AGH Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Optymalizacja GEMM

15 kwietnia 2024

Jakub Szaredko

1 Architektura procesora

Zadanie dostosowywałem pod urządzenie z wbudowanym procesorem Apple M1 Pro. Specyfikacja techniczna tego procesora nie jest publicznie dostępna, bazowałem na Wikipedii, gdzie zostały zapisane podstawowe parametry jednostki. Niestety, nie mogłem się doszukać liczby operacji zmiennoprzecinkowych na sekundę, jedyne informacje jakie znalazłem dotyczyły zintegrowanej karty graficznej.

Producent	Apple
Model	M1 Pro
Mikroarchitektura	Firestorm i Icestorm
Architektura instrukcji	ARMv8.5-A
Technologia	5nm
Liczba rdzeni	8
Taktowanie maksymalne	3.22 GHz
Cache L1	320 / 192 KB
Pamięć RAM	16 GB

2 Dostosowanie kompilacji

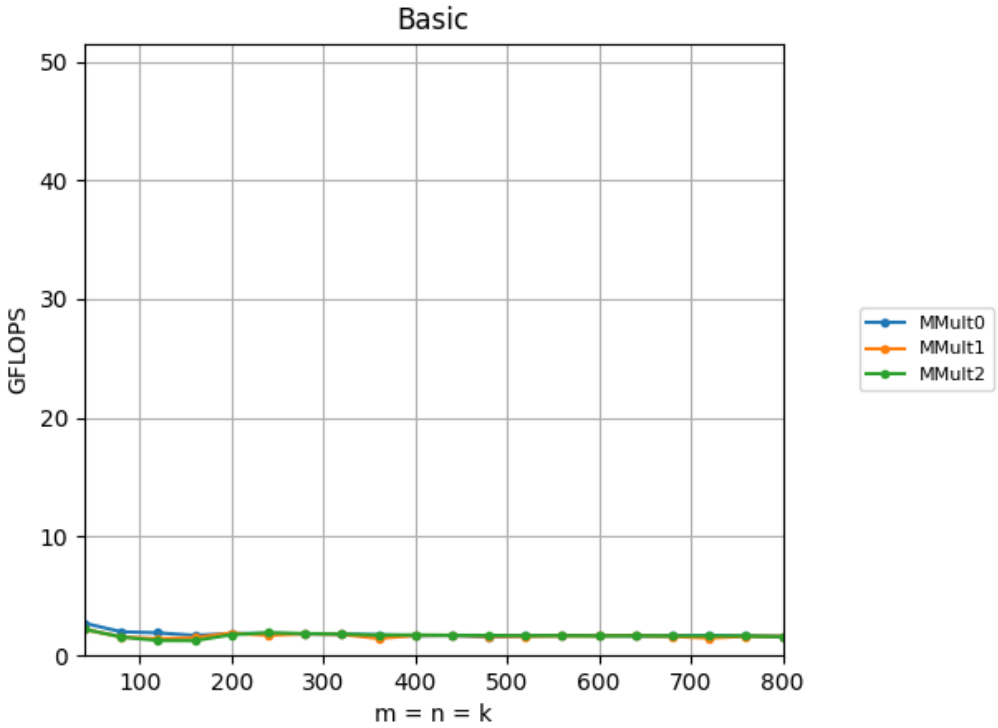
1. Dodanie parametru `-march`.
2. Wykorzystanie forsowanie kompilatora GCC za pomocą komendy `gcc-13`, **macOS linkuje Clang ze standardową komendą** `gcc`.

