Optymalizacja kodu na różne architektury
Zadanie domowe nr 2

Wojciech Michaluk

30.05.2025

1 Opis zadania

Celem zadania jest implementacja i optymalizacja algorytmu eliminacji Gaussa z uwzględnieniem mikroarchitektury posiadanego procesora oraz praktycznego wykorzystania technik niskopoziomowej optymalizacji kodu. Należy wzorować się na laboratoriach oraz **wskazanej instrukcji**. Trzeba również napisać algorytm weryfikujący poprawność rozwiązania oraz oszacować GFLOPS-y (uwzględniając rzeczywistą złożoność obliczeniową). Przy dopasowaniu rozwiązania do posiadanego modelu procesora szczególnie istotne są:

- mierzenie wysycenia FLOPS-ów procesora,
- poprawna wektoryzacja obliczeń,
- lokalność danych.

1.1 Weryfikacja parametrów posiadanego procesora

Parametr procesora	**Wartość parametru**
Producent	Intel
Model	i5-8300H
Liczba rdzeni	4
Liczba procesorów logicznych (wątków)	8
Częstotliwość podstawowa	2.30 GHz
Częstotliwość turbo	4.00 GHz
Pamięć podręczna (poziom 1)	256 kB
Pamięć podręczna (poziom 2)	1 MB
Pamięć podręczna (poziom 3)	8 MB
GFLOPS	147.2

Tabela 1: Podstawowe parametry procesora i ich wartości

1.2 Wyznaczenie maksymalnej teoretycznej wartości GFLOPS-ów

Wartość GFLOPS-ów dla procesora znalazłem, korzystając z udostępnionych przez firmę Intel danych pod **tym linkiem**. W tabeli dla mojego procesora (**i5-8300H**) można odczytać wartość: 147.2 GFLOPS, którą umieściłem w powyższej tabeli.
Szczególnie interesuje mnie wartość GFLOPS/rdzeń, zatem biorąc pod uwagę, że liczba rdzeni wynosi 4, otrzymuję $\frac{147.2}{4} = 36.8$ GFLOPS/rdzeń. Jest to maksymalna teoretyczna wartość, jaką mogę osiągnąć. W dalszej części sprawdzę, w jakim stopniu uda się do niej zbliżyć.

1.3 Dopasowanie rozwiązania do posiadanego modelu procesora

Biorąc pod uwagę parametry procesora i wzorując się na podlinkowanej instrukcji, będę testował algorytm dla rozmiarów macierzy od 10 do 1200 z krokiem wynoszącym 10, co pozwoli zaobserwować przypadek, kiedy macierz już się nie mieści w pamięci cache.
Przy kompilacji programu będę ponadto dodawał flagi `-march=native`, aby uwzględnić mikroarchitekturę procesora oraz `-mfma`.

1.4 Weryfikacja poprawności rozwiązania

Aby rozwiązania były porównywalne, w każdej wersji programu przed wykonaniem algorytmu macierze będą inicjalizowane losowymi wartościami, ale z ustalonym ziarnem losowania (*seed*). Przyjmuję, że pierwsza wersja algorytmu (bazowa, bez optymalizacji) będzie stanowić wersję referencyjną. Dla każdego badanego rozmiaru macierzy zapisuję sumy wszystkich wartości w macierzy po wykonaniu algorytmu. Sumowanie odbywa się za pomocą prostej funkcji:

```
// used to check algorithm correctness
double calculate_matrix_sum(const double ** A, int SIZE) {
  int i, j;
  double check = 0.0;

  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      check += A[i][j];
    }
  }

  return check;
}
```

Listing 1: Sumowanie wszystkich wartości macierzy

To właśnie dla tej bazowej wersji zapiszę kolejne uzyskane w taki sposób wartości w pliku `ref.txt` (w kolejnych wierszach dla kolejnych rozmiarów macierzy). Z kolei dla następnych wersji, z optymalizacjami, będę porównywał wartości uzyskanych sum z referencyjnymi za pomocą prostego algorytmu:

```
// check algorithm correctness
int check_correctness(const double ** A, int SIZE) {
  FILE* checksums = fopen("ref.txt", "r");
  double matrix_sum = calculate_matrix_sum(A, SIZE);
  double check;
  int retval, curr_size = 10;

  while (curr_size <= SIZE) {
    retval = fscanf(checksums, "%f\n", &check);
    curr_size += 10;
  }

  return check != matrix_sum ? -1 : 0;
}
```

Listing 2: Weryfikacja poprawności algorytmu eliminacji Gaussa

W przypadku zwróconej wartości `-1` (co oznacza nieprawidłowy wynik) program kończy działanie. Pełne wykonanie programu oznacza zatem prawidłowe wyniki.

2 Implementacja i analiza kolejnych wersji rozwiązania

2.1 Podstawowa wersja algorytmu

Poniżej zamieszczam pełen kod funkcji realizującej algorytm, natomiast w kolejnych wersjach skupię się na poczynionych zmianach.

```
// Algorithm to be optimized
void ge(double ** A, int SIZE) {
  int i, j, k;

  for (k = 0; k < SIZE; k++) {
    for (i = k + 1; i < SIZE; i++) {
      for (j = k + 1; j < SIZE; j++) {
        A[i][j] -= A[k][j] * (A[i][k] / A[k][k]);
      }
    }
  }
}
```

Listing 3: Podstawowa wersja algorytmu - *ge1.c*

Jest to funkcja bardzo podobna do tej z laboratorium (ćwiczenie 3) dotyczącego algorytmu eliminacji Gaussa *bez pivotingu*.

