Optymalizacja kodu na różne architektury
Zadanie domowe 2

Mateusz Kocot

4 grudnia 2021

Spis treści

1 Wprowadzenie

Zadanie wykonałem na komputerze z systemem Windows 10 (64-bit). Skorzystałem z kompilatora GCC 9.3.0 dostępnego w WSL. Procesor w tym komputerze to Intel Core i7-6700K (Skylake) z maksymalną częstotliwością taktowania 4.5 GHz. Posiada on 4 rdzenie oraz wykonuje w sumie 16 operacji zmiennoprzecinkowych na cykl, co daje 4 operacje na rdzeń.

Eksperymenty przeprowadziłem na macierzach o rozmiarze od 200 do 2000, z próbkowaniem co 200. Każdą wersję kodu uruchomiłem po 10 razy dla każdego rozmiaru by uzyskać miarodajne wyniki wraz z odchyleniem standardowym.

Utworzone wykresy przedstawiają liczby operacji zmiennoprzecinkowych (MFlops) w zależności od rozmiaru macierzy. Liczbę operacji zmiennoprzecinkowych obliczałem korzystając z następującego wzoru:

$$\sum_{i=0}^{N-1} (\sum_{j=i+1}^{N-1} (1 + \sum_{k=i+1}^{N-1} 2))$$

Przed uruchomieniem obszernych testów, dla każdej wersji kodu sprawdziłem, czy zwracają one te same wyniki. W tym celu zliczałem sumę wszystkich wartości w macierzy. Ostatecznie, dla danego rozmiaru, każda wersja kodu zwracała tą samą sumę wartości.

2 Bazowy algorytm (*lu1*)

Jako bazowy algorytm wykorzystałem kod dostępny na stronie Wikipedii pozbawiony operacji na macierzy `P` oraz pivotingu:

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

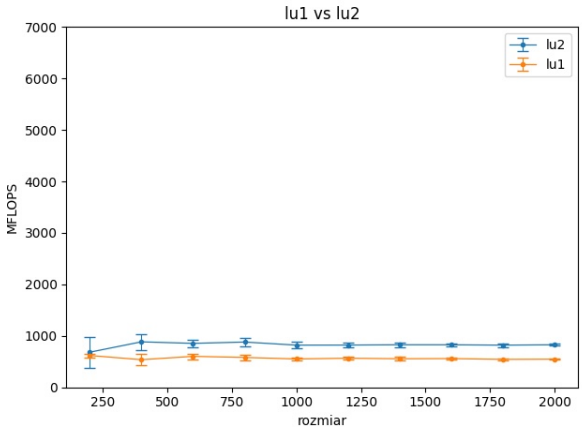
Wydajność bazowej wersji kodu przedstawiłem na rys. 1.

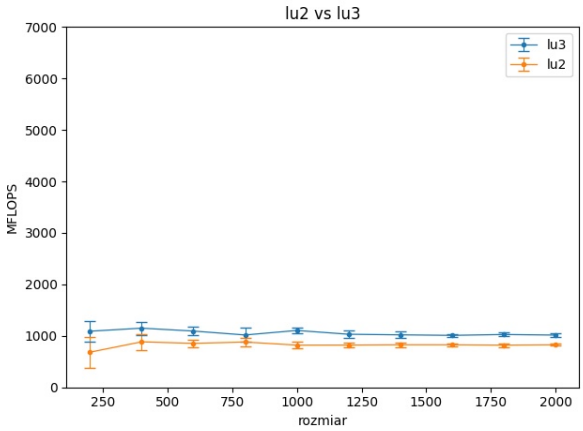
3 Optymalizacja 1 - rejestry (*lu2*)

Na początku procesu optymalizowania, do liczników pętli oraz często używanych wartości (macierz `A` oraz rozmiar `N`) dodałem słowo kluczowe `register`, przez co zachęciłem kompilator do umieszczenia tych zmiennych w rejestrach. Dzięki temu udało się zwiększyć liczbę flopsów niespełna dwukrotnie (rys. 1).

4 Optymalizacja 2 - rejestry 2 (*lu3*)

Tym razem wyciągnąłem często używane wartości `A[i][i]` oraz `A[j][i]` z odpowiednich miejsc w pętlach oraz zachęciłem kompilator do dodania ich do rejestrów. Po raz kolejny udało się zwiększyć wydajność (rys. 1).





