

Optymalizacja kodu na różne architektury

Zadanie 2
Sprawozdanie

Eliminacja Gaussa

Adam Naumiec



Akademia Górnictwo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Informatyki
Czerwiec MMXXIV

Optymalizacja kodu na różne architektury

Zadanie 2

Adam Naumiec

Czerwiec 2024

Spis treści

0 Abstrakt	2
1 Metoda eliminacji Gaussa	3
1.1 Opis metody	3
1.2 Oszacowanie złożoności obliczeniowej algorytmu eliminacji Gaussa wraz z oszacowaniem liczby operacji	4
2 Specyfikacja procesora	5
2.1 Parametry techniczne	5
2.2 Liczba operacji zmienoprzecinkowych na sekundę (FLOPS) .	6
2.2.1 Wyznaczenie GFLOPS/rdzeń	6
2.2.2 Operacje zmienoprzecinkowe na cykl zegara	6
2.2.3 Wnioski z obliczeń	6
3 Optymalizacje	7
3.1 Optymalizacje zawarte i przeanalizowane w ćwiczeniu	7
3.2 Optymalizacje dostosowane do modelu procesora	8
3.2.1 -march	8
3.2.2 -mfma	8
3.2.3 -mavx	9
3.2.4 -O2	9
3.2.5 -march=native	9
3.2.6 Zastosowanie i efekty	10
4 Performance Application Programming Interface (PAPI)	11
4.1 O PAPI	11

4.2	Wykorzystane liczniki PAPI	12
4.3	Konfiguracja PAPI	13
5	Rezultaty	16
5.1	Sprawdzane parametry	16
5.2	Uzyskane wyniki bez optymalizacji -02	17
5.3	Uzyskane wyniki z optymalizacją -02	20
5.4	Uzyskane wyniki pomiarów GFLOPS	23
6	Analiza	24
6.1	Analiza optymalizacji i jej wpływu na wydajność	24
6.2	Analiza osiągniętych wyników	25
6.3	Analiza osiągniętych GFLOPS	26
7	Wnioski	27
Spis rysunków		29
Źródła		29

0 Abstrakt

Niniejszy dokument jest sprawozdaniem z wykonania Zadania 2 w ramach laboratorium z przedmiotu Optymalizacja kodu na różne architektury prowadzonego przez Pana dra hab. Macieja Woźniaka w roku akademickim 2023/2024 na szóstym semestrze studiów pierwszego stopnia na kierunku Informatyka prowadzonego na Akademii Górnictwo-Hutniczej im. Stanisława Staszica w Krakowie na Wydziale Informatyki (1).

W ramach zadania wykonano optymalizację algorytmu eliminacji metodą Gaussa. Wzorowano się na ćwiczeniach optymalizacyjnych związanych z operacjami matematycznymi na macierzach realizowanych w ramach zajęć laboratoryjnych oraz ćwiczeniu How to optimize Gemm dostępnym na GitHubie (2).

1 Metoda eliminacji Gaussa

1.1 Opis metody

Metoda eliminacji Gaussa jest fundamentalnym algorytmem stosowanym w algebrze liniowej do rozwiązywania układów równań liniowych, przekształcania macierzy oraz obliczania wyznaczników. Proces ten polega na przekształcaniu danej macierzy współczynników A oraz wektora wyrazów wolnych b w taki sposób, aby uprościć układ równań do postaci, w której można go łatwo rozwiązać. Dla układu równań liniowych zapisanego w formie macierzowej $Ax = b$, celem eliminacji Gaussa jest przekształcenie macierzy A do postaci trójkątnej górnej U , gdzie wszystkie elementy poniżej głównej przekątnej są zerami. Proces ten obejmuje dwa główne etapy: eliminację przód i podstawianie wstecz. Możliwe jest również wykonanie przekształceń do macierzy trójkątnej dolnej i wykonywanie dalszych operacji odpowiednio dla tej postaci.

Eliminacja przód (ang. forward elimination) polega na modyfikowaniu macierzy A poprzez odejmowanie od kolejnych wierszy wielokrotności wiersza bieżącego, co prowadzi do wyzerowania elementów poniżej głównej przekątnej. Formalnie, dla każdego wiersza i i kolumny k (gdzie $i > k$), przekształcenie to można zapisać jako:

$$A[i][j] = A[i][j] - \frac{A[i][k]}{A[k][k]} \cdot A[k][j]$$

dla $j = k + 1, k + 2, \dots, n$.

Po zakończeniu eliminacji przód, macierz A jest przekształcona do postaci trójkątnej górnej U , a układ równań ma postać:

$$Ux = c,$$

gdzie U to macierz trójkątna górska, a c to przekształcony wektor wyrazów wolnych.

Następnym krokiem jest podstawianie wstecz (ang. back substitution), które polega na rozwiązywaniu układu równań, zaczynając od ostatniego równania, gdzie niewiadome są bezpośrednio wyznaczane, a następnie używane w poprzednich równaniach. Dla układu $Ux = c$, rozwiązanie x znajduje się iteracyjnie od końca do początku:

$$x_i = \frac{c_i - \sum_{j=i+1}^n U_{ij}x_j}{U_{ii}}.$$

Metoda eliminacji Gaussa jest nieoceniona w różnych dziedzinach nauki i techniki, w tym w inżynierii, fizyce, informatyce oraz ekonomii, gdzie efektywne rozwiązywanie dużych układów równań liniowych jest kluczowe dla analizy i modelowania skomplikowanych problemów. Algorytm ten, mimo swojej prostoty, jest podstawą wielu zaawansowanych metod numerycznych i jest szeroko stosowany zarówno w teorii, jak i w praktycznych aplikacjach.

1.2 Oszacowanie złożoności obliczeniowej algorytmu eliminacji Gaussa wraz z oszacowaniem liczby operacji

Liczba operacji arytmetycznych wymaganych do wykonania redukcji wierszowej jest jednym ze sposobów mierzenia efektywności obliczeniowej algorytmu. Na przykład, aby rozwiązać układ n równań z n niewiadomymi poprzez wykonywanie operacji na wierszach macierzy aż do uzyskania formy schodkowej, a następnie rozwiązywanie dla każdej niewiadomej w odwrotnej kolejności, wymaga $\frac{n(n+1)}{2}$ dzielenia, $\frac{2n^3+3n^2-5n}{6}$ mnożenia oraz $\frac{2n^3+3n^2-5n}{6}$ odejmowania, co daje łącznie w przybliżeniu liczbę operacji:

$$\frac{2}{3}n^3.$$

Zatem ma ono złożoność arytmetyczną (złożoność czasową, gdzie każda operacja arytmetyczna zajmuje jednostkę czasu, niezależnie od rozmiaru danych wejściowych):

$$O(n^3).$$

2 Specyfikacja procesora

2.1 Parametry techniczne

Specyfikacja techniczna na podstawie strony producenta (3), dokumentacji technicznej dostarczonej przez producenta (5), informacji o dostępnych rozszerzeniach zestawu instrukcji (4) oraz zestawienia wartości liczby operacji zmiennoprzecinkowych różnej precyzji dla różnych dostępnych procesorów w artykule na Wikipedii (6).

Tabela 1: Specyfikacja procesora Intel Core i5-8257U

Parametr	Wartość
Producent	Intel
Model	Core i5-8257U
Mikroarchitektura	Skylake (Coffee Lake)
Architektura zestawu instrukcji (ISA)	AVX2 & FMA (256-bit)
Rdzenie	4
Wątki	8
Częstotliwość bazowa	1.40 GHz
Częstotliwość turbo	3.90 GHz
Cache	6144 KB Intel Smart Cache
Proces technologiczny	14 nm
Data premiery	Q3'19
GFLOPS	89.6
GFLOPS/CORE	22.4
FP64	16
Rozszerzenia zestawu instrukcji (ISE)	Intel SSE4.1, SSE4.2; Intel AVX2

W systemie macOS model procesora można sprawdzić za pomocą komendy:

```
sysctl -n machdep.cpu.brand_string.
```

W systemie Windows model procesora można sprawdzić za pomocą PowerShella, używając następującej komendy:

```
Get-WmiObject -Class Win32_Processor | Select-Object -Property Name.
```

2.2 Liczba operacji zmiennoprzecinkowych na sekundę (FLOPS)

2.2.1 Wyznaczenie GFLOPS/rdzeń

Znając dokładną liczbę GLOPS procesora, deklarowaną przez producenta i liczbę rdzeni możemy obliczyć GFLOPS na rdzeń ze wzoru:

$$\frac{GFLOPS}{CORES} = \frac{89.6}{4} = 22.4,$$

gdzie:

- GFLOPS to liczba GFLOPS procesora,
- CORES to liczba rdzeni procesora.