2.1.1 Uruchomienie programu

Kompilacja i przykładowe wykonanie programu:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge1.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 5.711958
Checking correctness: -4.605781e+16
```

Flagi kompilacji są potrzebne ze względu na sposób instalacji i użycie przeze mnie biblioteki **PAPI** do pomiarów liczników oraz dostosowanie do architektury procesora. W przypadku takiego uruchomienia algorytm domyślnie testowany jest na macierzy o rozmiarze `1500x1500` oraz **bez** optymalizacji `-O2`. Dla analogicznego skompilowania z tą flagą i uruchomienia uzyskuję czas 1.353572 sekundy.

2.1.2 Oszacowanie złożoności obliczeniowej i FLOPS-ów

Patrząc na kod funkcji, obliczam wyrażenie oznaczające liczbę przebiegów wewnętrznej pętli:

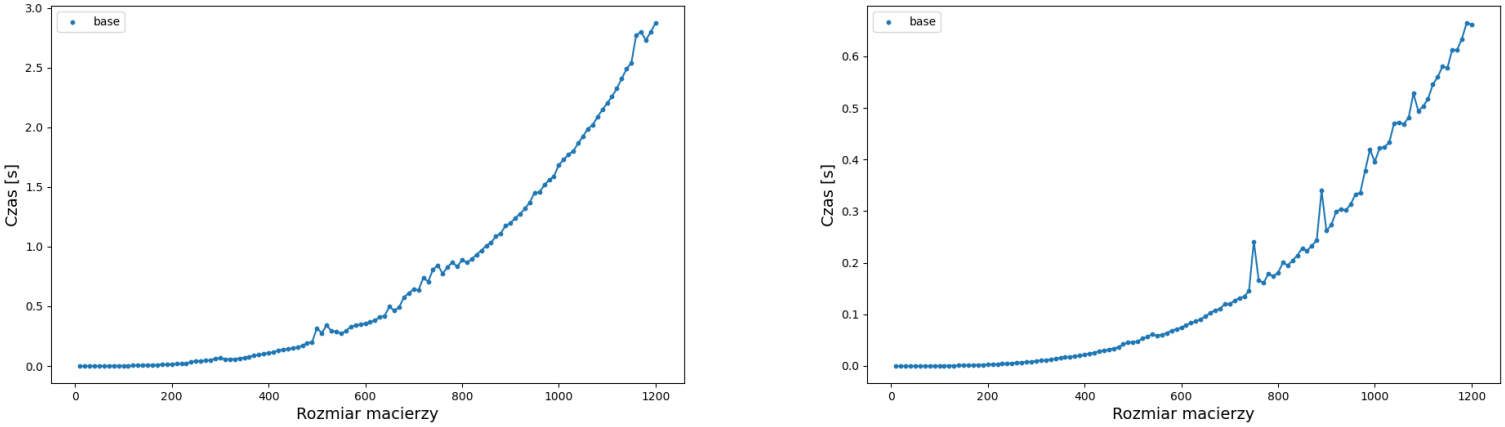
$$\sum_{k=1}^{n}(k-1)^2 = \frac{1}{6}n(2n^2-3n+1).$$

Jako że w każdym przebiegu wykonywane są 3 operacje zmiennoprzecinkowe (1x dzielenie, 1x mnożenie oraz 1x odejmowanie), podstawiając $n = \text{SIZE} = 1500$, wychodzi $3 \cdot \frac{1}{6} \cdot 1500 \cdot (2 \cdot 1500^2 - 3 \cdot 1500 + 1) = 3371625750$ operacji. Otrzymuję kolejno:

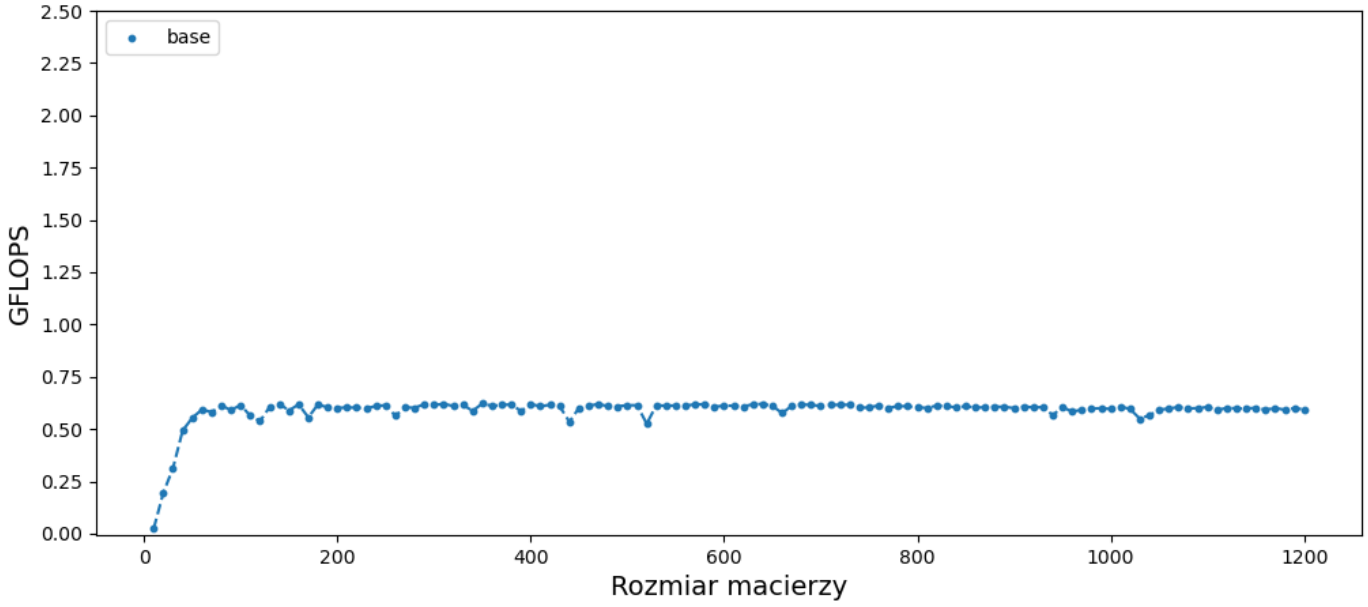
- bez optymalizacji `-O2`: FLOPS $\approx 3371625750/5.711958 \approx 590274955$, czyli około 0.59 GFLOPS-ów,
- z optymalizacją `-O2`: FLOPS $\approx 3371625750/1.353572 \approx 2490909793$, czyli około 2.49 GFLOPS-ów.

2.1.3 Pomiary czasowe i rzeczywista złożoność obliczeniowa - wykorzystanie PAPI

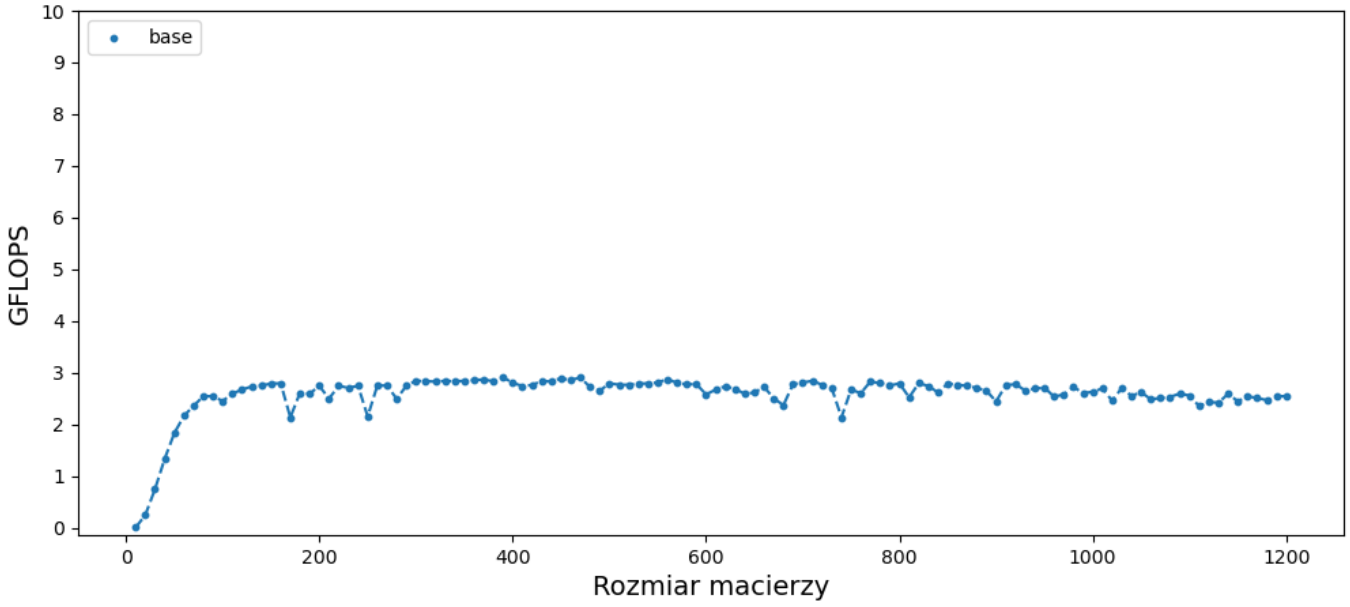
Z powodu ograniczenia dostępnych liczników na moim komputerze (`PAPI_FP_INS` nie jest dostępny), wykorzystuję licznik `PAPI_DP_OPS`, co powinno wystarczyć, jako że operuję na typie `double`. Wyniki pomiarów czasowych oraz liczników przedstawiam na wykresach (w wariancie bez optymalizacji `-O2` oraz z nią).



Wykres 1: Czasy wykonania dla wersji bazowej: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)



Wykres 2: FLOPS-y dla wersji bazowej, bez optymalizacji `-O2`



Wykres 3: FLOPS-y dla wersji bazowej, z optymalizacją `-O2`

2.2 Optymalizacja nr 1 - umieszczenie liczników pętli w rejestrach

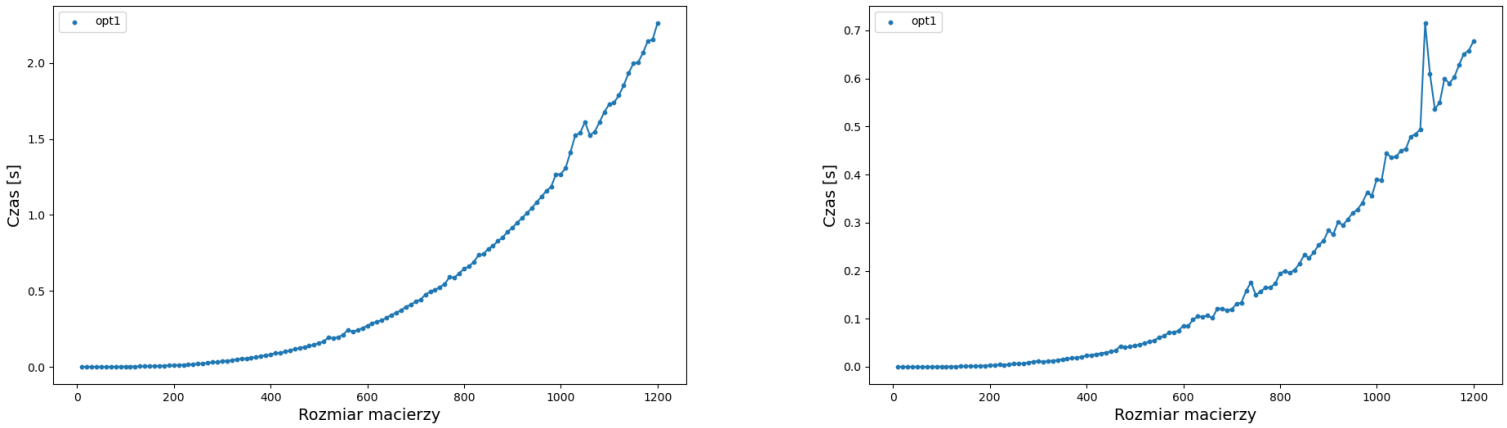
Prosta optymalizacja, polegająca na zmianie deklaracji zmiennych `int i, j, k` na `register int i, j, k`.

Pomiary czasowe i złożoność obliczeniowa

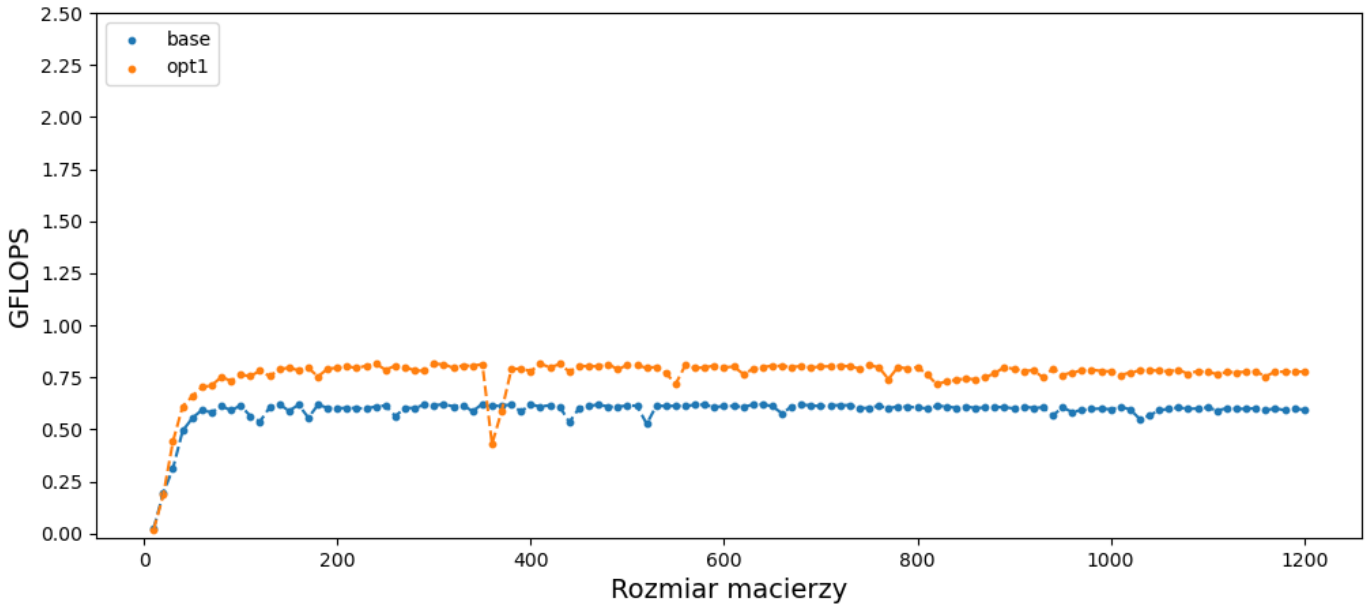
Kompilacja i przykładowe wykonanie programu:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge2.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 4.359846
Checking correctness: -4.605781e+16
```

