

[Opt Kodu] Zadanie 1

Autor sprawozdania: Maciej Sikora

Autor sprawozdania: Maciej Sikora

Wprowadzenie

Przebieg

Procesor na którym wykonywano ćwiczenie

Referencyjne czasy

Hiding computation in a subroutine

Computing four elements at a time

Further optimizing

Computing a 4 x 4 block of C at a time

Further optimizing 4x4

Blocking to maintain performance

Wykorzystanie standardu AVX

Ostateczne porównanie podstawowej implementacji do implementacji końcowej.


Wnioski

Wprowadzenie


Celem ćwiczenia było prześledzenie instrukcji znajdującej się pod linkiem oraz wykorzystanie do optymalizacji najnowszych instrukcji dostępnych na badanym procesorze.

Home · flame/how-to-optimize-gemm Wiki

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

 <https://github.com/flame/how-to-optimize-gemm/wiki>

flame/how-to-optimize-gemm




5 Contributors

5 Issues

866 Stars

228 Forks



W celu utrzymania zwięzłości sprawozdania umieszczałem tylko wykresy i bardzo krótkie komentarze do każdego etapu. Istotne wnioski znajdują się na samym końcu, w zwięzłej postaci.

Przebieg

Procesor na którym wykonywano ćwiczenie

AMD Ryzen 5 3600

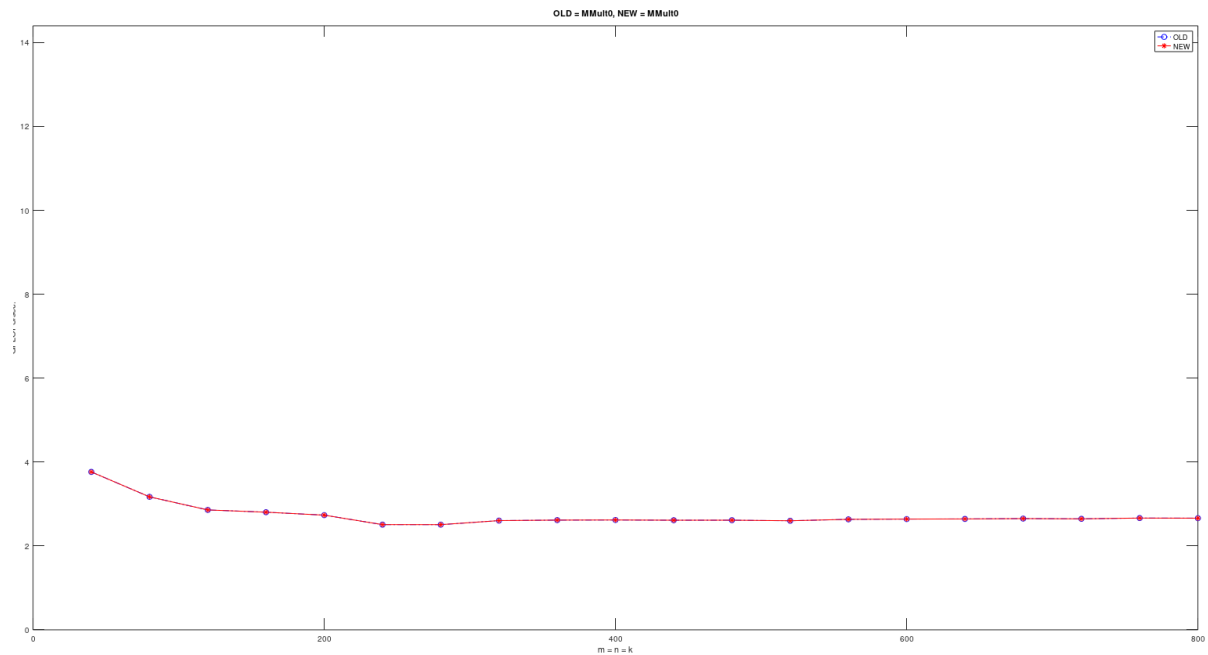
Base frequency: 3,600 MHz (3.6 GHz, 3,600,000 kHz)

Core count: 6

Thread count: 12

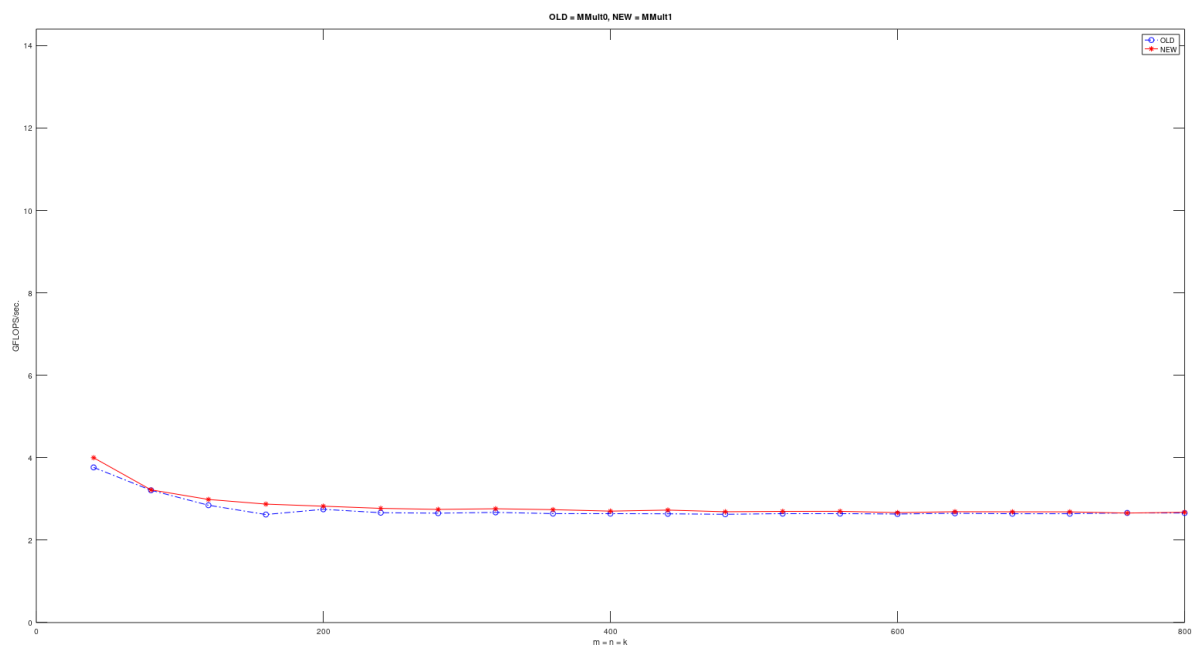
Referencyjne czasy

```
version = 'MMult0';
MY_MMult = [
40 3.764706e+00 0.000000e+00
80 3.170279e+00 0.000000e+00
120 2.858561e+00 0.000000e+00
160 2.804519e+00 0.000000e+00
200 2.734108e+00 0.000000e+00
240 2.507983e+00 0.000000e+00
280 2.507367e+00 0.000000e+00
320 2.603942e+00 0.000000e+00
360 2.615833e+00 0.000000e+00
400 2.618765e+00 0.000000e+00
440 2.613247e+00 0.000000e+00
480 2.613850e+00 0.000000e+00
520 2.600144e+00 0.000000e+00
560 2.634148e+00 0.000000e+00
600 2.640167e+00 0.000000e+00
640 2.643793e+00 0.000000e+00
680 2.651600e+00 0.000000e+00
720 2.646117e+00 0.000000e+00
760 2.664344e+00 0.000000e+00
800 2.662098e+00 0.000000e+00
];
```



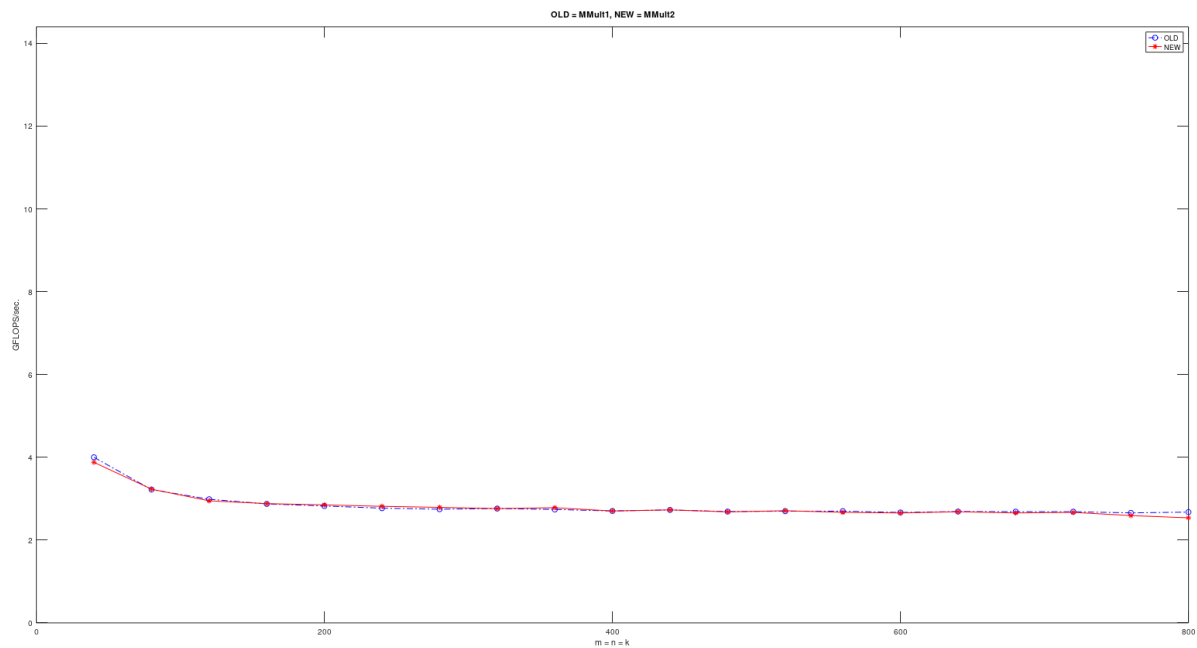
Widać z wyników, że pierwsze dwa rozmiary problemów były zauważalnie szybsze od pozostałych rozmiarów. Następnie od około rozmiaru 240 nie następowała żadna zauważalna różnica.

Hiding computation in a subroutine



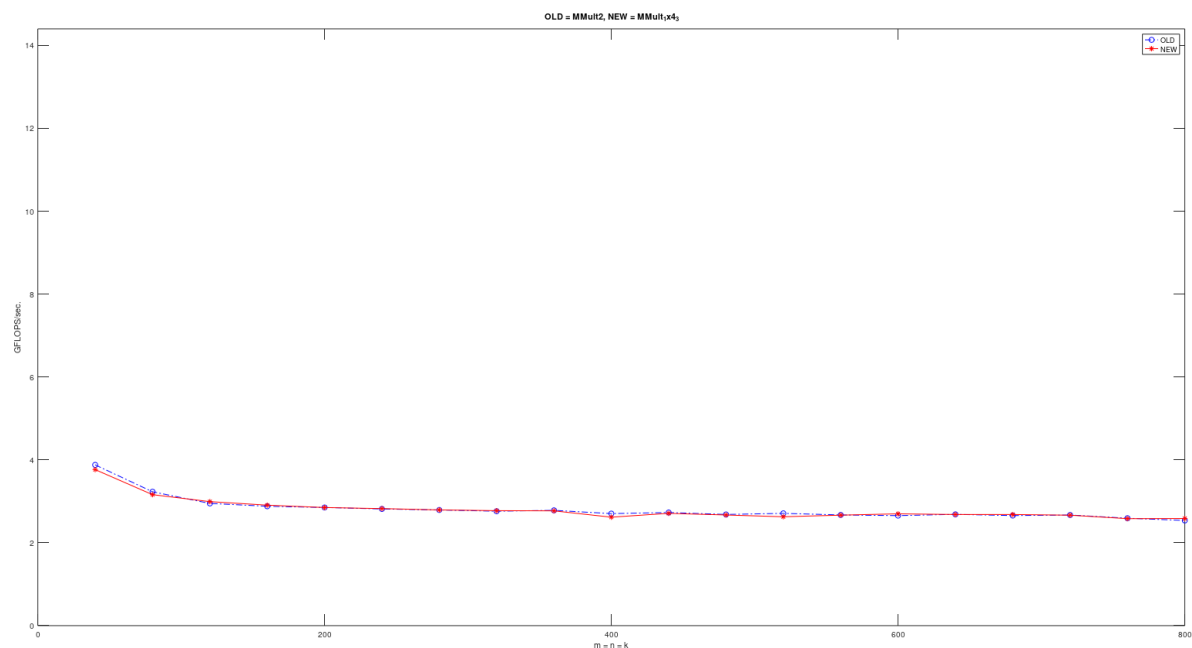
Pierwsza optymalizacja, która wyciągała operację do zewnętrznej funkcji nie pomogła za bardzo.