3 Zastosowane optymalizacje

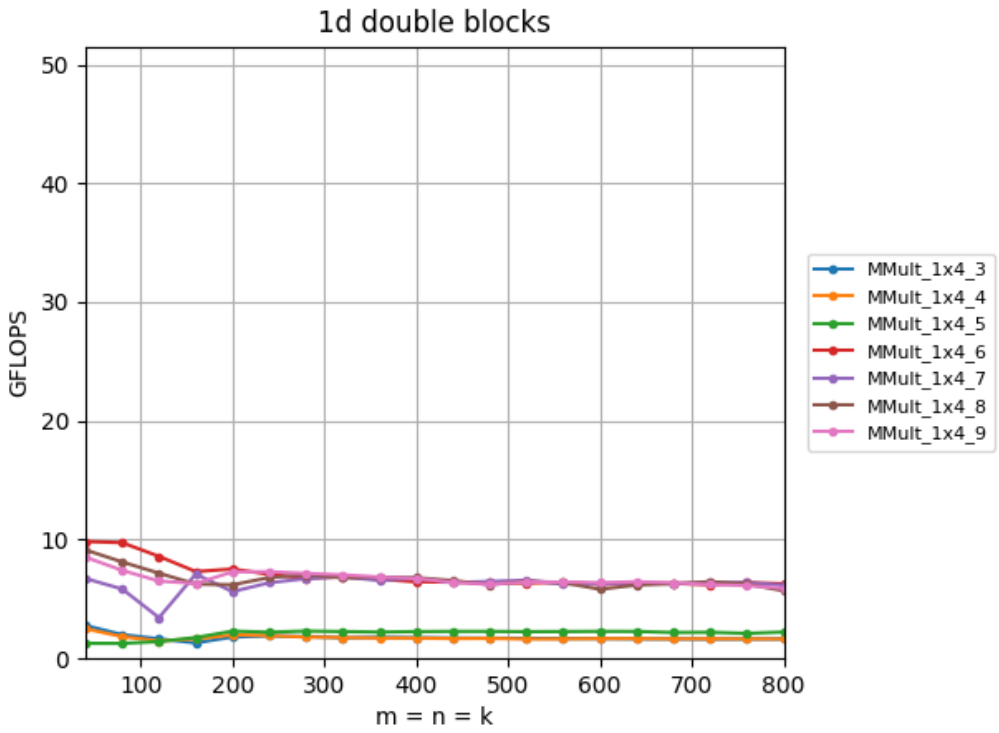
1. `MMult1`: Dodanie funkcji pomocniczej `AddDot`, która mnoży wiersz z kolumną.
2. `MMult2`: Mnożenie wierszy i kolumn w blokach o rozmiarze 4.
3. `MMult_Nx4_3`: Dodanie funkcji `AddDotNx4`, która mnoży wiersze i kolumny w blokach o rozmiarze 4.
   - $N = 1$, poruszanie się tylko po kolumnach.
   - $N = 4$, poruszanie się w obu wymiarach.
4. `MMult_Nx4_4`: Rozwinięcie funkcji `AddDot`.
5. `MMult_Nx4_5`: Połączenie wszystkich rozwinięć funkcji w jedną pętlę.
6. `MMult_Nx4_6`: Zdefiniowanie dodatkowych zmiennych, które zapisują wyniki mnożenia oraz zawartość komórki z A w rejestrach.
7. `MMult_Nx4_7`: Wskaźniki na komórki B i ich inkrementacja.
8. `MMult_Nx4_8`: Poruszanie się w blokach o rozmiarze 4 w `AddDotNx4`.
   - $N = 1$, poruszanie się po kolumnach (krok pętli = 4).
   - $N = 4$, poruszanie się po wierszach (krok pętli = 1).
9. `MMult_Nx4_9`: Zamiana inkrementacji na relatywne adresowanie (dodawanie do wskaźnika odpowiedniego indeksu).

10. `MMult_Nx4_10`: Rejestry i instrukcje wektorowe.
11. `MMult_Nx4_11`: Podział na dodatkowe bloki o rozmiarach 128 i 256, dodanie funkcji `InnerKernel`, która dokonuje bezpośrednio mnożenia.
12. `MMult_Nx4_12`: Dodanie funkcji `PackMatrixA`, która zapakowuje po 4 komórki w kolumnie A.
13. `MMult_Nx4_13`: Uproszczenie zapisu A.
14. `MMult_Nx4_14`: Dodanie funkcji `PackMatrixB`, która zapakowuje po 4 komórki w wierszu B, zamiana inkrementacji na relatywne adresowanie w `PackMatrixA` (dodawanie do wskaźnika odpowiedniego indeksu).
15. `MMult_Nx4_15`: Użycie statycznej tablicy `packedB`, wywoływanie `PackMatrixB` tylko, gdy wiersz $i = 0$.