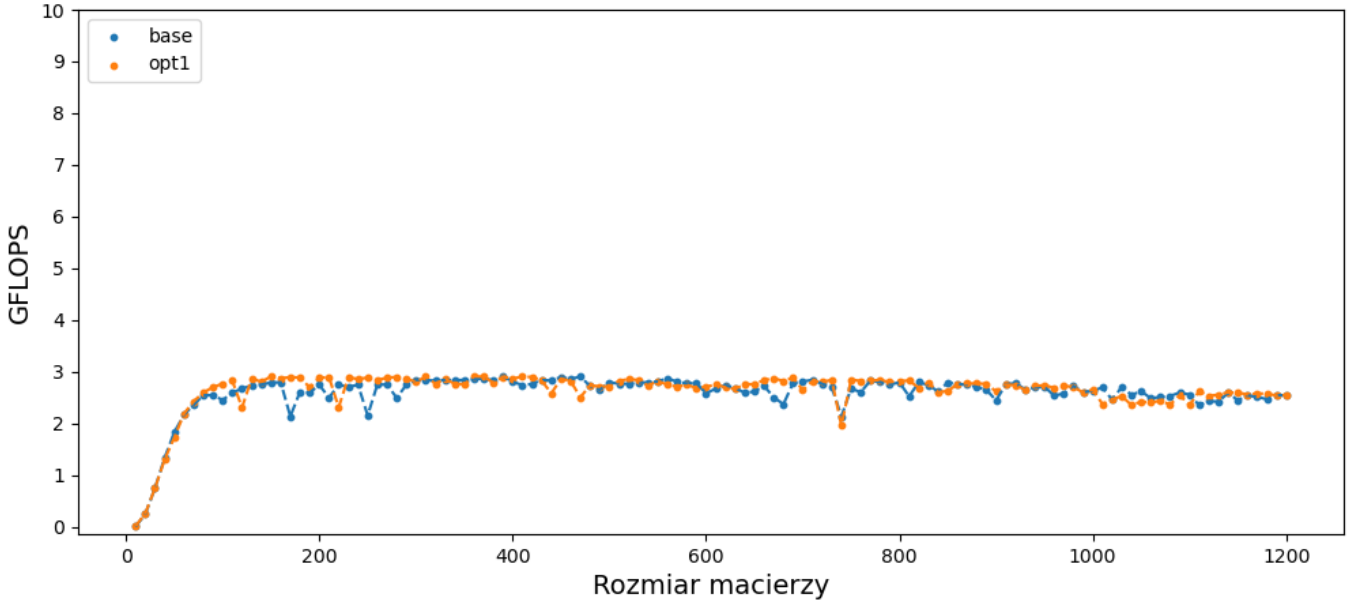
Czas wykonania jest zauważalnie krótszy niż dla wersji podstawowej (a wartość sumy jest taka sama). Przy kompilacji z flagą `-O2` różnica jest nieznaczna: uzyskuję czas 1.369004 sekundy. Szacując analogicznie FLOPS-y, otrzymuję około 0.77 GFLOPS-ów bez optymalizacji `-O2` oraz 2.46 GFLOPS-ów z tą optymalizacją.



Wykres 4: Czasy wykonania po 1. zmianie: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)



Wykres 5: Porównanie FLOPS-ów po 1. zmianie oraz dla wersji bazowej, bez optymalizacji `-O2`



Wykres 6: Porównanie FLOPS-ów po 1. zmianie oraz dla wersji bazowej, z optymalizacją `-O2`

2.3 Optymalizacja nr 2 - umieszczenie wartości mnożnika w rejestrze

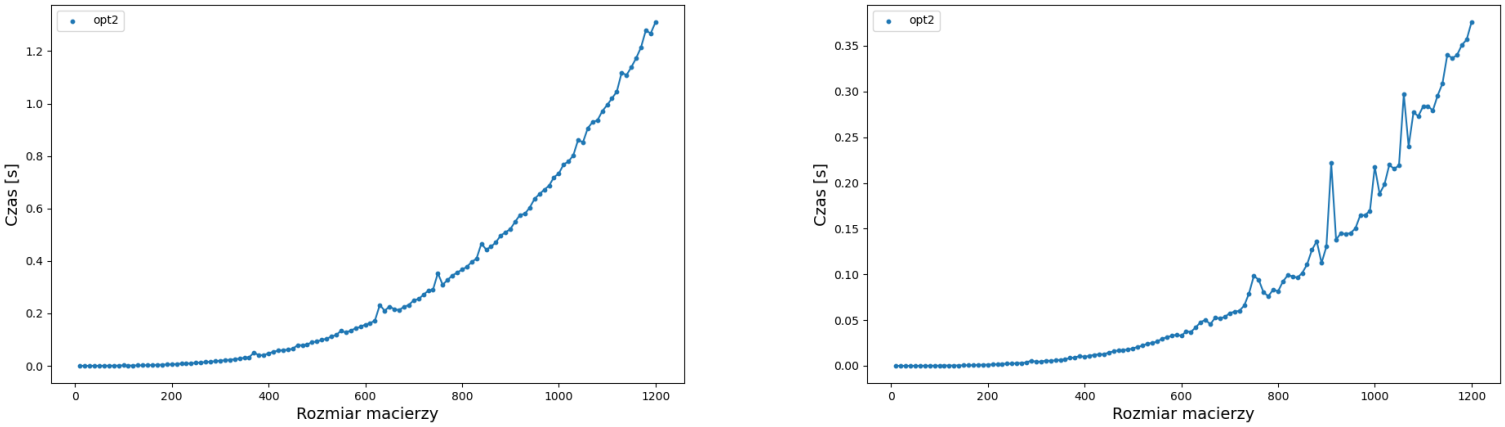
W najbardziej zagnieżdżonej pętli pojawia się wartość - niezależna od jej licznika - `(A[i][k] / A[k][k])`. Umieszczam ją zatem w rejestrze: `register double multiplier;` oraz przypisuję jej wartość **przed** wewnętrzną pętlą z licznikiem `j`: `multiplier = (A[i][k] / A[k][k]);`
Teraz wyrażenie w środku pętli wygląda następująco: `A[i][j] -= A[k][j] * multiplier;`

Pomiary czasowe i złożoność obliczeniowa

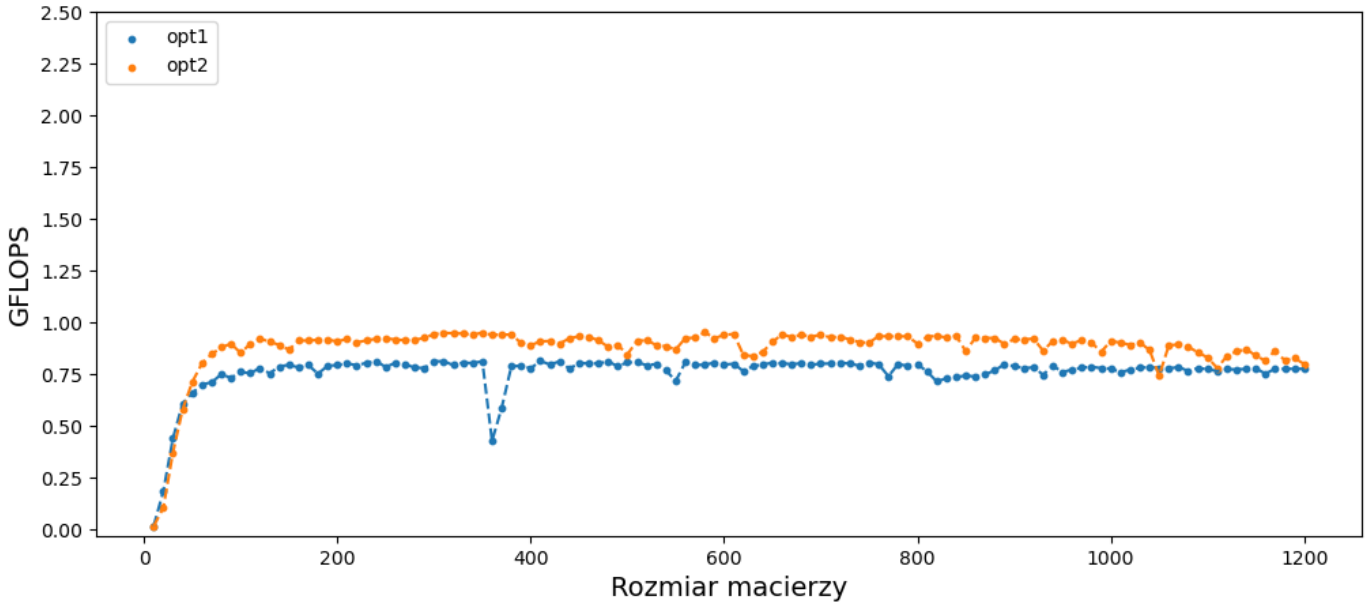
Kompilacja i przykładowe wykonanie programu:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge3.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 2.723208
Checking correctness: -4.605781e+16
```

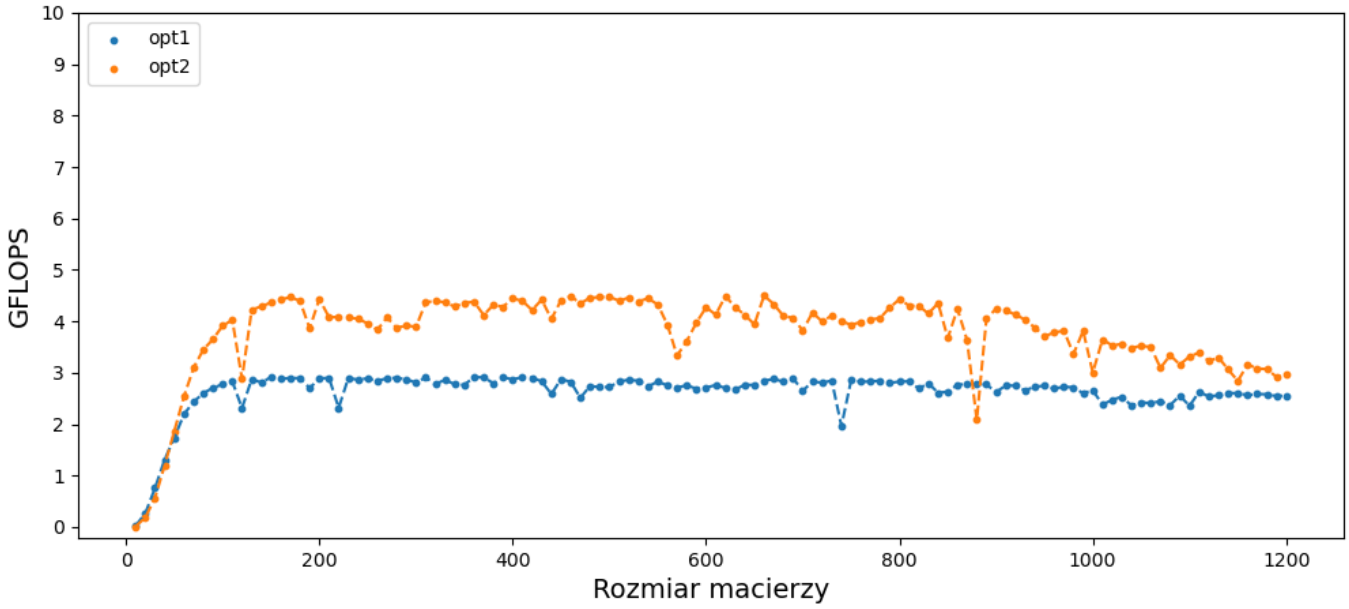
Kolejny raz udało się poprzez bardzo prostą optymalizację (niewielką zmianę) istotnie przyspieszyć działanie programu, zachowując równoważność. Przy optymalizacji `-O2` uzyskany czas to 1.264279 sekundy. Daje to około 1.24 GFLOPS-ów bez optymalizacji `-O2` oraz 2.67 GFLOPS-ów z tą optymalizacją.



Wykres 7: Czasy wykonania po 2. zmianie: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)



Wykres 8: Porównanie FLOPS-ów po 2. zmianie i po 1. zmianie, bez optymalizacji `-O2`



Wykres 9: Porównanie FLOPS-ów po 2. zmianie i po 1. zmianie, z optymalizacją `-O2`

2.4 Optymalizacja nr 3 - rozwinięcie pętli

Przetestuję rozwinięcie najbardziej zagnieżdżonej pętli (z licznikiem `j`) do najpierw 8 iteracji, następnie 16 iteracji (poniekąd w tym punkcie i w następnych będę je testował "równolegle"). Przypadek dla 8 iteracji wygląda następująco:

```
for (j = k + 1; j < SIZE; ) {
  if (j < (MAX(SIZE - BLKSIZE, 0))) {
    A[i][j] -= A[k][j] * multiplier;
    A[i][j+1] -= A[k][j+1] * multiplier;
    A[i][j+2] -= A[k][j+2] * multiplier;
    A[i][j+3] -= A[k][j+3] * multiplier;
    A[i][j+4] -= A[k][j+4] * multiplier;
    A[i][j+5] -= A[k][j+5] * multiplier;
    A[i][j+6] -= A[k][j+6] * multiplier;
    A[i][j+7] -= A[k][j+7] * multiplier;
    j += BLKSIZE;
  } else {
    A[i][j] -= A[k][j] * multiplier;
    j++;
  }
}
```

Listing 4: Ręczne rozwinięcie pętli do 8 iteracji - *ge4.c*

Pojawiające się tutaj `MAX` to pomocnicze makro, które - jak nazwa sugeruje - zwraca większą z liczb przyjmowanych jako argumenty: `#define MAX(a, b) (((a) > (b)) ? (a) : (b))`
Z kolei `BLKSIZE` to rozmiar rozwijanego bloku, który w tym przypadku wynosi 8: `#define BLKSIZE 8`
Oczywiście analogiczny jest przypadek dla `BLKSIZE` równego 16.

Pomiary czasowe i złożoność obliczeniowa

Kompilacja i przykładowe wykonanie programu - rozwinięcie do 8 iteracji:

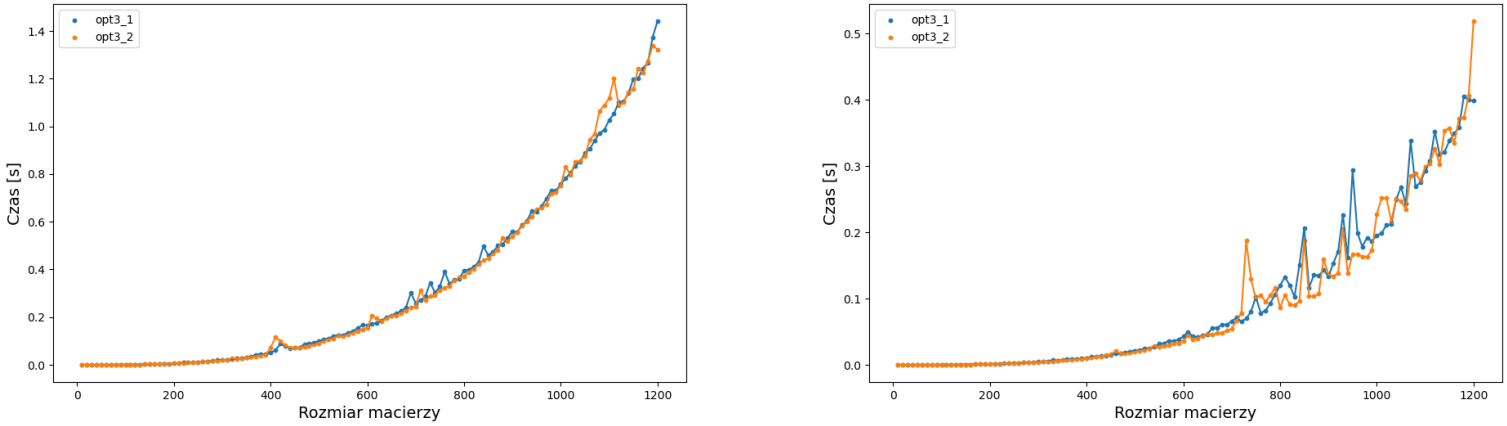
```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge4.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 3.241755
Checking correctness: -4.605781e+16
```

Jak widać, po tej zmianie czas wykonania się nieco pogorszył (z optymalizacją `-O2` wynosi 1.267277 sekundy, więc praktycznie identyczny). Zapewne wynika to z tego, że rozwinięcie pętli samo w sobie może nie dać poprawy, ale stanowi przygotowanie do dalszych optymalizacji. Bez optymalizacji `-O2` dostaję około 1.04 GFLOPS-ów, a z tą optymalizacją 2.66 GFLOPS-ów.

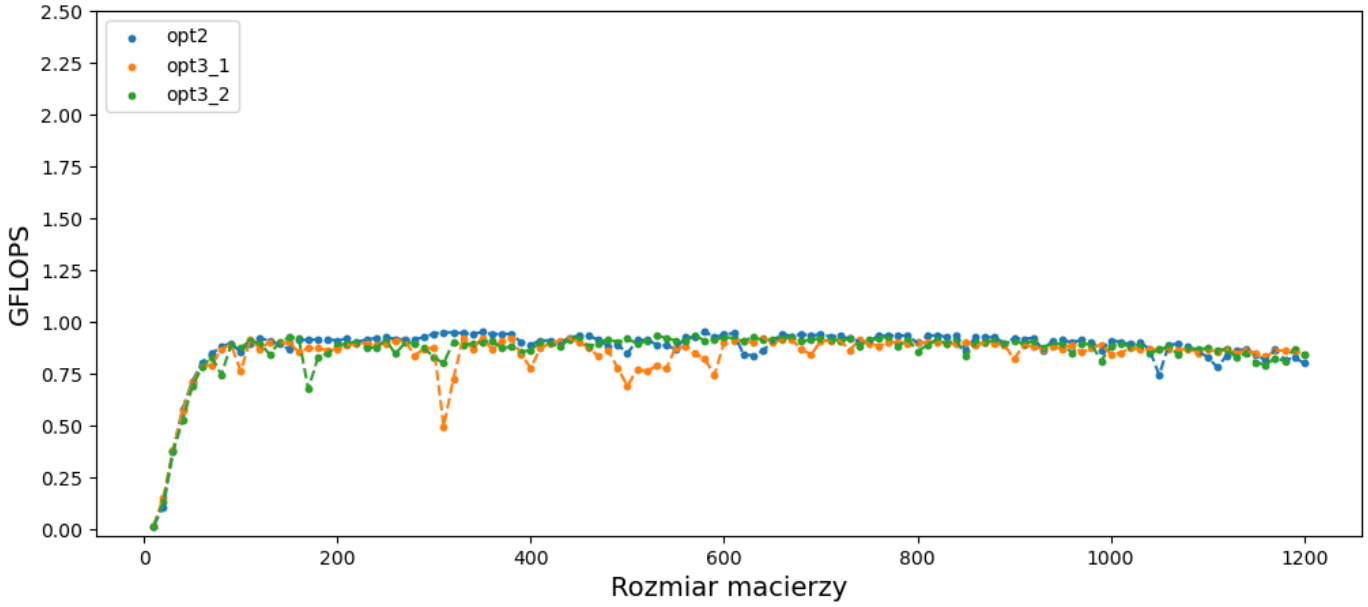
Kompilacja i przykładowe wykonanie programu - rozwinięcie do 16 iteracji:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge5.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 2.772802
Checking correctness: -4.605781e+16
```

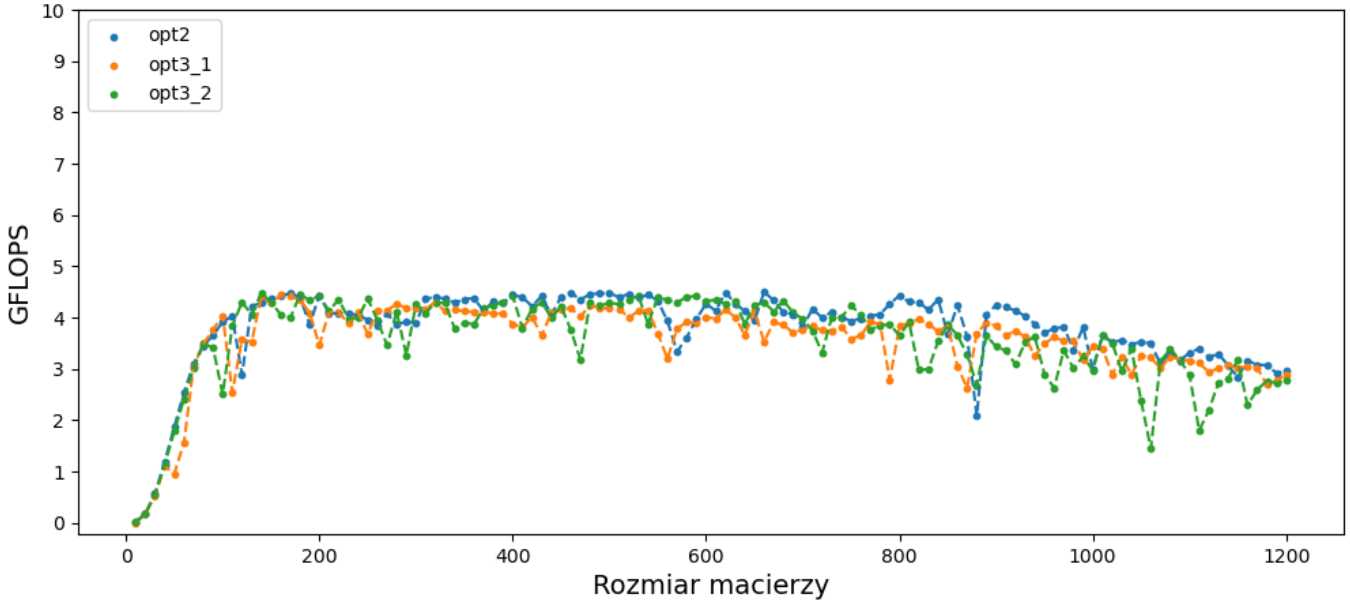
Dla tego wywołania czas jest praktycznie identyczny co dla poprzedniej optymalizacji, a z optymalizacją `-O2` wynosi 1.198155 sekundy, więc jest nawet nieco krótszy. Może to świadczyć o losowych, drobnych wahaniach (np. mimo wyłączenia innych programów, w tle coś akurat trochę bardziej obciążało procesor). Szacując FLOPS-y, bez optymalizacji `-O2` dostaję około 1.22 GFLOPS-ów, a z tą optymalizacją 2.81 GFLOPS-ów. Przedstawię wspólnie oba warianty tej wersji na wykresach, zarówno dla czasów, jak i porównania FLOPS-ów. Przebiegi `_1` dotyczą rozmiaru bloku 8, a `_2` rozmiaru bloku 16.



