Optymalizacja kodu na różne architektury

Adrian Madej 14.04.2024

1. Procesor

1.1. Sprawdzenie parametrów

Najpierw sprawdzamy model procesora, a następnie szukamy wybranych parametrów w intrenecie.

1.2. Obliczenia

Znając architekturę procesora odszukujemy FP64.

Na jej podstawie, jesteśmy w stanie obliczyć:

$$\frac{GFLOPS}{CORE} = FP64 * BASE\_CLOCK$$

1.3 Parametry

Uzyskane parametry zbieramy w tabelę

Parametr	Wartość
Producent	AMD Ryzen
Model	5 5600H
Mikroarchitektura	Cezanne-H (Zen 3)
Ilość rdzeni	6
Ilość wątków	12
Częstotliwość bazowa	3.3 GHz
Częstotliwość turbo	4.2 GHz
Cache	16 MB
FP64	16
GFLOPS/Rdzeń	52,8

Tabela 1. Parametry procesora

2. Optymalizacje

Oznaczenie Y oznacza, że bierzemy przypadek gdzie zachodzi coś dla wersji Y = 1 oraz Y = 4.

2.1 Optymalizacje z github *„How to optimize gemm”*

- MMult1 - Dodanie funkcji AddDot oraz zdefiniowanie makra X (brak zmian)
- MMult2 - Ręczne rozwinięcie pętli do 4 iteracji (brak zmian)
- MMult_Yx4_3 – Utworzenie funkcji AddDotYx4, przeniesienie do niej wywołań funkcji AddDot (brak zmian)
- MMult_Yx4_4 – Pozbycie się funkcji AddDot (brak zmian)
- MMult_Yx4_5 – Zastąpienie 4 pętli for 1 (powoli zauważalne zmiany)

  Od tego momentu zaczniemy zauważać wyraźne zmiany wydajności

- MMult_Yx4_6 – Używamy rejestrów dla C oraz A (zauważalna ogromna różnica)
- MMult_Yx4_7 – Odnosimy się do B poprzez wskaźnik (zauważalna różnica, gdyż zmniejsza to zasoby zużywane na indeksowanie danych)
- MMult_Yx4_8 –

  A = 1- Ręczne rozwinięcie pętli do 4 iteracji (lekko zmniejsza wydajność, prawdopodobnie wprowadza kompilator w błąd przez co nie stosuje on odpowiednich optymalizacji

  A = 4 – Zastosowanie rejestrów do przechowywania aktualnego rzędu z macierzy B (lekko zmniejsza wydajność)

- MMult_Ax4_9 –

  A = 1 - Zastosowanie pośredniego wskaźnika, inkrementacja go o 4 w kroku (wyniki są podobne, kompilator samemu przeprowadził tę optymalizację)

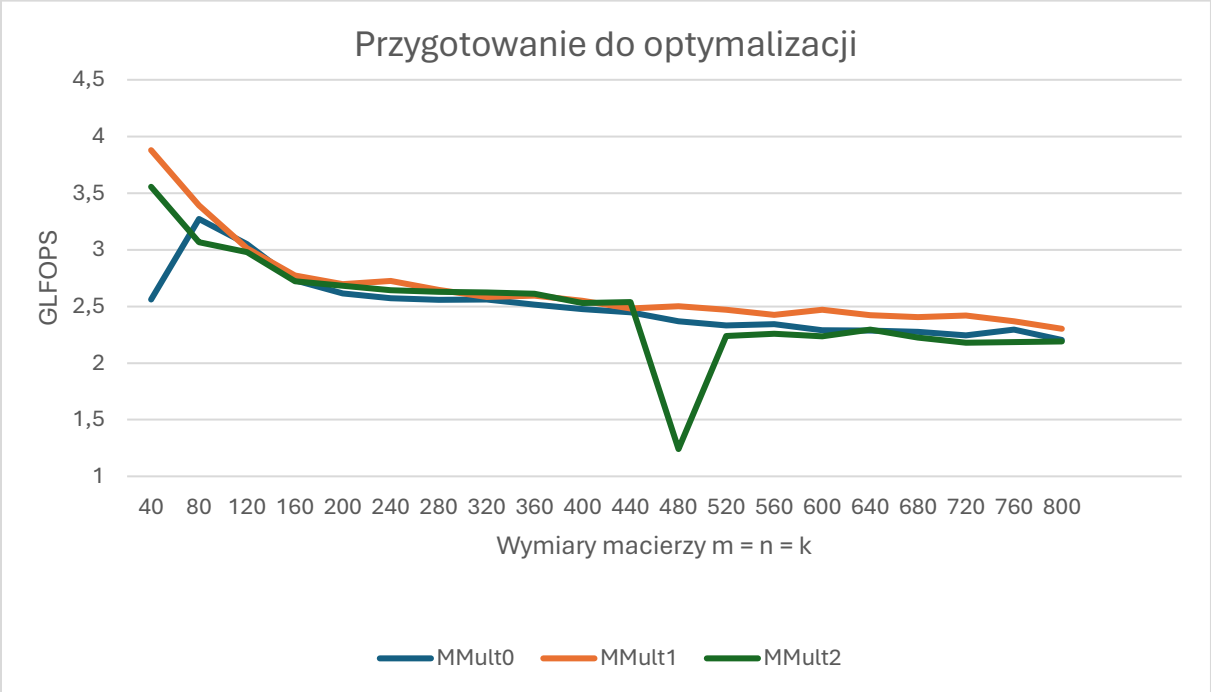
  A = 4 – Grupowanie 2 rzędów na raz, zamiast przechodzenie po nich po kolei (wyniki są podobne)

