Politechnika Wrocławska

Wydział Informatyki
i Telekomunikacji

Organizacja i architektura komputerów: projekt

Analiza i porównanie wybranych algorytmów mnożenia Montgomery’ego

Góra Dawid, 272241
Martynenka Anastasiya, 276720

Sprawozdanie z Projektu OIAK
Prowadzący: Dr hab. inż. S. Piestrak, prof. PWr
Termin zajęć: CZW 9:15-11:00 Parzysty

1. Wstęp teoretyczny

1.1. Arytmetyka modularna

Arytmetyka modularna, zwana również arytmetyką resztkową, polega na wykonywaniu operacji arytmetycznych względem pewnego ustalonego modułu m. W praktyce oznacza to, że wszystkie operacje – takie jak dodawanie, odejmowanie, mnożenie czy potęgowanie – wykonywane są na liczbach całkowitych, a wyniki są redukowane modulo m, czyli obliczane jako reszta z dzielenia przez m. Liczby całkowite zapisujemy w postaci:

$$a = q * m + r$$

Gdzie q jest pewną liczbą całkowitą, a r oznacza reszta z dzielenia liczby a przez q. W arytmetyce modularnej dla dwóch liczb całkowitych a i b mówimy że są do siebie kongruentne modulo m kiedy posiadają one tę samą resztę z dzielenia i zapisujemy to w postaci:

$$a \equiv b \, mod \, m$$

Arytmetyka modularna odgrywa kluczową rolę w informatyce szczególnie w kryptografii, systemach kodowania oraz algorytmach szyfrowania, takich jak RSA. W tych zastosowaniach operacje na bardzo dużych liczbach są wykonywane bardzo często, dlatego kluczowe znaczenie ma efektywność obliczeniowa algorytmów modularnych. Jednym z najważniejszych problemów obliczeniowych w arytmetyce modularnej jest szybkie mnożenie modularne, czyli obliczanie wartości $a \cdot b \ (mod \ n)$ dla dużych a, b i n. Standardowe podejście zakłada wykonanie zwykłego mnożenia całkowitego, a następnie operacji modulo. Dla dużych liczb takie podejście staje się jednak kosztowne obliczeniowo.

1.2. Arytmetyka Montgomery'ego

Aby przyspieszyć operacje modularne, stosuje się algorytmy optymalizujące tę procedurę. Jednym z najbardziej znanych i powszechnie używanych jest algorytm mnożenia Montgomery'ego[1]. Jego kluczową zaletą jest to, że eliminuje kosztowną operację dzielenia przez moduł n, zastępując ją operacjami przesunięć bitowych i redukcji modularnej w przestrzeni Montgomery'ego. Metoda ta wymaga uprzedniej transformacji operandów do tzw. reprezentacji Montgomery'ego. Jednak aby korzystać z tej reprezentacji zaniży spełnić pewne warunki początkowe mianowicie. Moduł n musi być k-bitową liczbą taką że $2^{k-1} \leq n < 2^k$ oraz najlepiej aby $r = 2^k$ jednak można stosować dowolne r które jest relatywnie pierwsze do wartości n czyli $NWD(n, \ r) = 1$ a warunek ten jest spełniony gdy n jest liczbą nieparzystą.

Jeśli spełnimy wspomniane dwa warunki to nasz zredukowany układ reszt modulo n definiujemy jako zbiór $\{a \cdot r \ (mod \, n) \mid 0 \leq a \leq n-1\}$ czyli istnieje dokładnie jeden element a taki że $a' \equiv a \cdot r (mod \, n)$

1.3. MonPro i ModExp

Kiedy zdefiniujemy dwie reszty modulo a' oraz b' możemy wyznaczyć Montgomery produkt który określa nam wzór:

$$MonPro(a', b') = u' = a' \cdot b' \cdot r^{-1} (mod\ n)$$

Gdzie liczby a, b, n są k-bitowymi liczbami binarnymi, $a, b < n$ natomiast $r - 1$ jest odwrotnością modularną $r\ (mod\ n)$, więc spełniona jest własność: $r^{-1} \cdot r = 1\ (mod\ n)$

Ostatnią potrzebną wartością do wyliczenia modularnej redukcji Montgomery'ego jest wartości dla n', która spełnia następujący warunek: $r^{-1} \cdot r - n \cdot n' = 1$. Wartości dla r^{-1} i n' mogą być policzone za pomocą rozszerzonego algorytmu Euklidesa.