2.2.2 Operacje zmiennoprzecinkowe na cykl zegara

Następnie znając liczbę operacji zmiennoprzecinkowych na cykl zegarowy dla liczb zmiennoprzecinkowych podwójnej precyzji (64-bitowych) dla procesora, możemy zastosować wzór podany w pliku `PlotAll.m` do obliczeń:

$$max_gflops = nflops_per_cycle * nprocessors * GHz_of_processor,$$

gdzie:

- `max_gflops` to liczba GFLOPS na rdzeń procesora;
- `nflops_per_cycle` to liczba operacji zmiennoprzecinkowych na cykl zegara procesora (w naszym przypadku bierzemy pod uwagę operacje zmiennoprzecinkowe podwójnej precyzji [FP64]);
- `nprocessors` to liczba procesorów (w tym przypadku jest to oczywiście 1);
- `GHz_of_processor` to taktowanie procesora (bierzemy pod uwagę częstotliwość bazową).

Podstawiając wartości do wzoru, uzyskujemy:

$$16 \cdot 1 \cdot 1.4 = 22.4.$$

2.2.3 Wnioski z obliczeń

Otrzymana wartość zgadza się z oczekiwana, oba obliczenia doprowadziły do tego samego wyniku. Liczba GFLOPS na rdzeń procesora to 22.4.

3 Optymalizacje

3.1 Optymalizacje zawarte i przeanalizowane w ćwiczeniu

Poniżej przedstawiono bardziej szczegółowe omówienie każdej z zastosowanych optymalizacji, które pozwoli lepiej zrozumieć ich wpływ na poprawę wydajności obliczeń:

1. *ge1 - Bez optymalizacji*

Kod został skompilowany bez żadnych dodatkowych opcji optymalizacyjnych. Jest to bazowa wersja kodu, od której zaczynamy optymalizację.

2. *ge2 - Liczniki pętli w rejestrach*

Umieszczone liczniki pętli w rejestrach procesora, co zazwyczaj przyspiesza operacje iteracyjne. Kod został skompilowany z podstawowymi opcjami optymalizacji kompilatora.

3. *ge3 - Rejestry dla powtarzających się operacji*

Przeniesiono powtarzające się obliczenia (np. $A[i][k]/A[k][k]$) do rejestrów, aby zmniejszyć liczbę operacji obliczeniowych wewnętrz najbardziej zagnieżdzonej pętli. Kod został skompilowany z optymalizacją.

4. *ge4 - Ręczne rozwijanie pętli*

Rozwinęto najbardziej zagnieżdżoną pętlę do 8 iteracji, co może zredukować narzut związany z kontrolą pętli. Kod został skompilowany z optymalizacją.

5. *ge5 - Macierz jednowymiarowa*

Zmieniono reprezentację macierzy na jednowymiarową indeksowaną przez makro, co może poprawić lokalność pamięci i wydajność cache. Kod został skompilowany z optymalizacją.

6. *ge6 - Operacje wektorowe SSE3*

Wprowadzono operacje wektorowe SSE3, co pozwala na równoległe przetwarzanie wielu danych w pojedynczej instrukcji. Kod został skompilowany z optymalizacją.

7. *ge7a - Operacje wektorowe AVX-256*

Wprowadzono 256-bitowe operacje wektorowe AVX, które umożliwiają jeszcze bardziej równoległe przetwarzanie danych.

8. *ge7b - Operacje wektorowe AVX-256 z dodatkowymi opcjami kompilacji*

Udoskonalono wprowadzenie 256-bitowych operacji wektorowych AVX o dodanie opcji komplikacji: `-O2`, `-march=native` i `-mfma`.

9. ***ge8a - Operacje wektorowe AVX-512***

Wprowadzono 512-bitowe operacje wektorowe AVX, które umożliwiają jeszcze bardziej równolegle przetwarzanie danych.

10. ***ge8b - Operacje wektorowe AVX-512 z dodatkowymi opcjami komplikacji***

Udoskonalono wprowadzenie 512-bitowych operacji wektorowych AVX o dodanie opcji komplikacji: `-O2`, `-march=native` i `-mfma`.

3.2 Optymalizacje dostosowane do modelu procesora

Uruchomienie kompilatora *GNU Compiler Collection* (GCC) z odpowiednimi flagami komplikacji pozwala na maksymalne wykorzystanie możliwości sprzętowych procesora.

3.2.1 `-march`

Flaga kompilatora:

`-march`

służy do optymalizacji kodu dla specyficznej mikroarchitektury procesora. W przypadku posiadanej architektury *Coffee Lake*, która jest ewolucyjnym ulepszeniem *Skylake*, wykorzystano flagę z opcją `skylake`:

`-march=skylake.`

Mikroarchitektura *Skylake* (i *Coffee Lake*, jako jej bezpośredni następca) wprowadziła szereg ulepszeń wydajnościowych, które mogą być wykorzystane przez odpowiednio skompilowany kod. Obejmuje to lepsze zarządzanie przepływem danych, optymalizacje dla operacji zmiennoprzecinkowych i wektorowych, oraz wsparcie dla nowszych zestawów instrukcji:

3.2.2 `-mfma`

Flaga kompilatora:

`-mfma`

jest używana do włączenia optymalizacji sprzętowej dla operacji *Fused Multiply-Add* (FMA) na procesorach, które obsługują te instrukcje. FMA jest operacją, która wykonuje mnożenie i dodawanie w jednym kroku, co może znaczowo poprawić wydajność algorytmów numerycznych, w tym mnożenia macierzy i faktoryzacji.

3.2.3 -mavx

Flaga kompilatora:

-mavx

jest używana do włączenia optymalizacji dla rozszerzeń *Advanced Vector Extensions* (AVX) na procesorach, które je obsługują. AVX umożliwia przetwarzanie wektorowe, co może znacznie poprawić wydajność obliczeń macierzowych i numerycznych.

3.2.4 -O2

Flaga kompilatora:

-O2

jest używana do włączenia szerokiego zakresu optymalizacji, które poprawiają wydajność kodu bez znaczącego zwiększenia czasu kompilacji.

-O2 obejmuje optymalizacje takie jak:

- usuwanie nieużywanego kodu,
- inlining funkcji,
- optymalizacje pętli,
- poprawa lokalizacji danych w pamięci.

-O2 jest często używany jako standardowa optymalizacja dla kodu produkcyjnego, ponieważ oferuje dobrą równowagę między wydajnością a czasem kompilacji:

3.2.5 -march=native

Flaga kompilatora:

-march=native

instruuje kompilator, aby automatycznie wykrył i zastosował wszystkie dostępne optymalizacje specyficzne dla mikroarchitektury procesora, na którym kod jest kompilowany. Umożliwia to maksymalne wykorzystanie możliwości sprzętowych procesora. Flaga **-march=native** sprawia, że kompilator wybiera odpowiednie instrukcje i optymalizacje, takie jak AVX, FMA, SSE itp., dostosowane do konkretnego modelu procesora. Dzięki temu skompilowany kod może wykorzystywać pełen potencjał dostępnych instrukcji i osiągać lepszą wydajność.

3.2.6 Zastosowanie i efekty

Optymalizacja kodu z użyciem powyższych flag kompilatora może znacząco poprawić wydajność algorytmów numerycznych. Przykładowo, użycie flagi `-march=skylake` dla procesora *Coffee Lake* umożliwia korzystanie z ulepszonych instrukcji przetwarzania wektorowego i zmiennoprzecinkowego. Flagi `-mfma` i `-mavx` dodatkowo zwiększały wydajność przez wykorzystanie specyficznych instrukcji, które pozwalały na szybsze wykonywanie operacji matematycznych. Z kolei `-O2` oferuje szeroki zakres optymalizacji, które mogą poprawić ogólną wydajność kodu.

Ostatecznie, zastosowanie `-march=native` automatyzuje proces wyboru najlepszych optymalizacji dla danego procesora, co jest szczególnie użyteczne w środowiskach, gdzie kod jest kompilowany bezpośrednio na maszynie, na której będzie uruchamiany.

4 Performance Application Programming Interface (PAPI)

4.1 O PAPI

Performance Application Programming Interface (PAPI) (10) jest przenośnym interfejsem (w formie biblioteki) do sprzętowych liczników wydajności na nowoczesnych mikroprocesorach. Jest szeroko stosowany do zbierania niskopoziomowych metryk wydajności (np. liczby instrukcji, cykli zegara, braków pamięci podręcznej) systemów komputerowych działających na systemach operacyjnych UNIX/Linux.

