Optymalizacja kodu na różne architektury

Zadanie 1
Sprawozdanie

How To Optimize Gemm

Adam Naumiec

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Informatyki
Kwiecień MMXXIV

Optymalizacja kodu na różne architektury
Zadanie 1

Adam Naumiec

Kwiecień 2024

Spis treści

0 Abstrakt

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

W ramach zadania wykonano ćwiczenie How to optimize Gemm dostępne na GitHubie (2), dopasowano rozwiązanie do możliwości posiadanego modelu procesora oraz przygotowano sprawozdanie wraz z opracowaniem wniosków.

1 Specyfikacja procesora

1.1 Parametry techniczne

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

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

1.2 Liczba operacji zmiennoprzecinkowych na sekundę (FLOPS)

1.2.1 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.

1.2.2 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:

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

1.2.3 Wnioski z obliczeń

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

2 Optymalizacje

2.1 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. **MMult1** - Wprowadzenie makra `X` oraz funkcji `AddDot` pozwoliło na bardziej efektywne zarządzanie operacjami, minimalizując redundancję kodu i poprawiając jego czytelność.
2. **MMult2** - Zmiana interwału iteracji dla indeksu `j` na wartość 4 umożliwiła lepsze wykorzystanie pamięci cache poprzez zredukowanie liczby potrzebnych odwołań do pamięci.
3. **MMult_Ax4_3** - Przeniesienie wywołań funkcji `AddDot` do specjalnie utworzonej funkcji `AddDotAx4` zwiększyło modularność kodu oraz ułatwiło dalsze optymalizacje.
4. **MMult_Ax4_4** - Rozwinięcie funkcji `AddDot` bezpośrednio w miejscach jej wywołania pozwoliło na eliminację dodatkowych skoków funkcji, co przyspieszyło wykonanie kodu.
5. **MMult_Ax4_5** - Stworzenie pojedynczej pętli dla rozwinięć z poprzedniej optymalizacji zminimalizowało narzut związany z wielokrotnymi skokami i testami warunków pętli.
6. **MMult_Ax4_6** - Użycie rejestrów procesora dla przechowywania często używanych zmiennych z macierzy `A` i `C` przyspieszyło dostęp do tych danych, redukując czas potrzebny na ich pobieranie z pamięci.
7. **MMult_Ax4_7** - Wprowadzenie wskaźników do macierzy `B` pozwoliło na bardziej efektywne zarządzanie dostępem do danych, co jest kluczowe w obliczeniach macierzowych.
8. **MMult_Ax4_8** - Zmiany w zarządzaniu pamięcią oraz rejestrach:
   (a) `A = 1` - Przeorganizowanie kroku pętli poprawiło lokalność odwołań do pamięci.
   (b) `A = 4` - Użycie rejestrów dla danych z macierzy `B` jeszcze bardziej zwiększyło wydajność, zmniejszając ilość dostępów do pamięci.

9. **MMult_Ax4_9** - Optymalizacja zarządzania wskaźnikami i zmiana kolejności operacji:
   (a) `A = 1` - Zmiana kroku wskaźników dla `B` na 4 pozwoliła na lepsze wykorzystanie pamięci cache.
   (b) `A = 4` - Zmiana kolejności wykonywania operacji pozwoliła na jeszcze efektywniejsze grupowanie operacji, co jest korzystne z punktu widzenia wydajności. Dokonano grupowania rzędów po dwa i przechodzenie w grupach po kolumnach, wcześniej przechodzono rzędami bez żadnego grupowania.
10. **MMult_4x4_enumerate10** - Wykorzystanie wektorów `__m128d` umożliwiło operacje na wielu danych jednocześnie, co znacząco przyspieszyło przetwarzanie.
11. **MMult_4x4_11** - Podział obliczeń na bloki oraz wprowadzenie funkcji `InnerKernel` pozwoliło na lepsze zarządzanie kompleksowymi obliczeniami, zwiększając ich efektywność poprzez ograniczenie narzutu związanego z zarządzaniem pętlami.
12. **MMult_4x4_12** - Funkcja `PackMatrixA` zorganizowała dane w sposób, który lepiej korzysta z właściwości pamięci cache, redukując niepotrzebne odwołania do pamięci.
13. **MMult_4x4_13** - Uproszczenie sposobu odwoływania się do elementów macierzy `A` skróciło czas potrzebny na przetwarzanie danych.
14. **MMult_4x4_14** - Dodanie funkcji `PackMatrixB` i zmiana kroku w `PackMatrixA` na 4 pozwoliły na jeszcze lepsze dostosowanie struktury danych do wykorzystania właściwości pamięci cache.
15. **MMult_4x4_15** - Warunkowe wykonanie funkcji `PackMatrixB` pozwoliło na elastyczne dostosowanie optymalizacji do różnych scenariuszy użycia, co jest korzystne w różnorodnych zastosowaniach praktycznych.