Tak samo druga, która w pętli wykonywała 4 operacje mnożeń zamiast jednej:



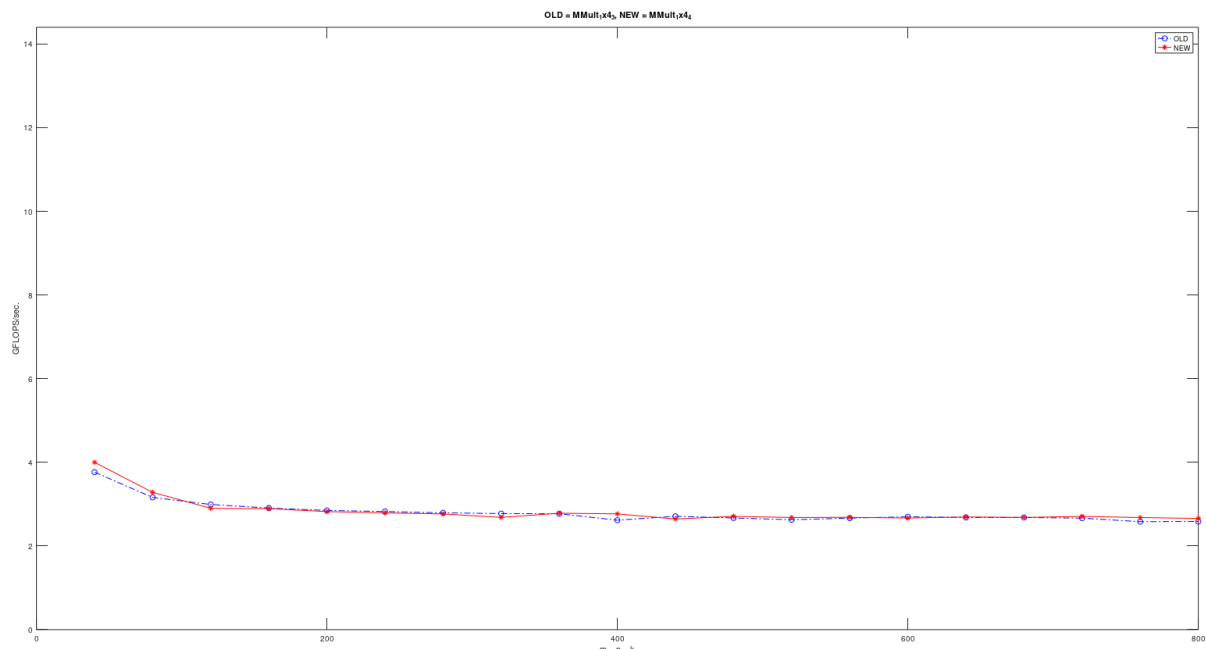
Computing four elements at a time

Jak i trzecia która wprowadzała ekstrakcję czterokrotnego mnożenia do zewnętrznej funkcji (analogicznie do optymalizacji 1)

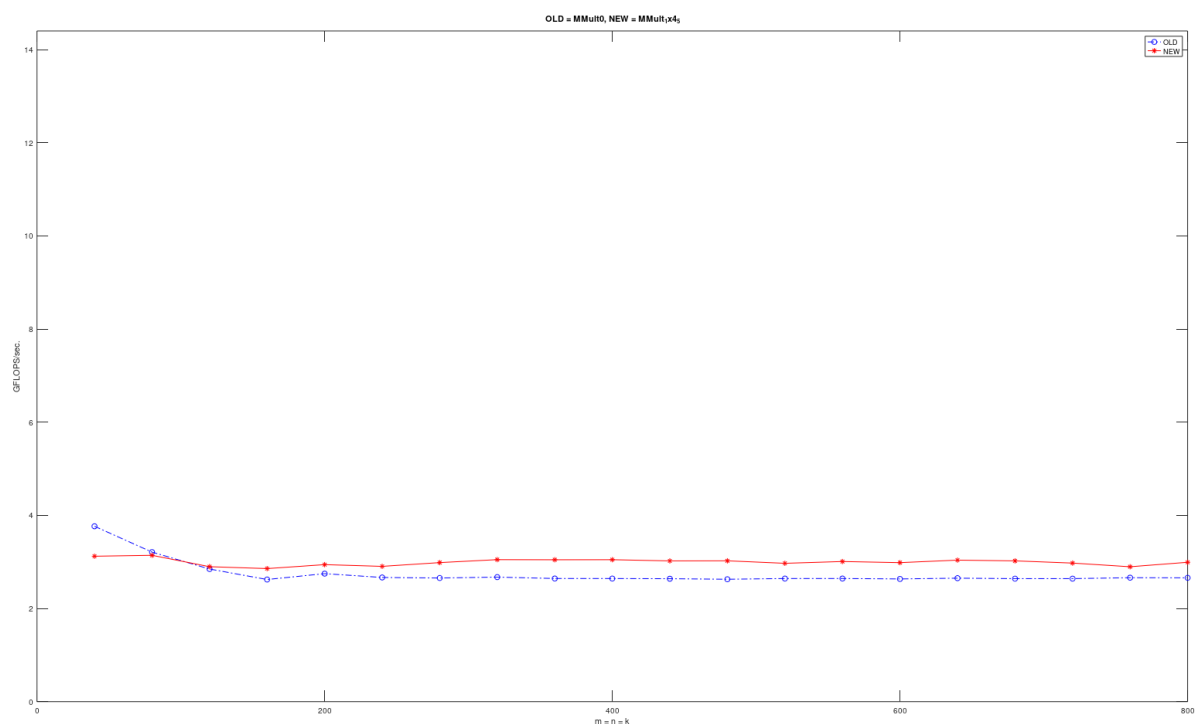


Czwarta optymalizacja zamienia funkcję którą stworzyliśmy w pierwszej optymalizacji na małe pętle które są wykonywane w funkcji która wykonuje cztery mnożenia w jednym cyklu nadrzędnej pętli.

W niektórych miejscach uzyskaliśmy minimalną poprawę, ale nie jest to nic szczególnie znaczącego.

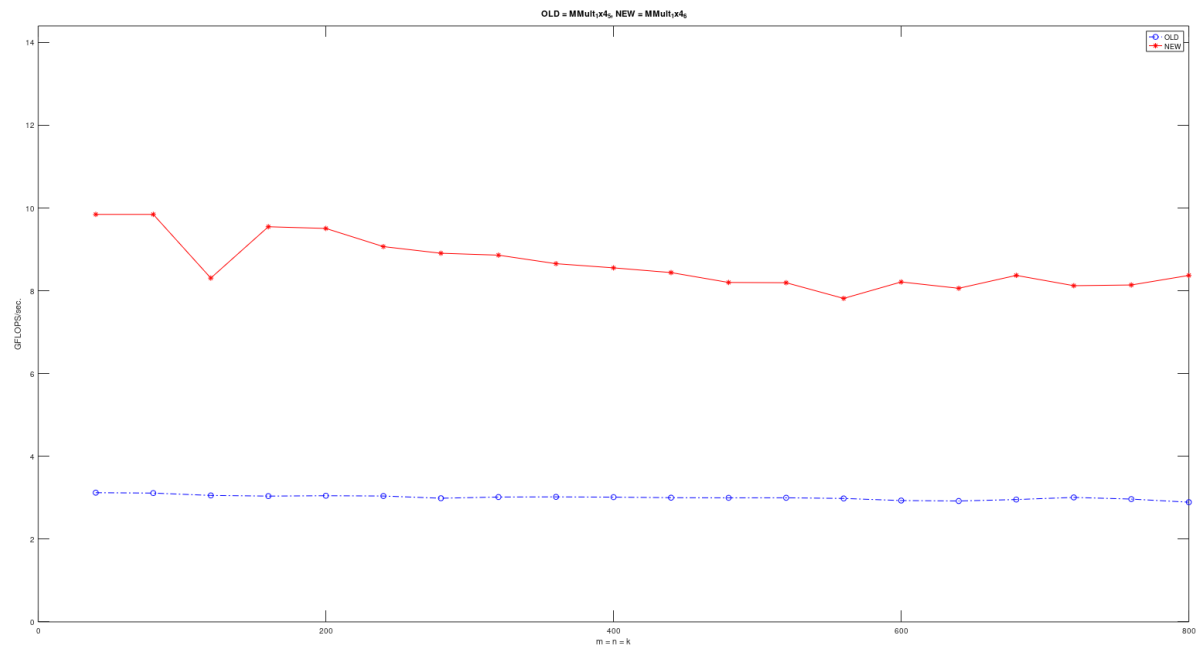


Następnie zamiast wywoływać cztery osobne pętle for wykonujemy te same obliczenia w jednym forze. Teraz można zaobserwować wzrost wydajności w większych macierzach, jednak w mniejszych widać lekki spadek wydajności. Warto zwrócić uwagę, że na moim wykresie nie widać bardzo znaczącego wzrostu wydajności po rozmiarze ok 460, ponieważ te macierze prawdopodobnie nadal mieszczą się w cache L2 mojego procesora. (Podczas prób zwiększenia rozmiarów macierzy otrzymywałem błąd "Segmentation fault", więc pozostałem przy testowaniu do max 800).

