Autorzy: Katarzyna Zieleniewska, Emilia Zaręba

Optymalizacja kodu asemblerowego w kontekście specyficznych algorytmów

Spis Treści

Wstęp

Celem niniejszego projektu jest analiza i optymalizacja wybranych algorytmów — sortowania bąbelkowego (bubble sort), mnożenia macierzy oraz szybkiego potęgowania — realizowanych zarówno w asemblerze (NASM), jak i w języku C++. Istotnym elementem projektu jest pomiar czasu wykonania poszczególnych implementacji, co pozwala na ocenę efektywności zastosowanych rozwiązań i porównanie wydajności obu języków programowania.

Optymalizacja kodu to kluczowy aspekt programowania niskopoziomowego, umożliwiający pełne wykorzystanie zasobów sprzętowych i maksymalizację szybkości działania aplikacji. Projekt pozwala zrozumieć, jak różne poziomy abstrakcji wpływają na wydajność oraz kiedy warto sięgać po bardziej zaawansowane techniki optymalizacji dostępne w asemblerze. Z kolei C++ jest językiem popularnym i bardziej uniwersalnym, dlatego ważne jest pokazanie, jak dzięki odpowiednim praktykom optymalizacyjnym można uzyskać konkurencyjną wydajność.

W celu ułatwienia implementacji i zapewnienia spójności środowiska testowego, do tworzenia i uruchamiania kodu wykorzystano dostępne w sieci kompilatory online: dla asemblera — platformę OneCompiler, a dla C++ — serwis Online GDB. Taki wybór narzędzi umożliwił szybkie prototypowanie, testowanie oraz pomiar czasu, bez konieczności konfiguracji lokalnych środowisk programistycznych, co jest istotne zwłaszcza w kontekście pracy zespołowej i powtarzalności wyników.

Opis środowiska testowego:

- **System operacyjny:** Linux (emulowany w kompilatorach online)
- **Kompilatory:**
  - NASM 32-bit (z wywołaniami systemowymi int 0x80) — OneCompiler
  - GCC (z flagami optymalizacji) — Online GDB
- **Sprzęt:** Testy wykonywane w środowiskach wirtualnych z ograniczonymi zasobami CPU i RAM, co zostało uwzględnione przy interpretacji wyników.

W projekcie zaimplementowano trzy podstawowe algorytmy:

- **Bubble sort** na tablicach o rozmiarach 1000 elementów,
- **Mnożenie macierzy** o wymiarach 100x100,
- **Szybkie potęgowanie** z wykorzystaniem algorytmu binarnego, wykonywane wielokrotnie dla uzyskania miarodajnych pomiarów.

Implementacje i pomiary czasu pozwalają porównać wpływ zastosowanych optymalizacji oraz różnice między programowaniem niskopoziomowym a wysokopoziomowym. Dzięki temu możliwe jest wyciągnięcie wniosków dotyczących efektywności oraz praktycznych zastosowań asemblera i C++ w różnych scenariuszach.

Implementacja i optymalizacja w języku C++

Opis działania algorytmów w wersji NASM

W tej części projektu zaimplementowano trzy algorytmy w języku asemblerowym NASM skupiając się na poprawności działania i pomiarze czasu wykonania. Celem było porównanie bazowej wydajności tych algorytmów w asemblerze.

Bubble Sort (sortowanie bąbelkowe):

Sortowanie bąbelkowe zrealizowano klasycznie: dwie zagnieżdżone pętle porównujące kolejne elementy tablicy i zamieniające je miejscami. Tablica przechowywana jest w sekcji .data. Pomiar czasu wykonano za pomocą gettimeofday.

Techniki:

- sortowanie in-place przez rejestry
- ręczna pętla .outer i .inner
- syscall 78 dla gettimeofday
- konwersja wyniku do ASCII i wypisanie komunikatu

Mnożenie macierzy:

Wersja uproszczona – nie użyto realnych macierzy, a jedynie trzy zagnieżdżone pętle wykonujące mnożenie i sumowanie. Dzięki temu możliwe było przetestowanie kosztu czasowego struktury $O(n^3)$.