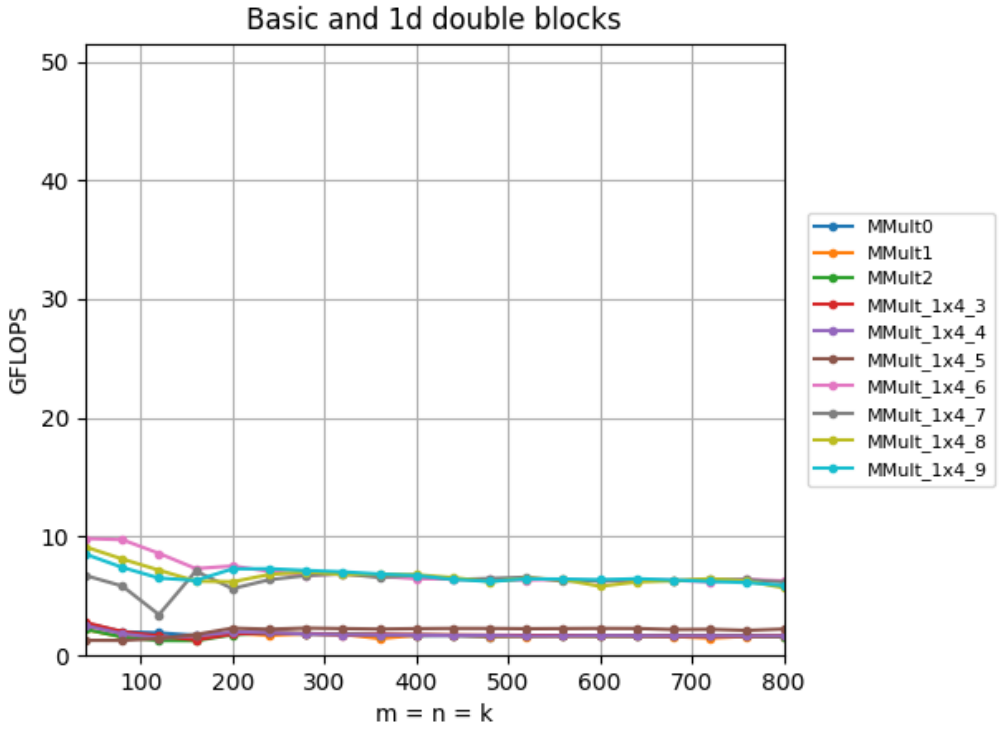
4 Wyniki



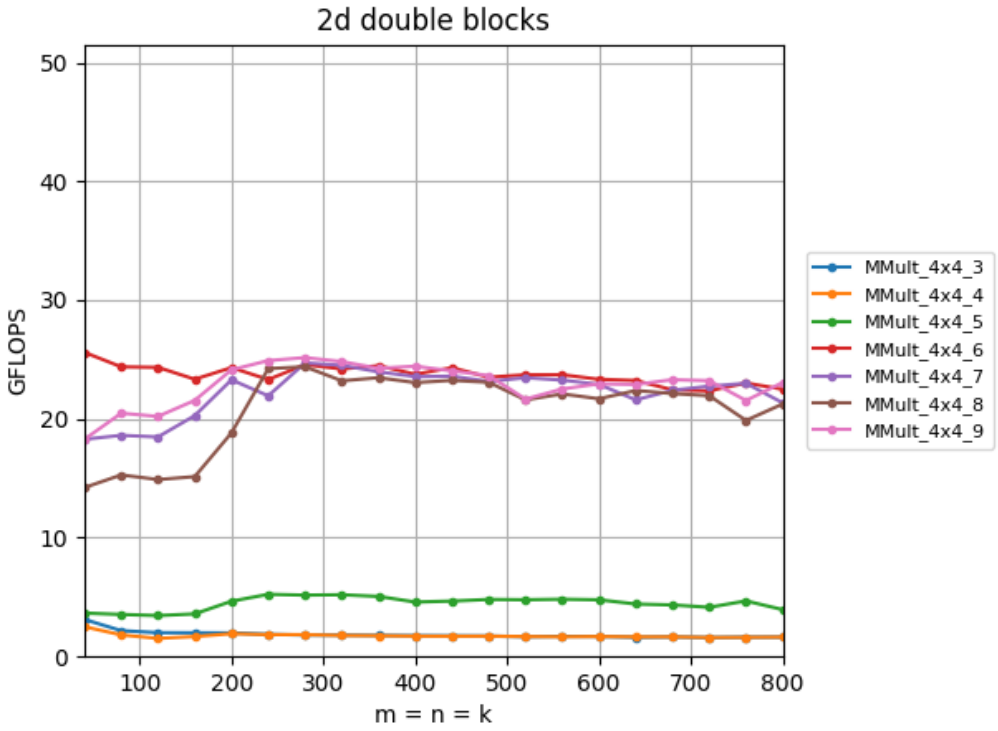
Wykres 1: Początkowa wersja, optymalizacja 1 i 2



