Optymalizacja Kodu na Różne Architektury:

Zadanie 1: How To Optimize Gemm

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:

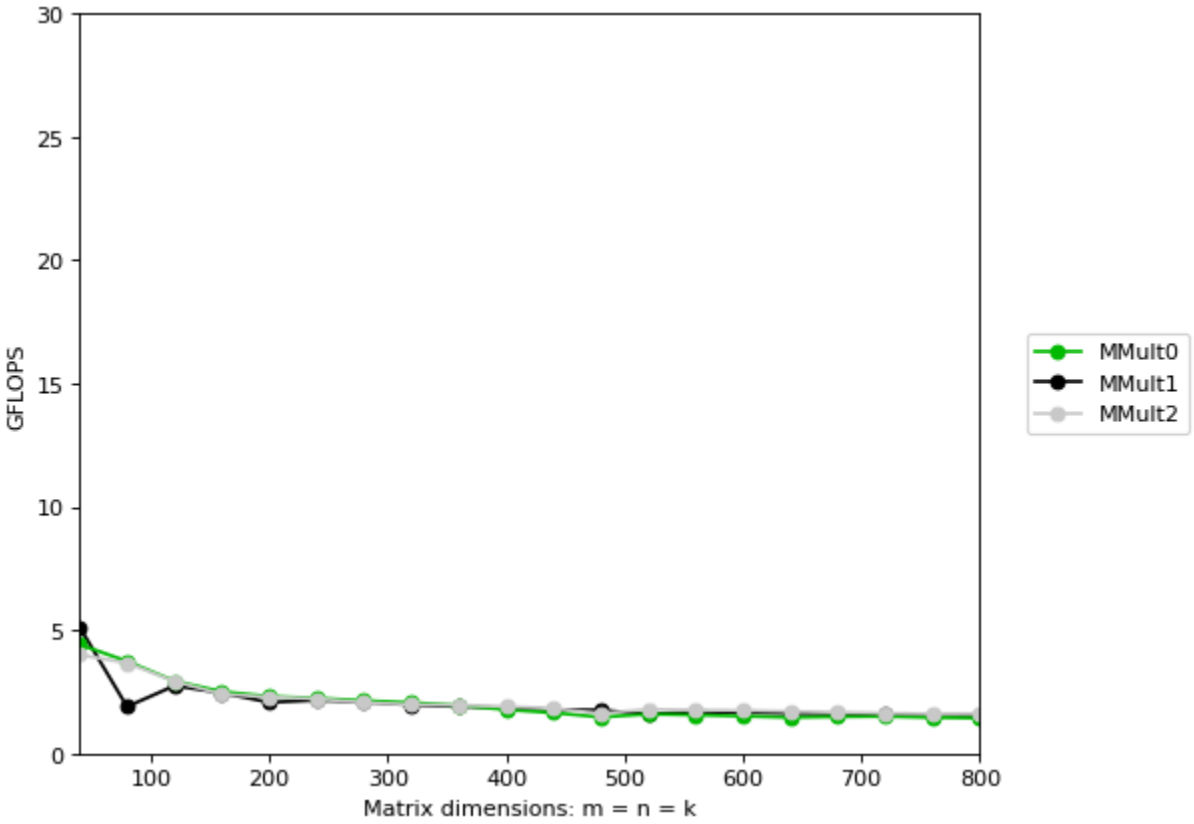
2.1. Opisane w “How To Optimize Gemm”:

1. **MMult1** - Dodanie makra X oraz funkcji AddDot
2. **MMult2** - Krok co 4 dla j, zwiększa wydajność pętli
3. **MMult_Ax4_3** - Przeniesienie wywołań AddDot do funkcji AddDotAx4, poprawia reużywalność kodu
4. **MMult_Ax4_4** - Rozwinięcie AddDot w miejscach wywoływania, redukuje narzut
5. **MMult_Ax4_5** - Jedna pętla dla rozwinięć z poprzedniej optymalizacji, stabilizuje wyniki
6. **MMult_Ax4_6** - Rejestry dla A i C, przyspieszają dostęp do pamięci, znacząco poprawiając wyniki
7. **MMult_Ax4_7** - Wskaźniki do B, upraszczają adresowanie
8. **MMult_Ax4_8**
   (a) A = 1 - Zmiana kroku pętli z optymalizacji 5. na 4
   (b) A = 4 - Rejestry dla B
9. **MMult_Ax4_9**
   (a) A = 1 - Zmiana kroku wskaźników do B na 4
   (b) A = 4 - Zmiana kolejności wykonywanych operacji (grupujemy rzędy po 2 a następnie w grupach przechodzimy kolumnami, dotychczas przechodziliśmy rzędami bez żadnego grupowania)
10. **MMult_4x4_10** - Wektory __m128d, zwiększają przepustowość podnosząc znacznie wydajność
11. **MMult_4x4_11** - Podział na bloki, dodanie funkcji InnerKernel, stabilizują wydajność
12. **MMult_4x4_12** - Dodanie funkcji PackMatrixA, zmniejsza wydajność do 12-17 GFLOPS/sec.
13. **MMult_4x4_13** - Uproszczenie odwołania do elementów z A, poprawia wydajnośc do poziomu bliskiego optymalizacji 11
14. **MMult_4x4_14** - Dodanie funkcji PackMatrixB, zmiana kroku na 4 w PackMatrixA
15. **MMult_4x4_15** - Warunkowe wykonanie PackMatrixB eliminuje niepotrzebne operacje, podwyższając wydajność programu.

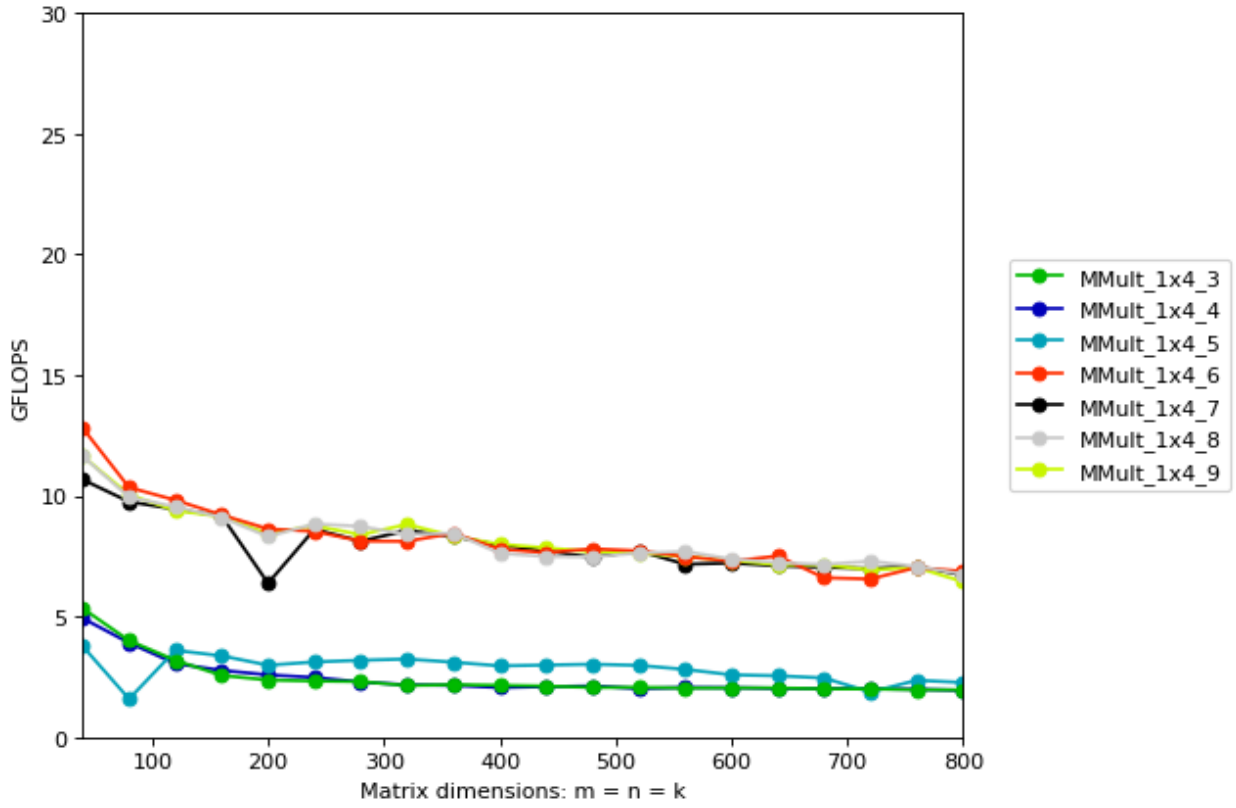
2.2. Dostosowanie do swojego procesora:

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

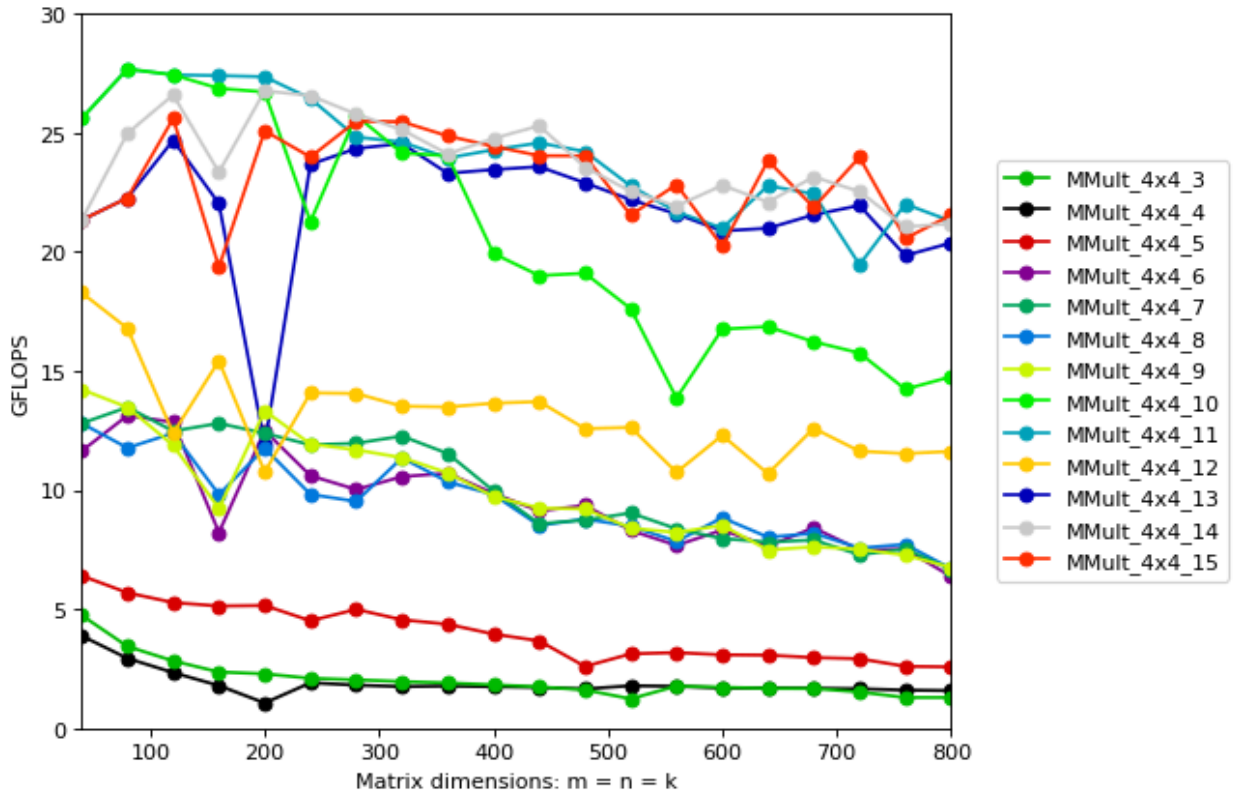
3. Wyniki:



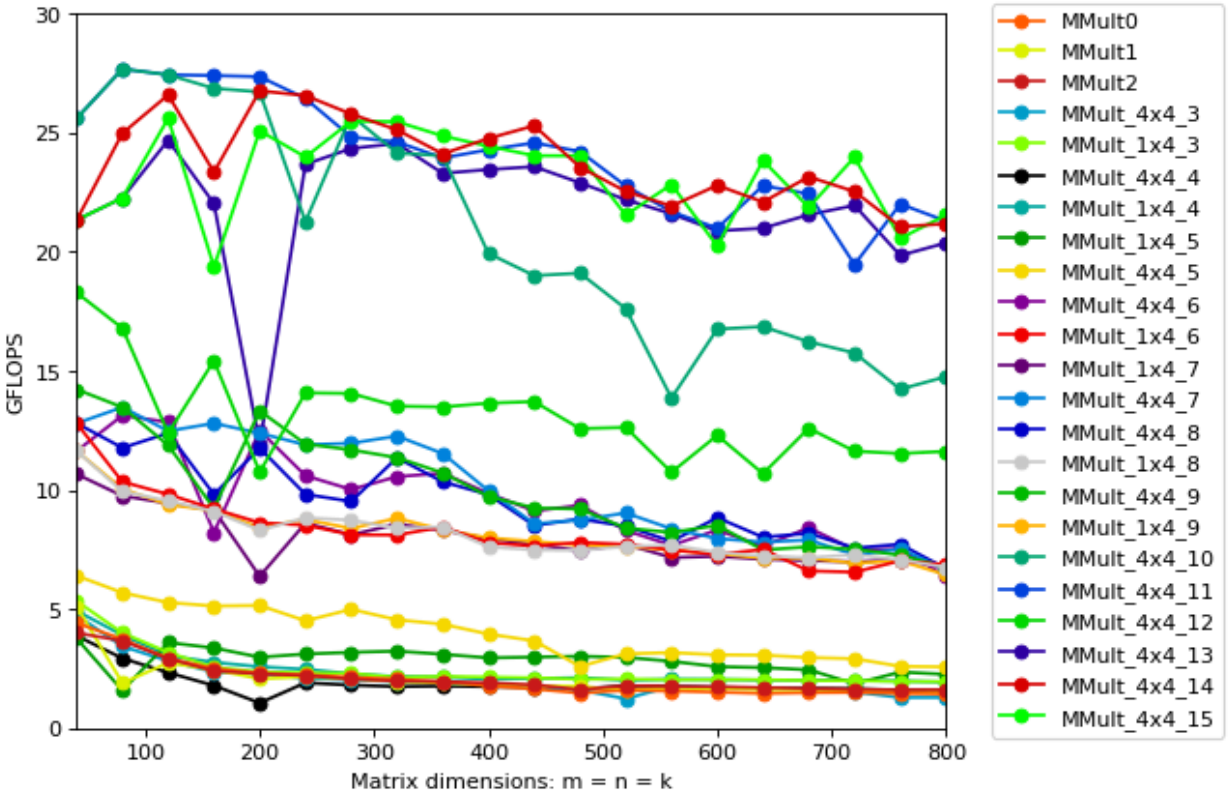
Rys.1: Wspólne optymalizacje



Rys.2: Optymalizacje 1x4



Rys.3: Optymalizacje 4x4



Rys.4: Wszystkie optymalizacje

4. Podsumowanie:

1. Najwyższy wynik 27.7 GFLOPS jest niski w porównaniu z teoretycznym 64 GFLOPS (43% maksymalnej wydajności).
2. Najwydajniejsze wersje programu:
    a. MMult_4x4_11
    b. MMult_4x4_13
    c. MMult_4x4_14
    d. MMult_4x4_15
3. Największe zyski wydajności wprowadziły:
    a. MMult_1x4_6
    b. MMult_4x4_10
    c. MMult_4x4_13
4. Największe straty wydajności wprowadziły:
    a. MMult_4x4_13 - dla n=200
    b. MMult_4x4_10 - dla dużych rozmiarów macierzy
    c. MMult_4x4_14,15 - dla n=160