- MMult_4x4_10 – Zastosowanie wektorów __128d (znacznie przyspiesza wydajność)
- MMult_4x4_11 – Dodanie funkcji InnerKernel, blokowanie macierz, tzn. dzielenie ich na mniejsze bloki
- MMult_4x4_12 – Dodanie funkcji PackMatrixA (znaczny spadek wydajności, ponieważ A jest pakowany wielokrotnie)
- MMult_4x4_13 – Optymalizacja zapisywanych bloków, tzn. zapisywanie wcześniej spakowanych oraz ich odczytywanie jeśli istnieją (znacznie przyspiesza wydajność)
- MMult_4x4_14 – Dodanie funkcji PackMatrixB oraz modyfikacja PackMatrixA - jednorazowo aktualizujemy wskaźnik (zauważalny lekki spadek wydajności)
- MMult_4x4_15 – podobnie jak w kroku 4x4_13, warunkowo wykonujemy PackMatrixB (trochę lepsze wniki)

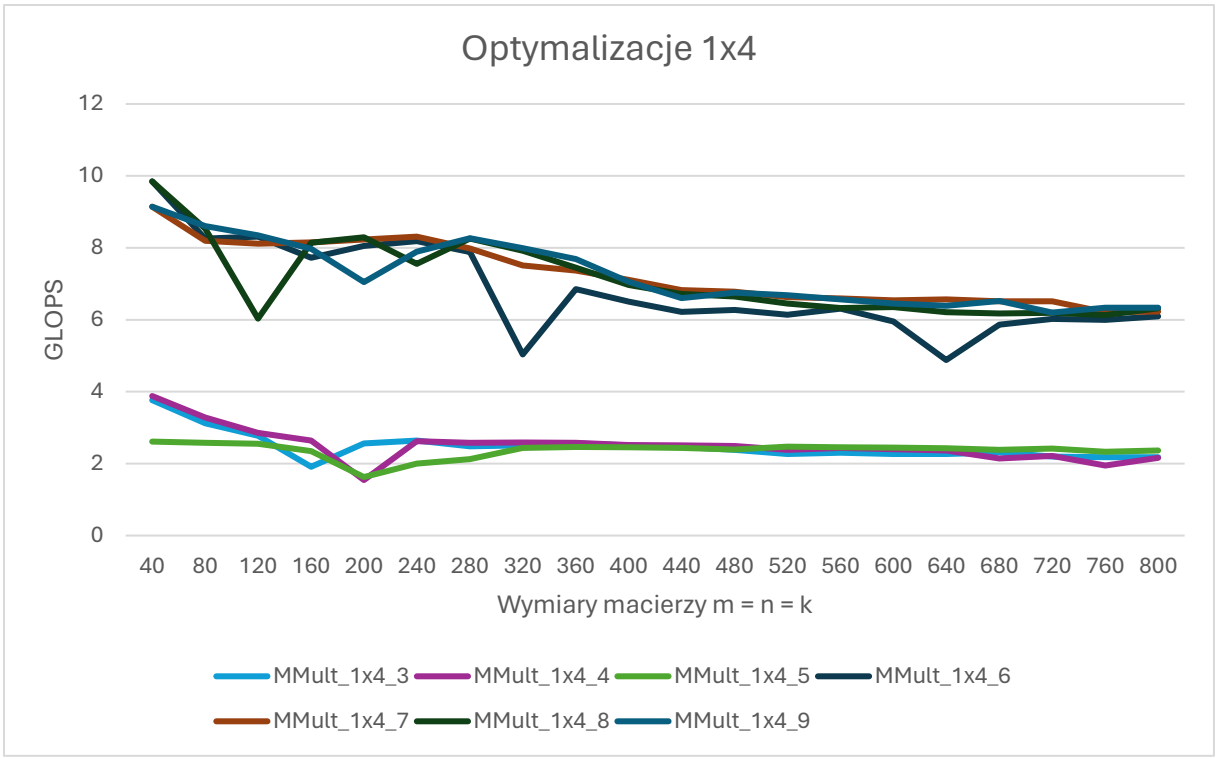
2.2 Optymalizacje dostosowane do procesora

