaszczenie macierzy): 0.544969 s
Średni czas dla Algorytmu 3 (SIMD): 0.482927 s
Średni czas dla Algorytmu 4 (OpenMP - wielowątkowość): 0.358445 s
Średni czas dla Algorytmu 5 (usunięcie nieefektywnych działań) : 0.351194 s
Średni czas dla Algorytmu 6 (cache blocking): 0.381662 s

Najlepszy średni czas uzyskany został przez OpenMP i usunięcie nieefektywnych działań - jednak widać że gdyby te algorytmy stosować osobno to OpenMP jest zdecydowanie najlepszy.

Ciekawym wynikiem jest to że cache blocking okazał się być minimalnie gorszy od algorytmu bez niego - może być to spowodowane brakiem testów dla innych rozmiarów niż 16x16, 32x32, 64x64 - gdzie finalny wynik pochodzi z podziału na bloki 16x16.

Na wykresie można zauważyć, że czasy dla małych wielkości macierzy tzn. dla n < 1000 są bardzo podobne więc jeszcze sprawdziłem średnią dla czasów mierzonych dla macierzy o wielkościach od 1000 do 2000.

Średni czas dla Algorytmu 1 (bazowa wersja): 0.970526 s
Średni czas dla Algorytmu 2 (spłaszczenie macierzy): 0.963705 s
Średni czas dla Algorytmu 3 (SIMD): 0.847845 s
Średni czas dla Algorytmu 4 (OpenMP - wielowątkowość): 0.635454 s
Średni czas dla Algorytmu 5 (usunięcie nieefektywnych działań): 0.625044 s
Średni czas dla Algorytmu 6 (cache blocking): 0.675543 s

Znów widać dużą zaletę wykorzystania OpenMP.
Cache blocking wypadł tu jeszcze gorzej.

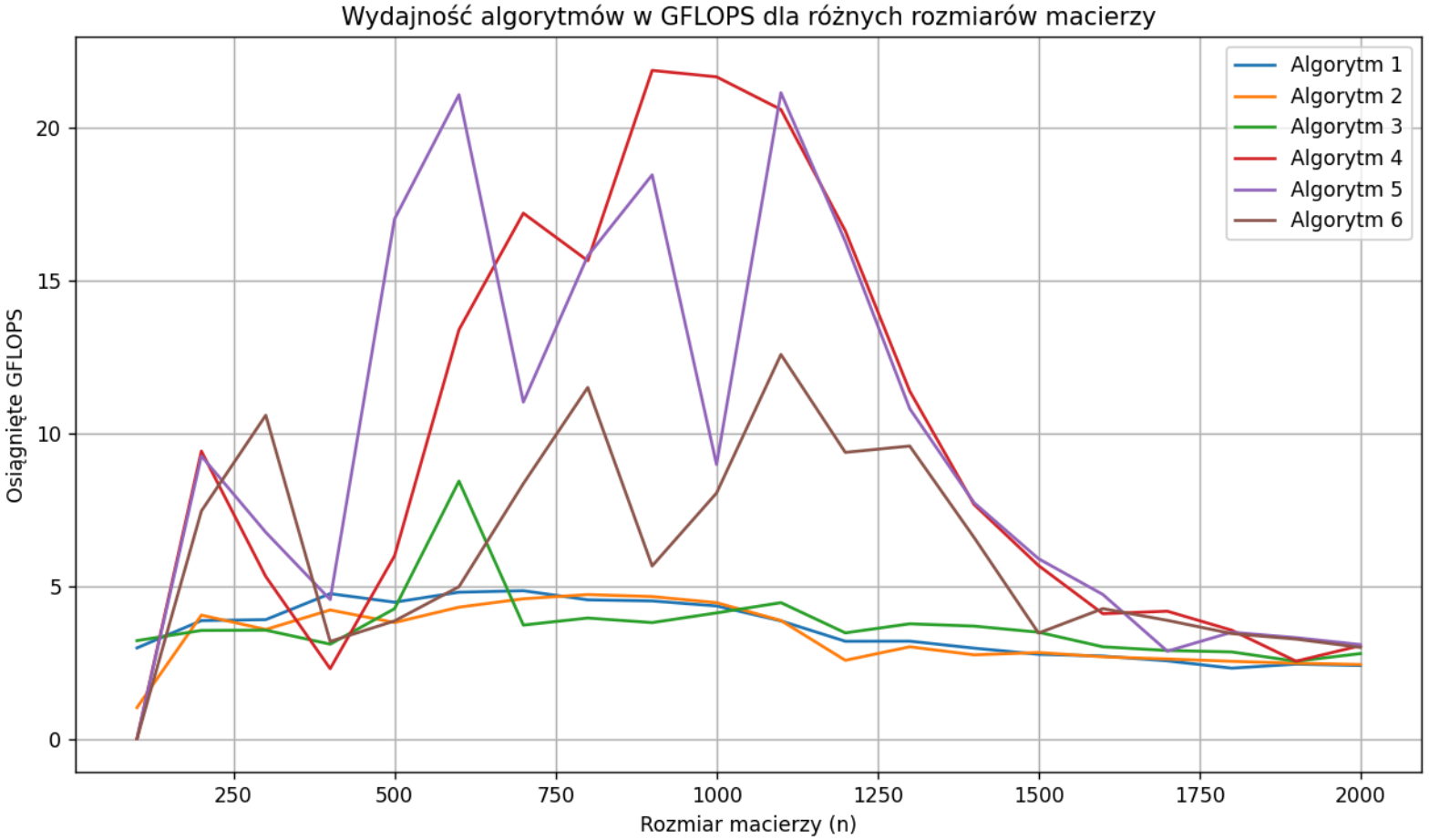
Ostatnia statystykę - sprawdziłem dla jakich rozmiarów macierzy (n) dana optymalizacja przyniosła najelpsze zyski czasowe (ile punktów procentowych spadł czas dla danej wielkości macierzy w danej optymalizacji w porównaniu do bazowego algorytmu)