Teraz możemy skorzystać z podanych pseudokodów[1] aby zaimplementować funkcje $MonPro(\overline{a}, \overline{b})$ (Pseudokod 1) oraz $ModExp(a; e; n)$ (Pseudokod 2) potrzebne nam do dalszej implementacji:

function MonPro$(\bar{a}, \bar{b})$
Step 1. $t := \bar{a} \cdot \bar{b}$
Step 2. $u := (t + (t \cdot n' \bmod r) \cdot n)/r$
Step 3. **if** $u \geq n$ **then return** $u - n$ **else return** u

Pseudokod 1: Funkcja MonPro[1]

function ModExp(a, e, n)
Step 1. $\bar{a} := a \cdot r \bmod n$
Step 2. $\bar{x} := 1 \cdot r \bmod n$
Step 3. for $i = j - 1$ downto 0
$\bar{x} :=$**MonPro**$(\bar{x}, \bar{x})$
if $e_i = 1$ **then** $\bar{x} :=$**MonPro**$(\bar{x}, \bar{a})$
Step 4. **return** $x :=$**MonPro**$(\bar{x}, 1)$

Pseudokod 2: Funkcja ModExp[1]

2. Opis algorytmów

W naszym projekcie zaimplementowaliśmy dwa algorytmy dla mnożenia Montgomery'ego, Separated Operand Scanning (SOS), oraz Coarsely Integrated Operand Scanning (CIOS), poniżej zostanie omówiona zasada działania obu algorytmów. Pseudokody były podstawą do implementacji ich w naszym projekcie. Na obu algorytmach przeprowadziliśmy testy sprawdzające ich poprawność oraz przeprowadziliśmy badania wydajności ze względu czasowego.

2.1. Algorytm Separated Operand Scanning(SOS)

Metoda SOS jest klasycznym podejściem do mnożenia Montgomery'ego, w którym proces mnożenia i redukcji modularnej są wykonywane osobno, w dwóch oddzielnych etapach. Najpierw następuje obliczenie iloczynu dwóch liczb wielowyrazowych $a \cdot b$, co skutkuje łańcuchem cyfr (słów) przechowywanym w tablicy pomocniczej t. Operacje prezentuje Pseudokod 3.

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[i+j] + a[j]*b[i] + C
        t[i+j] := S
    t[i+s] := C
```

Pseudokod 3: SOS Krok 1.

Następnie przechodzimy do redukcji tablicy używając $u' = \frac{(t+ m \cdot n)}{r}$, gdzie $m = t \cdot n'$. Redukcja odbywa się przez kolejne przeliczenia i dodawanie odpowiednio przesuniętego iloczynu $m \cdot n$ do tablicy t, a następnie wykonanie przesunięcia bitowego (podzielenie przez $R = 2^{sw}$) przez zignorowanie najmniej znaczących słów tablicy. Operacje te są zawarte w Pseudokod 4.

```
for i=0 to s-1
    C := 0
    m := t[i]*n'[0] mod W
    for j=0 to s-1
        (C,S) := t[i+j] + m*n[j] + C
        t[i+j] := S
    ADD (t[i+s],C)
```

Pseudokod 4: SOS Krok 2.

Na końcu wykonywana jest korekcja końcowa – jeśli wynik u która bierze się z tego że nasz wynik jest zapisany w słowie $s + 1$ więc aby zapewnić poprawność wyniku w zakresie $[0, n - 1]$ musimy dokonać odpowiedniej korekty którą prezentuje Pseudokod 5.

```
B := 0
for i=0 to s-1
    (B,D) := u[i] - n[i] - B
    t[i] := D
(B,D) := u[s] - B
t[s] := D
if B=0 then return t[0], t[1], ... , t[s-1]
       else return u[0], u[1], ... , u[s-1]
```

Pseudokod 5: SOS Krok 3.

Metoda SOS charakteryzuje się prostotą implementacyjną i stanowi punkt odniesienia w analizie innych algorytmów. Jednakże, jej wadą jest większe zapotrzebowanie na pamięć ($2s + 2$ słowa pomocnicze) oraz brak optymalizacji przepływu danych między mnożeniem a redukcją.

2.2. Algorytm Coarsely Integrated Operand Scanning(CIOS)

Algorytm CIOS jest zoptymalizowaną wersją SOS, w której mnożenie i redukcja modularna zostały zintegrowane w jednym przebiegu pętli zewnętrznej. W każdej iteracji głównej pętli (po słowach liczby b) wykonywane są zarówno obliczenia iloczynu cząstkowego $a \cdot b[i]$, jak i redukcja modularna operacje te opisuje Pseudokod 6. To pozwala na efektywne wykorzystanie danych tymczasowych oraz redukcję operacji pamięciowych.

