Optymalizacja Kodu na Różne Architektury:

Zadanie 2: Optymalizacja algorytmu eliminacji Gaussa

Autor: Krzysztof Solecki

1. Procesor:

1.1. Parametry:

Parametr	Wartość
Producent	Intel
Model	Core i5-12450H
Mikroarchitektura	Alder Lake
Rdzenie	8
Wątki	12
Częstotliwość bazowa	2 GHz
Częstotliwość turbo	4,4 GHz
Cache L3	12 MB
GFLOPS	256
GFLOPS/rdzeń	64

1.2. Wyznaczenie wartości GFLOPS/rdzeń:

$$\frac{GFLOPS}{rdzeń} = \frac{256}{4} = 64$$

Wartości GFLOPS dla procesorów firmy Intel można sprawdzić pod tym linkiem.

Można także skorzystać ze wzoru podanego w pliku *PlotAll.m:*

$$nflops\_per\_cycle * nprocessors * GHz\_of\_processor = 32 * 1 * 2 = 64$$

Najpierw należy sprawdzić mikroarchitekturę naszego procesora:

$$cat\ /sys/devices/cpu/caps/pmu\_name$$

Następnie sprawdzamy wartość $FP64 \equiv nflops\_per\_cycle$ pod tym linkiem.
Zarówno pierwszy jak i drugi sposób doprowadził do tego samego wyniku - 64

2. Optymalizacje:

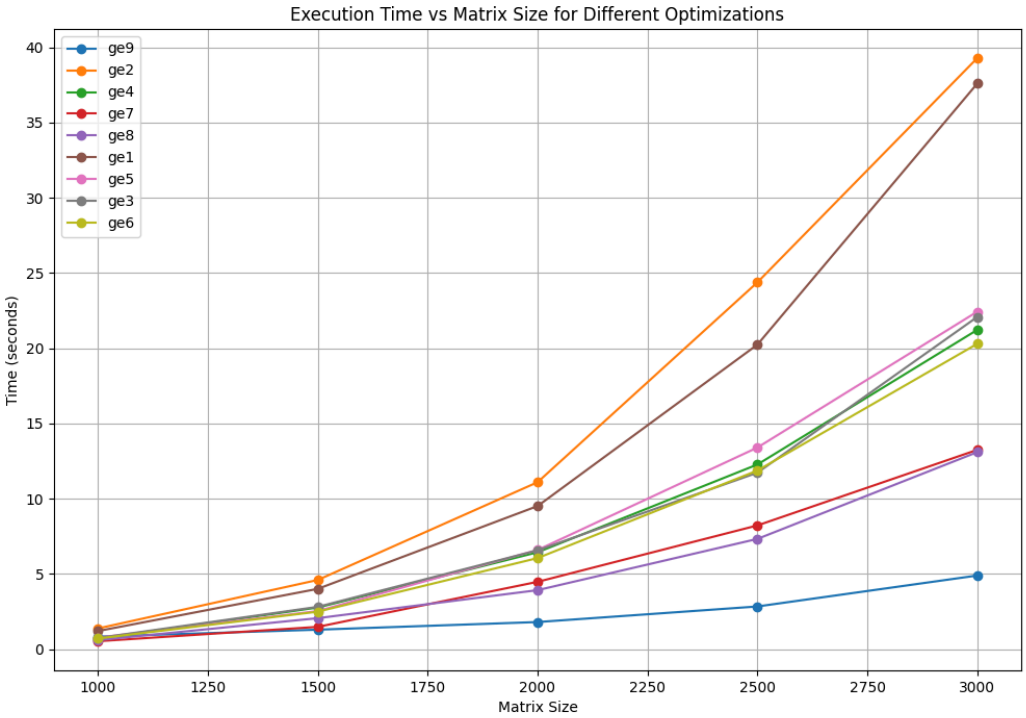
1. **ge1.c - Bazowa wersja programu:** Ta wersja służy jako punkt odniesienia dla wszystkich innych optymalizacji.
2. **ge2.c - Umieszczenie liczników pętli w rejestrach:** Rejestry są najszybszym rodzajem pamięci dostępnej dla procesora. Przeniesienie liczników pętli do rejestrów może przyspieszyć operacje na tych licznikach.
3. **ge3.c - Umieszczenie w rejestrze wartość multiplayer, która najczęściej się powtarza:** Podobnie jak w przypadku liczników pętli, przeniesienie często używanej wartości do rejestru może przyspieszyć operacje na tej wartości.
4. **ge4.c - Rozwinięcie najbardziej zagnieżdżonej pętli do 8 iteracji:** Rozwinięcie pętli może zredukować koszt operacji skoku związanych z pętlą, co może przyspieszyć wykonanie kodu.
5. **ge5.c - Indeksowanie macierzy przez makro:** Użycie makra do indeksowania macierzy może poprawić czytelność kodu i zredukować ryzyko błędów.
6. **ge6.c - Zastosowanie operacji wektorowych _m128d:** Operacje wektorowe mogą przetwarzać wiele danych jednocześnie, co może znacznie przyspieszyć obliczenia. Instrukcje _m128d mogą przetwarzać 2 liczby zmiennoprzecinkowe podwójnej precyzji na raz.
7. **ge7.c - Zastosowanie operacji wektorowych _m256d:** Podobnie jak _m128d, ale instrukcje _m256d mogą przetwarzać 4 liczby zmiennoprzecinkowe podwójnej precyzji na raz, co może przynieść jeszcze większe przyspieszenie.
8. **ge8.c - Zastosowanie prefetchingu w celu minimalizacji opóźnień w dostępie do pamięci:** Prefetching polega na wczytywaniu danych do pamięci podręcznej z wyprzedzeniem, zanim będą one potrzebne, co może zredukować opóźnienia związane z dostępem do pamięci.
9. **ge9.c - Zastosowanie biblioteki OpenMP w celu zrównoleglenia części programu wykorzystując potencjał wielu rdzeni CPU:** Zrównoleglenie kodu za pomocą OpenMP może znacznie przyspieszyć wykonanie programu na systemach wielordzeniowych, dzieląc obliczenia między rdzenie.