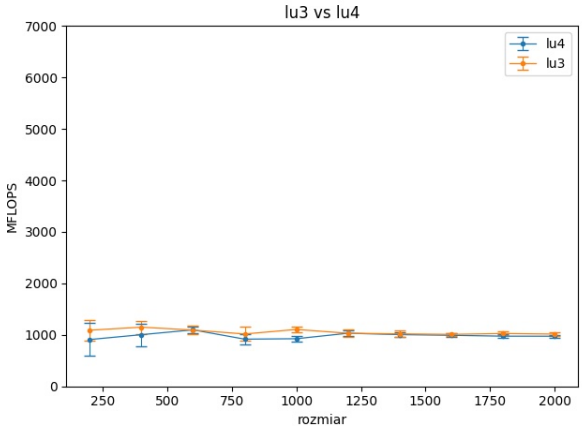
Rys. 1: Porównanie wydajności lu1, lu2 oraz lu3

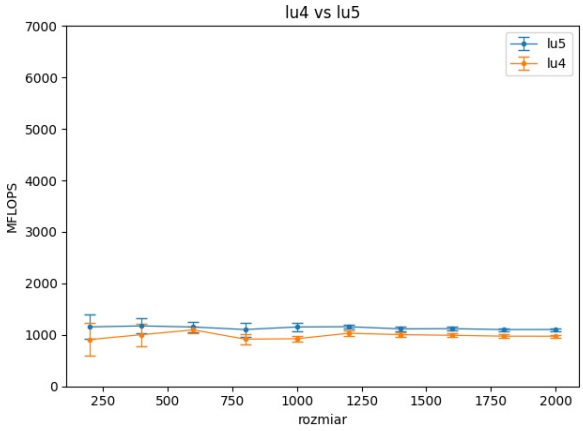
5 Optymalizacja 3 - rozwinięcie pętli (*lu4*)

W tej wersji rozwinąłem najgłębszą pętlę do 8 iteracji. Operacja ta nie przyniosła wzrostu wydajności (rys. 2), lecz dzięki niej kod został przygotowany pod kolejne kroki.

6 Optymalizacja 4 - zamiana 2D na 1D (*lu5*)

Zamieniłem macierz dwuwymiarową na jednowymiarową oraz dodałem odpowiednie makro, dzięki któremu możliwe jest odwoływanie się do konkretnego elementu w macierzy. Operacja ta przyniosła niewielki wzrost w liczbie MFlopsów (rys. 2).





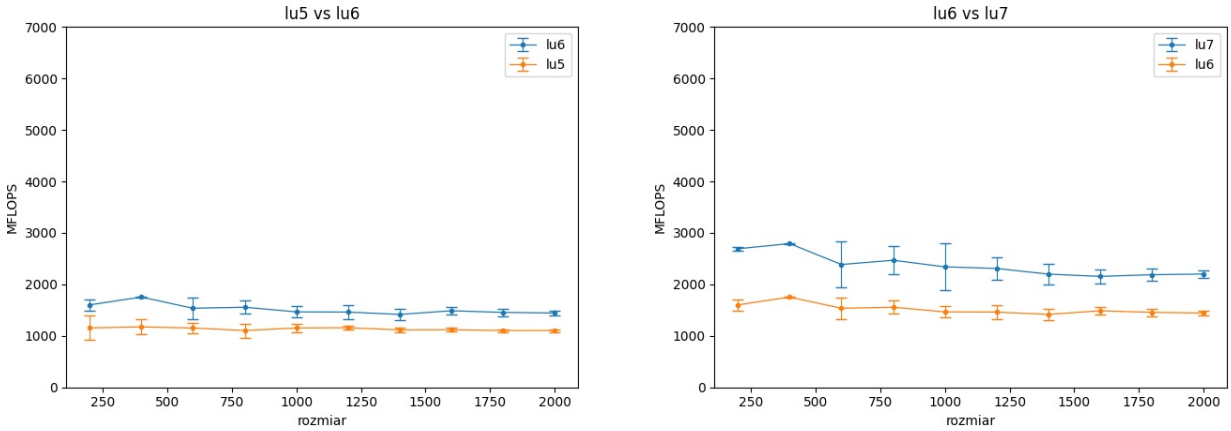
Rys. 2: Porównanie wydajności lu3, lu4 oraz lu5

7 Optymalizacja 5 - SSE3 (*lu6*)

W tym kroku po raz pierwszy użyte zostały operacje wektorowe, na razie tylko SSE3. Pozwoliło to na uzyskanie całkiem sporego wzrostu wydajności (rys. 3).)