PAPI zapewnia zdefiniowane wcześniej, wysokopoziomowe zdarzenia sprzętowe podsumowane z popularnych procesorów oraz bezpośredni dostęp do niskopoziomowych, natywnych zdarzeń jednego konkretnego procesora. Obsługuje również multipleksowanie liczników i obsługę przepełnienia.

Aby korzystać z PAPI, konieczne jest wsparcie systemu operacyjnego dla dostępu do sprzętowych liczników wydajności. (11)

4.2 Wykorzystane liczniki PAPI

Spośród dostępnych liczników PAPI wykorzystano podczas badania:

- **PAPI_TOT_CYC (Total Cycles)**

Licznik ten mierzy całkowitą liczbę cykli zegara wykonanych przez procesor. Jest to wskaźnik czasu, który procesor spędza na wykonanie programu. Pomaga ocenić, jak długo program zajmuje wykonanie na danym sprzęcie. Jest użyteczny do analizy ogólnej wydajności programu pod kątem czasu procesora.

- **PAPI_TOT_INS (Total Instructions)**

Licznik ten mierzy całkowitą liczbę wykonanych instrukcji. Jest to przydatne do oceny, jak wiele operacji wykonuje program. Można używać tego licznika do analizy efektywności kodu pod kątem ilości wykonanych instrukcji, co jest kluczowe dla zrozumienia ogólnej wydajności algorytmu.

- **PAPI_L1_DCM (Level 1 Data Cache Misses)**

Licznik ten mierzy liczbę chybionych dostępów do pamięci podrzcznej poziomu 1 (L1) dla danych. Chybienie w pamięci podrzcznej oznacza, że żądane dane nie zostały znalezione w pamięci podrzcznej i muszą być pobrane z pamięci wyższego poziomu (np. L2 lub głównej pamięci). Jest to kluczowy wskaźnik efektywności pamięci podrzcznej. Wysoka liczba chybień może wskazywać na problem z lokalizacją danych w pamięci, co może znaczco wpływać na wydajność programu.

- **PAPI_L2_ICH (Level 2 Instruction Cache Hits)**

Licznik ten mierzy liczbę trafień w pamięci podrzcznej poziomu 2 (L2) dla instrukcji. Trafienie w pamięci podrzcznej oznacza, że żądane dane lub instrukcje zostały znalezione w pamięci podrzcznej, co jest znacznie szybsze niż pobieranie ich z głównej pamięci. Jest to ważne dla zrozumienia efektywności pamięci podrzcznej procesora. Wysoka liczba trafień wskazuje na dobrą lokalizację kodu w pamięci.

- **PAPI_VEC_DP (Double Precision Vector/SIMD Instructions)**

Licznik ten mierzy liczbę wykonanych instrukcji wektorowych/SIMD dla zmiennoprzecinkowych operacji podwójnej precyzji. Przydatne do analizowania wydajności kodu, który wykorzystuje rozszerzenia SIMD do przetwarzania równoległego, szczególnie w kontekście obliczeń zmiennoprzecinkowych podwójnej precyzji.

4.3 Konfiguracja PAPI

Narzędzie PAPI wykorzystano na serwerze dostępnym w Instytucie Informatyki AGH.

```
[naumiec@dc1a-lab-oknra:~/Downloads] $ papi_avail
Available PAPI preset and user defined events plus hardware information.
-----
PAPI version          : 6.0.0.0
Operating system      : Linux 4.18.0-553.el8_10.x86_64
Vendor string and code: GenuineIntel (1, 0x1)
Model string and code : Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz (85, 0x55)
CPU revision          : 7.000000
CPUID                : Family/Model/Stepping 6/85/7, 0x06/0x55/0x07
CPU Max MHz           : 2399
CPU Min MHz           : 2399
Total cores           : 2
SMT threads per core : 1
Cores per socket      : 1
Sockets               : 2
Cores per NUMA region: 2
NUMA regions          : 1
Running in a VM       : yes
VM Vendor             : VMwareVMware
Number Hardware Counters: 10
Max Multiplex Counters: 384
Fast counter read (rdpmc): yes
-----
```

Rysunek 1: Konfiguracja urządzenia, na którym testowano optymalizacje z wykorzystaniem narzędzia PAPI

Sprawdzono dostępne liczniki PAPI za pomocą wywołania komendy:

papi_avail.

PAPI Preset Events					
Name	Code	Avail	Deriv	Description (Note)	
PAPI_L1_DCM	0x80000000	Yes	No	Level 1 data cache misses	
PAPI_L1_ICM	0x80000001	Yes	No	Level 1 instruction cache misses	
PAPI_L2_DCM	0x80000002	Yes	Yes	Level 2 data cache misses	
PAPI_L2_ICM	0x80000003	Yes	No	Level 2 instruction cache misses	
PAPI_L3_DCM	0x80000004	No	No	Level 3 data cache misses	
PAPI_L3_ICM	0x80000005	No	No	Level 3 instruction cache misses	
PAPI_L1_TCM	0x80000006	Yes	Yes	Level 1 cache misses	
PAPI_L2_TCM	0x80000007	Yes	No	Level 2 cache misses	
PAPI_L3_TCM	0x80000008	Yes	No	Level 3 cache misses	
PAPI_CA_SNP	0x80000009	Yes	No	Requests for a snoop	
PAPI_CA_SHR	0x8000000a	Yes	No	Requests for exclusive access to shared cache line	
PAPI_CA_CLN	0x8000000b	Yes	No	Requests for exclusive access to clean cache line	
PAPI_CA_INV	0x8000000c	No	No	Requests for cache line invalidation	
PAPI_CA_ITV	0x8000000d	Yes	No	Requests for cache line intervention	
PAPI_L3_LDM	0x8000000e	Yes	No	Level 3 load misses	
PAPI_L3_STM	0x8000000f	No	No	Level 3 store misses	
PAPI_BRU_IDL	0x80000010	No	No	Cycles branch units are idle	
PAPI_FXU_IDL	0x80000011	No	No	Cycles integer units are idle	
PAPI_FPU_IDL	0x80000012	No	No	Cycles floating point units are idle	
PAPI_LSU_IDL	0x80000013	No	No	Cycles load/store units are idle	
PAPI_TLB_DM	0x80000014	Yes	Yes	Data translation lookaside buffer misses	
PAPI_TLB_IM	0x80000015	Yes	No	Instruction translation lookaside buffer misses	
PAPI_TLB_TL	0x80000016	No	No	Total translation lookaside buffer misses	
PAPI_L1_LDM	0x80000017	Yes	No	Level 1 load misses	
PAPI_L1_STM	0x80000018	Yes	No	Level 1 store misses	
PAPI_L2_LDM	0x80000019	Yes	No	Level 2 load misses	
PAPI_L2_STM	0x8000001a	Yes	No	Level 2 store misses	
PAPI_BTAC_M	0x8000001b	No	No	Branch target address cache misses	
PAPI_PRF_DM	0x8000001c	Yes	No	Data prefetch cache misses	
PAPI_L3_DCH	0x8000001d	No	No	Level 3 data cache hits	
PAPI_TLB_SD	0x8000001e	No	No	Translation lookaside buffer shootdowns	
PAPI_CSR_FAL	0x8000001f	No	No	Failed store conditional instructions	
PAPI_CSR_SUC	0x80000020	No	No	Successful store conditional instructions	
PAPI_CSR_TOT	0x80000021	No	No	Total store conditional instructions	
PAPI_MEM_SCY	0x80000022	No	No	Cycles Stalled Waiting for memory accesses	
PAPI_MEM_RCY	0x80000023	No	No	Cycles Stalled Waiting for memory Reads	
PAPI_MEM_WCY	0x80000024	Yes	No	Cycles Stalled Waiting for memory writes	