Wykres 10: Czasy wykonania po 3. zmianie: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)



Wykres 11: Porównanie FLOPS-ów po 3. zmianie i po 2. zmianie, bez optymalizacji `-O2`



Wykres 12: Porównanie FLOPS-ów po 3. zmianie i po 2. zmianie, z optymalizacją `-O2`

2.5 Optymalizacja nr 4 - macierz jednowymiarowa indeksowana poprzez makro

W związku z tą zmianą, we **wszystkich** miejscach zmieniam wystąpienia `double**` na `double*` (czyli też w funkcjach używanych do weryfikacji poprawności algorytmu w sekcji na początku dokumentu). Makro do indeksowania: `#define IDX(i, j, n) (((j) + (i) * (n)))`
Odniesienia do pól macierzy zmieniają się na zasadzie: zamiast `A[i][j]` teraz jest `A[IDX(i, j, SIZE)]`.

Oczywiście dla innych indeksów zamiana wygląda analogicznie. Testuję tę zmianę dla rozwinięcia do 8 iteracji oraz do 16 iteracji.

Pomiary czasowe i złożoność obliczeniowa

Kompilacja i przykładowe wykonanie programu - rozmiar bloku 8:

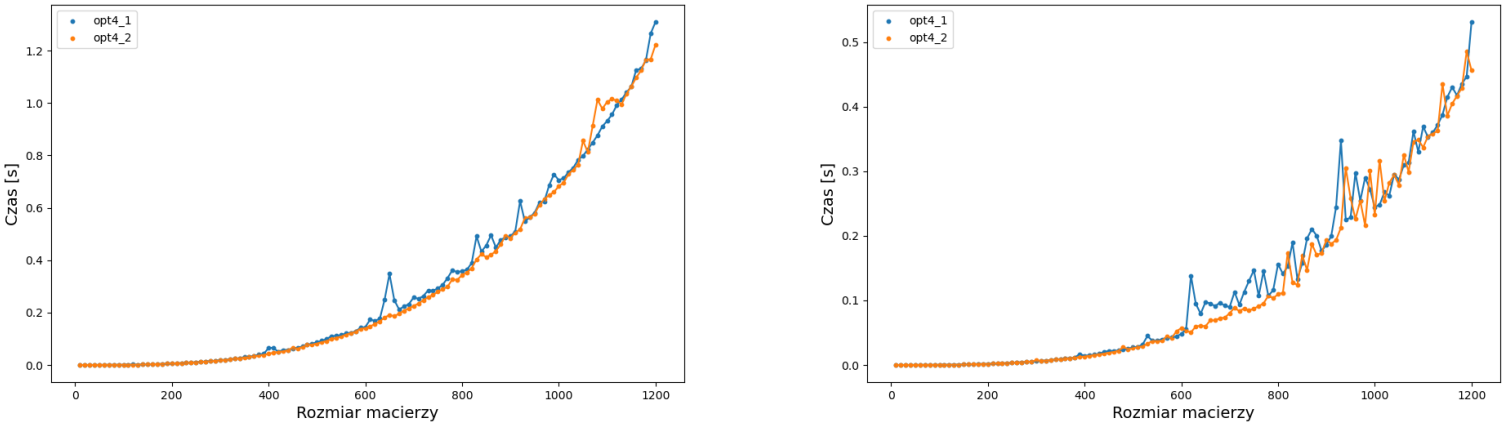
```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge6.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 2.624456
Checking correctness: -4.605781e+16
```

Udało się nieco obniżyć czas wykonania (także z optymalizacją `-O2`: 1.015555 sekundy). Tak naprawdę nadal jest to optymalizacja przygotowująca do czegoś większego: **jednostek wektorowych**. Szacując FLOPS-y, dostaję odpowiednio około 1.28 GFLOPS-ów bez optymalizacji `-O2` oraz 3.32 GFLOPS-ów z tą optymalizacją.

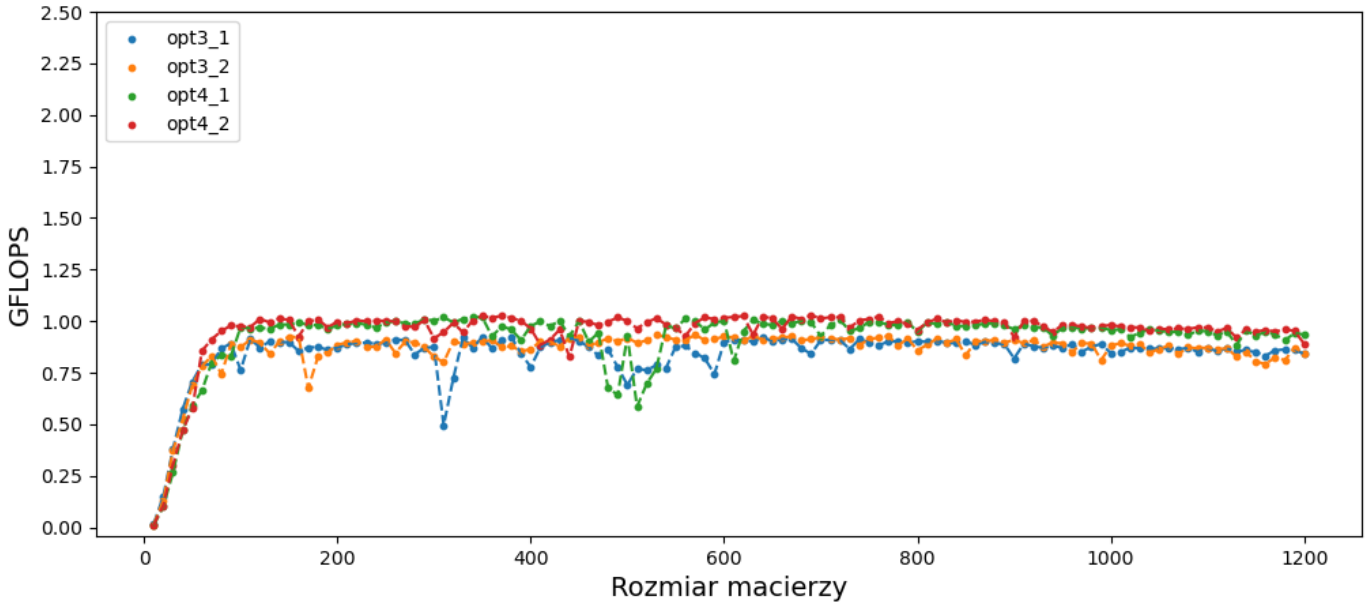
Kompilacja i przykładowe wykonanie programu - rozmiar bloku 16:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge7.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 2.482357
Checking correctness: -4.605781e+16
```

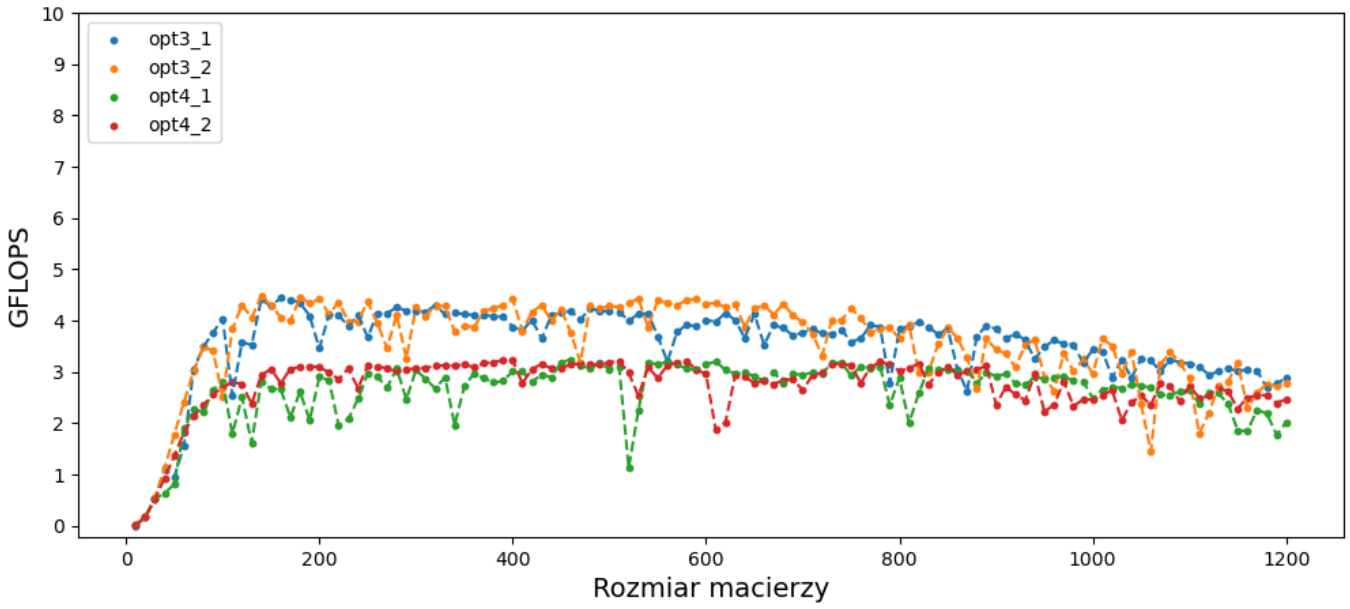
Po raz kolejny czas bez optymalizacji `-O2` jest nieco niższy, z optymalizacją `-O2` podobny - 1.049540 sekundy. Dostaję odpowiednio około 1.36 GFLOPS-ów bez optymalizacji `-O2` oraz 3.21 GFLOPS-ów z tą optymalizacją.



Wykres 13: Czasy wykonania po 4. zmianie: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)



Wykres 14: Porównanie FLOPS-ów po 4. zmianie i po 3. zmianie, bez optymalizacji `-O2`



Wykres 15: Porównanie FLOPS-ów po 4. zmianie i po 3. zmianie, z optymalizacją `-O2`

2.6 Optymalizacja nr 5 - operacje wektorowe SSE3

Aby wykorzystać jednostki wektorowe, wprowadzam następujące zmiany (dla rozwinięcia do 8 iteracji):

1. Dodaję deklaracje:

```
double multiplier[2];
register __m128d mm_multiplier;
register __m128d tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7;
```

2. Zmiana w pętli:

```
for (i = k + 1; i < SIZE; i++) {
  multiplier[0] = (A[IDX(i, k, SIZE)] / A[IDX(k, k, SIZE)]);
  multiplier[1] = multiplier[0];
  mm_multiplier = _mm_loadu_pd(multiplier);

  for (j = k + 1; j < SIZE; ) {
    if (j < (MAX(SIZE - BLKSIZE, 0))) {
      // load
      tmp0 = _mm_loadu_pd(A + IDX(i, j, SIZE));
      tmp1 = _mm_loadu_pd(A + IDX(k, j, SIZE));
      tmp2 = _mm_loadu_pd(A + IDX(i, j+2, SIZE));
      ...

      // multiply
      tmp1 = _mm_mul_pd(tmp1, mm_multiplier);
      tmp3 = _mm_mul_pd(tmp3, mm_multiplier);
      ...

      // subtract
      tmp0 = _mm_sub_pd(tmp0, tmp1);
      tmp2 = _mm_sub_pd(tmp2, tmp3);
      ...

      // store
      _mm_storeu_pd(A + IDX(i, j, SIZE), tmp0);
      _mm_storeu_pd(A + IDX(i, j+2, SIZE), tmp2);
      ...

      j += BLKSIZE;
    } else {
      A[IDX(i, j, SIZE)] -= A[IDX(k, j, SIZE)] * multiplier[0];
      j++;
    }
  }
}
```

Dla rozwinięcia do 16 iteracji zmiennych (i odpowiednich operacji) jest `2x` więcej, ale wyglądają one tak samo.

Pomiary czasowe i złożoność obliczeniowa

Kompilacja i przykładowe wykonanie programu dla rozmiaru bloku równego 8:

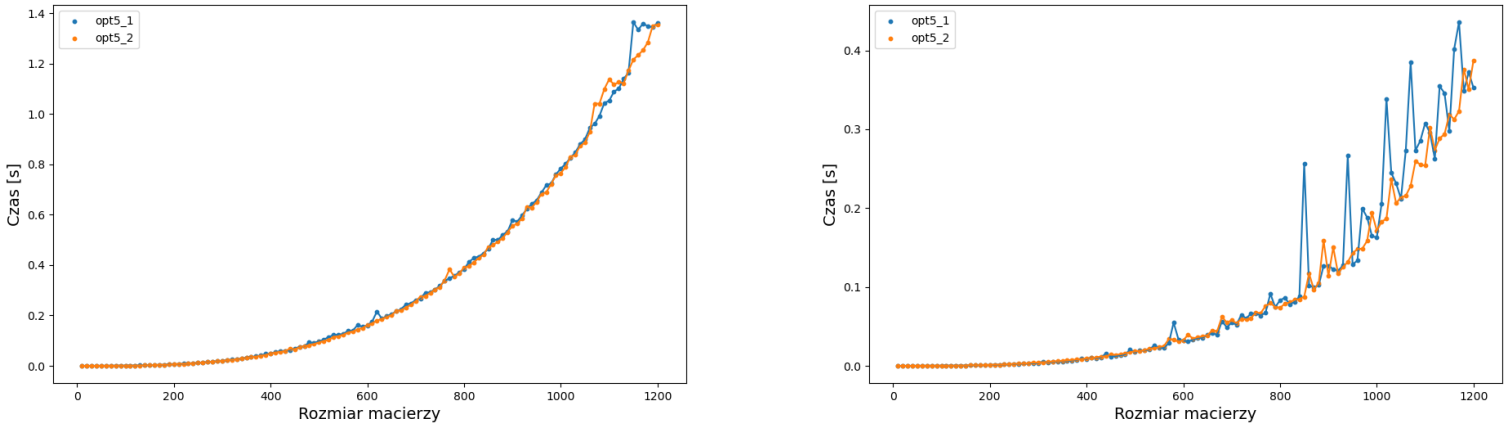
```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge8.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 2.726138
Checking correctness: -4.605781e+16
```

Dla przykładowego wywołania (bez optymalizacji `-O2`) nie widać poprawy, nawet jest lekkie pogorszenie! Wynika to zapewne z rozmiaru macierzy (1500), nie mieści się ona bowiem w całości w pamięci cache. Lepiej jest już z optymalizacją `-O2`, wtedy uzyskuję czas 0.908771 sekundy. Daje to odpowiednio około 1.24 GFLOPS-ów bez optymalizacji `-O2` oraz 3.71 GFLOPS-ów z tą optymalizacją.

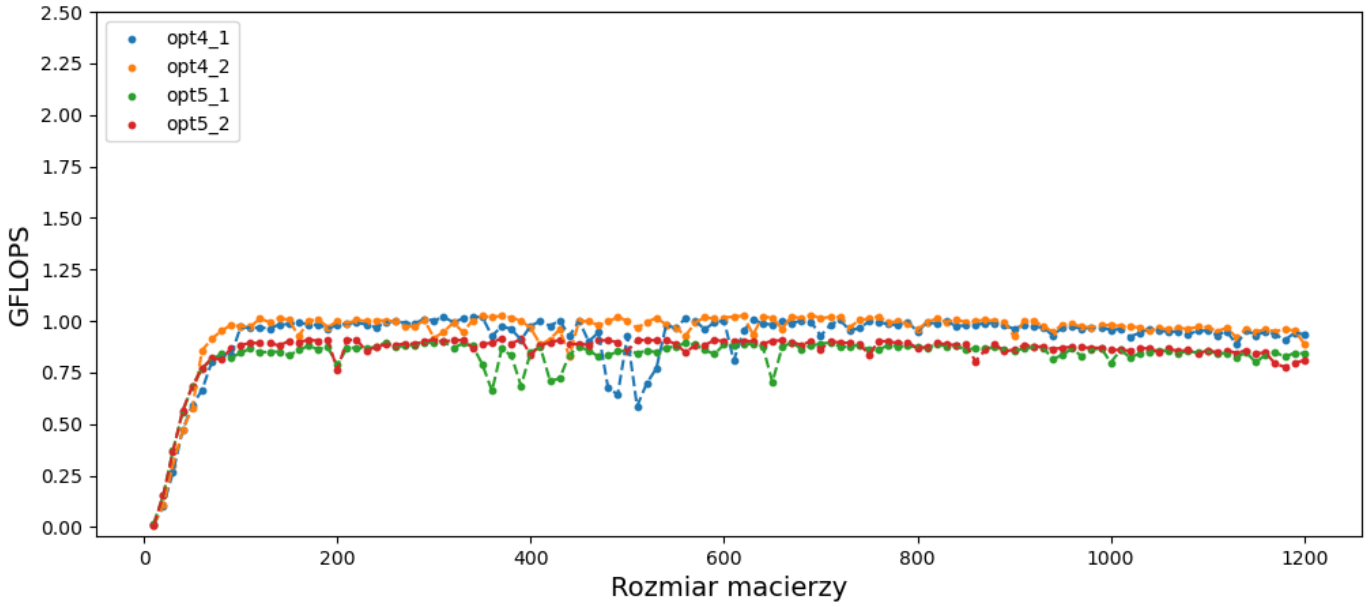
Kompilacja i przykładowe wykonanie programu dla rozmiaru bloku równego 16:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma ge9.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 2.687005
Checking correctness: -4.605781e+16
```

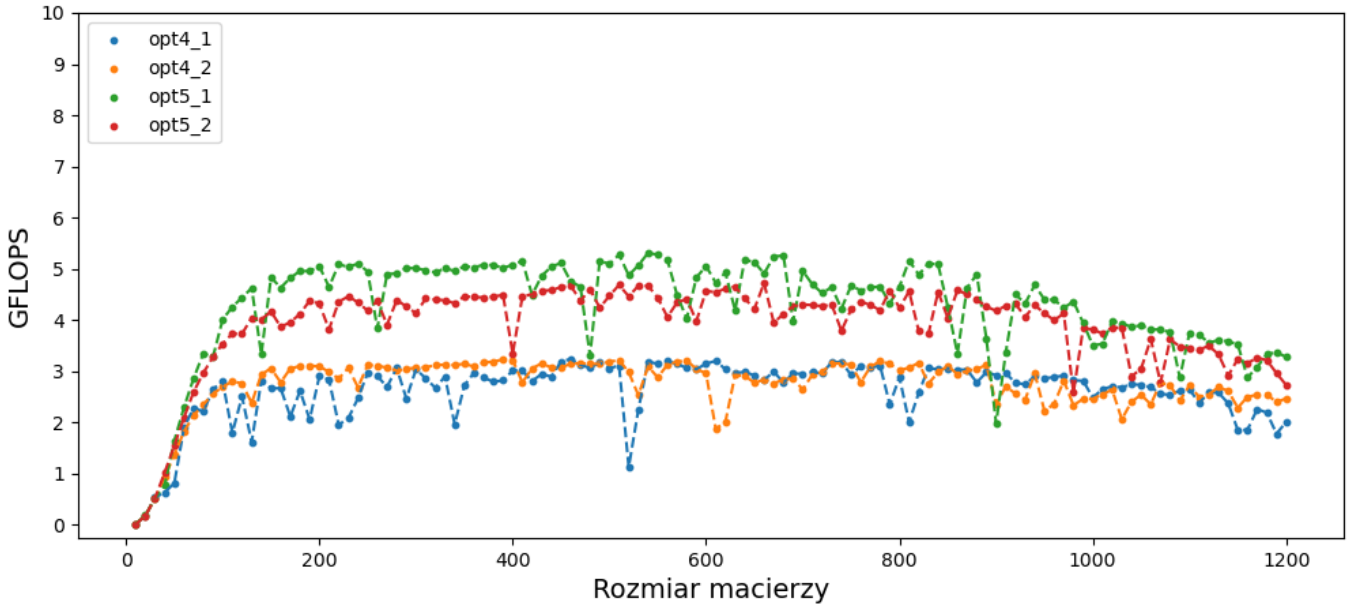
Czas jest podobny jak dla bloku o rozmiarze 8. Z optymalizacją `-O2` uzyskuję czas 1.061949 sekundy, więc gorszy, ale może to być wahnięcie akurat przy uruchamianiu tego programu. Takie rezultaty dają około 1.26 GFLOPS-ów bez optymalizacji `-O2` oraz 3.17 GFLOPS-ów z tą optymalizacją.