3. Dostosowanie do swojego CPU:

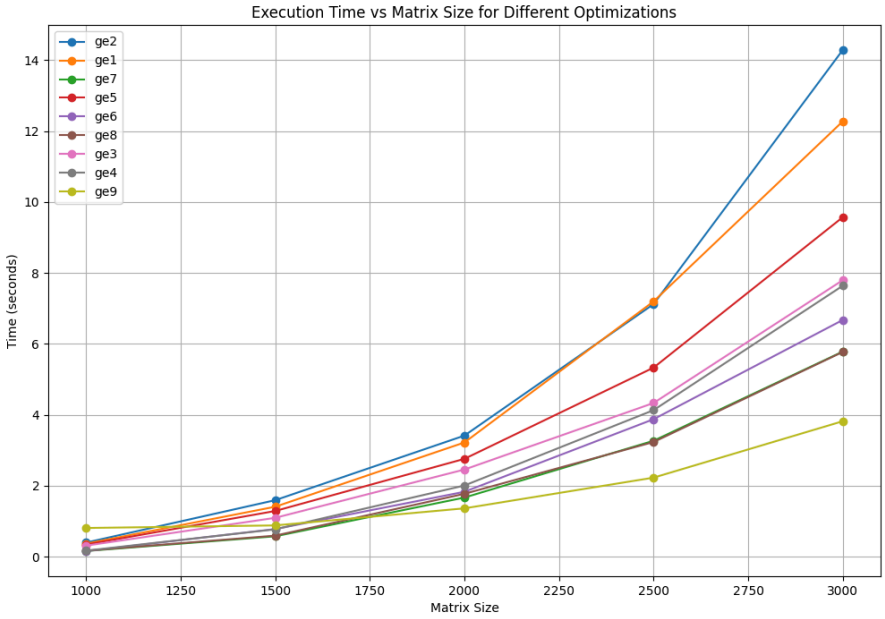
Uruchomienie z flagą optymalizującą dla danej mikroarchitektury: $-\ march\ =\ alderlake$