2.2 Optymalizacje dostosowane do modelu procesora

Uruchomienie kompilatora *GNU Compiler Collection* (GCC) z flagą optymalizacji dla posiadanej mikroarchitektury procesora marki *Intel*. Wykorzystano flagę `skylake` z uwagi na to, że posiadana architektura *Coffee Lake* jest ewolucyjnym ulepszeniem Skylake i używa tej samej mikroarchitektury pod względem instrukcji procesora:

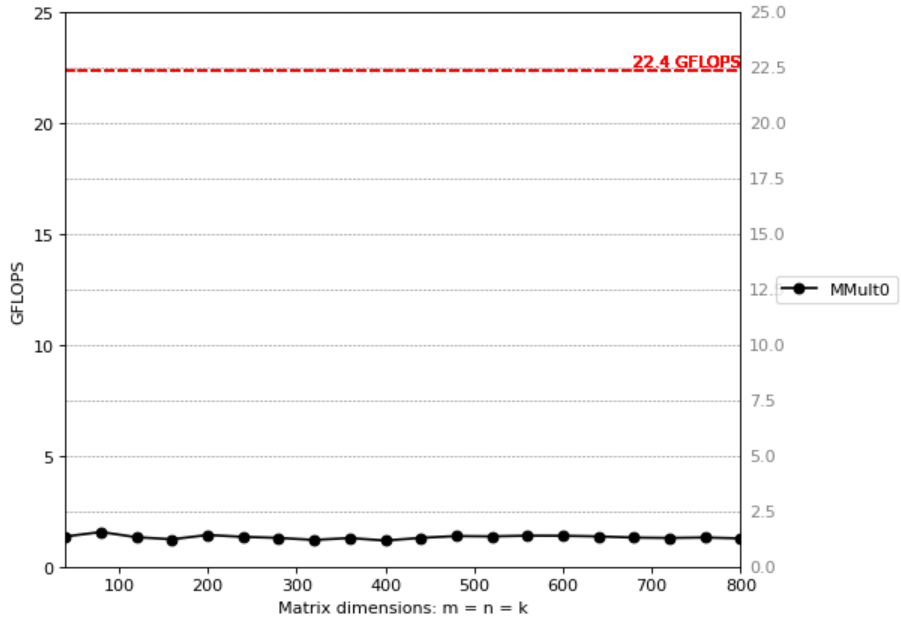
```
-march=skylake
```

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.

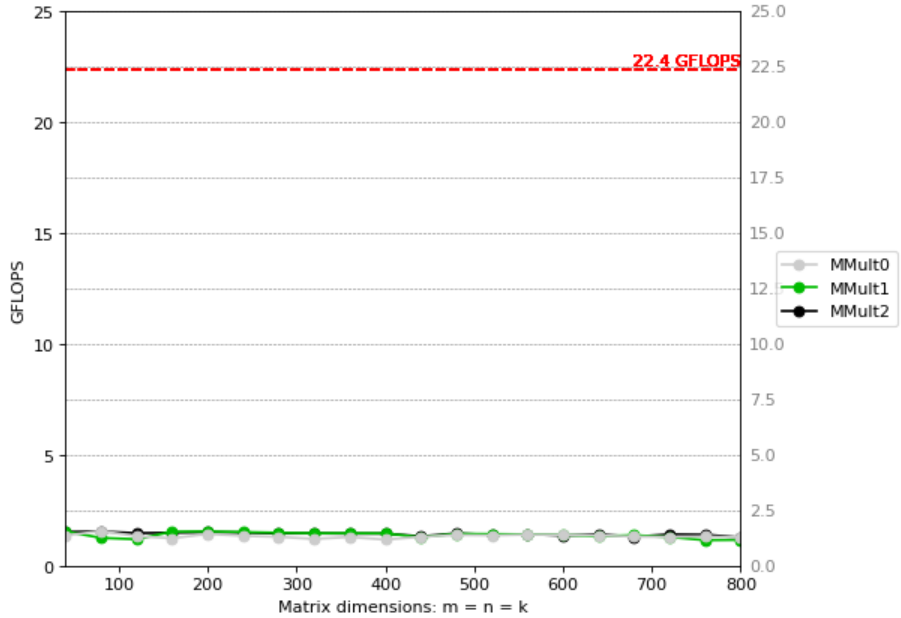
Kompilacja kodu z wykorzystaniem `-march=skylake` zapewnia, że oprogramowanie wydajnie wykorzysta te możliwości na procesorach *Coffee Lake*.