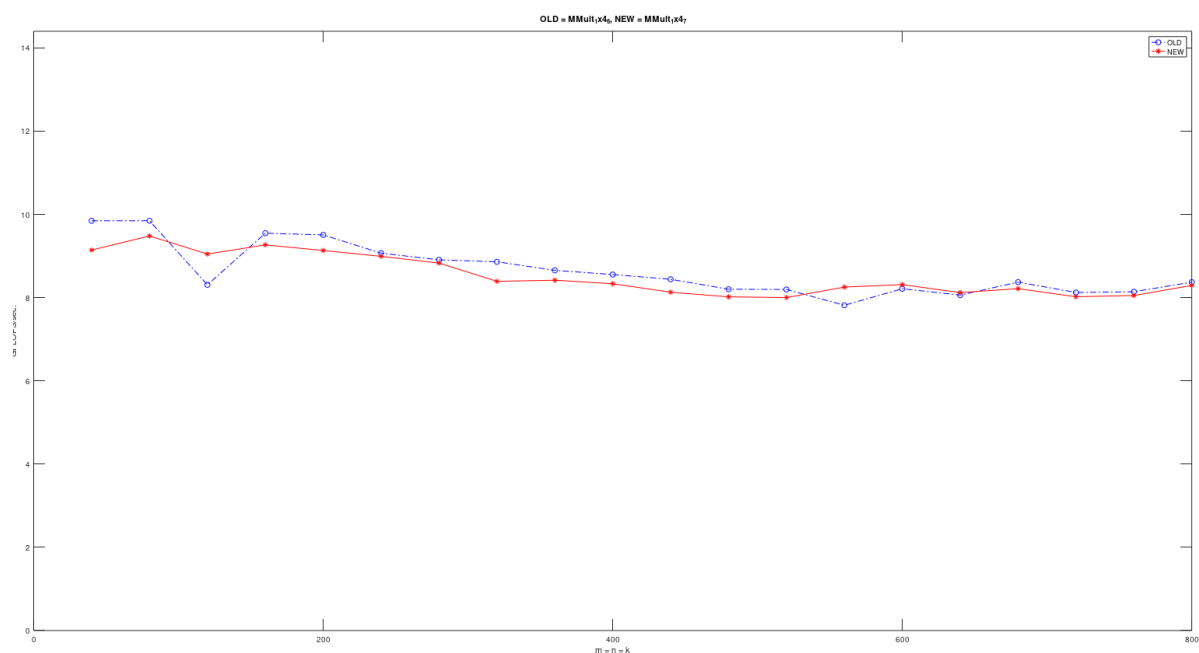


Further optimizing

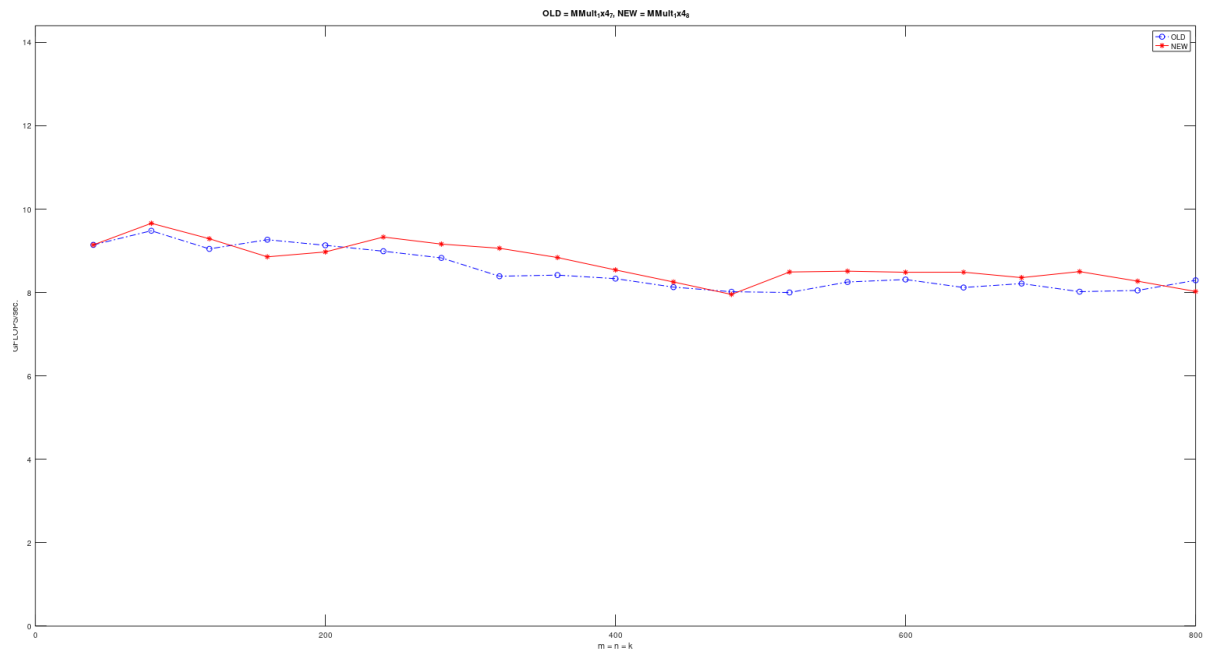
Kolejna optymalizacja wprowadza korzystanie z rejestrów i widać tutaj potężny wzrost w wydajności.



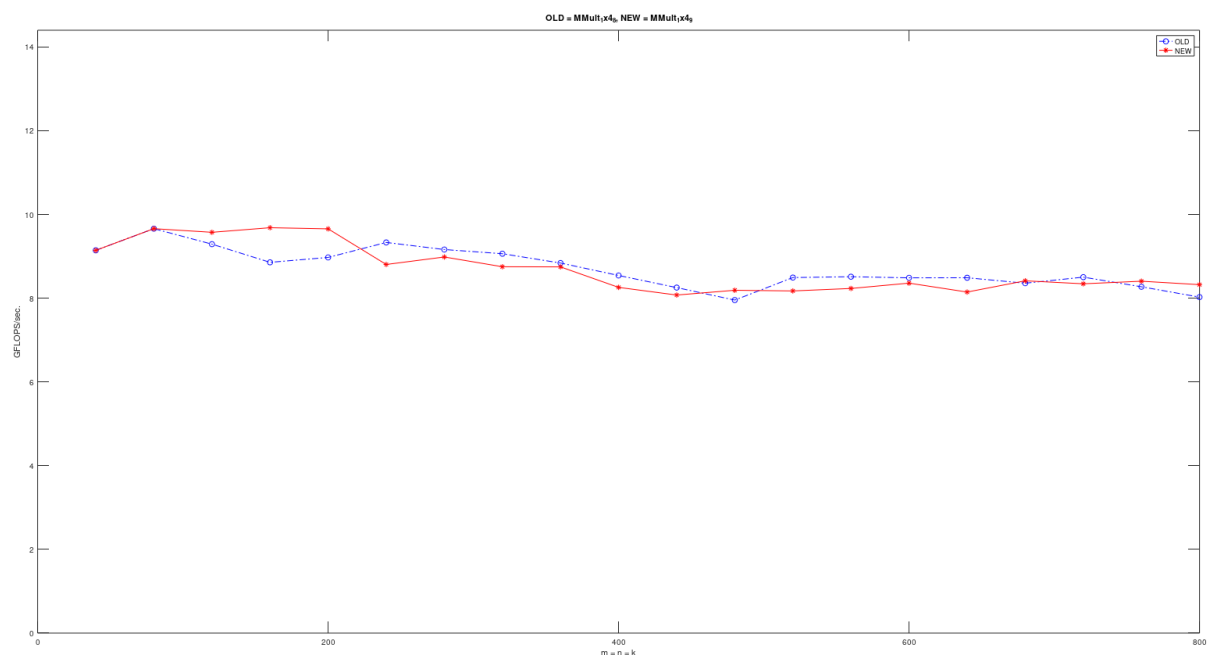
Kolejna optymalizacja polegała na wykorzystaniu wskaźników do pamięci zawierającej elementy $B(0,x)$ co powoduje zmniejszenie nakładu na indexowanie. Poprawiła ona lekko najgorsze przypadki, ale również zmniejszyła dobre przypadki. Prawdopodobnie część optymalizacji którą wykonywał kompilator przepadła.



W kolejnej optymalizacji pozbywamy się kolejnych pętli. Wyniki są podobne i średnio lepsze. Nie widać tutaj znacznego pogorszenia które było widoczne w dostarczonych materiałach

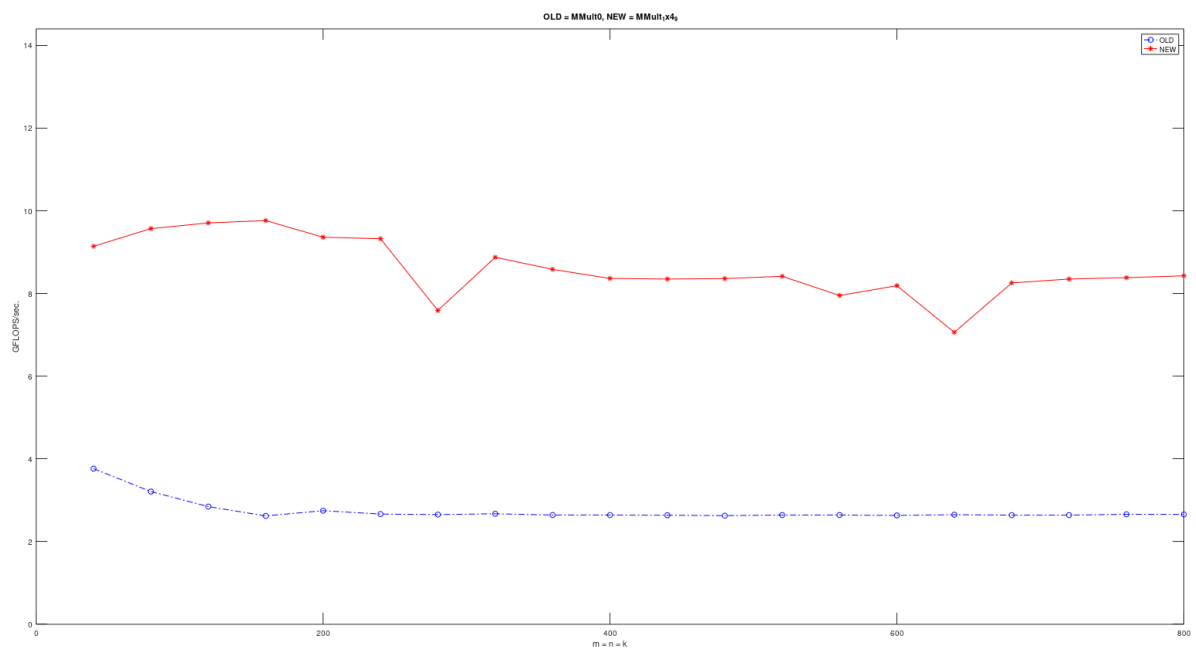


W kolejnej optymalizacji staramy się ręcznie zastosować trick z aktualizacją pointerów.



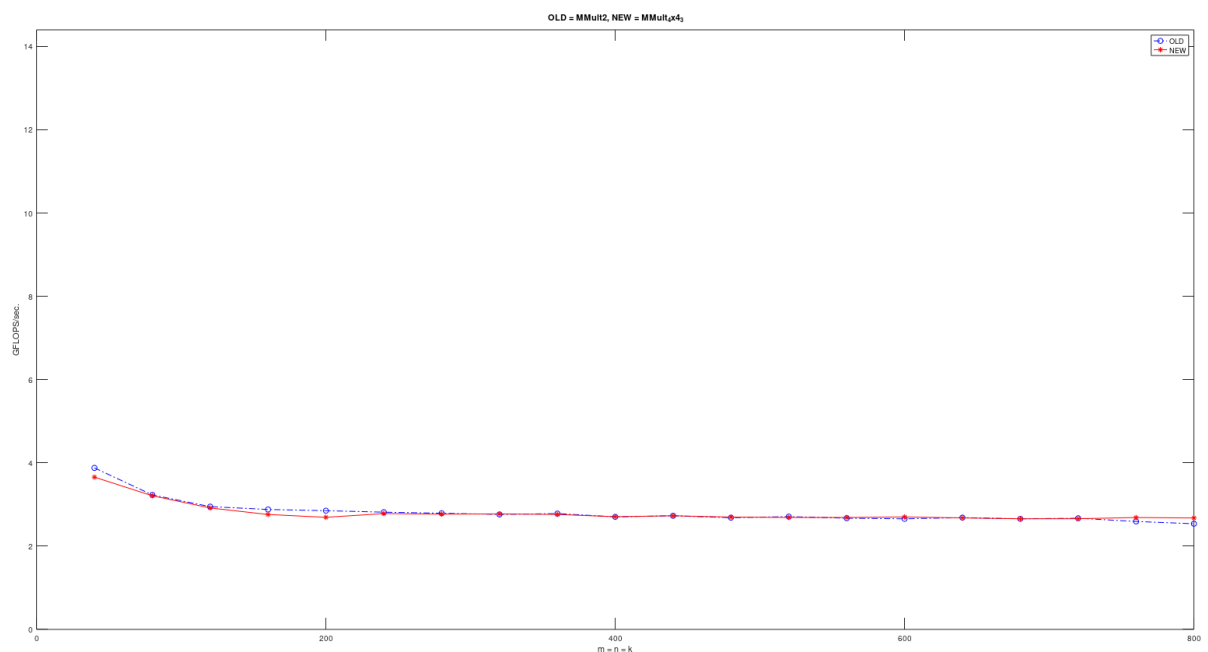
Pierwsza optymalizacja w tym dziale wprowadziła masowy skok w wydajności. Pozostałe ulepszenia wprowadzały dość niekonsystentne zmiany, bo nigdy nie było

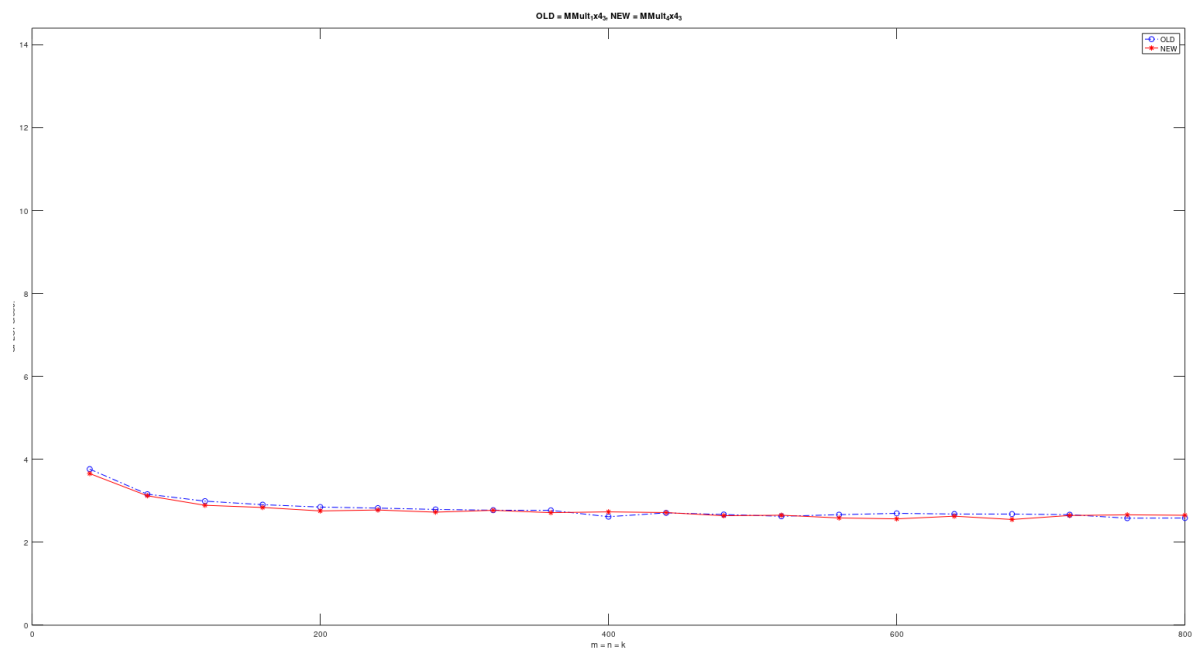
jasnego pogorszenia/polepszenia wyników. Poniżej znajduje się porównanie do wersji bazowej.