Wykres 16: Czasy wykonania po 5. zmianie: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)



Wykres 17: Porównanie FLOPS-ów po 5. zmianie i po 4. zmianie, bez optymalizacji `-O2`



Wykres 18: Porównanie FLOPS-ów po 5. zmianie i po 4. zmianie, z optymalizacją `-O2`

2.7 Optymalizacja nr 6 - 256-bitowe operacje wektorowe AVX

W tym przypadku dokonuję analogicznych zmian co poprzednio, tylko dostosowuję jednostki wektorowe do przechowywania czterech liczb zmiennoprzecinkowych typu `double` zamiast dwóch (dzięki temu zmiennych i wywołań funkcji jest odpowiednio mniej). Wykorzystuję typ `__m256d` oraz odpowiednie funkcje: `_mm256_loadu_pd`, `_mm256_mul_pd` itd.

Pomiary czasowe i złożoność obliczeniowa

Kompilacja i przykładowe wykonanie programu - rozmiar bloku równy 8:

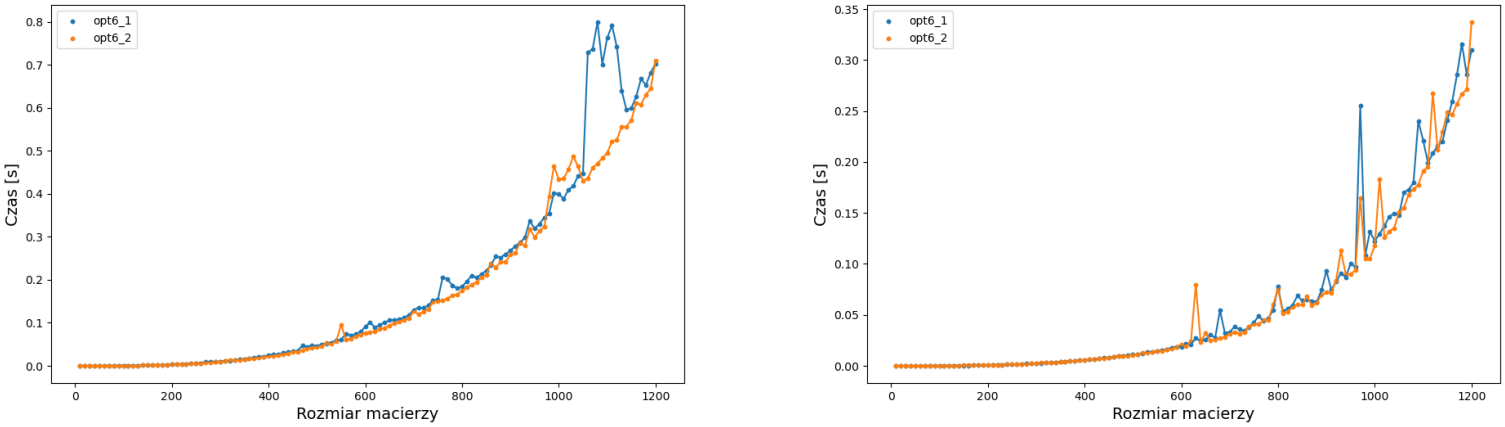
```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma -mavx ge10.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 1.434436
Checking correctness: -4.605781e+16
```

Pojawiła się dodatkowa flaga `-mavx`, związana z użyciem jednostek wektorowych tego typu. Widać znaczącą poprawę (dwukrotną) szybkości działania algorytmu, poprawa jest również widoczna z optymalizacją `-O2` (0.820616 sekundy). Otrzymuję około 2.35 GFLOPS-ów (bez optymalizacji `-O2`!) oraz 4.11 GFLOPS-ów z tą optymalizacją.

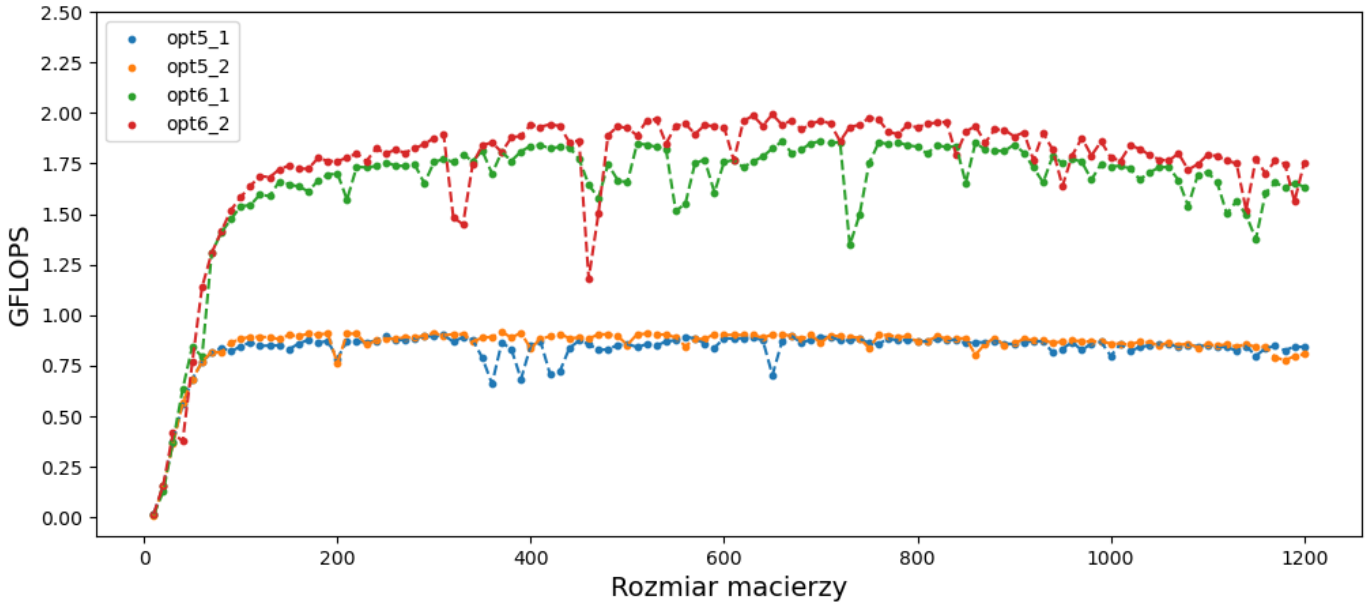
Kompilacja i przykładowe wykonanie programu - rozmiar bloku równy 16:

```
...OKNRA/zadania/zad2$ gcc -I/usr/include -L/usr/lib/x86_64-linux-gnu -march=
native -mfma -mavx ge11.c -lpapi
...OKNRA/zadania/zad2$ ./a.out
Calling Gauss elimination algorithm
Execution time: 1.568826
Checking correctness: -4.605781e+16
```

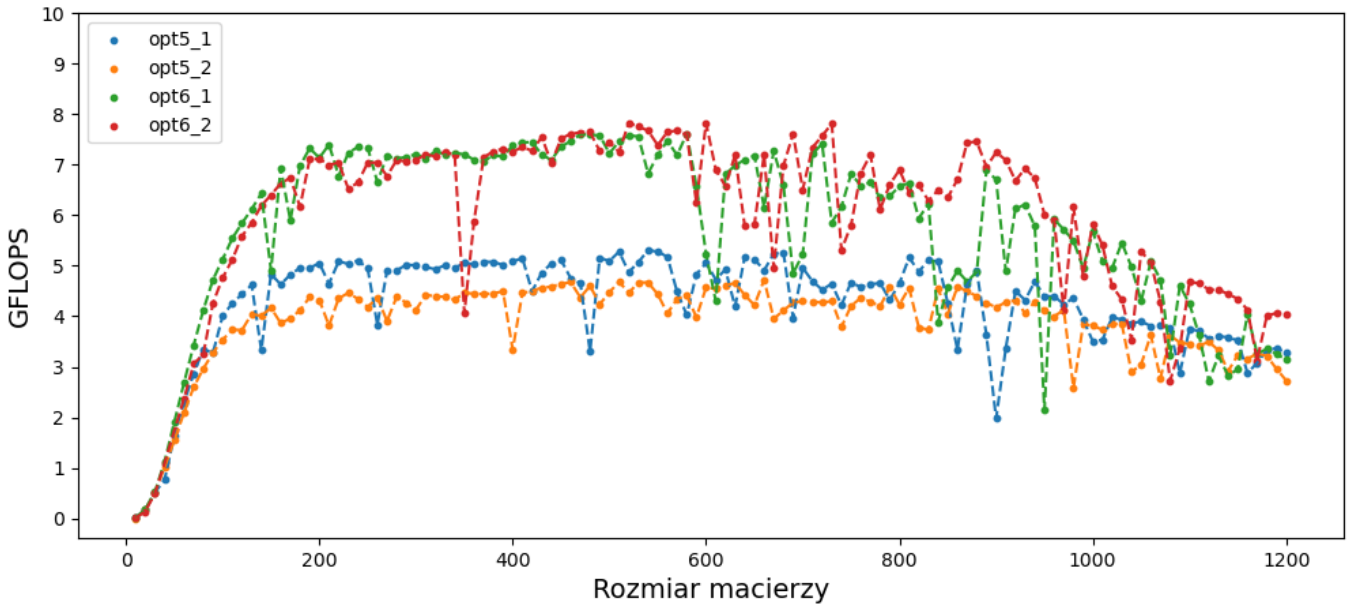
Także w tym przypadku algorytm działa szybko, z optymalizacją `-O2` uzyskuję podobny czas - 0.856615 sekundy. Powyższe czasy oznaczają około 2.15 GFLOPS-ów bez optymalizacji `-O2` oraz 3.94 GFLOPS-ów z tą optymalizacją.