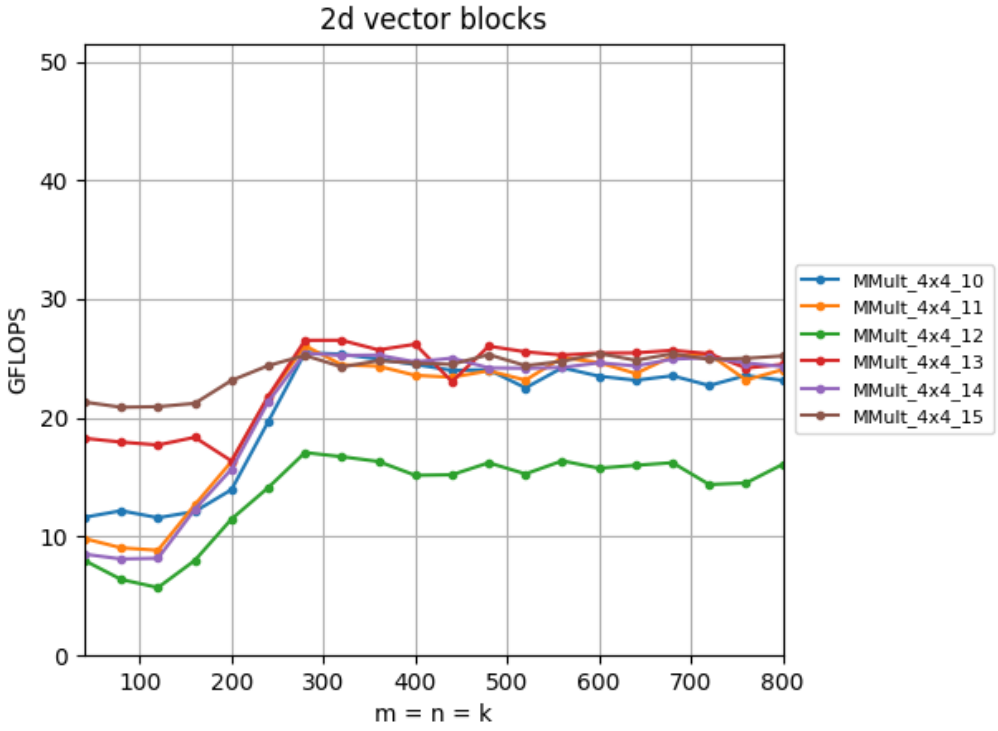
Wykres 2: Optymalizacje blokowe 3-9 (jeden wymiar)