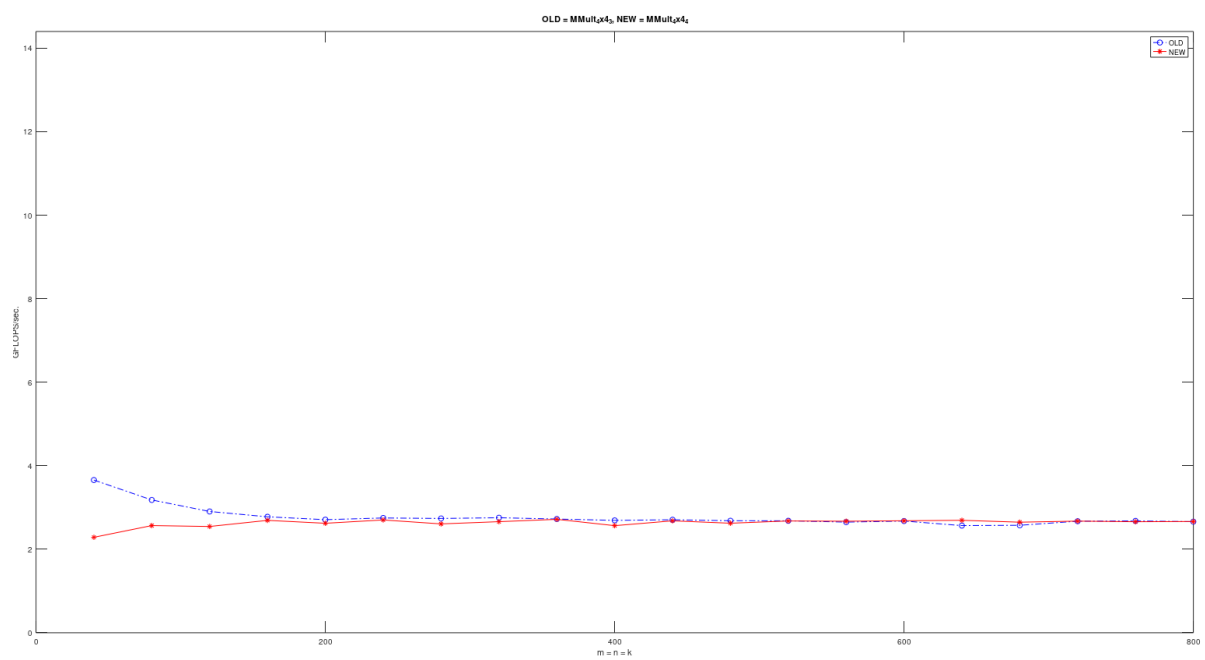
Computing a 4 x 4 block of C at a time

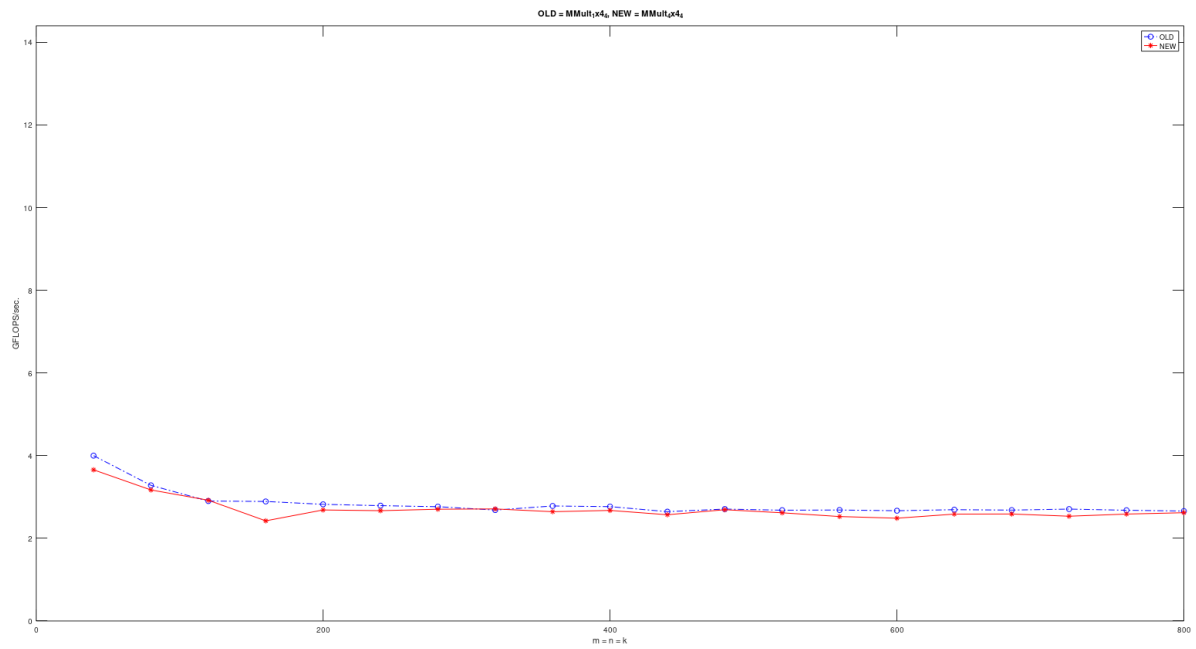
Pierwsza optymalizacja niewiele wprowadziła



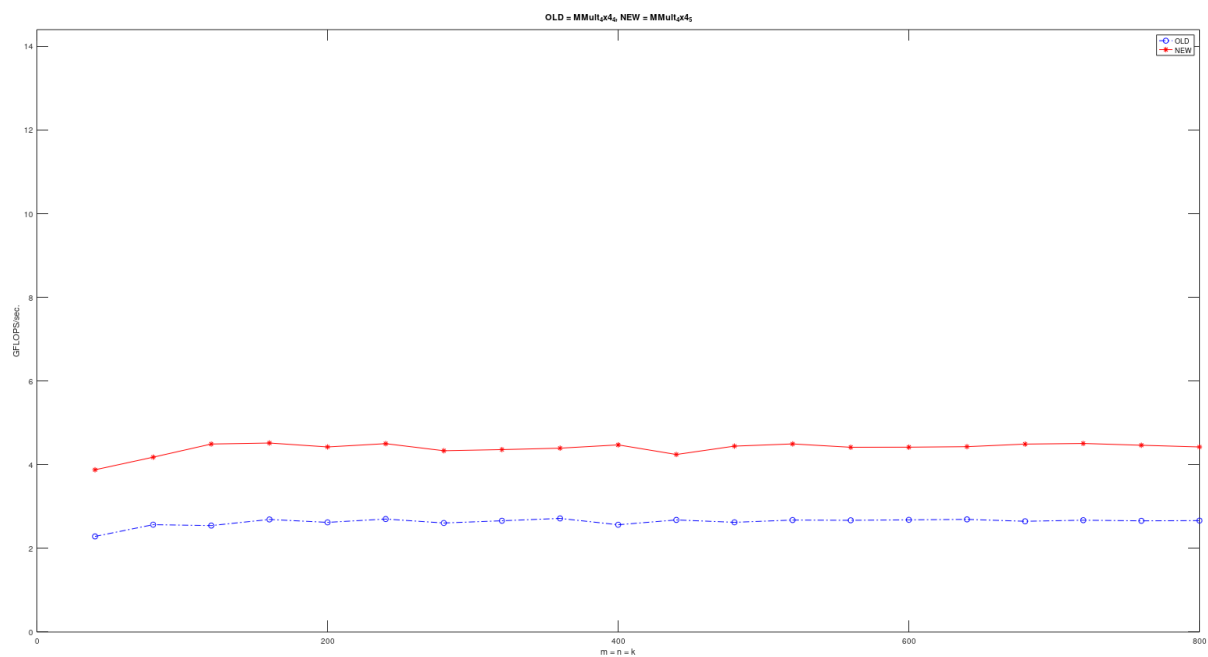


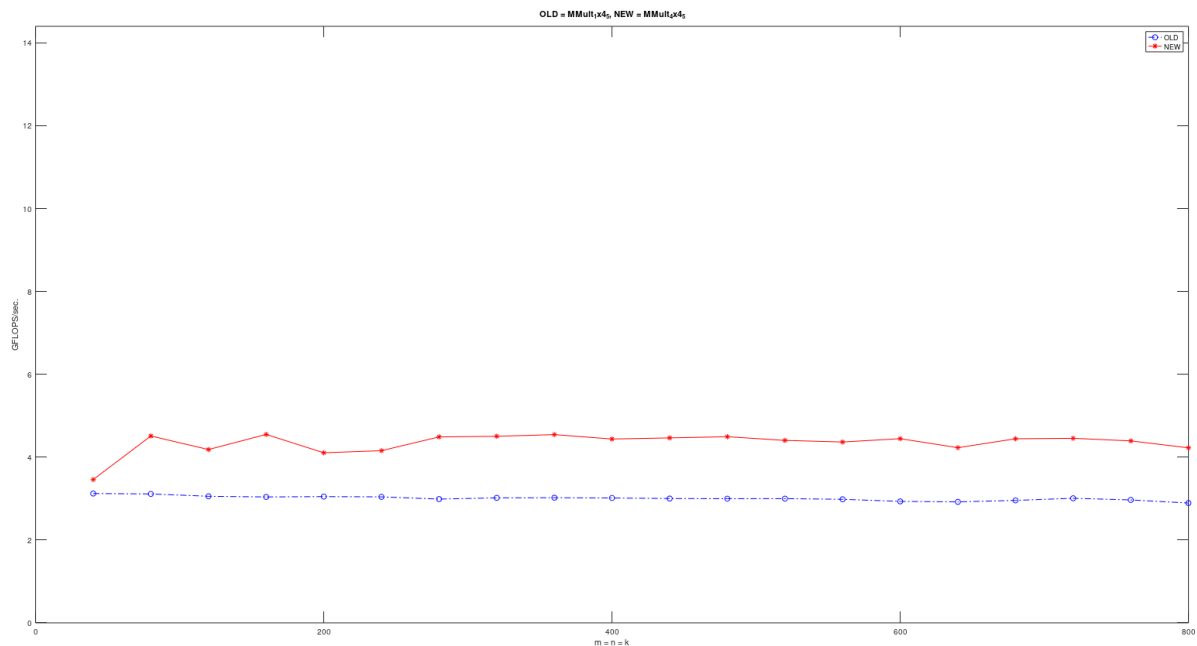
Dруга podobnie, a nawet lekko pogorszyła wydajność dla małych rozmiarów.