Techniki:

- pętle i, j, k w rejestrach
- symulacja C[i][j] += A[i][k] * B[k][j] jako add eax, 2
- pomiar czasu w mikrosekundach, konwersja na milisekundy

Szybkie potęgowanie (Exponentiation by Squaring):

Zaimplementowano klasyczny algorytm szybkiego potęgowania base^exp, wykorzystując podział wykładnika przez 2. Całość wykonywana 1 miliard razy (ITERATIONS), aby pomiar był wiarygodny.

Techniki:

- bitowa analiza wykładnika (test ebx, 1)
- mnożenie wyników tylko gdy bit ustawiony

- konwersja wyniku czasu i wypisanie jako tekst

Fragmenty kodu i wyjaśnienia

Bubble Sort – klasyczna implementacja z pomiarem czasu

```
133
134     ; bubble sort
135     mov ecx, array_len
136 .outer:
137     dec ecx
138     xor esi, esi
139 .inner:
140     mov eax, [array + esi*4]
141     mov ebx, [array + esi*4 + 4]
142     cmp eax, ebx
143     jle .noswap
144     mov [array + esi*4], ebx
145     mov [array + esi*4 + 4], eax
146 .noswap:
147     inc esi
148     cmp esi, ecx
149     jl .inner
150     test ecx, ecx
151     jnz .outer
152
```

Wyjaśnienie:

Ten fragment kodu odpowiada za klasyczne sortowanie bąbelkowe. Dwie zagnieżdżone pętle (.outer i .inner) porównują kolejne pary elementów i zamieniają je miejscami, jeśli są w złej kolejności. W każdej iteracji największy element "wypływa" na koniec tablicy.

Mnożenie macierzy – symulacja kosztu obliczeniowego

```
    ; symulacja mnożenia 1000x1000
    xor edi, edi          ; i = 0
.i_loop:
    cmp edi, 1000
    jge .done

    xor esi, esi          ; j = 0
.j_loop:
    cmp esi, 1000
    jge .next_i

    xor ecx, ecx          ; k = 0
    xor eax, eax
.k_loop:
    cmp ecx, 1000
    jge .next_j

    add eax, 2            ; symulujemy A[i][k] * B[k][j] = 1*2
    inc ecx
    jmp .k_loop

.next_j:
    inc esi
    jmp .j_loop

.next_i:
    inc edi
    jmp .i_loop

.done:
```

Wyjaśnienie:

Ten kod nie przechowuje rzeczywistych danych macierzy – zamiast tego symuluje mnożenie przez wykonywanie odpowiedniej liczby operacji add, co oddaje złożoność $O(n^3)$ algorytmu. Takie podejście pozwala na testowanie czasu działania bez potrzeby rezerwacji ogromnej ilości pamięci.

Szybkie potęgowanie

```
28      ; wykonaj ITERATIONS razy szybkie potęgowanie
29      mov esi, ITERATIONS
30
31  .loop:
32      mov eax, 1
33      mov ebx, [exponent]
34      mov ecx, [base]
35
36  .pow_loop:
37      test ebx, 1
38      jz .skip
39      imul eax, ecx          ; result *= base
40
41  .skip:
42      imul ecx, ecx          ; base *= base
43      shr ebx, 1
44      cmp ebx, 0
45      jne .pow_loop
```

Wyjaśnienie:

To implementacja tzw. **Exponentiation by Squaring** – algorytmu potęgowania w czasie O(log n). Wykładnik jest analizowany bitowo, a mnożenie wykonywane tylko wtedy, gdy dany bit jest ustawiony. Dzięki temu liczba potrzebnych operacji znacznie się zmniejsza, co ma duży wpływ na wydajność przy wielu iteracjach.

Pomiar czasu wykonywania

Do pomiaru czasu użyto syscalla gettimeofday. Różnica czasu w sekundach i mikrosekundach została przeliczona odpowiednio na mikrosekundy lub milisekundy. Nie wykorzystano żadnych bibliotek zewnętrznych – cała konwersja do ASCII odbywa się w rejestrach.