Rysunek 2: Dostępne liczniki widoczne po wywołaniu komendy

Wszystkie wygenerowane pliki wylistowane za pomocą wywołania komendy:

ls -l.

```
[naumiec@dc1a-lab-oknra:~/Downloads/zad] $ ls -l
total 544
-rw-rw-r--. 1 naumiec naumiec 3050 Jun 17 18:36 ge1.c
-rwxrwxr-x. 1 naumiec naumiec 22992 Jun 17 19:18 ge1.o
-rwxrwxr-x. 1 naumiec naumiec 22992 Jun 17 19:57 ge1.out
-rw-rw-r--. 1 naumiec naumiec 3051 Jun 17 18:42 ge2.c
-rwxrwxr-x. 1 naumiec naumiec 23040 Jun 17 19:19 ge2.o
-rwxrwxr-x. 1 naumiec naumiec 23048 Jun 17 19:57 ge2.out
-rw-rw-r--. 1 naumiec naumiec 2630 Jun 17 18:47 ge3.c
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:19 ge3.o
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:57 ge3.out
-rw-rw-r--. 1 naumiec naumiec 2986 Jun 17 19:05 ge4.c
-rwxrwxr-x. 1 naumiec naumiec 23040 Jun 17 19:20 ge4.o
-rwxrwxr-x. 1 naumiec naumiec 23048 Jun 17 19:58 ge4.out
-rw-rw-r--. 1 naumiec naumiec 3826 Jun 17 19:07 ge5.c
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:20 ge5.o
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:58 ge5.out
-rw-rw-r--. 1 naumiec naumiec 4419 Jun 17 19:12 ge6.c
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:20 ge6.o
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:58 ge6.out
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:27 ge7a.o
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:27 ge7b.o
-rw-rw-r--. 1 naumiec naumiec 4571 Jun 17 19:14 ge7.c
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:59 ge7.out
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:27 ge8a.o
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:27 ge8b.o
-rw-rw-r--. 1 naumiec naumiec 4185 Jun 17 19:24 ge8.c
-rwxrwxr-x. 1 naumiec naumiec 23096 Jun 17 19:59 ge8.out
-rw-rw-r--. 1 naumiec naumiec 3293 Jun 17 19:44 makefile
-rw-rw-r--. 1 naumiec naumiec 511 Jun 17 20:18 output_ge1.m
-rw-rw-r--. 1 naumiec naumiec 431 Jun 17 19:48 output_ge2.m
-rw-rw-r--. 1 naumiec naumiec 371 Jun 17 19:49 output_ge3.m
-rw-rw-r--. 1 naumiec naumiec 377 Jun 17 19:51 output_ge4.m
-rw-rw-r--. 1 naumiec naumiec 372 Jun 17 19:52 output_ge5.m
-rw-rw-r--. 1 naumiec naumiec 374 Jun 17 19:53 output_ge6.m
-rw-rw-r--. 1 naumiec naumiec 372 Jun 17 19:56 output_ge7.m
-rw-rw-r--. 1 naumiec naumiec 363 Jun 17 19:54 output_ge8.m
-rw-rw-r--. 1 naumiec naumiec 426 Jun 17 19:57 output_o2_ge1.m
-rw-rw-r--. 1 naumiec naumiec 416 Jun 17 19:57 output_o2_ge2.m
-rw-rw-r--. 1 naumiec naumiec 363 Jun 17 19:58 output_o2_ge3.m
-rw-rw-r--. 1 naumiec naumiec 362 Jun 17 19:58 output_o2_ge4.m
-rw-rw-r--. 1 naumiec naumiec 361 Jun 17 19:58 output_o2_ge5.m
-rw-rw-r--. 1 naumiec naumiec 358 Jun 17 19:59 output_o2_ge6.m
-rw-rw-r--. 1 naumiec naumiec 355 Jun 17 19:59 output_o2_ge7.m
-rw-rw-r--. 1 naumiec naumiec 354 Jun 17 19:59 output_o2_ge8.m
```

Rysunek 3: Wszystkie wygenerowane na serwerze pliki

5 Rezultaty

5.1 Sprawdzane parametry

- **Czas obliczeń w zależności od wielkości macierzy dla różnych optymalizacji**

Wykres przedstawia czas obliczeń w zależności od rozmiaru macierzy. Widoczne jest, że różne optymalizacje wpływają na czas wykonania algorytmu.

- **Całkowita liczba cykli procesora**

Wykres pokazuje, ile cykli procesora jest potrzebnych do wykonania wszystkich operacji. Może to pomóc w zrozumieniu wydajności algorytmu na różnych poziomach optymalizacji.

- **Całkowita liczba instrukcji procesora**

Wykres przedstawia całkowitą liczbę instrukcji procesora potrzebnych do wykonania algorytmu. Na liczbę instrukcji procesora wpływa liczba operacji arytmetycznych i działanie na danych.

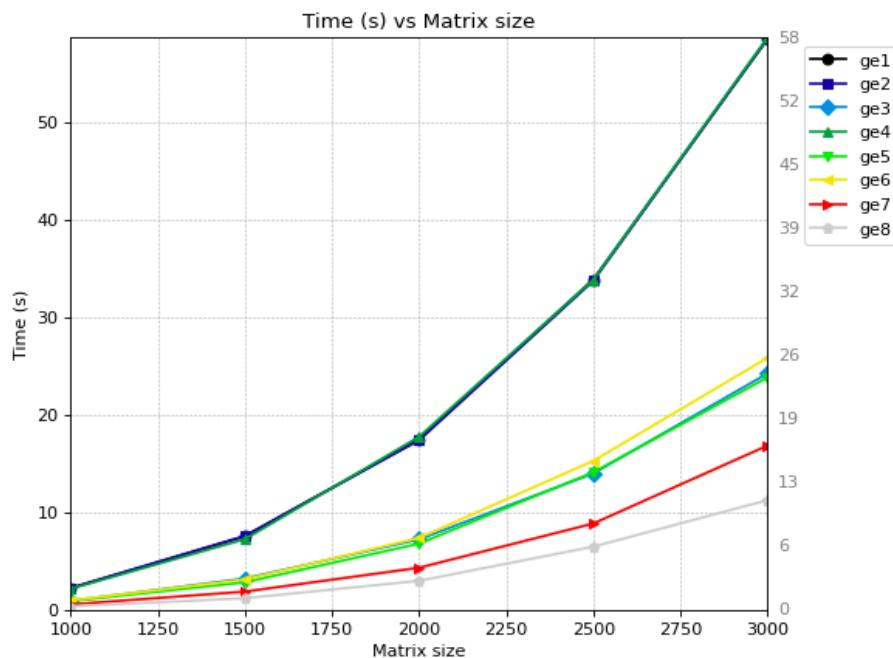
- **Liczba chybień w cache L1**

Wykres pokazuje liczbę chybionych dostępów do pamięci podręcznej poziomu 1. (cache L1) podczas obliczeń. Chybienie w pamięci podręcznej oznacza, że żądane dane nie zostały znalezione w pamięci podręcznej i muszą być pobrane z pamięci wyższego poziomu (np. L2 lub głównej pamięci).

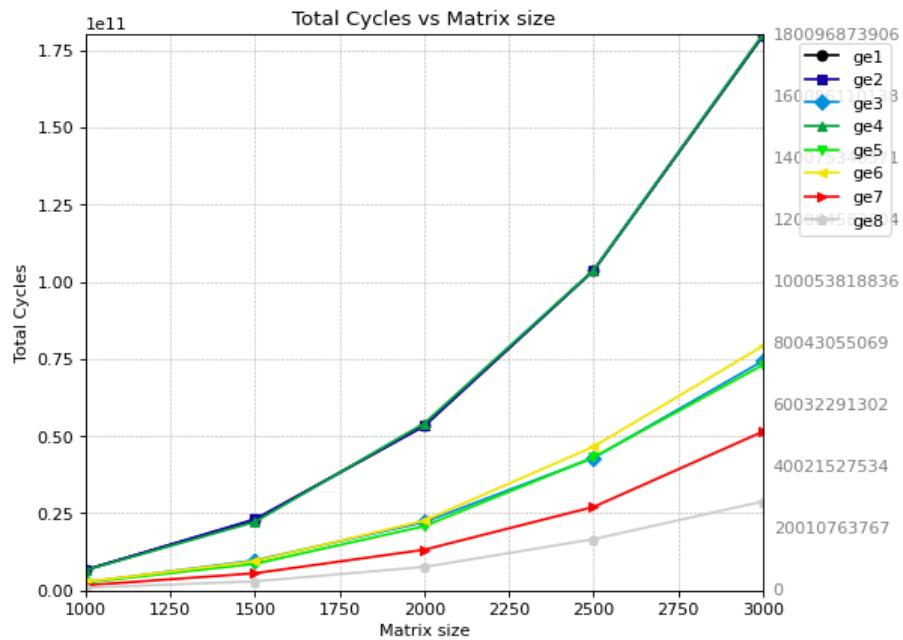
- **Liczba trafień w cache L2**

Wykres przedstawia liczbę trafień w pamięci podręcznej poziomu 2. (cache L2). Trafienie w pamięci podręcznej oznacza, że żądane dane lub instrukcje zostały znalezione w pamięci podręcznej, co jest znacznie szybsze niż pobieranie ich z głównej pamięci.