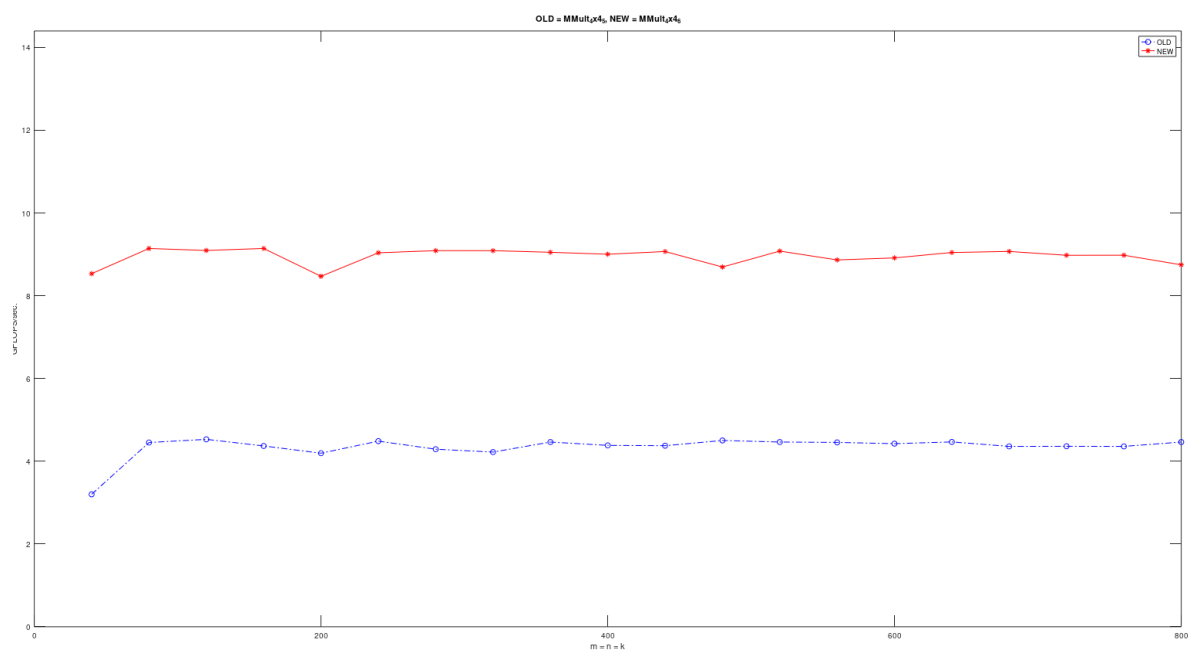


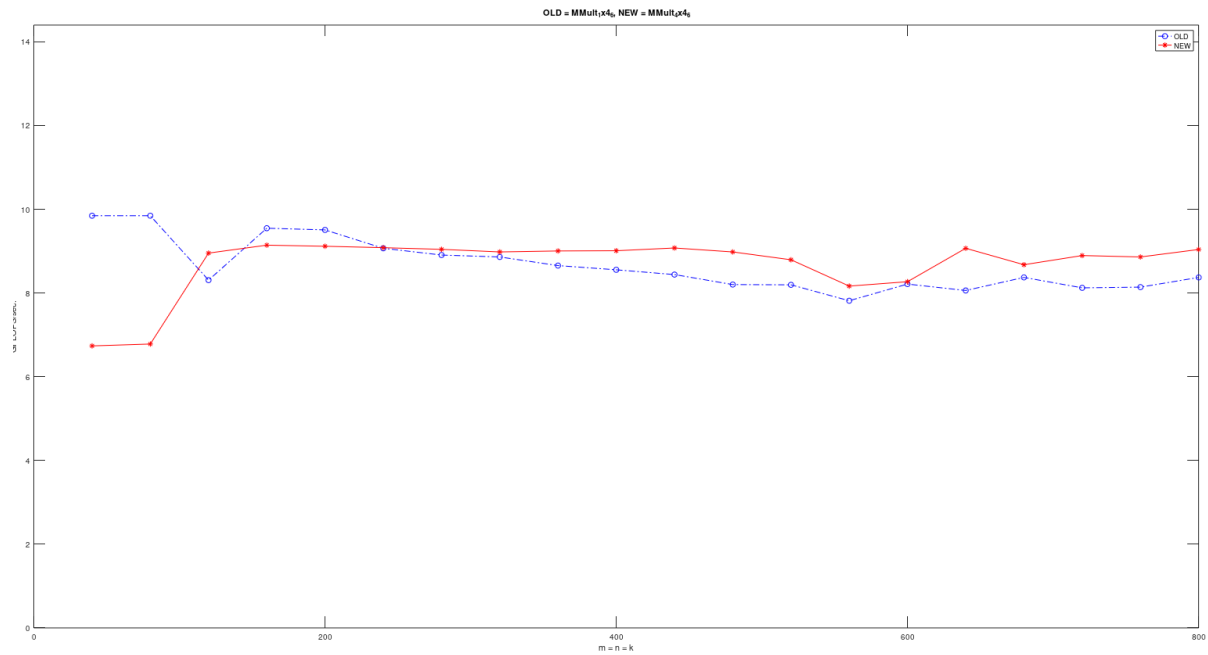
W trzecim ulepszeniu widać poprawę w stosunku do poprzedniego ulepszenia algorytmu działającego na bloku 4x4, jak również w stosunku do tego samego stopnia optymalizacji dla obliczeń dla wektora 1x4



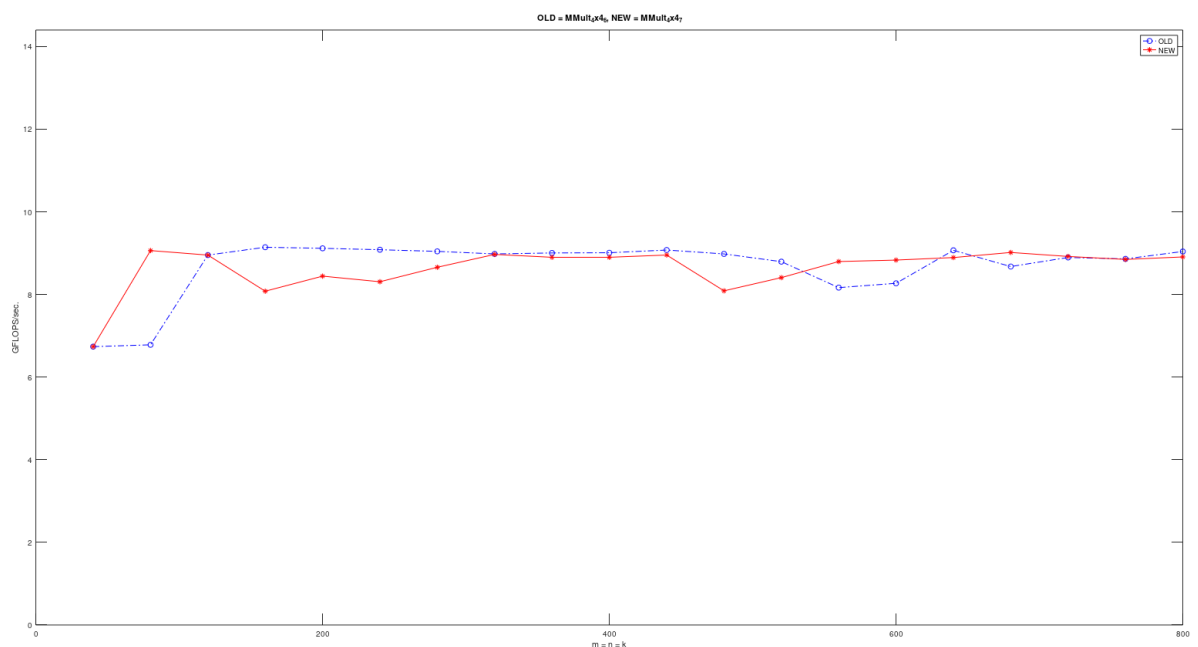


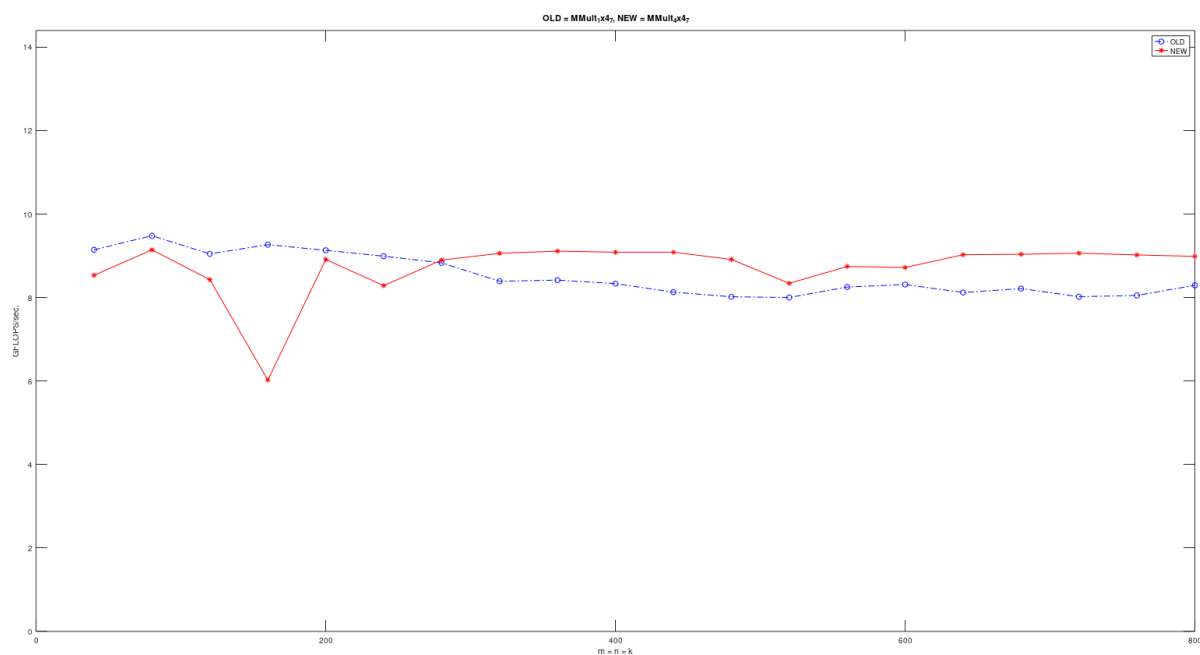
Wprowadzenie rejestrów daje nam bardzo znaczący wzrost w wydajności, jednak nadal jest porównywalny z implementacją 1x4. Implementacja blokowa jest lepsza dla większych macierzy, a implementacja vectorowa jest lepsza dla mniejszych macierzy na tym etapie.



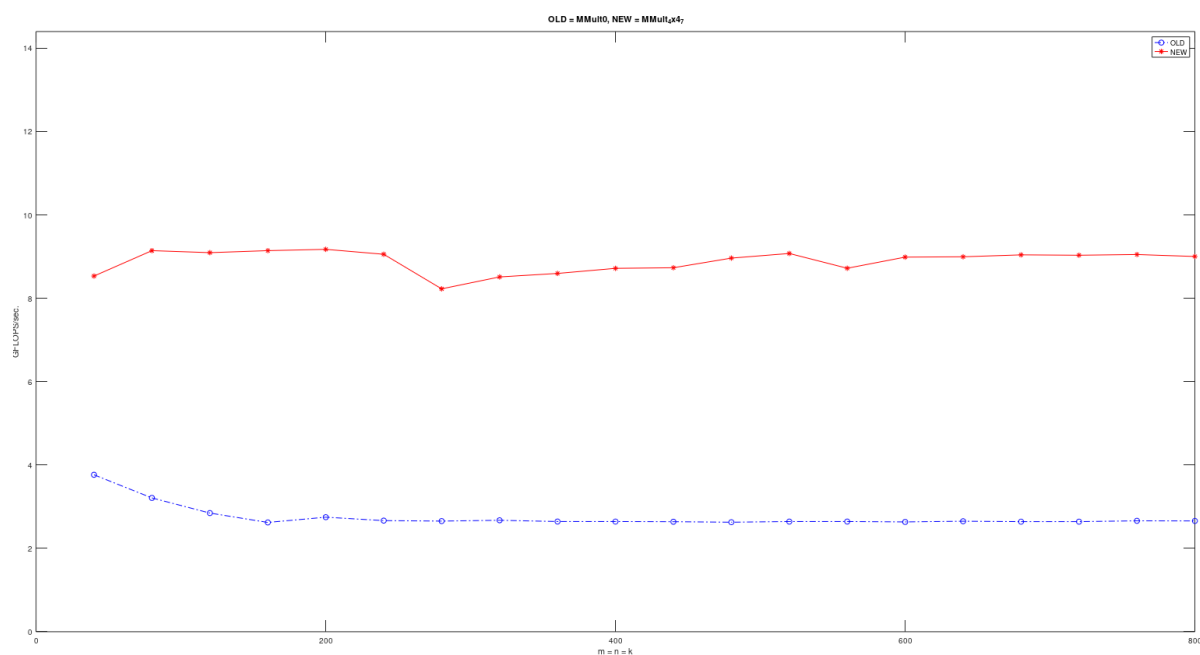


Wprowadzenie sztuczki z pointerami dało jednakże niekonsekwentne wyniki. Widać miejsca w których to ulepszenie poprawia wydajność, ale patrząc na wykres można powiedzieć, że nowy wykres jest częściej pod starym wykresem. Odwrotne spostrzeżenie można zaobserwować dla porównania z tą samą optymalizacją dla algorytmu 1x4.