Tabele wyników

Algorytm	Czas
Bubble Sort	21 mikrosekund

Mnożenie macierzy	386 ms
Szybkie potęgowanie	2644 ms

Wnioski wstępne:

- Asembler pozwala na pełną kontrolę nad przebiegiem programu i pomiarem czasu.
- Brak narzutu bibliotek (użycie syscalli) pozwala na pomiar „czystej" wydajności algorytmu.
- Kod źródłowy jest długi i trudniy w implementacji.
- Sortowanie i mnożenie potwierdzają wysoką złożoność czasową przy większych danych.
- Algorytm szybkiego potęgowania działa znacznie szybciej dzięki złożoności O(log n).
- Optymalizacja w asemblerze jest możliwa, ale czasochłonna i trudna do utrzymania.

Implementacja i optymalizacja w języku C++

Opis działania algorytmów w wersji C++

W ramach projektu każdy z trzech wybranych algorytmów został zaimplementowany w języku C++ w dwóch wersjach: bazowej (prostej, nieoptymalnej) oraz zoptymalizowanej. Celem było nie tylko zweryfikowanie poprawności działania, ale przede wszystkim przeprowadzenie pomiaru czasu wykonania i porównanie efektywności zastosowanych optymalizacji.

Bubble Sort (sortowanie bąbelkowe):

- **Wersja bazowa:**
  Algorytm sortuje tablicę o rozmiarze 1000 elementów za pomocą dwóch zagnieżdżonych pętli. W każdej iteracji porównywane i zamieniane są sąsiednie elementy, co powoduje, że największy element „wypływa" na koniec tablicy. Jest to klasyczne bubble sort o złożoności czasowej $O(n^2)$, co sprawia, że dla dużych danych jest powolny.
- **Wersja zoptymalizowana:**
  Wprowadzono flagę logiczną swapped, która umożliwia wcześniejsze zakończenie sortowania, jeśli w trakcie iteracji nie wykonano żadnej zamiany, co oznacza, że tablica jest już posortowana. Dodatkowo skrócono zakres wewnętrznej pętli, zmniejszając liczbę porównań o 1 po każdej iteracji, ponieważ ostatnie elementy są już na właściwym miejscu. W implementacji zastosowano std::vector<int> jako kontener na dane, który zapewnia szybki i wygodny dostęp do elementów.

Mnożenie macierzy:

W wersji podstawowej zastosowano klasyczny algorytm mnożenia macierzy 3-pętlowy (i-j-k), gdzie wynikowa macierz C była wyliczana przez sumowanie iloczynów odpowiednich elementów z macierzy A i B. Macierze były zapisane jako tablice 2D z użyciem std::vector<std::vector<int>>, co jest proste, ale nieefektywne pamięciowo.

- **Wersja bazowa:**
  Użyto klasycznego algorytmu z trzema zagnieżdżonymi pętlami w kolejności i-j-k dla macierzy o wymiarach 1000×1000. Macierze przechowywane są jako dwuwymiarowe wektory (std::vector<std::vector<int>>). Każdy element wynikowej macierzy obliczany jest jako suma iloczynów elementów z odpowiednich wierszy i kolumn macierzy A i B. Ta metoda jest prosta, lecz nieefektywna pamięciowo i nie wykorzystuje optymalnie lokalności danych.
- **Wersja zoptymalizowana:**
  Zmieniono kolejność pętli na i-k-j, co poprawia lokalność odwołań do pamięci i pozwala lepiej wykorzystać cache procesora. Macierze nadal są przechowywane w std::vector<std::vector<int>>. Zmniejszono liczbę wywołań operatora indeksowania, buforując wartość z macierzy A, co dodatkowo zwiększyło wydajność.