Porównanie optymalizacji względem bazowego algorytmu:
Największa oszczędność czasu dla Algorytmu 2: 8.85% przy n = 1800
Największa oszczędność czasu dla Algorytmu 3: 43.08% przy n = 600
Największa oszczędność czasu dla Algorytmu 4: 81.24% przy n = 1100
Największa oszczędność czasu dla Algorytmu 5: 81.73% przy n = 1100
Największa oszczędność czasu dla Algorytmu 6: 73.37% przy n = 1200

Algorytm 2 (spłaczenie macierzy) miał bardzo podobne wyniki.
Dla reszty optymalizacji widać, że największe zyski były przy większych macierzach dla n ~1000.

GFLOPS

Mierzyłem również szacowane GFLOPS dla każdej wersji programu.
Tak prezentuje się wykres złożony ze wszystkich tych wyników:



Widać, że algorytmy 4,5,6 znów są na przodzie i najlepiej wykorzystują zasoby komputera (6ty algorytm trochę gorzej).
Zaobserwować można “turbulencje” przy środkowych wielkościach macierzy - od n = 5000 do n = 1500)

Do tego zebrałem jeszcze dane a średnich GFLOPSach każdej wersji programu, maksymalnych GFLOPS i dla jakiej wielkości macierzy wystąpiły oraz dla jakiego n wystąpił największy wzrost GFLOPS w porównaniu z kodem bazowym:

Algorytm 1 (bazowy algorytm):
- Maksymalne GFLOPS: 4.8532 przy n = 700
- Średnie GFLOPS: 3.5767

Algorytm 2 (spłaszczenie macierzy):
- Maksymalne GFLOPS: 4.7275 przy n = 800
- Średnie GFLOPS: 3.3636
- Największy wzrost GFLOPS względem bazowego: +9.71% przy n = 1800.0

Algorytm 3 (SIMD):
- Maksymalne GFLOPS: 8.4417 przy n = 600
- Średnie GFLOPS: 3.7381
- Największy wzrost GFLOPS względem bazowego: +75.69% przy n = 600.0

Algorytm 4 (OpenMP):
- Maksymalne GFLOPS: 21.8909 przy n = 900
- Średnie GFLOPS: 9.6162
- Największy wzrost GFLOPS względem bazowego: +433.19% przy n = 1100.0

Algorytm 5 (Usunięcie nieefektywnych działań):
- Maksymalne GFLOPS: 21.1574 przy n = 1100
- Średnie GFLOPS: 9.6219
- Największy wzrost GFLOPS względem bazowego: +447.33% przy n = 1100.0

Algorytm 6 (cache blocking - rozmiar bloku = 16x16):
- Maksymalne GFLOPS: 12.5876 przy n = 1100
- Średnie GFLOPS: 6.1591
- Największy wzrost GFLOPS względem bazowego: +225.63% przy n = 1100.0

Widać znowu wydajne działanie algorytmów od momentu zastosowania wielowątkowości.
Usunięcie nieefektywnych działań nie zmieniło tutaj dużo.
I znów widać pogorszenie się programu po zastosowaniu cache blockingu.