8 Optymalizacja 6 - AVX (*lu7*)

Tym razem, operacje SSE3 zostały zmienione na operacje AVX, które są optymalne dla używanego procesora. Przyniosło to olbrzymi wzrost liczby MFlopsów (rys. 3).



Rys. 3: Porównanie wydajności lu5, lu6 oraz lu7

9 Optymalizacja 7 - rozwinięcie pętli 2 (*lu8*)

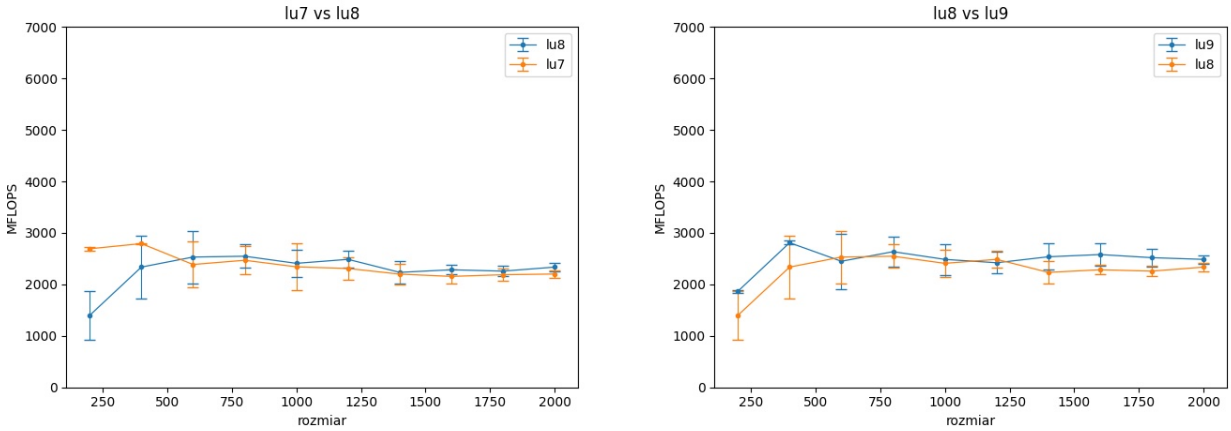
W przedostatnim kroku spróbowałem rozszerzyć liczbę operacji w jednym wykonaniu wewnętrznej pętli do 16. Zwiększyło to lekko wydajność dla dużych rozmiarów problemu, ale, co ciekawe, wydajność dla macierzy o rozmiarach nie większych niż 400 znacząco spadła (rys. 4).

10 Optymalizacja 8 - zblokowanie macierzy (*lu9*)

Na wykresach z kilku poprzednich optymalizacji można zaobserwować lekki spadek wydajności przy zwiększaniu problemu. Jest to spowodowane tym, że coraz mniejsza część macierzy mieści się w pamięci cache, a algorytm skonstruowany jest tak, że dość często odwołuje się do odległych od siebie części macierzy. Problem ten można rozwiązać zamieniając nieco kolejność operacji tak, by wynik pozostał ten sam, ale w danym momencie skupiać się tylko na konkretnym fragmencie macierzy.

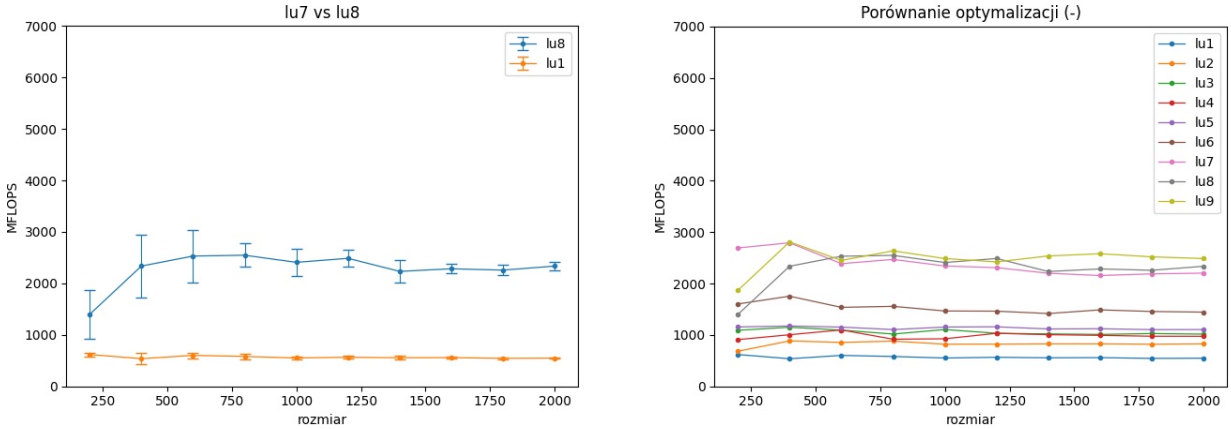
W ostatnim kroku optymalizacji skupiłem się właśnie na tym problemie. Rozwiązałem go poprzez dodanie funkcji `innerKernel`, która wykonuje głównie na bloku macierzy o ustalonej liczbie wierszy. Nie jest w pełni możliwe zblokowanie się tylko na konkretnym bloku, gdyż zewnętrzna pętla musi iterować po wierszach od `i=0` do `i=[koniec_bloku]`. Jednakże, jako że odniesienia do wierszy macierzy wykorzystujące zmienną `i` są stosunkowo rzadkie w porównaniu do tych wykorzystujących zmienną `j`, nie stanowi to problemu.