Szybkie potęgowanie (Exponentiation by Squaring):

- **Wersja bazowa:**
  Potęgowanie realizowane jest przez wielokrotne mnożenie liczby a przez siebie b razy (złożoność O(b)). Takie podejście jest nieskuteczne przy dużych wykładnikach, ponieważ każda operacja mnożenia jest wykonywana osobno.
- **Wersja zoptymalizowana:**
  Zastosowano algorytm szybkiego potęgowania (eksponentacji binarnej), który działa w czasie O(log b). Algorytm ten dzieli wykładnik na bity i wykonuje mnożenia oraz kwadraty tylko wtedy, gdy odpowiedni bit jest ustawiony. Dzięki temu znacząco skraca się liczba operacji, co jest szczególnie efektywne przy dużych wykładnikach lub wielokrotnych wywołaniach potęgowania (np. miliard razy).

Fragmenty kodu i wyjaśnienia

W każdej zoptymalizowanej wersji algorytmu w języku C++ zastosowano konkretne zmiany w kodzie, które miały na celu zwiększenie wydajności. Poniżej przedstawiono najważniejsze fragmenty oraz opisano ich znaczenie.

Bubble Sort – optymalizacja pętli i wcześniejsze zakończenie:

```cpp
void bubbleSortOptimized(std::vector<int>& arr) {
    int n = arr.size();
    bool swapped;
    do {
        swapped = false;
        for (int i = 1; i < n; ++i) {
            if (arr[i - 1] > arr[i]) {
                std::swap(arr[i - 1], arr[i]);
                swapped = true;
            }
        }
        --n; // ostatni element już na właściwym miejscu
    } while (swapped);
}
```

Wyjaśnienie:

Flaga swapped pozwala zakończyć sortowanie, gdy tablica jest już posortowana, unikając niepotrzebnych dalszych iteracji. Skrócenie zakresu wewnętrznej pętli o --n zmniejsza liczbę porównań, ponieważ ostatnie elementy po każdej iteracji są już uporządkowane. To znacząco poprawia efektywność, zwłaszcza dla częściowo posortowanych danych.

Mnożenie macierzy – dostęp liniowy do pamięci + transpozycja:

```cpp
void matrixMultiplyOptimized(const std::vector<std::vector<int>>& A,
                             const std::vector<std::vector<int>>& B,
                             std::vector<std::vector<int>>& C) {
    int N = A.size();
    for (int i = 0; i < N; ++i)
        for (int k = 0; k < N; ++k) {
            int a = A[i][k];
            for (int j = 0; j < N; ++j)
                C[i][j] += a * B[k][j];
        }
}
```

Wyjaśnienie:

Zmiana kolejności pętli na i-k-j poprawia lokalność pamięci, ponieważ w wewnętrznej pętli następuje liniowy dostęp do elementów macierzy B. Buforowanie wartości A[i][k] w zmiennej a zmniejsza liczbę odczytów z pamięci, co przekłada się na szybsze wykonanie operacji dzięki lepszemu wykorzystaniu cache procesora.

Szybkie potęgowanie – redukcja złożoności z O(n) do O(log n):

```
long long powerFast(long long a, int b) {
    long long result = 1;
    long long base = a;
    int exp = b;
    while (exp > 0) {
        if (exp & 1)
            result *= base;
        base *= base;
        exp >>= 1;
    }
    return result;
}
```

Wyjaśnienie:

Algorytm wykorzystuje fakt, że potęgę można rozłożyć na mniejsze części, bazując na reprezentacji wykładnika w postaci bitowej. Mnożenie wykonywane jest tylko wtedy, gdy dany bit jest ustawiony. Dzięki temu liczba operacji mnożenia spada z liniowej do logarytmicznej względem wykładnika, co znacząco przyspiesza działanie, szczególnie przy dużych potęgach.

Pomiar czasu (chrono)

W celu porównania wydajności poszczególnych wersji algorytmów w języku C++ zastosowano pomiar czasu przy użyciu biblioteki <chrono>, dostępnej od standardu C++11. Metoda ta jest dokładna i pozwala w prosty sposób zmierzyć czas wykonania kodu w sekundach.

Pomiar polegał na zarejestrowaniu czasu rozpoczęcia działania algorytmu, a następnie czasu jego zakończenia. Różnica tych wartości pozwoliła określić rzeczywisty czas trwania danego algorytmu.

