Pierwiastkowanie wielkich liczb

Architektura komputerów 2 – projekt 2019

Sabina Michalczewska 245580
Tobiasz Puślecki 241354

Prowadzący:
Prof. dr hab. inż. Janusz Biernat

Spis treści

1 Wprowadzenie

1.1 Cel projektu

Celem projektu było opracowanie algorytmu obliczania pierwiastka kwadratowego działającego na wielkich liczbach, a następnie zaimplementowaniu go w dowolnym wysokopoziomowym języku programowania. Wynikiem pracy powinien być wydajny algorytm operujący na liczbach składających się z ponad dwóch tysięcy bitów (600 cyfr w reprezentacji dziesiętnej), na którym przeprowadzone zostaną eksperymenty prowadzące do finalnej analizy efektywności rozwiązań.

1.2 Sformułowanie problemu

Główny problem zadania stanowi rozmiar argumentów, a dokładniej sposób składowania w pamięci. Nie istnieje prosty typ zmiennej, pozwalający przechowywać zadany argument.

Konieczne więc było znalezienie sposobu na przechowywanie liczb większych niż pozwalają na to standardowe typy, oraz zaadaptowanie znanych algorytmów (m.in dodawania lub odejmowania pozycyjnego) tak, aby operować na liczbach zapisanych opracowanym na potrzeby zadania sposobem przechowywania. Ze względu na wybrany algorytm pierwiastkowania nieuniknione było również napisanie funkcji pozwalającej na wykonywanie przesunięć bitowych.

1.3 Podstawy teoretyczne

Specyfika obliczania pierwiastka n-tego stopnia pewnymi metodami wymaga przyjęcia oszacowania wartości początkowej. Następnie, w kolejnych iteracjach, uzyskuje się coraz dokładniejsze estymacje wyniku, które w zależnosci od metody, bądź liczby wykonanych kroków algorytmu, różnią się mniej lub bardziej od rzeczywistego wyniku. Do najpopularniejszych metod tego typu należą metoda babilońska, metoda równego podziału czy aproksymacja Bakhshali.

Istnieją również sposoby obliczania cyfra po cyfrze, które są proste do zastosowania w obliczeniach ręcznych, oraz łatwe do zaimplementowania, ponieważ nie wymagają np. dzielenia, jak w metodzie Newtona.

1.4 Założenia

Pierwotnym sposobem przechowywania argumentu była tablica o długości argumentu, w której w każdej komórce mieścił się jeden bit argumentu. To rozwiązanie okazało się jednak nieefektywne, ponieważ marnowany był spory obszar pamięci oraz wymagało ono wykonania większej ilości operacji niż w przypadku przyjętego później rozwiązania.

Zdecydowano więc, że zostanie wykorzystana tablica zawierająca w każdej komórce część argumentu w postaci 32 bitowej liczby typu unsigned int - uint_32 w języku C++, który gwarantuje 32 bitowość na dowolnym kompilatorze i na dowolnym sprzęcie. Typ ten jest zawarty w bibliotece standardowej języka C++.

Został użyty alogrytm "szukaj i sprawdź", ponieważ pozwalał on na najefektywniejsze obliczenie wartości pierwiastka. Dodatkowo nie wymagał implementacji żadnych dodatkowych działań, poza dodawaniem i odejmowaniem pozycyjnym, a w systemie dwójkowym operacje dzielenia można było zastąpić przesunięciami bitów.

1.5 Etapy projektu

- Etap 1 - Wybór i opracowanie algorytmu
- Etap 2 - Implementacja algorytmu dla liczb 128 bitowych
- Etap 3 - Implementacja algorytmu dla liczb reprezentowanych przez słowa 32 bitowe
- Etap 4 - Opracowanie i implementacja sposobu weryfikacji wyników
- Etap 5 - Refaktoryzacja kodu, wprowadzenie ulepszeń do algorytmu, testy szybkości

1.6 Algorytm w pseudokodzie

```
In  : input[]
Out : result[]

Temporary value : one[]

i := 0
Set second MSB of one to 1

while(input[i++]==0)
    one[i] = 1 << 30

while (one > input)
    one = one >> 2

while (one != 0)
    if ( input >= (result + one) )
        input = input - (result + one)
        result = result >> 2 + one
    else
        result = result >> 2
        one = one >> 4
```