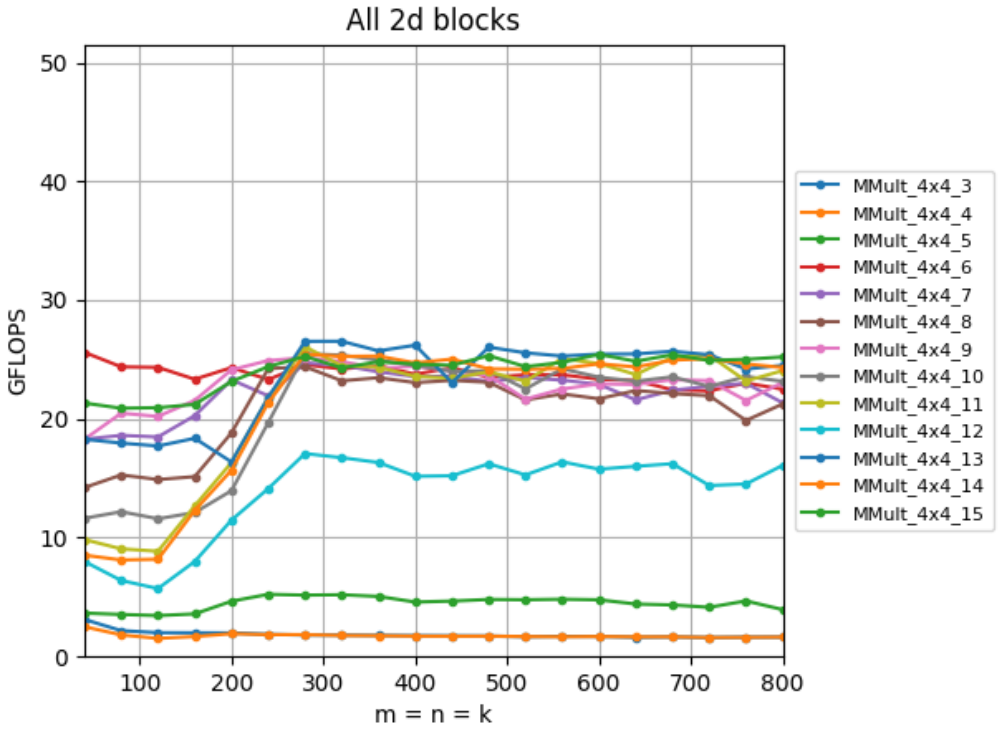
Wykres 3: Początkowa wersja, optymalizacje 1-9 (jeden wymiar)



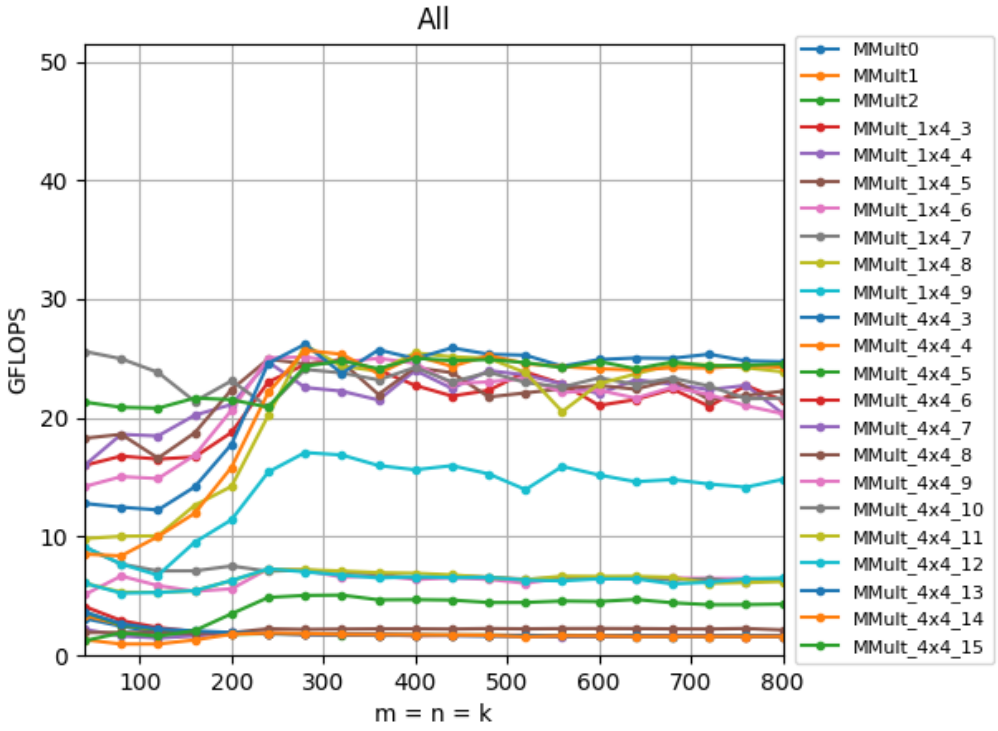
Wykres 4: Optymalizacje blokowe 3-9 (dwa wymiary)