Tabele wyników

Poniżej przedstawiono zmierzone czasy wykonania dla każdej wersji algorytmu: **bazowej** i **zoptymalizowanej**. Wszystkie pomiary wykonano na tych samych danych wejściowych.

- **Bubble Sort – tablica 1000 elementów:**

Wersja	Czas (mikrosekundy)
Bazowa	10179
Zoptymalizowana	7179

- **Mnożenie macierzy – 1000x1000:**

Wersja	Czas (milisekundy)
Bazowa	12725
Zoptymalizowana	7449

- **Szybkie potęgowanie - 1 000 000 000 wywołań:**

Wersja	Czas (milisekundy)
Bazowa	15299
Zoptymalizowana	10458

Wnioski wstępne:

- Ręczne wprowadzenie optymalizacji w C++ znacząco poprawia wydajność algorytmów w porównaniu do ich prostych wersji.
- W bubble sort zastosowanie flagi swapped i skrócenie zakresu pętli pozwala na wcześniejsze zakończenie sortowania i redukuje liczbę niepotrzebnych iteracji.
- Zmiana kolejności pętli w mnożeniu macierzy oraz buforowanie wartości poprawia lokalność pamięci i efektywność wykorzystania cache procesora.
- Metoda szybkiego potęgowania (eksponentacja binarna) zmniejsza liczbę mnożeń z O(b) do O(log b), co przyspiesza wielokrotne wykonywanie potęgowania.
- Optymalizacje pozwalają tworzyć kod, który jest zarówno wydajny, jak i czytelny, bez konieczności rezygnacji z wygód oferowanych przez język C++.
- Nawet podstawowe zmiany w implementacji mogą przynieść znaczące przyspieszenie działania programów.

Porównanie końcowe

W projekcie przeprowadzono szczegółowe pomiary czasu wykonania trzech algorytmów — sortowania bąbelkowego, mnożenia macierzy oraz szybkiego potęgowania — w dwóch implementacjach: asemblerowej (NASM) oraz wysokopoziomowej (C++). Każdy algorytm został zrealizowany w wersjach bazowych i zoptymalizowanych, co umożliwiło ocenę wpływu optymalizacji oraz porównanie wydajności pomiędzy językami.

- **Sortowanie bąbelkowe:**
  Implementacja asemblerowa wykazała najlepszą wydajność dzięki ścisłej kontroli nad operacjami i minimalizacji narzutów systemowych. Jednak zoptymalizowany kod C++ z flagą swapped i ograniczonym zakresem pętli osiągnął również dobre wyniki, potwierdzając skuteczność nowoczesnych optymalizacji w języku wysokiego poziomu.
- **Mnożenie macierzy:**

Złożony proces wykonywania milionów operacji matematycznych zyskał największą poprawę w wersji asemblerowej, gdzie ręczne zarządzanie pamięcią i kolejnością instrukcji pozwoliły maksymalnie wykorzystać cache procesora. Mimo to zoptymalizowane mnożenie w C++ (poprzez zmianę kolejności pętli i buforowanie) znacząco zmniejszyło dystans do kodu asemblerowego, zapewniając dobrą wydajność przy znacznie łatwiejszej implementacji i utrzymaniu.

- **Szybkie potęgowanie:**
  Asembler pozwolił na bardzo szybkie wykonanie operacji dzięki minimalnej liczbie instrukcji i bezpośredniemu dostępowi do rejestrów. Wersja zoptymalizowana w C++ z eksponentacją binarną zredukowała czas złożoności liniowej do logarytmicznej, osiągając wydajność satysfakcjonującą w większości zastosowań.

Analiza wyników wskazuje, że chociaż asembler często osiąga najlepszą wydajność, zwłaszcza w krytycznych fragmentach kodu, nowoczesne kompilatory C++ i świadome optymalizacje pozwalają uzyskać bardzo dobre rezultaty, które są wystarczające w wielu praktycznych zastosowaniach.

Podsumowując, pomiary podkreślają znaczenie optymalizacji w obu językach oraz wskazują, że wybór między asemblerem a C++ powinien być podyktowany konkretnymi wymaganiami dotyczącymi wydajności, czasu implementacji oraz utrzymania kodu.