3 Wyniki i analiza

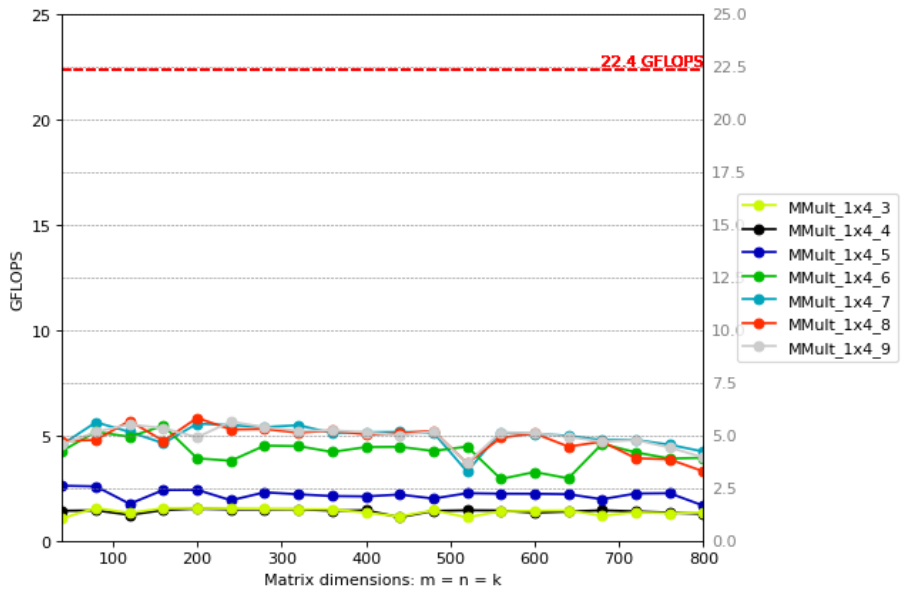
3.1 Uzyskane wyniki



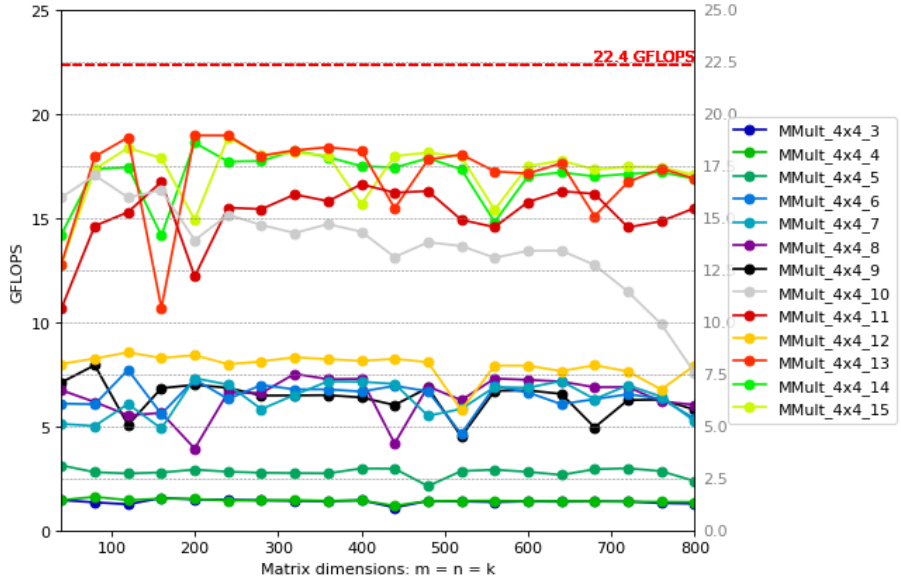
Rysunek 1: Wstępne optymalizacje



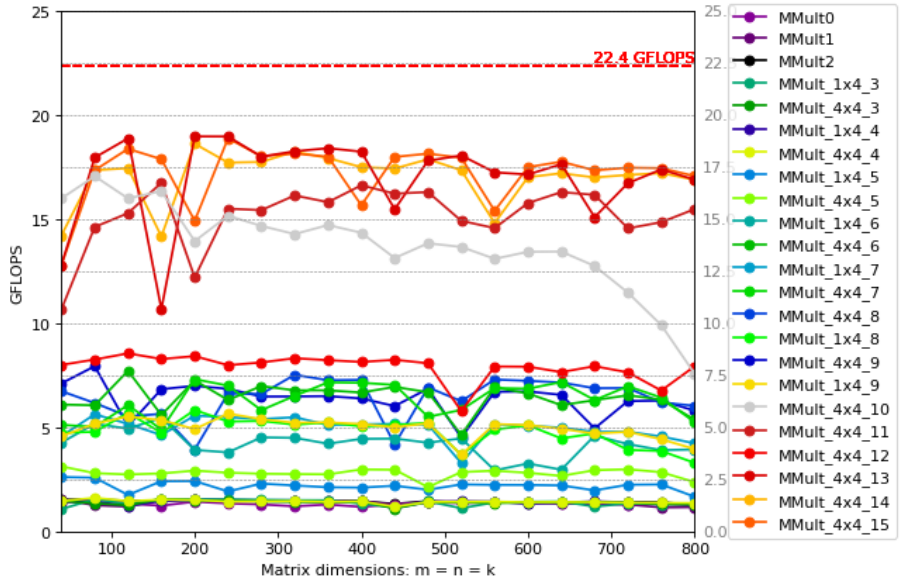
Rysunek 2: Wspólne optymalizacje



Rysunek 3: Optymalizacje 1x4



Rysunek 4: Optymalizacje 4x4



Rysunek 5: Wszystkie optymalizacje

3.2 Analiza rezultatów

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

- **Osiągnięta wydajność w porównaniu z teoretyczną**
  Najlepsze wersje programu, choć znacząco poprawiły wydajność, nie osiągnęły wydajności teoretycznego maksimum. To wskazuje na potencjalne obszary dalszych optymalizacji.
- **Najwydajniejsze wersje programu**
  Wśród testowanych wariantów, największą wydajność osiągnęły wersje `MMult_4x4_13` (uproszczenie odwołań do macierzy), `MMult_4x4_15` (optymalizacja wykorzystania pamięci cache), oraz `MMult_4x4_14` (poprawa zarządzania pamięcią cache).
- **Wpływ optymalizacji na wydajność**
  Największy wzrost wydajności zanotowaliśmy po wprowadzeniu wykorzystania wskaźników do macierzy w wersji `MMult_1x4_7` oraz zastosowaniu wektoryzacji w wersji `MMult_4x4_10`. Te zmiany 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 `MMult_4x4_10` dla dużych macierzy, `MMult_4x4_13` i `MMult_4x4_14` dla macierzy o rozmiarze `n = 160`, oraz `MMult_4x4_11` i `MMult_4x4_15` dla `n = 200`.
- **Rola kompilatora w optymalizacji**
  Analiza pokazała, że kompilator potrafi samodzielnie wprowadzać pewne optymalizacje, co może zdecydowanie wpłynąć na wydajność programu.
- **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.

4 Wnioski

Wykonanie zadania pozwoliło bliżej przyjrzeć się zagadnieniowi 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 kompilacji, podczas gdy `-O3` maksymalizuje wydajność kosztem dłuższego czasu kompilacji i potencjalnie większego rozmiaru końcowego binarnego. Wybór odpowiedniego poziomu zależy od priorytetów projektu: czasu wykonania versus czasu kompilacji.