4. Porównanie wyników:

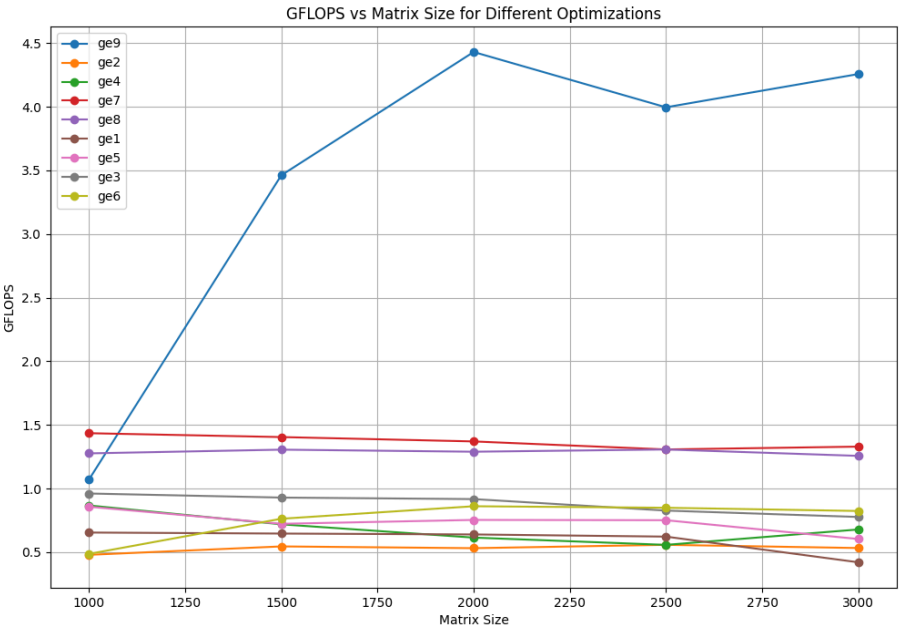
4.1 Wyniki czasowe bez optymalizacji -O2:



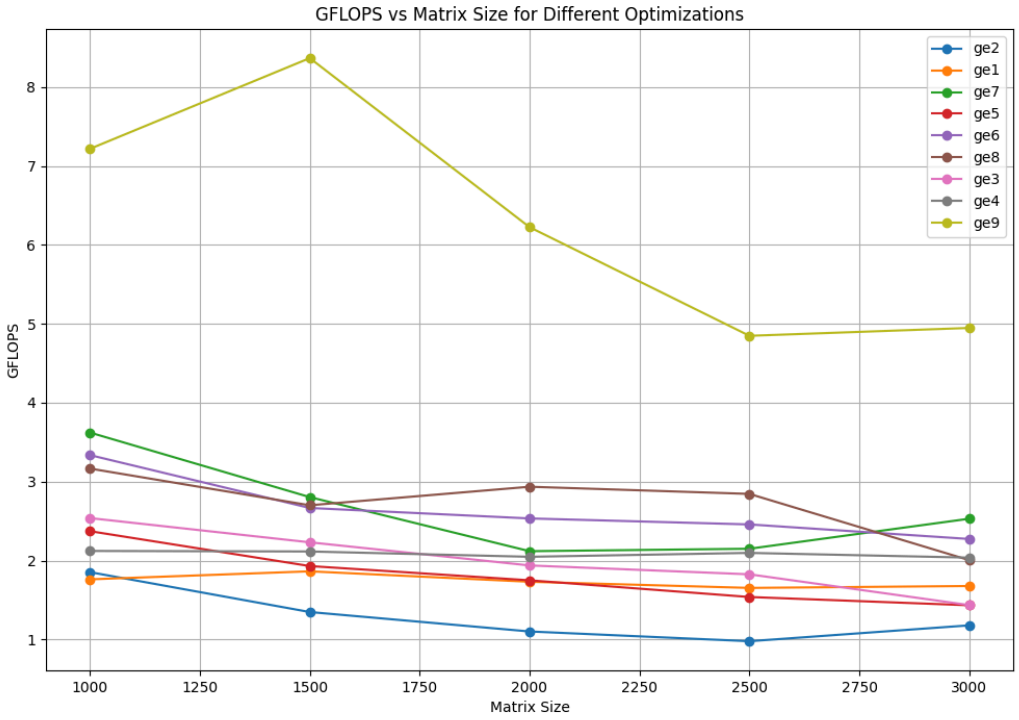
4.2 Wyniki czasowe z optymalizacją -O2:



4.3 Wyniki GFLOPS bez optymalizacji -O2:



4.4 Wyniki GFLOPS z optymalizacją -O2:

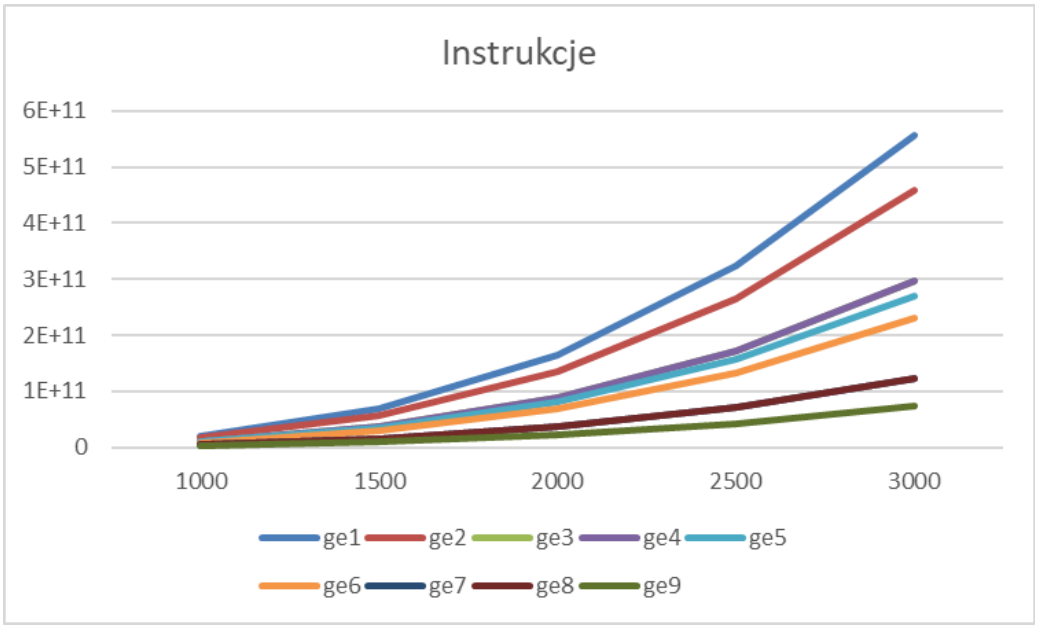


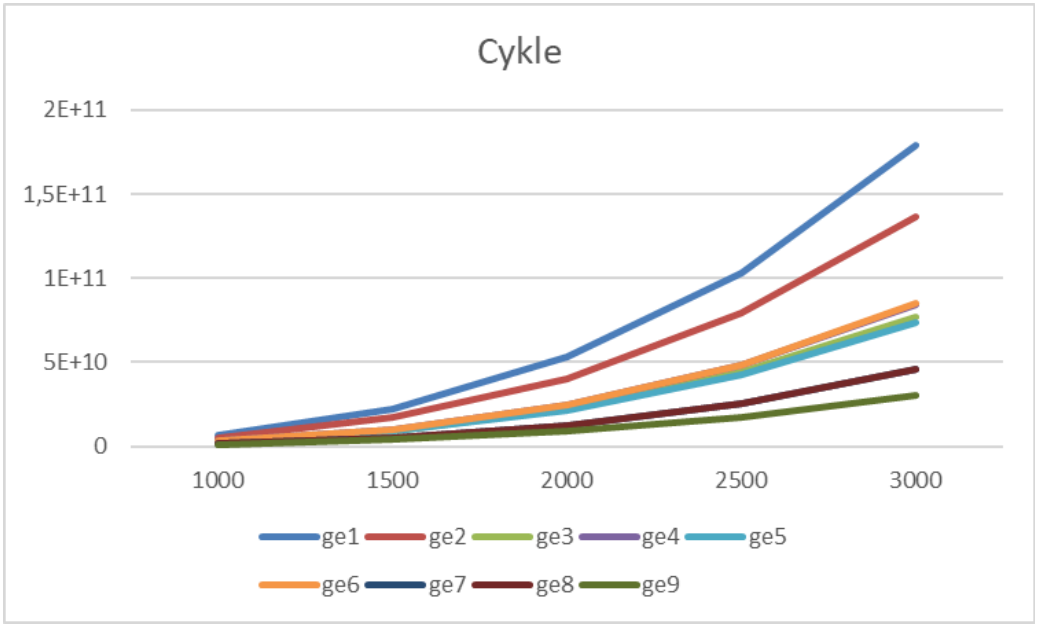
4.5 Prosty algorytm weryfikujący rozwiązanie:

```
double check = 0.0;
   for (i = 0; i < SIZE; i++) {
      for (j = 0; j < SIZE; j++) {
         check = check + matrix[i][j];
      }
   }
```

Ten fragment kodu został dodany do każdej wersji optymalizacji algorytmu eliminacji Gaussa w celu zweryfikowania, czy dla danego rozmiaru macierzy wynikowe macierze się nie różnią.
W moim przypadku wszystkie wartości zmiennej *check* były prawidłowe.

4.5 Pomiary liczników procesora PAPI:





5. Wnioski:

- Analiza wyników:
    - Najwyższy wynik GFLOPS uzyskany w trakcie obliczeń był równy ~9 co stanowi 14% teoretycznej wartości; mogły mieć na to wpływ m. in. inne procesy w systemie, które ograniczyły dostępność mocy obliczeniowej procesora.
    - Rozwinięcie pętli do 8 iteracji przyniosło zauważalną poprawę.