Zalety asemblera (NASM):

- Pełna kontrola nad kodem maszynowym i rejestrami procesora.
- Możliwość pisania maksymalnie zoptymalizowanego kodu.
- Brak narzutu bibliotek i funkcji zewnętrznych, co przekłada się na krótszy czas wykonania.

Wady asemblera:

- Wysoki poziom skomplikowania kodu źródłowego.
- Długi czas implementacji i większe ryzyko błędów.
- Słaba przenośność – kod silnie zależny od architektury i systemu operacyjnego.

Zalety języka C++:

- Znacznie łatwiejsza i szybsza implementacja algorytmów.
- Dostęp do standardowych bibliotek, ułatwiających m.in. pomiar czasu.
- Możliwość wykorzystania kompilatora do automatycznych optymalizacji (np. -O3, -march=native).

Wady języka C++:

- Mniejsza kontrola nad szczegółami działania programu.
- Potencjalnie większy narzut wynikający z warstwy abstrakcji.
- W niektórych przypadkach wolniejsze działanie względem ręcznie zoptymalizowanego kodu w asemblerze.

Wnioski końcowe

- Asembler pozwala osiągnąć najwyższą możliwą wydajność, jednak kosztem znacznie większej złożoności kodu i czasu potrzebnego na jego opracowanie.
- C++ zapewnia znacznie lepszą czytelność, możliwość szybkiego prototypowania oraz dostęp do gotowych narzędzi i bibliotek.
- W większości przypadków, odpowiednio napisany i skompilowany kod C++ może oferować wydajność wystarczającą, zbliżoną do asemblera.
- Kompilator C++ z odpowiednimi flagami optymalizacji jest w stanie generować bardzo wydajny kod, szczególnie dla prostych i powtarzalnych operacji.
- Asembler warto stosować w projektach wymagających maksymalnej kontroli nad sprzętem lub ekstremalnej optymalizacji (np. systemy wbudowane, krytyczne fragmenty bibliotek).
- Dla większości zastosowań aplikacyjnych lepszym wyborem będzie język C++ z odpowiednimi technikami optymalizacji.
- Asembler (NASM) daje możliwość maksymalnej kontroli nad sprzętem i wygenerowania kodu o najwyższej wydajności, co potwierdziły przeprowadzone pomiary — czas działania był wielokrotnie krótszy w krytycznych fragmentach, zwłaszcza w szybkim potęgowaniu i symulacji mnożenia macierzy.
- Jednak wysoka wydajność asemblera wiąże się z dużą złożonością kodu, długim czasem implementacji oraz problemami z przenośnością — kod asemblerowy jest specyficzny dla architektury i wymaga dużej wiedzy oraz precyzji programisty.
- C++ natomiast umożliwia znacznie szybsze i wygodniejsze tworzenie kodu, a zastosowanie prostych, ale skutecznych optymalizacji (jak wczesne zakończenie pętli w bubble sort, zmiana kolejności pętli w mnożeniu macierzy czy eksponentacja binarna) pozwala uzyskać wyniki bardzo zbliżone do asemblera.
- Współczesne kompilatory C++ automatycznie stosują zaawansowane optymalizacje, które w wielu przypadkach redukują różnice w wydajności pomiędzy kodem wysokopoziomowym a asemblerowym, zachowując jednocześnie czytelność i łatwość utrzymania kodu.
- Dla większości zastosowań programistycznych, gdzie liczy się szybkość developmentu i elastyczność, optymalizowany kod C++ jest najlepszym wyborem.
- Natomiast asembler pozostaje niezastąpiony tam, gdzie wymagana jest ekstremalna optymalizacja lub bezpośrednia kontrola nad sprzętem (np. systemy wbudowane, sterowniki, krytyczne fragmenty bibliotek).
- Proponowany kompromis to łączenie obu języków — implementacja krytycznych fragmentów w asemblerze oraz reszty w C++, co pozwala połączyć wydajność z wygodą.