1.7 Opis słowny algorytmu

```
Wejscie: tablica liczb 32bit
Wyjscie: tablica liczb 32bit

1. Przypisz i:=0
2. Ustaw drugi MSB w slowie o najwyzszym indeksie tablicy one
3. Dopoki input[i]==0 wykonuj
    3.1 Ustaw drugi MSB w slowie o najwyzszym indeksie tablicy one
4. Dopoki one > number wykonuj
    4.1 Przypisz one:= one >> 2
5. Dopoki one!=0
    5.1 Jesli number >= result+one
        5.1.1 Przypisz input := input - (result + one)
        5.1.2 Przypisz result := result >> 2 + one
    5.2 W przeciwnym razie wykonuj
        5.2.1 Przypisz result := result >> 2
        5.2.2 Przypisz one := one >> 4
```

2 Kod źródłowy w języku C++

2.1 Algorytm pierwiastkowania

Poniższy kod reprezentuje wybrany algorytm pierwiastkowania w języku C++

```
void root(uint32_t input[], uint32_t result[], int n)
{
  for (int i = 0; i < n; i++)
  {
    if(input[i]!=0) // pomijanie zerowych komorek tablicy
       one[i] = 1L << 30;
  }
  while (isBigger(one, input, n))
  {
     bitshift_right(one, n); // przesuwanie "jedynki" dopoki one nie bedzie
     bitshift_right(one, n); // mniejsze/rowne input
  }
  while (!isZero(one, n))
  {
     copy(empty_arr, var, n); //zerowanie var
     add_word(result, one, var, n); // var := result + one

     if (isBigger(input, var, n) || equals (input,var, n ))
       {
         copy(empty_arr, buff, n); //zerowanie buff
         subtract_word(input, var, buff, n+1);
         copy(buff, input, n);           // input -=var
         bitshift_right(result, n);
         copy(empty_arr, x, n); // zerowanie x

         add_word(result, one, x, n);
         copy(x, result, n);
       }
     else
       {
         bitshift_right(result, n);
       }
     bitshift_right(one, n);
     bitshift_right(one, n);
  }
}
```

2.2 Funkcja dodawania

Poniższa funkcja wykonuje dodawanie dwóch tablic i umieszcza wynik tego działania w trzeciej tablicy - result:

```
void add_word(uint32_t a[], uint32_t b[], uint32_t result[], int len)
{
  int carry = 0;

  for (int i = len - 1; i >= 0; i--) //iteracja po indeksach od LSB
  {
    result[i] = a[i] + b[i] + carry;
    carry = (a[i] > result[i] || b[i] > result[i]) ? 1 : 0;
  }
}
```

2.3 Funkcja odejmowania

Poniższa funkcja wykonuje odejmowanie dwóch tablic i umieszcza wynik tego działania w trzeciej tablicy - result:

```
void subtract_word(uint32_t a[], uint32_t b[], uint32_t result[], int len)
{
  int carry = 0;

  for (int i = len - 1; i >= 0; i--)
  {
        result[i] = a[i] - b[i] - carry;
        carry = (b[i] + carry > a[i]) ? 1 : 0;
  }
}
```