- Uruchomienie z flagą optymalizacyjną dla mikroarchitektury procesora: -march = native (lekko zwiększa wydajność)

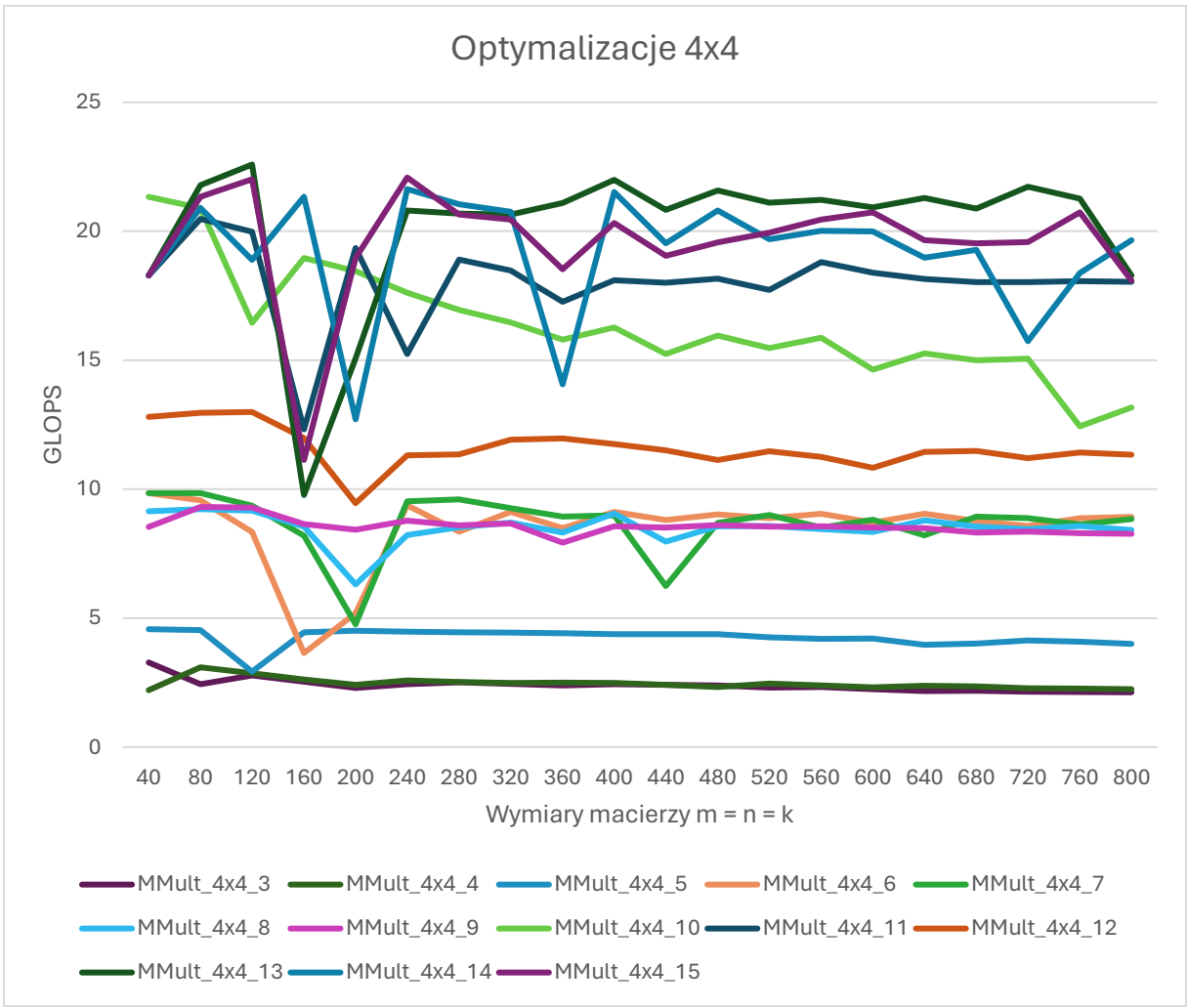
3. Wyniki



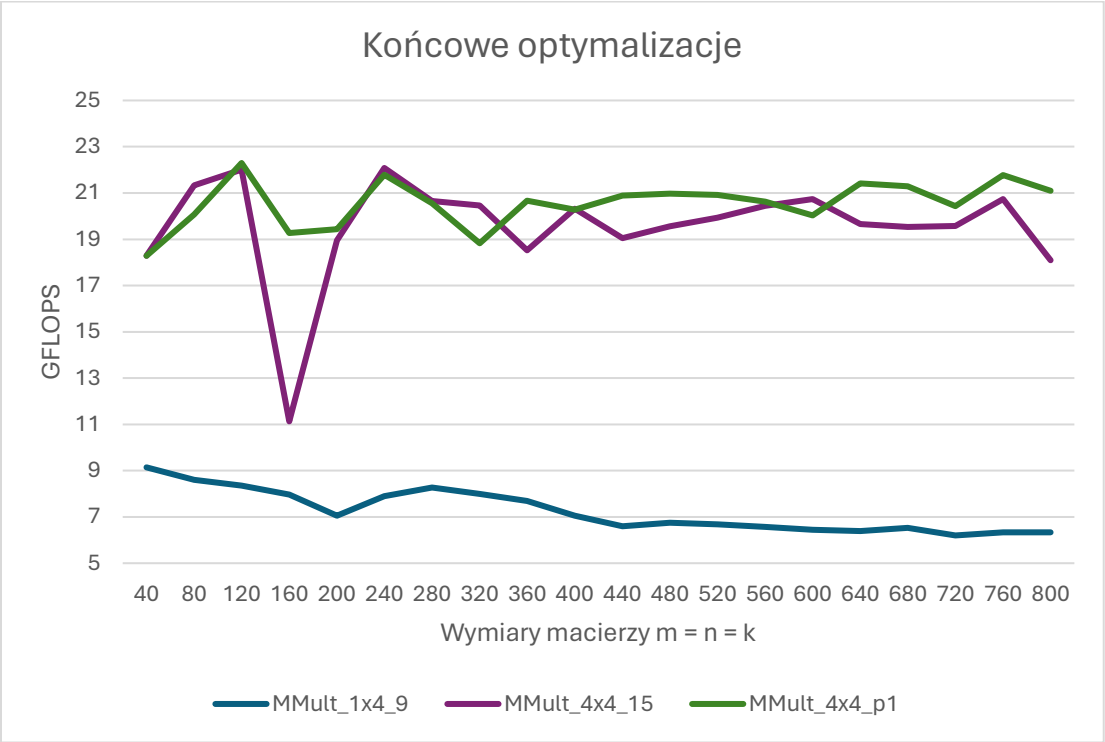
Wykres 1. Przygotowanie do optymalizacji



Wykres 2. Optymalizacje 1x4



Wykres 3. Optymalizacje 4x4



Wykres 4. Końcowe optymalizacje

4. Wnioski

4.1 Analiza wyników

- Najwyższy wynik GFLOPS uzyskany w trakcie obliczeń był równy 22.3 co stanowi 42% teoretycznej wartości; mogły mieć na to wpływ m. in. inne procesy w systemie, które ograniczyły dostępność mocy obliczeniowej procesora.

- Największe zyski wprowadziły:

  a) MMult_1x4_6
  b) MMult_4x4_10
  c) MMult_4x4_13

- Największe wahania wyników w obrębie pomiaru:

  a) MMult_4_4_6, 11, 13, 15 dla rozmiaru macierzy m = 160
  b) MMult_4_4_7 dla rozmiaru macierzy m = 200
  c) MMult_4_4_14 dla rozmiaru macierzy m = 200 i m = 360

- Największa strata wydajności nastąpiła dla wersji MMult_4x4_12 z racji tego, iż zapisywaliśmy bloki podczas każdej iteracji
- Największy zysk uzyskaliśmy dla wersji MMult_4x4_13 gdzie zaczęliśmy optymalizować zapisywanie bloków
- Najlepszą wersją okazała się wersja MMult_4x4_p1, czyli wersja MMult_4x4_15 odpalona z flagą optymalizacyjną procesora

4.2 Wnioski ogólne

- Umieszczanie zmiennych w rejestrze może znacznie przyspieszyć działanie programu
- Kompilator samemu wprowadza część optymalizacji (np. MMult_1x4_9)
- Pisząc kod trzeba uważać, aby nie wprowadzić kompilatora w błąd, przez co może on nie zastosować niektórych optymalizacji (np. MMult_1x4_8)
- Zastosowanie wektorów __128d znacznie przyspiesza wydajność (np. MMult_4x4_10)

5. Źródła

- Gitub - Home · flame/how-to-optimize-gemm Wiki (github.com)
- Specyfikacja procesora - AMD Ryzen™ 5 5600H Drivers & Support | AMD
- FLOPS - FLOPS - Wikipedia
- Co oznacza FLOPS - Gflops real world meaning | Overclockers UK Forums
- Polecenie - oknra: Zadanie domowe | UPeL (agh.edu.pl)