```
for i=0 to s-1
    C := 0
    for j=0 to s-1
        (C,S) := t[j] + a[j]*b[i] + C
        t[j] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := C
    C := 0
    m := t[0]*n'[0] mod W
    for j=0 to s-1
        (C,S) := t[j] + m*n[j] + C
        t[i] := S
    (C,S) := t[s] + C
    t[s] := S
    t[s+1] := t[s+1] + C
    for j=0 to s
        t[j] := t[j+1]
```

Pseudokod 6: CIOS Krok 1.

Po każdej iteracji mnożenia, natychmiast wyliczany jest odpowiedni współczynnik redukcji m na podstawie wartości $t[0]$, co umożliwia natychmiastową redukcję przez n. Następnie następuje przesunięcie słów w tablicy t, co odpowiada dzieleniu przez W, bez konieczności przechowywania pełnego łańcucha $2s$ słów. Ten fragment algorytmu opisuje Pseudokod 7. Dzięki temu CIOS potrzebuje jedynie $s + 3$ słów pomocniczych.

```
m := t[0]*n'[0] mod W
(C,S) := t[0] + m*n[0]
for j=1 to s-1
    (C,S) := t[j] + m*n[j] + C
    t[j-1] := S
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s+1] + C
```

Pseudokod 7: CIOS Krok 2.

3. Metodologia i pomiary

Implementacja wcześniej opisanych algorytmów została przeprowadzona w języku Python. W celu analizy ich wydajności, przeprowadzono pomiary czasu działania przy użyciu danych pseudolosowych z uwzględnieniem warunków początkowych. Testy przeprowadzono dla różnych długości danych wejściowych, odpowiadających rozmiarom: 32, 64, 128, 256, 512 oraz 1024 bitów. Dla każdej z tych bitowości przeprowadzono 150 niezależnych pomiarów czasu wykonania, co pozwoliło na uzyskanie wiarygodnych i statystycznie istotnych wyników.

4. Analiza wyników

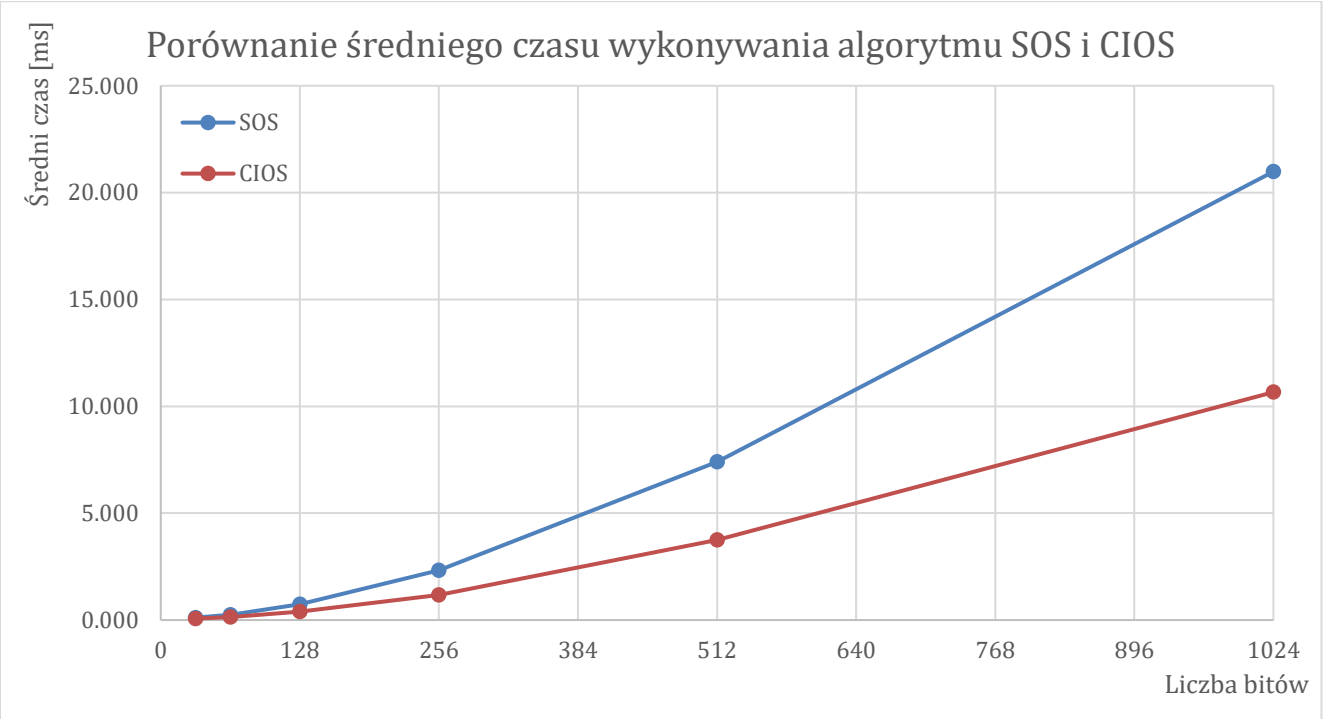
Dla każdego algorytmu z osobna obliczyliśmy średni czas wykonywania algorytmu i wyciągnęliśmy wartość minimalną, maksymalną i medianą. Dane prezentuje Tabela 1 i 2. Średni czas wykonania algorytmu porównaliśmy i zaprezentowaliśmy na Wykres 1.

SOS - czasy w milisekundach				
Liczba bitów	Średnia	Minimum	Maksimum	Mediana
32	0.116	0.036	0.265	0.082
64	0.249	0.046	0.944	0.107
128	0.733	0.071	3.263	0.146
256	2.319	0.148	12.705	0.470
512	7.407	0.355	52.790	1.199
1024	20.983	1.322	197.795	2.515

Tabela 1: Wyniki pomiarów dla algorytmu SOS