- **Wpływ lokalności danych**
  Wykorzystanie lokalności danych, zarówno przestrzennej jak i czasowej, poprzez odpowiednie rozmieszczanie 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.

- **Wykorzystanie rejestrowe**
  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 (tutaj przykład z `__m128d`) 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 związanego z zarządzaniem pętlą (np. inkrementacja i sprawdzanie warunku) oraz zwiększa lokalność czasową operacji. W przeprowadzonych optymalizacjach, szczególnie efektywne było rozwijanie pętli w kontekście funkcji `AddDot`, co umożliwiało lepsze wykorzystanie pipeline'ów procesora.

- **Dynamiczne dostosowanie technik optymalizacji**
  Warunkowe zastosowanie niektórych technik, jak pokazano w optymalizacji z `PackMatrixB`, pozwala na dostosowanie strategii wykonania do aktualnego rozmiaru i struktury danych. Daje to możliwość elastycznego reagowania na różnorodne scenariusze użytkowania.
- **Modularyzacja i dekompozycja**
  Podział algorytmu na mniejsze funkcje, jak `InnerKernel` czy `PackMatrix`, nie tylko poprawia czytelność kodu, ale również ułatwia jego optymalizację i testowanie.
- **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 nasze badania podkreśliły kluczowe znaczenie dogłębnego 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

Źródła

[1] Wykłady i laboratoria prowadzone przez Pana Doktora 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] 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`

[5] 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`

[6] Artykuł dotyczący FLOPS (*floating point operations per second*) (pol. liczba operacji zmiennoprzecinkowych na sekundę) dostępny na *Wikipedii*, `https://en.wikipedia.org/wiki/FLOPS`

[7] Artykuł dotyczący architektury zestawu instrukcji procesora (*instruction set architecture* [ISA]) dostępny na *Wikipedii*, `https://en.wikipedia.org/wiki/Instruction_set_architecture`

***