Po zastosowaniu zblokowania można zaobserwować ustabilizowanie się liczby MFlopsów dla większych rozmiarów macierzy (rys. 4).



Rys. 4: Porównanie wydajności lu7, lu8 oraz lu9

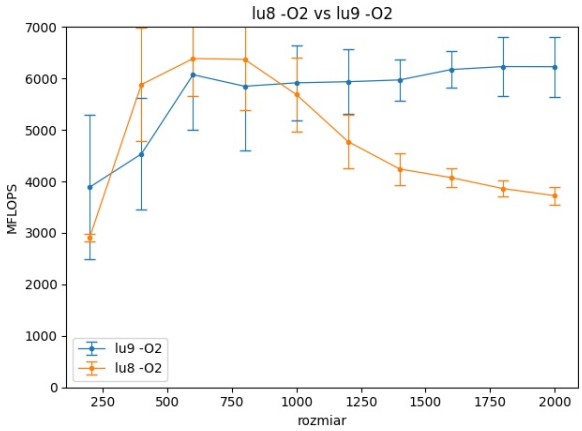
Ostatecznie udało się uzyskać wzrost z ok. 600 MFlopsów do ok. 2500 MFlopsów (rys. 5). W porównaniu wszystkich wersji (rys. 5) widać wyraźnie, że największy wzrost przyniosło użycie operacji wektorowych AVX.

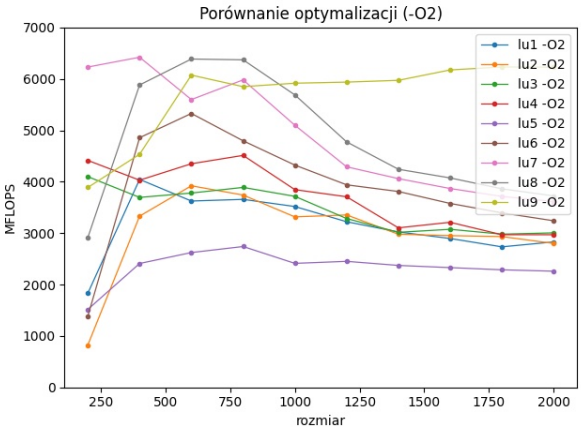


Rys. 5: Końcowe wyniki wykonanych optymalizacji. Kompilacja bez opcji -O2. Żeby zwiększyć czytelność, na wykresie przedstawiającym wyniki wszystkich wersji zrezygnowałem z odchyleń.

11 Optymalizacja z wykorzystaniem flagi -O2

Te same eksperymenty wykonałem także dla programów kompilowanych z opcją -O2. Na wynikowych wykresach (rys. 6) widać podobne zależności pomiędzy poszczególnymi wersjami. Tym razem jednak spadek wydajności dla dużych macierzy jest olbrzymi. Widać wyraźnie jak pomocne w tym przypadku wykonanie ostatniego kroku moich optymalizacji, tj. sprawienie by najczęściej wykonywane operacje nie potrzebowały co chwilę dostępu do bardzo odległych fragmentów macierzy.





Rys. 6: Końcowe wyniki wykonanych optymalizacji. Kompilacja z wykorzystaniem opcji -O2. Żeby zwiększyć czytelność, na wykresie przedstawiającym wyniki wszystkich wersji zrezygnowałem z odchyleń.