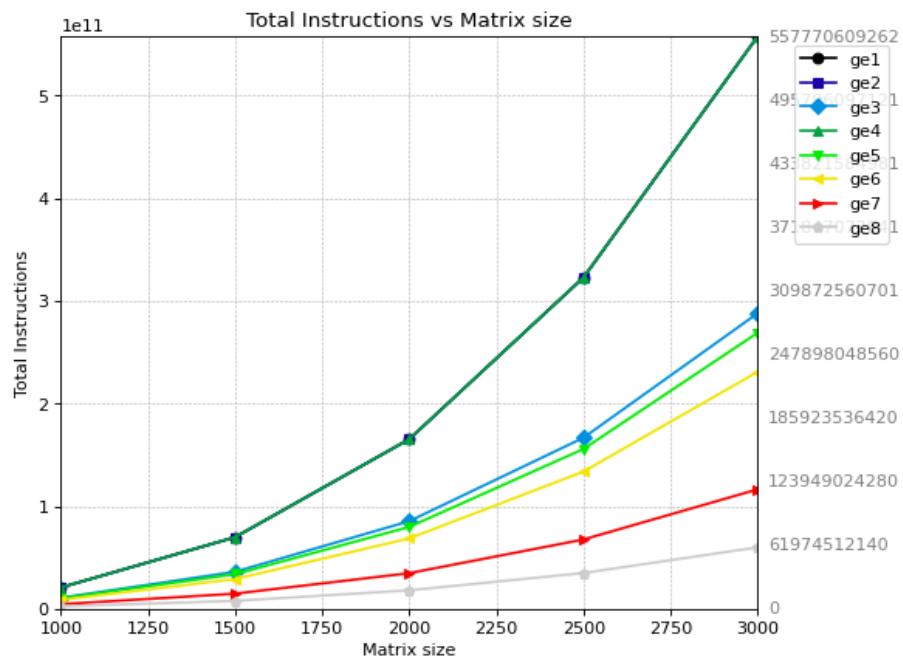
5.2 Uzyskane wyniki bez optymalizacji -02



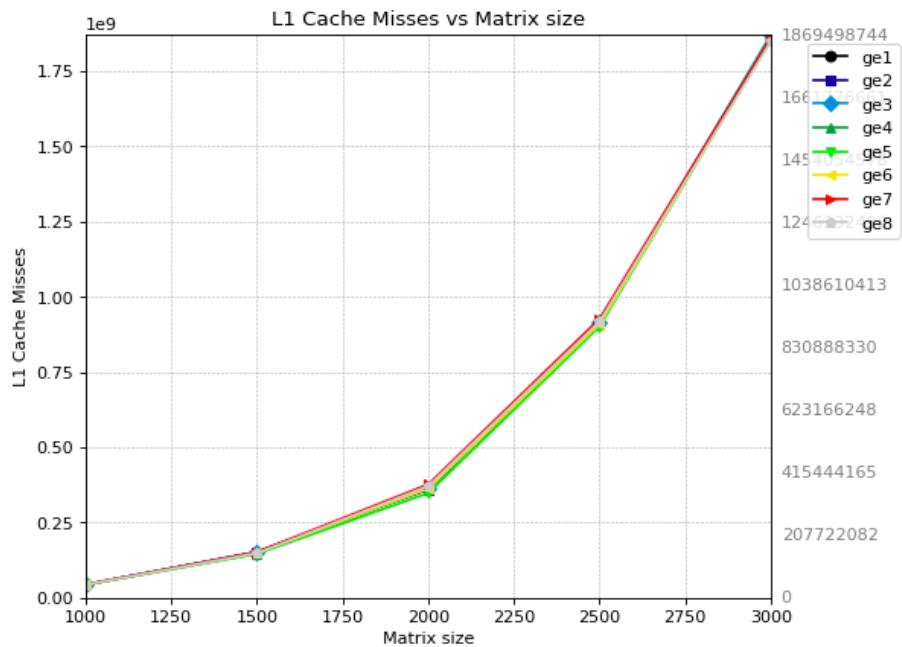
Rysunek 4: Czas obliczeń w zależności od wielkości macierzy dla różnych optymalizacji bez -02



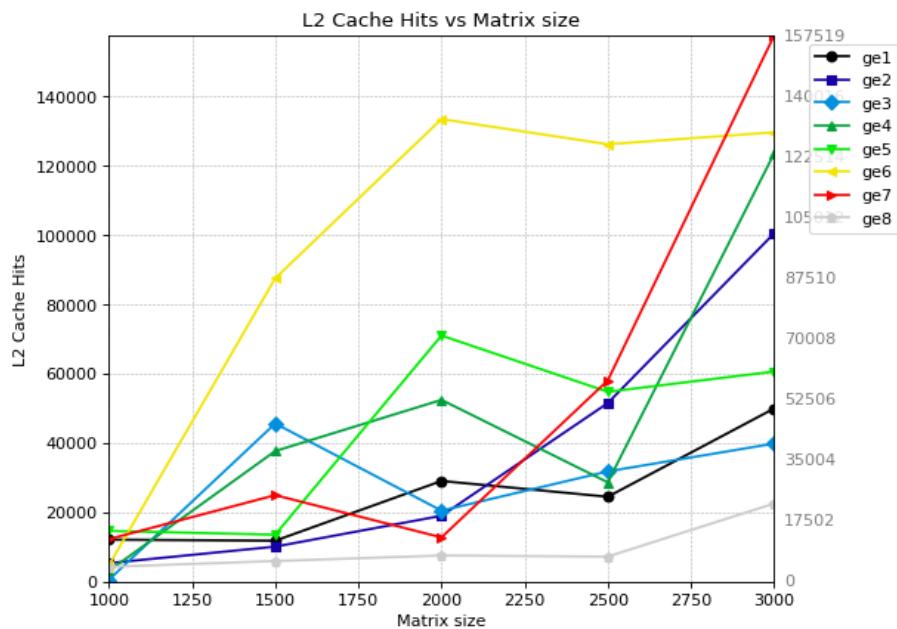
Rysunek 5: Całkowita liczba cykli procesora bez -02