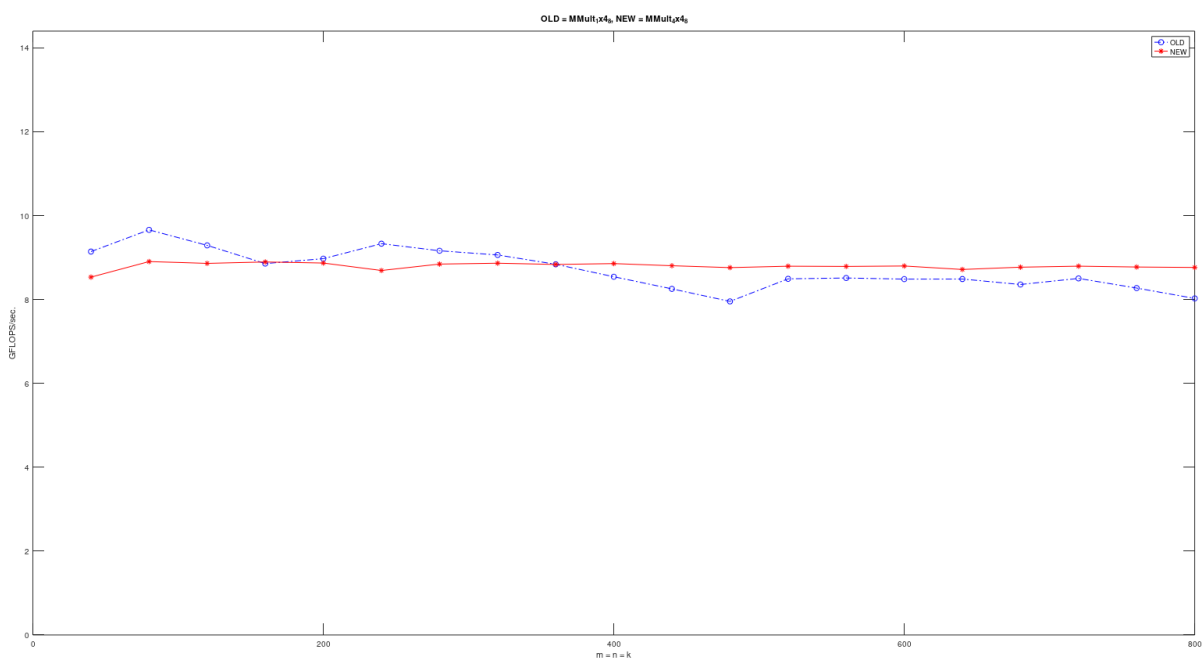
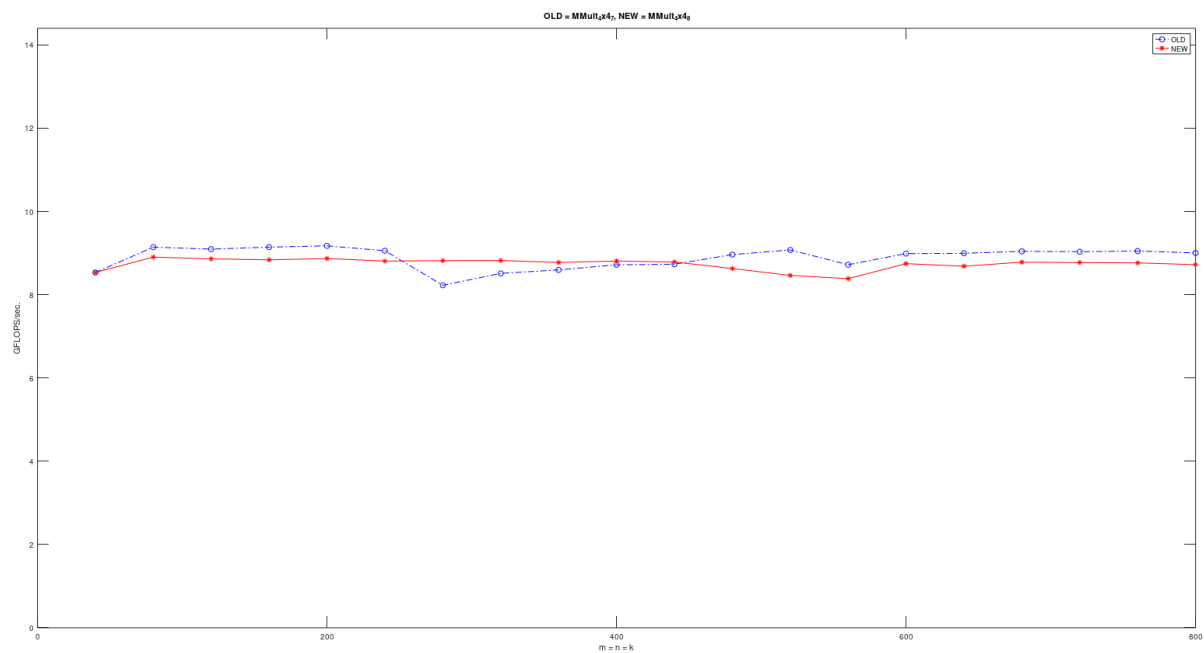


Porównanie z wersją bazową. Widać 2,5-3 krotny wzrost wydajności dzięki zastosowanym optymalizacjom.

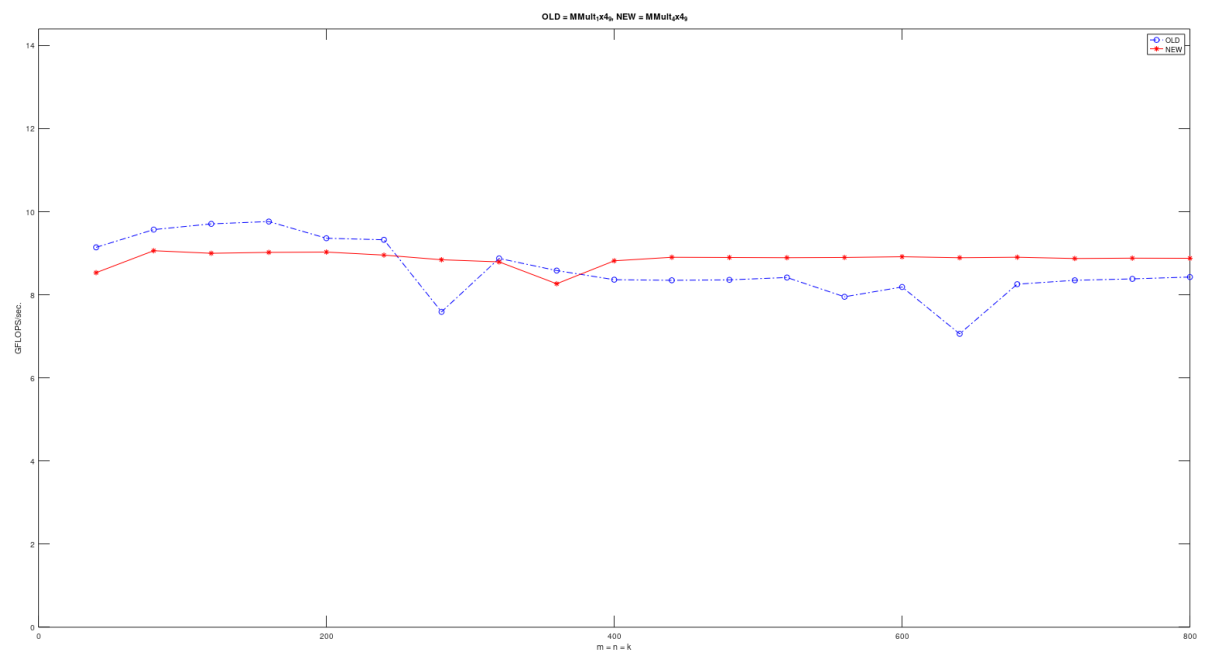
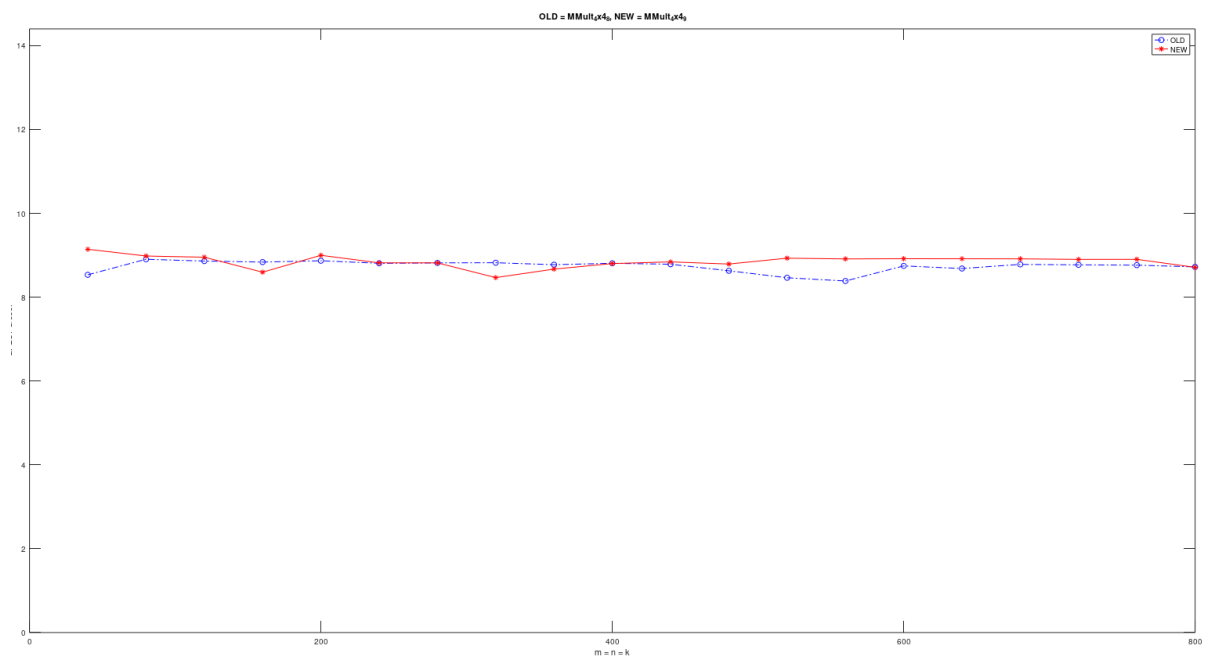


Further optimizing 4x4

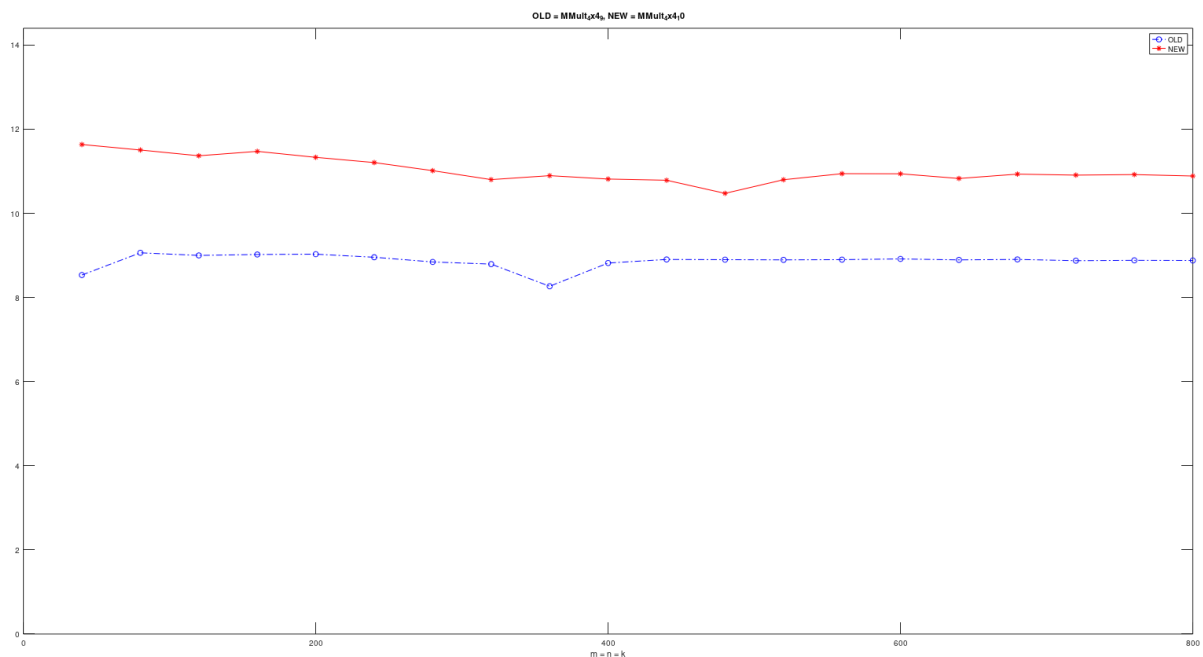
Wprowadzamy przechowywanie pointerów w rejestrach. Te zmiany nie wprowadzają konsystentnej zmiany w wynikach w porównaniu do poprzedniej wersji, jednak można zauważyć, że nastąpiła poprawa dla większych rozmiarów macierz względem algorytmu 1x4.