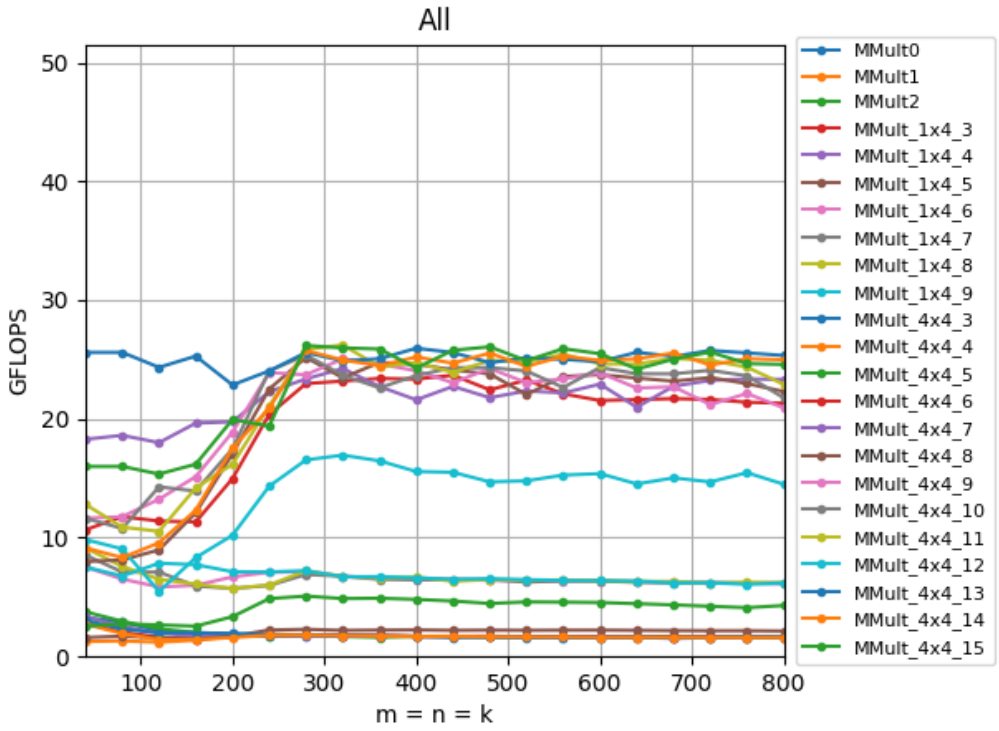
Wykres 5: Optymalizacje wektorowe 10-15



Wykres 6: Wszystkie optymalizacje blokowe (dwa wymiary)



Wykres 7: Początkowa wersja i wszystkie dokonane optymalizacje (GCC)



Wykres 8: Początkowa wersja i wszystkie dokonane optymalizacje (Clang)

5 Podsumowanie

1. Najefektywniejszym rozwiązaniem wydaje się być program `MMult_4x4_13`, jest on tylko **nieznacznie lepszy** od niektórych pozostałych rozwiązań. Osiągnął on wynik maksymalny 24.7 GFLOPS, natomiast dodając stabilność lepszym pomysłem mogłby być wykorzystanie `MMult_4x4_6`.

2. Najgorszymi rozwiązaniami okazały się pierwsze wersje programów, osiągały one wyniki rzędu 1.6 GFLOPS.

3. Operacje wektorowe zupełnie nie poprawiły wydajności. Wygląda na to, że część pozostałych rozwiązań skorzystała z mechanimzu autowektoryzacji lub odwrotnie, natomiast uważam to za mniej prawdopodobną sytuację.

4. Największy wzrost można odnotować między programem `MMult_4x4_5` a `MMult_4x6_5` - wprowadzenie zmiennych rejestrowych.