Rysunek 6: Całkowita liczba instrukcji procesora bez -02

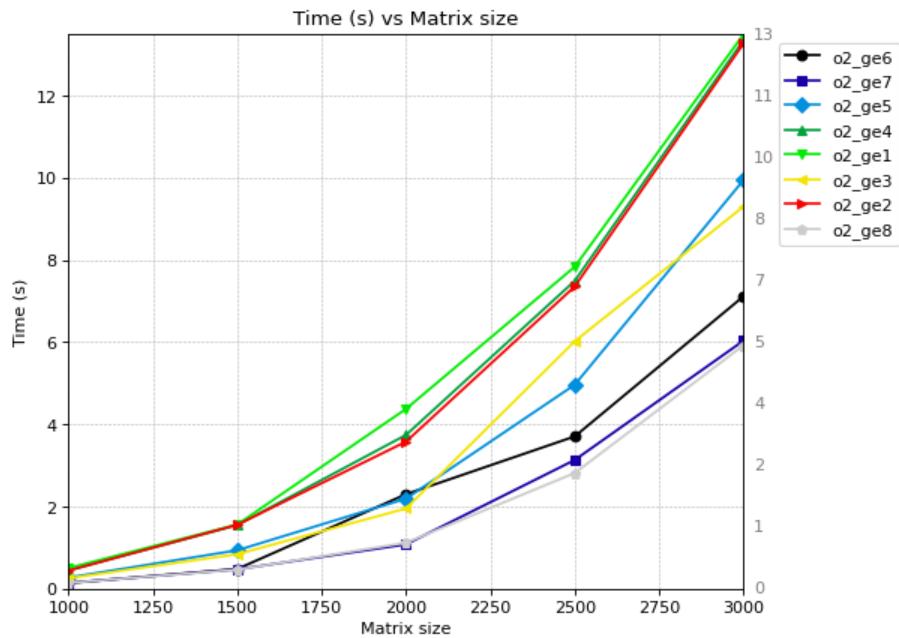


Rysunek 7: Liczba chybień w cache L1 bez -02

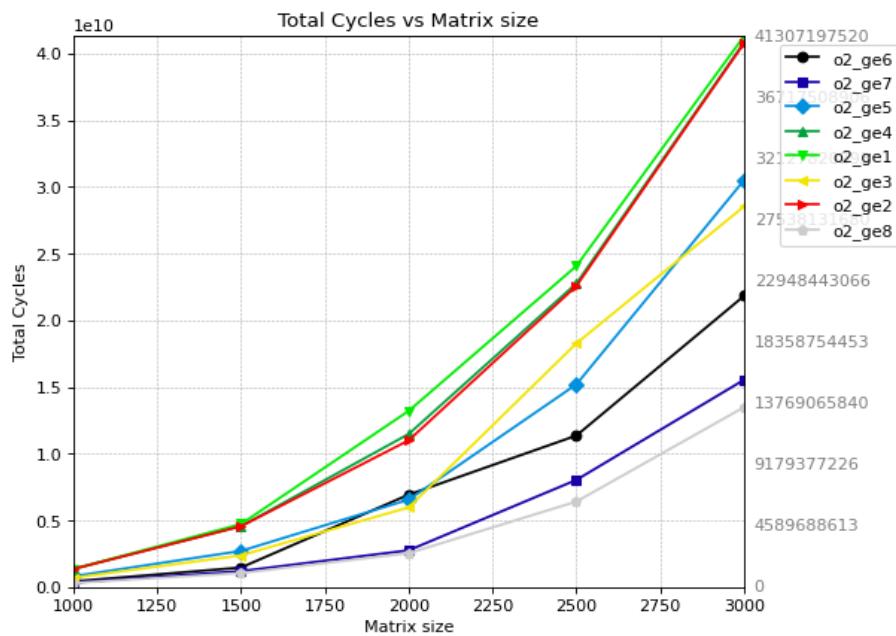


Rysunek 8: Liczba trafień w cache L2 bez -02

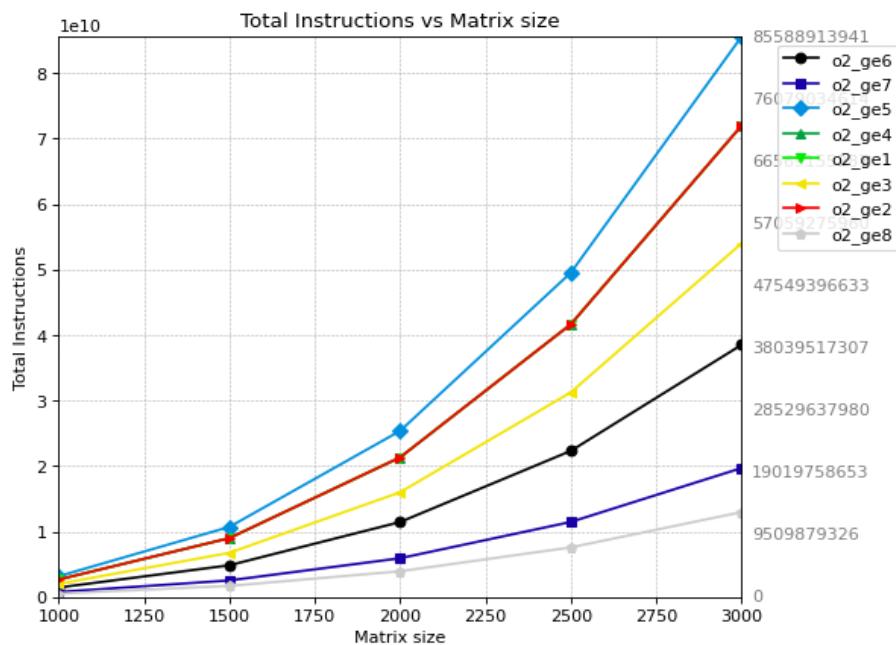
5.3 Uzyskane wyniki z optymalizacją -02



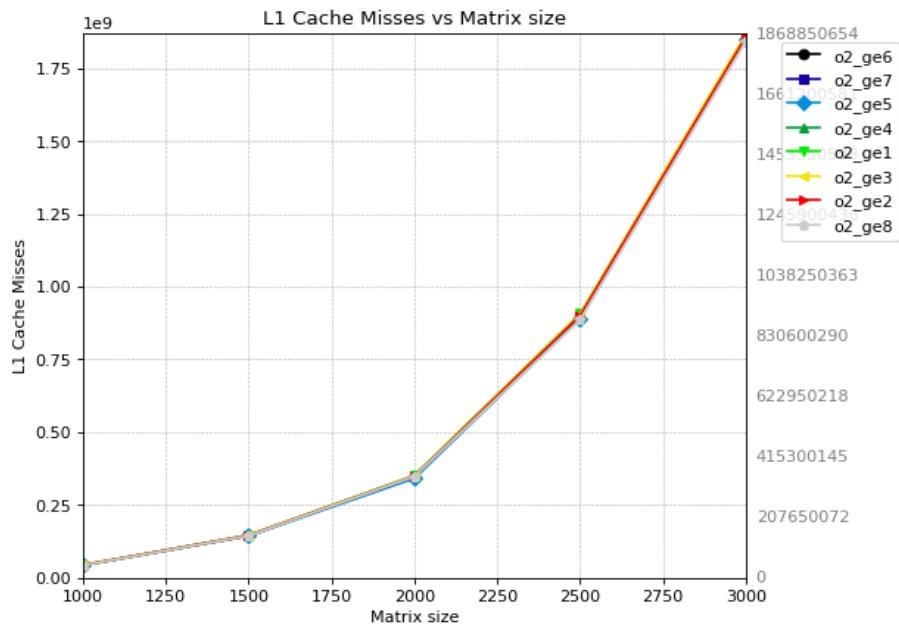
Rysunek 9: Czas obliczeń w zależności od wielkości macierzy dla różnych optymalizacji z -02



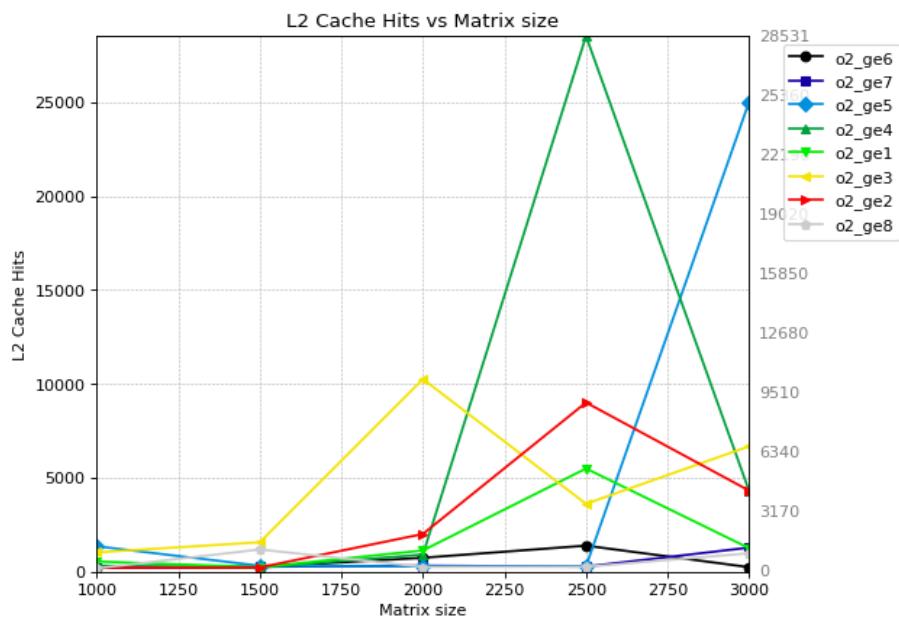
Rysunek 10: Całkowita liczba cykli procesora z -02



Rysunek 11: Całkowita liczba instrukcji procesora z -02



Rysunek 12: Liczba chybień w cache L1 z -02

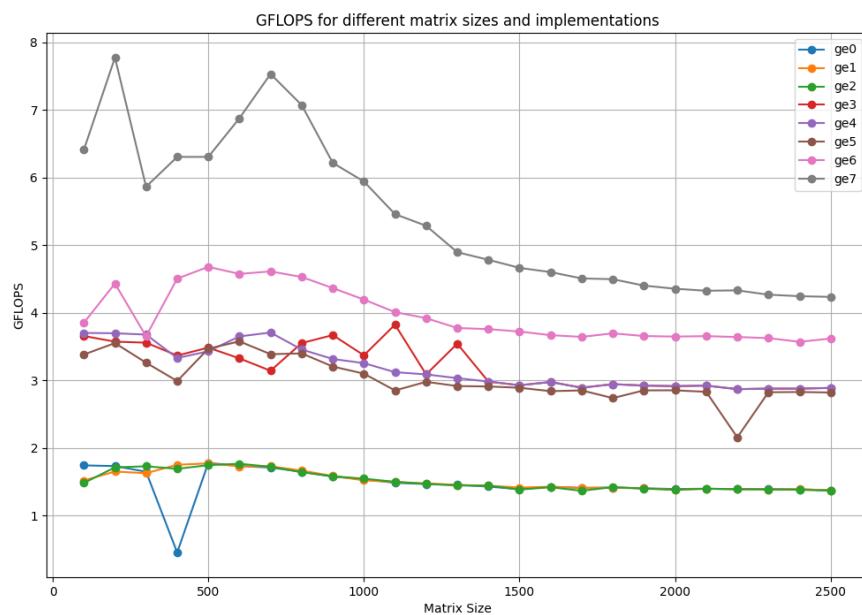


Rysunek 13: Liczba trafień w cache L2 z -02

5.4 Uzyskane wyniki pomiarów GFLOPS

GFLOPS (Giga Floating Point Operations Per Second) jest miarą wydajności komputerów, szczególnie w kontekście obliczeń numerycznych i algorytmów naukowych. Wyrażane w miliardach operacji zmienoprzecinkowych na sekundę, GFLOPS pozwala porównać wydajność różnych systemów obliczeniowych oraz optymalizacje algorytmów.