CIOS - czasy w milisekundach				
Liczba bitów	Średnia	Minimum	Maksimum	Mediana
32	0.069	0.024	0.277	0.051
64	0.141	0.034	0.613	0.067
128	0.390	0.058	1.749	0.103
256	1.177	0.123	6.156	0.296
512	3.756	0.329	28.402	0.766
1024	10.662	1.279	94.127	1.975

Tabela 2: Wyniki pomiaru dla algorytmu CIOS



Wykres 1: Porównanie średniego czasu dla algorytmów SOS i CIOS

5. Wnioski

Analiza teoretyczna[1] oraz przeprowadzone pomiary wskazują, że algorytm CIOS przewyższa algorytm SOS pod względem efektywności czasowej, zwłaszcza przy przetwarzaniu danych o dużych długościach bitowych. W miarę wzrostu rozmiaru operandów (np. 512 lub 1024 bity), różnice w czasie wykonania stają się coraz bardziej wyraźne i istotne z praktycznego punktu widzenia.

Główną przyczyną tej przewagi jest optymalna organizacja pamięciowa algorytmu CIOS. W odróżnieniu od SOS, który wymaga przechowywania aż $2s + 2$ słów tymczasowych, CIOS ogranicza tę liczbę do zaledwie $s + 3$ słów. Zmniejszone zapotrzebowanie na pamięć tymczasową przekłada się bezpośrednio na redukcję liczby operacji zapisu i odczytu z pamięci — które, jak wiadomo, stanowią istotny czynnik wpływający na ogólną wydajność algorytmu na współczesnych architekturach komputerowych.

Warto podkreślić, że oba algorytmy wykonują identyczną liczbę operacji mnożenia ($2s^2 + s$), jednak to właśnie lepsze zarządzanie przepływem danych i zintegrowanie faz mnożenia oraz redukcji w CIOS pozwala na osiągnięcie znaczących oszczędności czasowych. Dzięki temu CIOS jest nie tylko bardziej wydajny, ale również lepiej skalowalny — co czyni go preferowaną metodą w zastosowaniach praktycznych i implementacjach kryptograficznych wymagających intensywnych obliczeń modularnych, takich jak RSA czy Diffie-Hellman.

6. Bibliografia

[1] C. Kaya Koc, T. Acar and B. S. Kaliski, "Analyzing and comparing Montgomery multiplication algorithms," in *IEEE Micro*, vol. 16, no. 3, pp. 26-33, June 1996, doi: 10.1109/40.502403.

[2] Warren, Henry. "Montgomery multiplication."

[3] Montgomery, Peter L. "Modular multiplication without trial division." *Mathematics of computation* 44.170 (1985): 519-521.