Zmiana kolejności wykonywania obliczeń z rzędów na na bloki 2x2 idące od lewej do prawej. Wyniki są nadal bardzo podobno, jednak widać lekkie polepszenie.

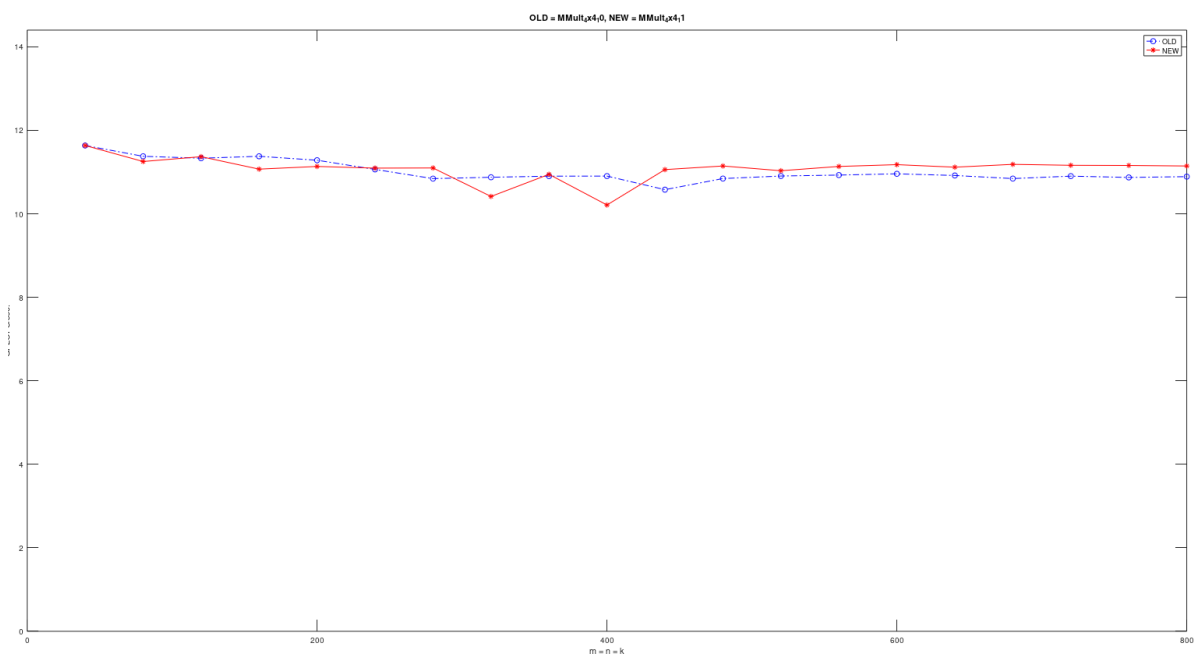


Wprowadzenie funkcji i struktur wektorowych dla standardów SSE, SSE2, SSE3.
Znacząco widać polepszenie wydajności na całej przestrzeni.

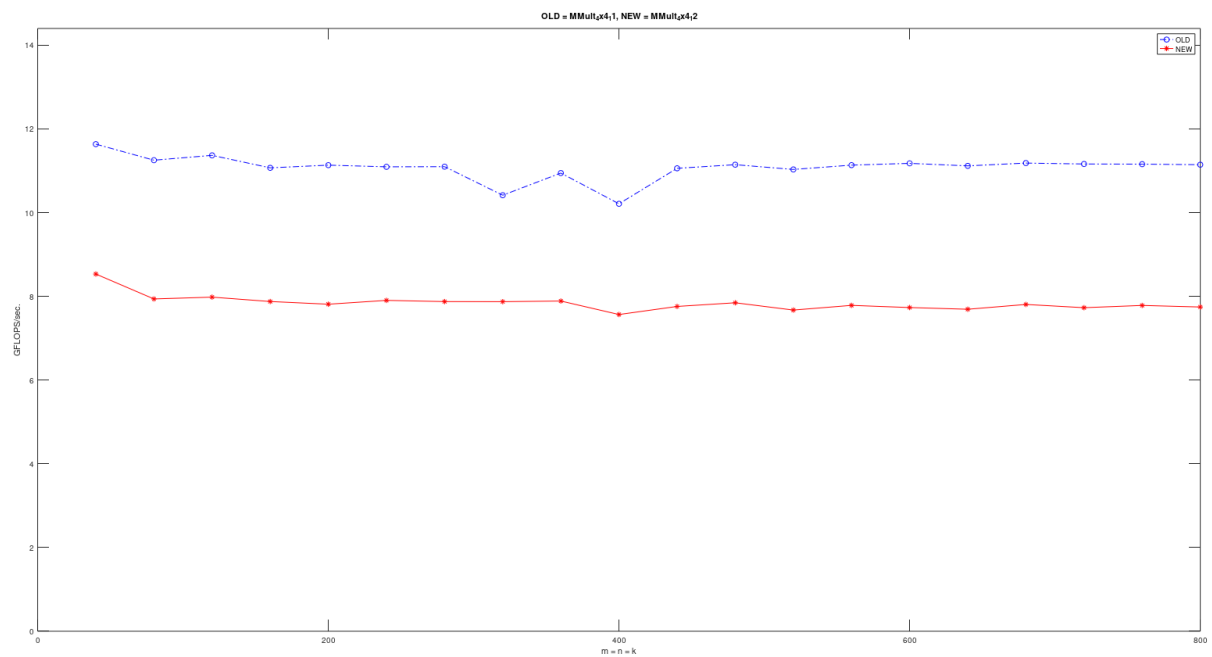


Blocking to maintain performance

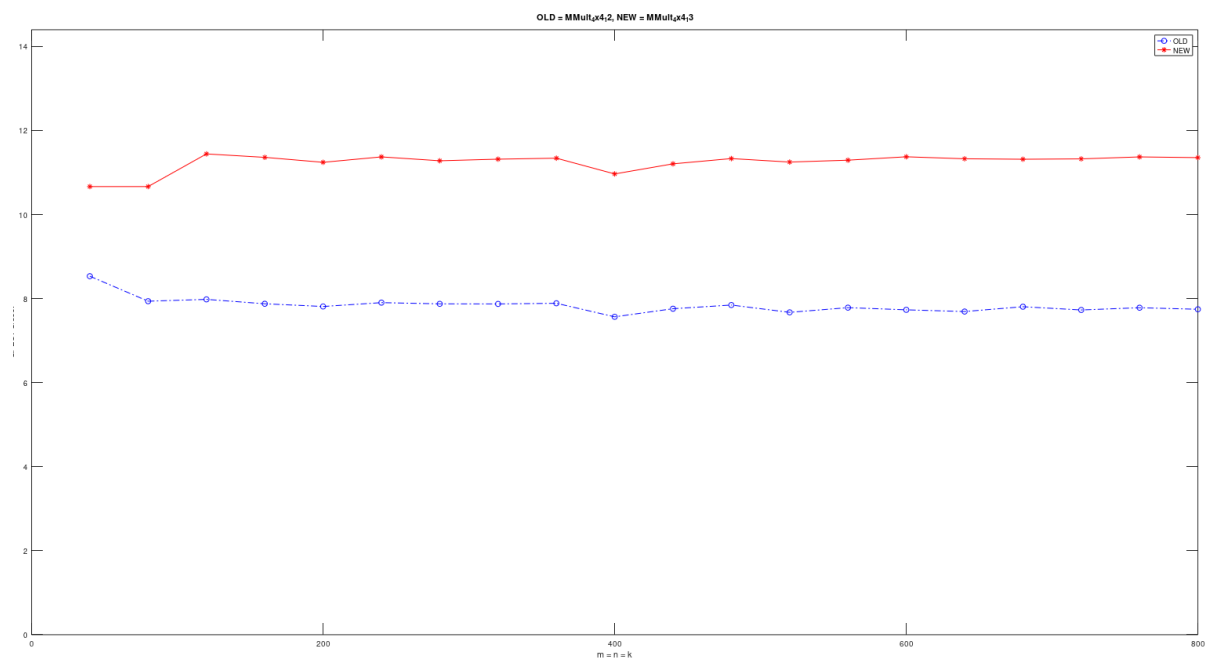
W tym momencie zamiast liczyć całą macierz, tworzymy funkcję która oblicza tylko blok całej macierzy. To ulepszenie daje konsystentnie lepsze wyniki dla macierzy większych od 400. Ta optymalizacja miała służyć poprawieniu sytuacji w której macierz nie mieściła się do cache L2, w moim przypadku ciężko było to zaobserwować, ponieważ mój cache L2 jest prawdopodobnie większy od cache używanego do stworzenia ćwiczenia.



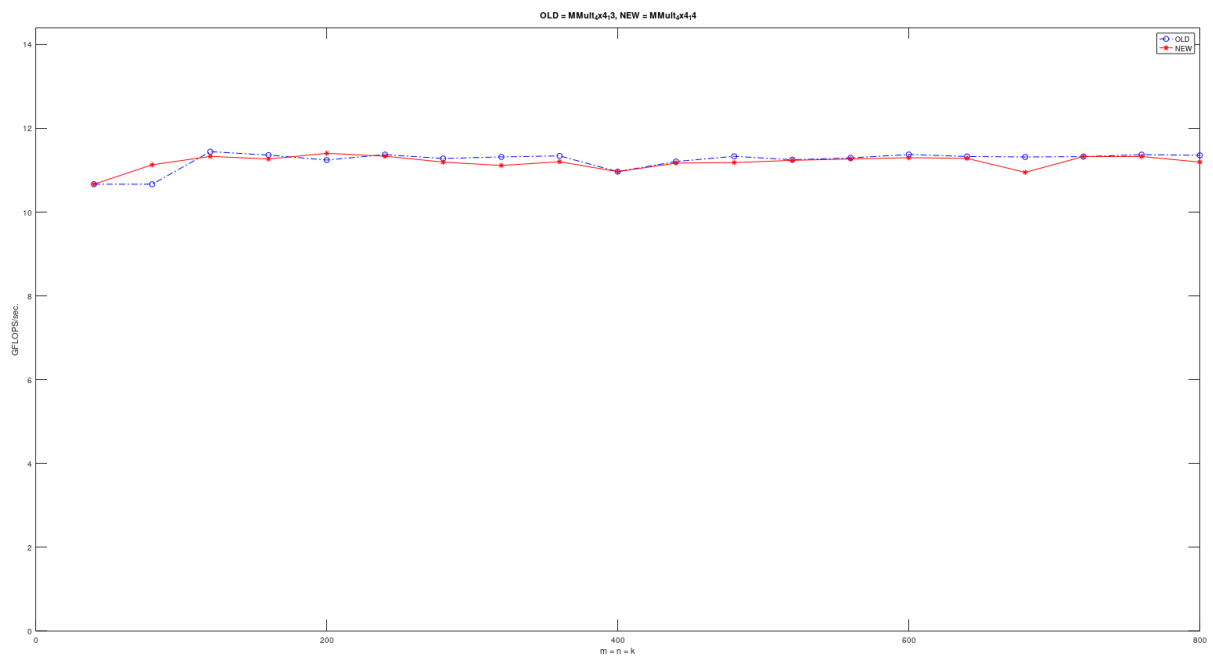
Dodanie pakowania macierzy znacząco pogarsza wydajność programu.