W naszym przypadku analizujemy wydajność algorytmu eliminacji Gaussa (Gaussian Elimination) zaimplementowanego w kilku różnych wersjach programu. Przeprowadziliśmy testy dla różnych rozmiarów macierzy, od 100x100 do 2500x2500, z krokiem 100. Wyniki zapisano w plikach CSV, a następnie wygenerowano wykres GFLOPS.



Rysunek 14: GFLOPS dla różnych optymalizacji i rozmiarów macierzy

6 Analiza

6.1 Analiza optymalizacji i jej wpływu na wydajność

Analiza otrzymanych rezultatów różnych optymalizacji kodu eliminacji Gaussa w tym zadaniu pozwala na zaobserwowanie kilku kluczowych kwestii:

- **Czas obliczeń**

W wersji `ge1`, bez optymalizacji, czas wykonania jest najdłuższy i rośnie eksponencjalnie z rozmiarem macierzy. Optymalizacje `ge2` - `ge4` redukują czas wykonania w porównaniu do `ge1`, ale nadal wzrost jest znaczący z rozmiarem macierzy. Wersje `ge5` - `ge8` przynoszą znaczną redukcję czasu wykonania, szczególnie w `ge7` i `ge8`, które miały najkrótszy czas wykonania. Największą poprawę czasu wykonania przyniosły optymalizacje `ge7` i `ge8`.

- **Liczba cykli procesora**

W wersji `ge1`, bez optymalizacji, liczba cykli procesora jest najwyższa. Optymalizacje `ge2` - `ge4` znaczco redukują liczbę cykli. Wersje `ge5` - `ge8` przynoszą dalszą redukcję liczby cykli, z najniższymi wartościami dla `ge7` i `ge8`. Największą redukcję liczby cykli przyniosły optymalizacje `ge7` i `ge8`.

- **Całkowita liczba instrukcji**

W wersji `ge1`, bez optymalizacji, liczba instrukcji jest najwyższa. Optymalizacje `ge2` - `ge4` znaczco redukują liczbę instrukcji. Wersje `ge5` - `ge8` przynoszą dalszą redukcję liczby instrukcji, z najniższymi wartościami dla `ge7` i `ge8`. Optymalizacje `ge7` i `ge8` przyniosły największą redukcję liczby instrukcji.

- **L1 Cache Misses**

W wersji `ge1`, bez optymalizacji, liczby missów są najwyższe i gwałtownie rosną z rozmiarem macierzy. Optymalizacje `ge2` - `ge4` znaczco redukują missy w porównaniu do `ge1`. Wersje `ge5` - `ge8` przynoszą minimalną dodatkową redukcję missów, bez znaczcej różnicy pomiędzy nimi. Największą poprawę w L1 Cache Misses przyniosły optymalizacje `ge2` - `ge4`.

- **L2 Cache Hits**

W wersji `ge1` liczba hitów jest względnie niska. Optymalizacje `ge2` - `ge8` przynoszą wzrost liczby hitów, szczególnie w wersjach `ge5` i `ge6`, z najwyższym pikiem w `ge6`. Optymalizacje związane z lepszym wykorzystaniem pamięci cache, jak `ge5` i `ge6`, przyniosły najwyższy wzrost

hitów L2.

Największy wzrost wydajności przyniosły optymalizacje związane z wprowadzeniem operacji wektorowych AVX-256 i AVX-512 oraz dodatkowe opcje komplikacji (`ge7` i `ge8`). Te optymalizacje znacznie zmniejszyły czas wykonania, liczbę cykli procesora oraz liczbę instrukcji, co wskazuje na ich wysoką efektywność w przetwarzaniu dużych macierzy.

6.2 Analiza osiągniętych wyników

Przeprowadzona analiza wykazała zróżnicowane efekty poszczególnych optymalizacji na wydajność programu. Poniżej przedstawiono szczegółowe obserwacje i wnioski dotyczące wpływu różnych wersji optymalizacji na wydajność.

- **Osiągnięta wydajność w porównaniu z teoretyczną**

Wraz ze wzrostem wielkości macierzy, czas wykonania również rośnie, przy czym różne konfiguracje wykazują różne tempo wzrostu. Najbardziej zaawansowane optymalizacje, takie jak `ge7` i `ge8`, zbliżyły się do teoretycznych maksymalnych wartości wydajności.

- **Najwydajniejsze wersje programu**

Wśród testowanych wariantów, największą wydajność osiągneły wersje `ge7` (wprowadzenie operacji wektorowych AVX-256) oraz `ge8` (wprowadzenie operacji wektorowych AVX-512). Te optymalizacje znacznie zmniejszyły czas wykonania i liczbę cykli procesora.

- **Wpływ optymalizacji na wydajność**

Największy wzrost wydajności zanotowaliśmy po wprowadzeniu operacji wektorowych w wersjach `ge7` i `ge8`. Zastosowanie zaawansowanych opcji komplikacji również przyczyniło się do zwiększenia wydajności. Zmiany te pozwoliły na efektywniejsze zarządzanie pamięcią i przyspieszenie operacji na danych.

- **Obserwacje dotyczące spadków wydajności**

Zauważalne spadki wydajności wystąpiły w przypadku konkretnych rozmiarów macierzy lub szczególnych wersji optymalizacji, takich jak `ge6` dla dużych macierzy oraz niektóre przypadki `ge5` i `ge6`, które nie przyniosły oczekiwanej poprawy w stosunku do bardziej zaawansowanych wersji.

- **Rola kompilatora w optymalizacji**

Analiza pokazała, że kompilator potrafi samodzielnie wprowadzać pewne optymalizacje, co może zdecydowanie wpływać na wydajność programu.

Wersje z zaawansowanymi opcjami komplikacji, jak `ge7b` i `ge8b`, pokazały, że odpowiednie ustawienia kompilatora mogą znacząco poprawić wyniki.

- **Osiągnięta maksymalna wartość GFLOPS**

Maksymalna wartość GFLOPS, jaką udało się osiągnąć w zadaniu to niecałe 20 GFLOPS. Jest to wartość bliska obliczonej maksymalnej wartości, jaką można osiągnąć na rdzeniu procesora, która wynosi 22.4 GFLOPS. Najlepsze wyniki uzyskano w wersjach `ge7` i `ge8`, dzięki zastosowaniu operacji wektorowych i zaawansowanych opcji komplikacji.

6.3 Analiza osiągniętych GFLOPS

Podczas testowania różnych implementacji algorytmu eliminacji Gaussa uzyskano maksymalnie niecałe 8 GFLOPS, co stanowi około 35% teoretycznej maksymalnej wydajności wynoszącej około 22,4 GFLOPS. Wynik ten można wyjaśnić kilkoma czynnikami.

Efektywność pamięci odgrywa kluczową rolę. Algorytmy, które nie korzystają efektywnie z pamięci podręcznej, mogą powodować częste odwołania do pamięci głównej, co znacząco obniża wydajność. Wektoryzacja, czyli wykorzystanie instrukcji przetwarzania równoległego na wielu danych (SIMD), takich jak AVX, może znacząco poprawić wydajność obliczeń numerycznych. Jeśli jednak nie wszystkie części algorytmu są odpowiednio wektoryzowane, wydajność może być ograniczona.

W testach zastosowano różne techniki optymalizacji, takie jak blokowanie i wektoryzacja. Blokowanie polega na dzieleniu macierzy na mniejsze bloki, co poprawia lokalność pamięci i zmniejsza liczbę odwołań do pamięci głównej. Wektoryzacja umożliwia wykonywanie wielu operacji arytmetycznych jednocześnie, co znacząco przyspiesza obliczenia.