Wykres 19: Czasy wykonania po 6 zmianie: bez optymalizacji `-O2` (po lewej), z tą optymalizacją (po prawej)

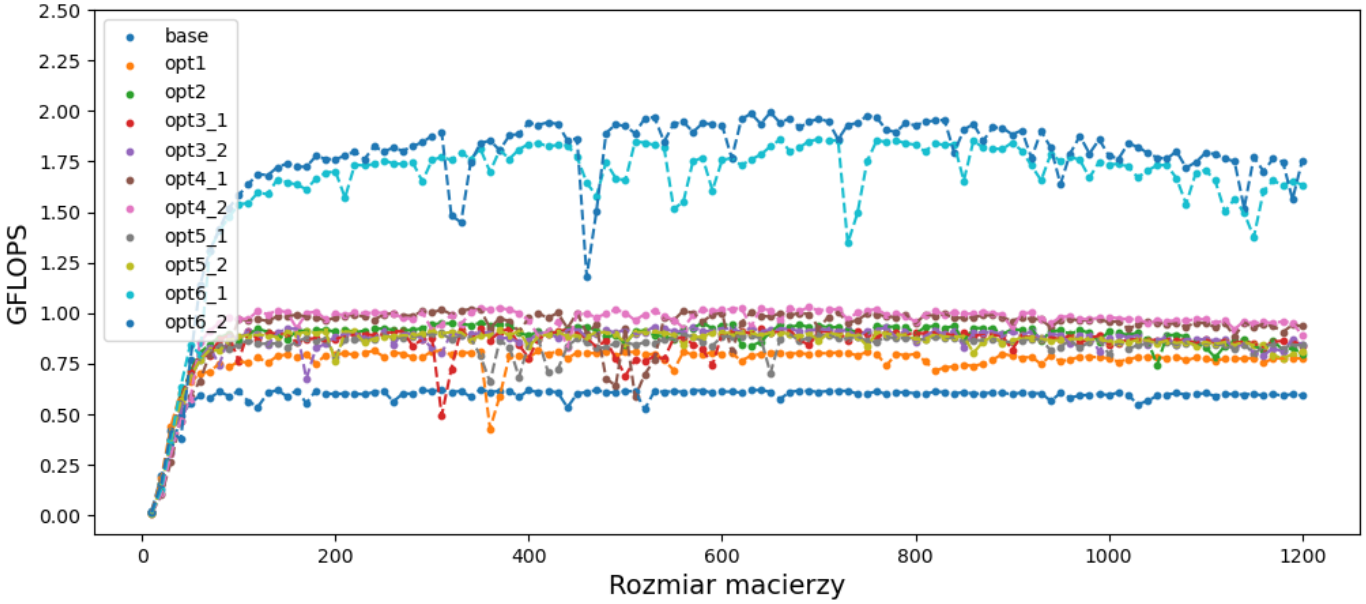


Wykres 20: Porównanie FLOPS-ów po 6. zmianie i po 5. zmianie, bez optymalizacji `-O2`

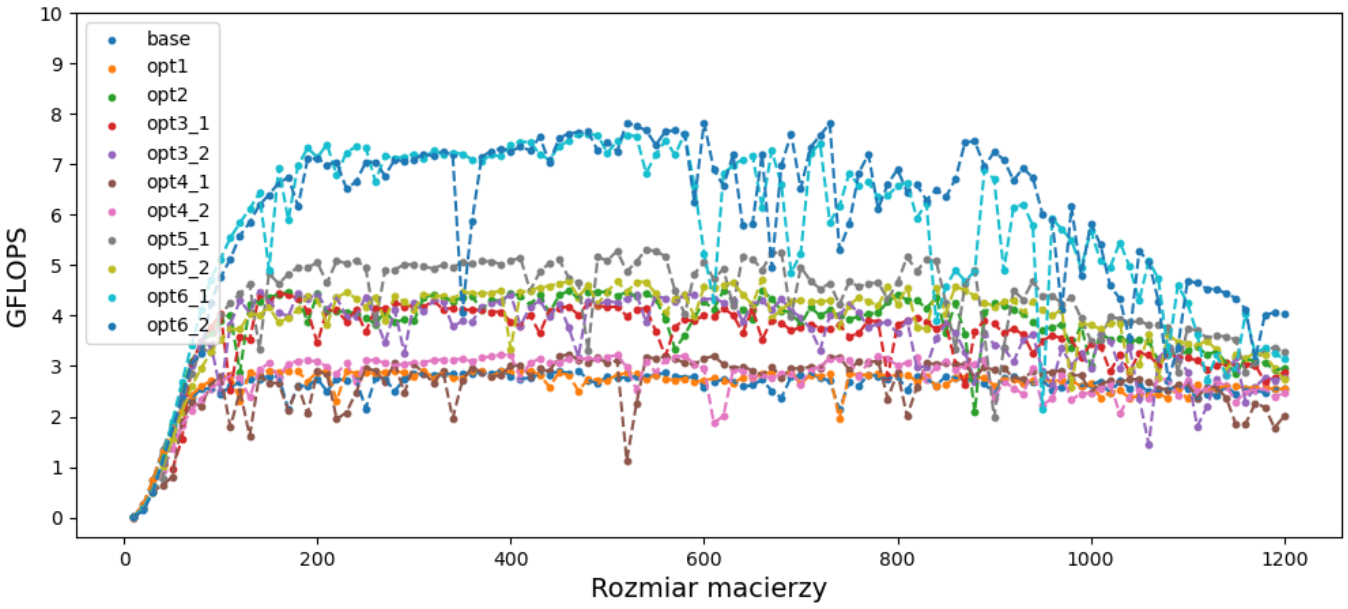


Wykres 21: Porównanie FLOPS-ów po 6. zmianie i po 5. zmianie, z optymalizacją `-O2`

3 Porównanie wszystkich wersji programu - FLOPS-y



Wykres 22: Porównanie FLOPS-ów dla wszystkich wersji, bez optymalizacji `-O2`



Wykres 23: Porównanie FLOPS-ów dla wszystkich wersji, z optymalizacją `-O2`

4 Wnioski

Zanim przejdę do analizy uzyskanych wyników, poruszę kwestię **lokalności danych**. Na podstawie laboratorium (ćwiczenie: *Optymalizacja użycia pamięci cache*) chciałem przeprowadzić analogiczne eksperymenty, przede wszystkim co do *tilingu* oraz kopiowania danych do lokalnych buforów.
W teorii pozwoliłoby to dostosować rozwiązanie do poziomów L1 oraz L2 cache, natomiast w praktyce sam tiling nie dał poprawy rozwiązania, a operowanie na danych w lokalnych buforach okazało się problematyczne i zwracało błędne wyniki. Finalnie nie udało mi się rozwiązać tych kwestii w sensowny sposób, więc porzuciłem tę drogę. Mam nadzieję, że przeprowadzone testy pozwoliły zaobserwować zjawisko lokalności danych chociaż w pewnym stopniu, na poziomie pamięci cache L3.

Po analizie wyników dla wszystkich wersji algorytmu, można zaobserwować następujące zjawiska:

- Stopień poprawy szybkości działania algorytmu zależy od tego, czy przy kompilacji użyłem flagi `-O2` czy nie. Z kolei dla różnych rozmiarów bloków (8 oraz 16) w późniejszych optymalizacjach w obu przypadkach różnica była niewielka, jednak zwykle na korzyść rozmiaru bloku 16.
- W przypadku braku flagi `-O2`:
  1. początkowe 2 optymalizacje (przeniesienie odpowiednich zmiennych do rejestru) przyniosły zauważalną poprawę i zwiększenie FLOPS-ów,
  2. kolejne 2 optymalizacje stanowiły optymalizacje *przejściowe*, wyniki były podobne co dla poprzednich wersji, natomiast umożliwiły one wprowadzenie jednostek wektorowych,
  3. jednostki wektorowe `SSE3` nie spowodowały poprawy wyników, były one wręcz nieco słabsze! Z kolei jednostki wektorowe `AVX` przyniosły blisko dwukrotną poprawę wydajności,
  4. najwięcej udało mi się uzyskać około 2 GFLOPS-y (dla operacji wektorowych `AVX`), co stanowi zaledwie około 5.43% maksymalnej wydajności.
- Przy użyciu flagi `-O2`:
  1. widoczna jest duża poprawa wydajności - wersja bazowa ma więcej FLOPS-ów niż najlepsza optymalizacja bez tej flagi,

2. pierwsza optymalizacja nie powoduje zauważalnej zmiany, natomiast druga już tak. Ponownie trzecia zmiana daje podobne osiągi do drugiej, z kolei optymalizacja nr 4 daje gorsze wyniki!
3. wprowadzenie jednostek wektorowych obu typów daje znaczącą poprawę FLOPS-ów, natomiast od rozmiaru macierzy około 1000 następuje znaczny spadek wydajności, przewaga jednostek wektorowych spada i jest niewielka,
4. najwięcej udało mi się uzyskać około 8 GFLOPS-ów (ponownie dla jednostek wektorowych `AVX`). Stanowi to około 21.74% maksymalnej wydajności - jest to już lepszy rezultat, ale wciąż wydaje się poniżej oczekiwań.

- W przypadku pojedynczych, przykładowych wykonań programu oraz niektórych uruchomień testowych widoczne były zaburzenia (dłuższy czas - "skok" na wykresie czasu oraz analogiczne wychylenie, ale w dół na wykresie FLOPS-ów). Zapewne było to spowodowane tymczasowym zwiększonym obciążeniem procesora, mimo moich prób zapewnienia niezmiennego środowiska w czasie testów.
- Wspomniane spadki FLOPS-ów dla większych rozmiarów macierzy mogą wynikać z faktu, iż macierze te nie mieszczą się w całości w pamięci cache (zgodnie z parametrami mojego procesora). Nie ma wtedy możliwości zapewnienia pełnej lokalności danych, co skutkuje spowolnieniem działania. Jest to widoczne dla wszystkich wersji z flagą `-O2`, w szczególności dla ostatnich optymalizacji - z użyciem jednostek wektorowych.