    - Najwyższe wyniki GFLOPS oraz najmniejszą ilość instrukcji i cykli uzyskaliśmy przy zastosowaniu wektorów __m256d z prefechingiem i wykorzystaniem biblioteki OpenMP.
    - Zastosowanie flagi optymalizacyjnej O2 przy kompilacji znacznie przyspieszyło działanie kodu.
- Wnioski ogólne:
    - Umieszczanie zmiennych w rejestrze może znacznie przyspieszyć działanie programu
    - Ręczne rozwinięcie pętli może znacząco zwiększyć wydajność
    - Zastosowanie wektorów __256d znacznie przyspiesza wydajność
    - Flaga 02 znacznie przyspiesza wykonywanie programu
    - Zrównoleglenie często wykonywanych fragmentów kodu za pomocą OpenMP znacząco przyspiesza wydajność, ale trzeba liczyć się z potencjalnymi problemami typu race condition.

6. Źródła:

- Github - https://github.com/flame/how-to-optimize-gemm/wiki
- GFLOPS - https://en.wikipedia.org/wiki/FLOPS
- Algorytm eliminacji Gaussa - https://en.wikipedia.org/wiki/Gaussian_elimination
- CPU Specyfikacja - https://www.intel.com/content/www/us/en/products/sku/132222/intel-core-i512450h-processor-12m-cache-up-to-4-40-ghz/specifications.html