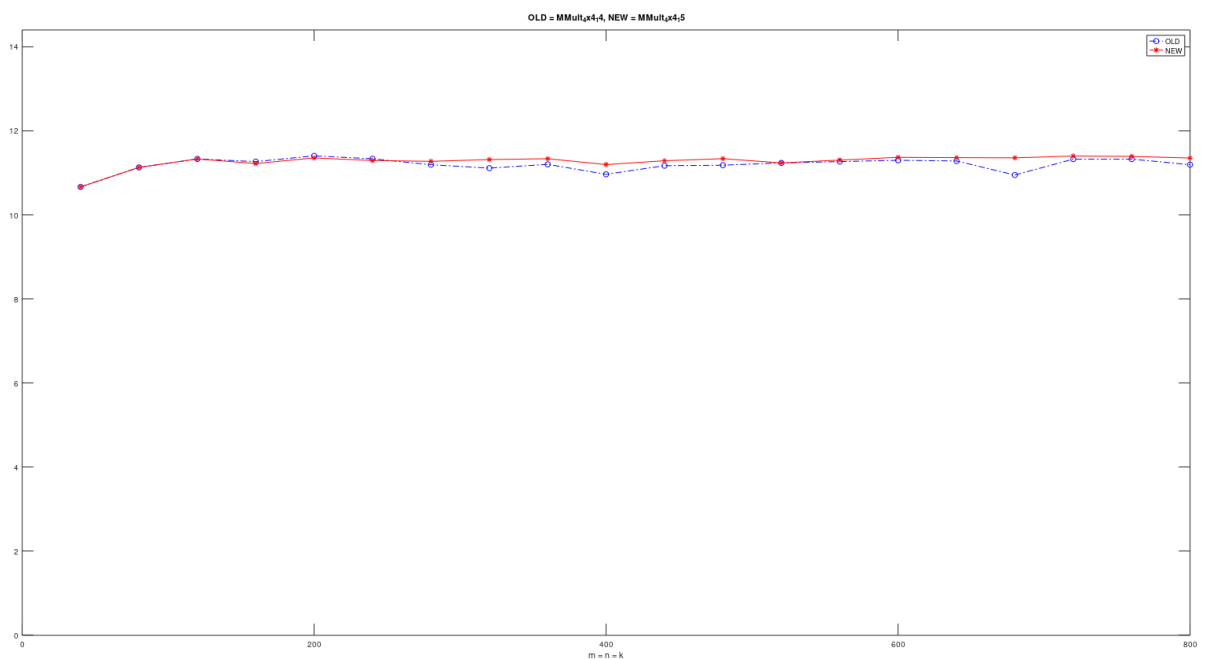
Teraz dodajemy zapamiętywanie bloku po pierwszy użyciu co daje znaczący wzrost wydajności.



Następnie dodajemy pakowanie kx4 bloków. Zmiana ta nie poprawie bardzo widocznie programu.



Dodajemy ostatecznie kod sprawiający, że nie pakujemy wielokrotnie tych samych bloków.



Wykorzystanie standardu AVX

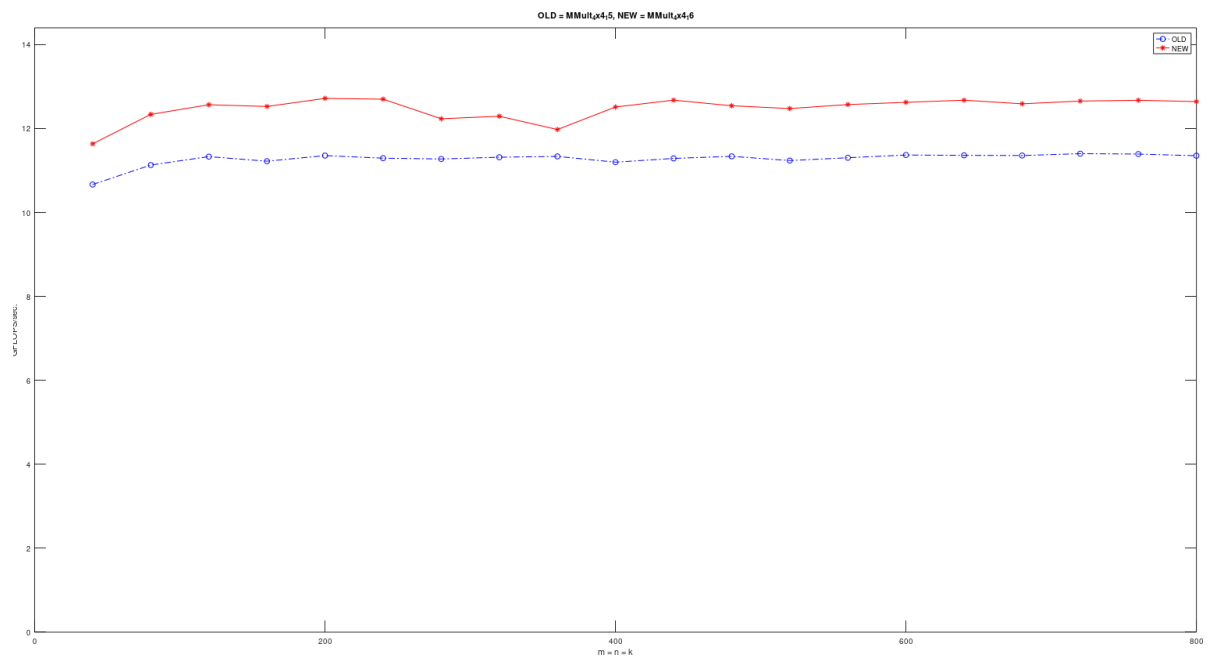
Tą część zadania wykonałem bazując na instrukcjach dostępnych na stronie



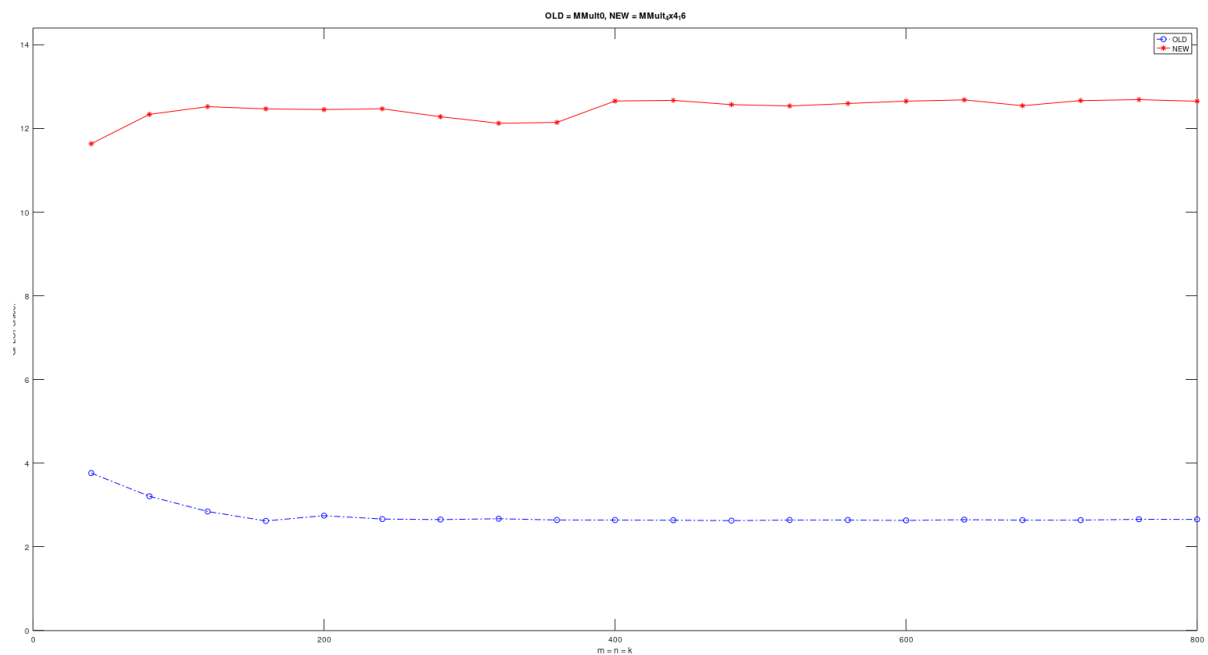
Zamiana instrukcji na AXV i korzystanie z `__m256d` spowodowała zauważalną różnicę.

Mapowanie funkcji:

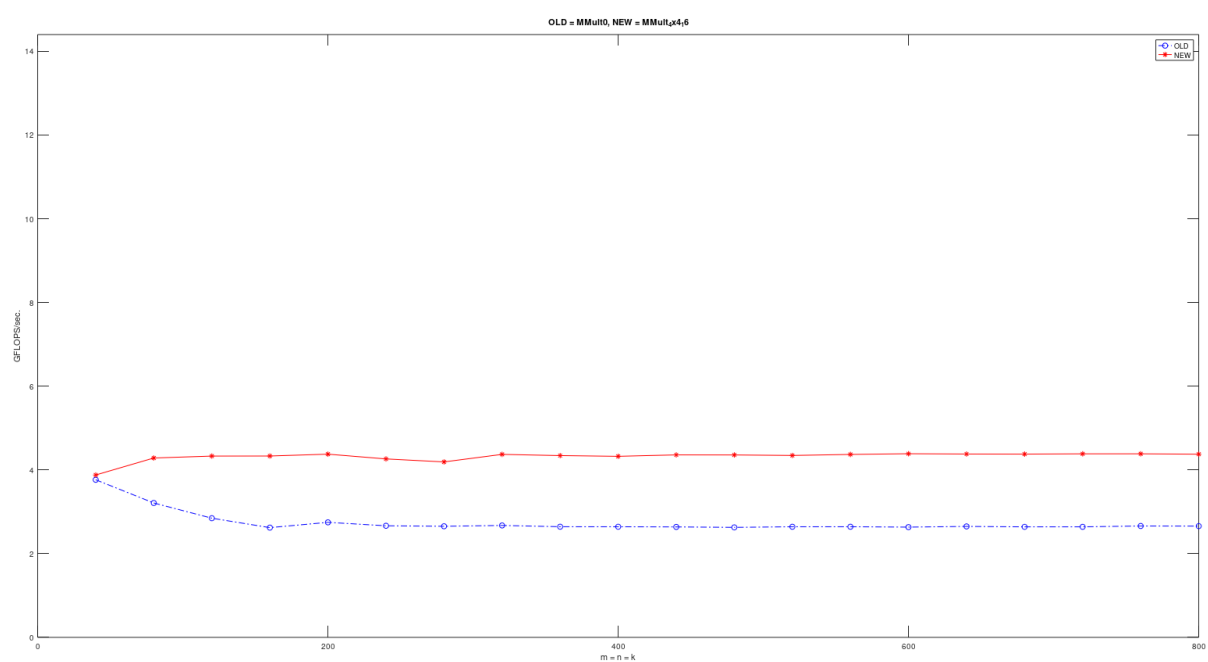
```
_mm_setzero_pd() -> _mm256_setzero_pd()
_mm_load_pd -> _mm256_loadu_pd()
_mm_loadup_pd((double *) b) -> _mm256_broadcast_sd((double *) b)
```



Ostateczne porównanie podstawowej implementacji do implementacji końcowej.



Porównanie algorytmów bez wykorzystania żadnej flagi optymalizującej.



Wnioski

Większość kroków optymalizujących przynosiła małe albo niekonsystentne efekty. Największymi ulepszeniami wydajności było skorzystanie z rejestrów, wprowadzenie operacji blokowych oraz wykorzystanie instrukcji AVX. Dzięki tym zmianom byłem w stanie uzyskać blisko pięciokrotne przyspieszenie operacji zmiennie przecinkowych.

Bardzo ciekawą obserwacją jest to, że nie wykorzystanie flagi do zoptymalizowania kodu pogorszyło wydajność prawie trzy krotnie.

Z tego zadania warto wyciągnąć następującą wiedzę:

- jeżeli nie ma jasnych przeciwwskazań to najlepiej zawsze korzystać z flag optymalizujących
- jeżeli jesteśmy w stanie zidentyfikować zmienne do których często się odwołujemy warto je umieścić `explicit` w rejestrach
- w celu poprawy wydajności programu im więcej operacji jesteśmy w stanie napisać "z palca" oraz mniej korzystając z pomocy pętli i odwołań do tablic w pamięci tym lepiej
- wykorzystanie najnowszych operacji dostępnych dla procesora, pomimo tego, że może być cięższe w implementacji, potrafi znacząco poprawić wydajność