2.4 Funkcja przesunięcia bitowego

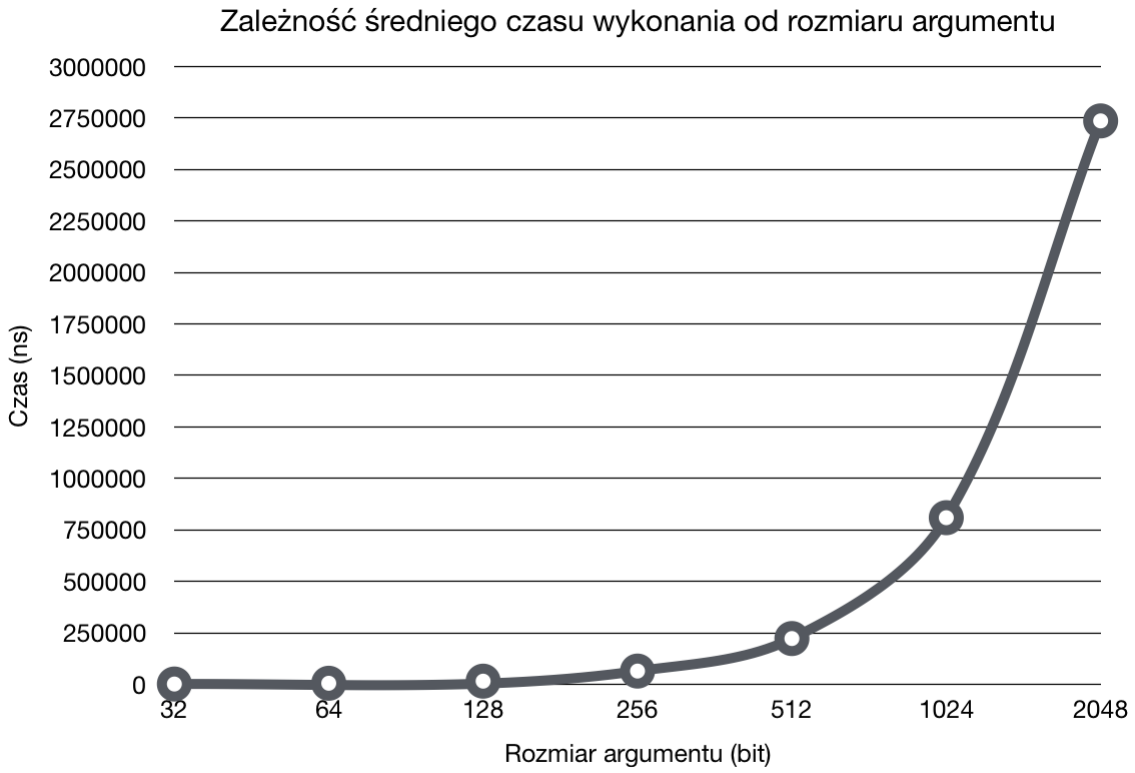
```
void bitshift_right(uint32_t array[], int n)
{
   if (n == 0)
     return;

   for (int i = n - 1; i >= 0; i--)
    {
      array[i] = array [i] >> 1; // przesuniecie w prawo slowa

     // carry bit ustawiony w nastepnym slowie
      if (i > 0)
        array[i] = array[i] | (array[i - 1] & 0x1) << 31;
    }
}
```

3 Wyniki eksperymentów

3.1 Weryfikacja wyników

Sprawdzenie uzyskanego wyniku wymagało zaprojektowania mechanizmu weryfikacji, opartego na funkcji Square, której argumentem był wynik funkcji Root, a wynikiem argument podniesiony do kwadratu. Następnie tak uzyskana liczba była odejmowana od liczby input. Do sprawdzenia ograniczenia górnego i ograniczenia dolnego wykorzystano wzory skróconego mnożenia. W toku eksperymentów zauważono, co jest zgodne z teorią, że wynik pierwiastkowania metodą "szukaj i sprawdź" jest zawsze niemniejszy niż rzeczywista wartość pierwiastka.

3.2 Pomiary czasowe



Najlepszą aproksymacją funkcji przybliżającej powyższy wykres, jest aproksymacja wykładnicza, której wynikiem jest: $y = 370e^{1.3x}$

4 Podsumowanie

4.1 Problemy

Podczas realizacji projektu napotkano następujące problemy:

- Przy zmianie sposobu przechowywania liczb, pojawiły się błędy, co wymusiło przeprojektowanie funkcji sprawdzających relacje między danymi liczbami

4.2 Wnioski

Wybrany algorytm oraz sposób przechowywania argumentu pozwoliły na opracowanie efektywnego algorytmu dla liczb dłuższych niż 2000 bitów, dzięki wykorzystaniu tablicy uint_32 zaoszczędzono sporo czasu, ponieważ można było operować bezpośrednio na bitach bez konieczności konwersji argumentu lub implementacji dzielenia. Wynik pierwiastkowania

Literatura

[1] Janusz Biernat, *Artmetyka komputerów*, ss. 53–55, 1996.

[2] Dennis M. Ritchie, Brian W. Kernighan, *The C Programming Language*, 1988.

[3] C. Woo , M. Guy, "Square Root by Abacus Algorithm", 1985,

[4] Opracowanie zbiorowe, "Methods of computing square root"
`https://en.wikipedia.org/wiki/Methods_of_computing_square_roots`

[5] C++ Reference, `https://cppreference.com`.