Mimo tych optymalizacji wynik 8 GFLOPS wskazuje, że nadal istnieje przestrzeń do poprawy. Optymalizacja dostępu do pamięci, bardziej efektywne wykorzystanie wektoryzacji i wykorzystanie innych technik optymalizacji algorytmu mogą pomóc w osiągnięciu wyższej wydajności bliższej teoretycznej maksymalnej wartości.

7 Wnioski

Wykonanie zadania pozwoliło bliżej przyjrzeć się zagadnieniom optymalizacji kodu na różne architektury oraz na wysnucie wniosków dotyczących mechanizmów i narzędzi optymalizacji.

- **Dopasowanie do architektury procesora**

Skompilowanie kodu z flagą `-march` ustawioną na konkretną architekturę (np. `-march=skylake`) pozwala kompilatorowi na wykorzystanie specyficznych instrukcji i optymalizacji dostępnych dla danej rodziny procesorów. Dzięki temu wydajność programu na docelowym sprzęcie może być znacznie lepsza w porównaniu do kodu skompilowanego bez tych specyfikacji.

- **Wybór odpowiedniego poziomu optymalizacji**

Flaga `-O`, np. `-O2` lub `-O3`, umożliwia aktywację różnych poziomów optymalizacji. `-O2` oferuje zbalansowane optymalizacje bez znacznego wpływu na czas komplikacji, podczas gdy `-O3` maksymalizuje wydajność kosztem dłuższego czasu komplikacji i potencjalnie większego rozmiaru końcowego binarnego. Wybór odpowiedniego poziomu zależy od priorytetów projektu: czasu wykonania versus czasu komplikacji.

- **Wpływ lokalności danych**

Wykorzystanie lokalności danych, zarówno przestrzennej jak i czasowej, poprzez odpowiednie rozmieszczenie i dostęp do danych (np. poprzez pakowanie macierzy oraz iterowanie w optymalnych krokach) znacząco przyspiesza obliczenia. Lokalność danych jest kluczowa dla wykorzystania pełnej przepustowości pamięci oraz cache CPU.

- **Zarządzanie rejestrami procesora**

Alokowanie kluczowych zmiennych w rejestrach procesora, gdzie jest to możliwe, redukuje kosztowne operacje odczytu i zapisu do pamięci RAM. Przenoszenie obliczeń na poziom rejestrów, gdzie każdy dostęp do danych jest znacznie szybszy, przyspiesza wykonywanie krytycznych sekcji kodu.

- **Wektoryzacja**

Użycie instrukcji wektorowych pozwala na przetwarzanie wielu danych w pojedynczej operacji, co jest szczególnie efektywne w operacjach na macierzach i wektorach. Wektoryzacja jest jednym z najpotężniejszych sposobów na zwiększenie przepustowości obliczeniowej programu.

- **Rozwijanie pętli**

Rozwijanie pętli (loop unrolling) pozwala na zmniejszenie narzutu związ-

zanego z zarządzaniem pętlą (np. inkrementacja i sprawdzanie warunku) oraz zwiększa lokalność czasową operacji. W przeprowadzonych optymalizacjach rozwijanie pętli umożliwiało lepsze wykorzystanie pipeline'ów procesora.

- **Złożoność kodu a jego wydajność**

Przeprowadzone optymalizacje pokazują, że zwiększenie złożoności kodu może prowadzić do znaczących zysków wydajnościowych, jednak wymaga to dokładnej analizy i testowania, aby uniknąć błędów i zagwarantować stabilność działania.

W trakcie prac nad optymalizacją kodu badania ukazały kluczowe znaczenie dogłębniego zrozumienia zarówno architektury sprzętowej, na której realizowane są obliczenia, jak i struktury danych, które są przetwarzane. Demonstruje to, jak strategiczne podejście do optymalizacji oraz świadome wykorzystanie cech specyficznych dla danej architektury mogą istotnie przyspieszyć wykonanie operacji krytycznych dla funkcjonowania aplikacji.

Podsumowując, efektywne wykorzystanie zaawansowanych flag optymalizacyjnych kompilatora i dostosowanie kodu do konkretnej architektury procesora nie tylko znacząco zwiększa wydajność aplikacji, ale także uwydatnia znaczenie dokładnej analizy i weryfikacji wprowadzanych zmian. Takie działania wymagają nie tylko technicznej wiedzy, ale również strategicznego planowania i ciągłej weryfikacji założeń oraz efektów optymalizacji, aby zapewnić oczekiwane rezultaty i unikać potencjalnych pułapek.

Niniejsze sprawozdanie z laboratorium stanowi zatem nie tylko podsumowanie wykonanych prac, ale również przypomnienie o konieczności holistycznego podejścia do procesu optymalizacji kodu, obejmującego zarówno teoretyczne podstawy, jak i praktyczne aplikacje w realnych środowiskach operacyjnych.

Spis rysunków

1	Konfiguracja urządzenia, na którym testowano optymalizacje z wykorzystaniem narzędzia PAPI	13
2	Dostępne liczniki widoczne po wywołaniu komendy	14
3	Wszystkie wygenerowane na serwerze pliki	15
4	Czas obliczeń w zależności od wielkości macierzy dla różnych optymalizacji bez -O2	17
5	Całkowita liczba cykli procesora bez -O2	18
6	Całkowita liczba instrukcji procesora bez -O2	18
7	Liczba chybień w cache L1 bez -O2	19
8	Liczba trafień w cache L2 bez -O2	19
9	Czas obliczeń w zależności od wielkości macierzy dla różnych optymalizacji z -O2	20
10	Całkowita liczba cykli procesora z -O2	21
11	Całkowita liczba instrukcji procesora z -O2	21
12	Liczba chybień w cache L1 z -O2	22
13	Liczba trafień w cache L2 z -O2	22
14	GFLOPS dla różnych optymalizacji i rozmiarów macierzy	23

Źródła

- [1] Wykłady i laboratoria prowadzone przez Pana dra hab. Macieja Woźniaka w ramach przedmiotu *Optymalizacja kodu na różne architektury* prowadzone na AGH w Krakowie
- [2] Ćwiczenie *How to optimize Gemm* dostępne na platformie *GitHub* wraz z zawartą stroną *Wiki* oraz zawartością pomocniczą i źródłową autorstwa Profesora Roberta van de Geijna, <https://github.com/flame/how-to-optimize-gemm>
- [3] Specyfikacja techniczna procesora *Intel Core i5-8257U* na stronie producenta, <https://www.intel.com/content/www/us/en/products/sku/191067/intel-core-i58257u-processor-6m-cache-up-to-3-90-ghz/specifications.html>
- [4] Informacja o rozszerzeniach zestawu instrukcji (ang. *Instruction Set Extensions* [ISE]) dostępnych dla procesora *Intel Core i5-8257U* na stronie producenta, <https://ark.intel.com/content/www/us/en/ark/products/191067/intel-core-i5-8257u-processor-6m-cache-up-to-3-90-ghz.html>

- [5] Dokumentacja techniczna producenta *Intel* wraz z deklarowanymi wartościami FLOPS, <https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf>
- [6] Artykuł dotyczący zestawienia liczby operacji zmiennoprzecinkowych dla wybranych procesorów dostępny na *Wikipedii*, https://en.wikipedia.org/wiki/FLOPS#Floating-point_operations_per_clock_cycle_for_various_processors, na podstawie <https://en.wikichip.org/wiki/flops>
- [7] Artykuł dotyczący FLOPS (floating point operations per second) (pol. *liczba operacji zmiennoprzecinkowych na sekundę*) dostępny na *Wikipedii*, <https://en.wikipedia.org/wiki/FLOPS>
- [8] Artykuł dotyczący architektury zestawu instrukcji procesora (ang. *instruction set architecture* [ISA]) dostępny na *Wikipedii*, https://en.wikipedia.org/wiki/Instruction_set_architecture
- [9] Artykuł dotyczący metody eliminacji Gaussa (ang. *Gaussian elimination*) dostępny na *Wikipedii*, https://en.wikipedia.org/wiki/Gaussian_elimination
- [10] Informacje dotyczące nrzędzia Performance Application Programming Interface (PAPI) dostępny na stronie *The University of Tennessee, Knoxville*, <https://icl.utk.edu/papi/>
- [11] Artykuł dotyczący nrzędzia Performance Application Programming Interface (PAPI) dostępny na *Wikipedii*, https://en.wikipedia.org/wiki/Performance_Application_Programming_Interface
