Projekt Zespołowy

Etap projektu – projektowanie rozwiązania na zadaną architekturę

Autorzy:
Biernacka Kamila
Kania Dominik
Leśniak Mateusz
Maziarz Wojciech

kwiecień 2021

Streszczenie

Poniższe sprawozdanie jest wynikiem naszej pracy na drugim etapie projektu zespołowego z implementacji metody indeksu w architekturach GPU. Przedstawimy w nim przygotowane przez nas projekty i rysunki koncepcyjne wymaganych do zaimplementowania algorytmów.

Spis treści

1 Mnożenie modularne dużych liczb **3**

2 Poszukiwanie relacji i faktoryzacja w bazie **4**
2.1 Szybkie potęgowanie modularne 4
2.2 Faktoryzacja w bazie 4
2.3 Budowa relacji 5

3 Eliminacja Gaussa w pierścieniu $\mathbb{Z}_{p-1}$ **6**
3.1 Algorytm Euklidesa 6
3.2 Rozszerzony algorytm Euklidesa 6

1 Mnożenie modularne dużych liczb

W celu wykonania mnożenia dużych liczb $a, b \in \mathbb{F}_p$ wykorzystamy algorytm 2. Pierwszym krokiem jest przedstawienie liczb a, b w postaci $a = x_1 \cdot 2^{32} + y_1$ oraz $b = x_2 \cdot 2^{32} + y_2$.
Wtedy
$r_p = r_p(ab) = r_p((x_1 \cdot 2^{32} + y_1)(x_2 \cdot 2^{32} + y_2)) = r_p(x_1 x_2 \cdot_p 2^{64}) +_p r_p(x_1 y_2 \cdot_p 2^{32}) +_p r_p(x_2 y_1 \cdot_p 2^{32}) +_p r_p(y_1 y_2)$.
Do wyznaczenia pośrednich wartości r_p wykorzystywany jest algorytm 1. Algorytm mnożenia pośredniego działa analogicznie do algorytmu 2. Różnicą jest przedstawienie czynników jako $x \cdot 2^{16} + y$.

Algorithm 1: Mnożenie pośrednie, `halfMult`

Input: a, b - dwie liczby całkowite, p - modulnik
Output: $result$ - wynik mnożenia

$x_1 \leftarrow a >> 16$
$y_1 \leftarrow a \;\&\&\; 0xffff$
$x_2 \leftarrow b >> 16$
$y_2 \leftarrow b \;\&\&\; 0xffff$
$half_a \leftarrow (((x_1 x_2)\%p) \cdot r_p(2^{32}))\%p$
$half_b \leftarrow (((x_1 y_2)\%p) \cdot r_p(2^{16}))\%p$
$half_c \leftarrow (((x_2 y_1)\%p) \cdot r_p(2^{16}))\%p$
$half_d \leftarrow (y_1 y_2)\%p$
return $(half_a + half_b + half_c + half_d)\%p$

Algorithm 2: Pełne mnożenie modularne dwóch liczb, `mult`

Input: a, b - dwie liczby całkowite, p - modulnik
Output: $result$ - wynik mnożenia

$x_1 \leftarrow a >> 32$
$y_1 \leftarrow a \;\&\&\; 0xffffffff$
$x_2 \leftarrow b >> 32$
$y_2 \leftarrow b \;\&\&\; 0xffffffff$
$half_a \leftarrow (halfMult(x_1, x_2) \cdot r_p(2^{64}))\%p$
$half_b \leftarrow (halfMult(x_1, y_2) \cdot r_p(2^{32}))\%p$
$half_c \leftarrow (halfMult(x_2, y_1) \cdot r_p(2^{32}))\%p$
$half_a \leftarrow halfMult(y_1, y_2)$
return $(half_a + half_b + half_c + half_d)\%p$

2 Poszukiwanie relacji i faktoryzacja w bazie

2.1 Szybkie potęgowanie modularne

Metoda indeksu wymaga obliczenia wartości typu a^b *mod* n. Szybkie potęgowanie modularne jest prostym algorytmem pozwalającym zredukować liczbę mnożeń i dzieleń modulo z b do *O(log b)*.

Algorithm 3: Szybkie potęgowanie modularne, `fastPow`

Input : podstawa potęgi a, wykładnik potęgi b, modulnik n
Output: a^b *mod* n

```
bits ← to_bin(b)
nbits ← length(bits)
a ← a%n
result ← 1
x ← a
for i ← 0 to nbits do
    if bits[i]==1 then
        result ← result * x
        result ← result%n
    x ← x * x
    x ← x%n
return result
```

2.2 Faktoryzacja w bazie

Mamy bazę $\mathcal{N} = \{2, 3, \dots p_k\}$, gdzie $p_i \in \mathcal{P}$ i p_k jest największą liczbą pierwszą mniejszą B. W celu faktoryzacji wykorzystamy algorytm 4.

Algorithm 4: Faktoryzacja w bazie, `factor`

Input: a - faktoryzowana liczba, $\mathcal{N}$ - baza rozkładu
Output: $result = [e_1, e_2, \dots, e_k]$ - czynniki

```
for i ← 0 to k do
    counter ← 0
    while a%N[i] do
        a ← a / N[i]
        counter ← counter + 1
    end
    result[i] ← counter
end
return result
```

Algorithm 5: Sprawdzenie czy liczba faktoryzuje się w wybranej bazie, `IsFactor`

```
Input: a - faktoryzowana liczba, 𝒩 - baza rozkładu
Output: result = [e_1, e_2, ..., e_k] - czynniki
for i ← 0 to k do
    counter ← 0
    while a%N[i] do
        a ← a / N[i]
        counter ← counter + 1
    end
    result[i] ← counter
end
if a == 1 then
    return True
end
else
    return False
end
```

2.3 Budowa relacji

W celu zbudowania relacji $\mathcal{R}$ generujemy l liczb losowych e_i. A następnie wykorzystując algorytm 3 wyznaczamy wartości a^{e_i}.
Niech e będzie tablicą l- elementową, jako $fastPow(a, e, p)$ rozumiemy jednoczesne wykonanie $fastPow(a, e_i, p)$ dla każdego elementu tablicy z wykorzystaniem karty graficznej.

Algorithm 6: Budowa relacji, `relationBuild`

```
Input: a - generator, l - wielkość zbioru relacji, p - modulnik
Output: R - zbiór relacji
for i ← 0 to l do
    e[i] ← randomInt()
end
R ← fastPow(a, e, p)
for i ← 0 to l do
    while not isFactored(R[i]) do
        R[i] ← fastPow(a, randomInt(), p)
    end
end
return R
```

3 Eliminacja Gaussa w pierścieniu $\mathbb{Z}_{p-1}$

3.1 Algorytm Euklidesa

Poniższy algorytm wykorzystuje algorytm Euklidesa do sprawdzenia, czy podane na wejściu dwie liczby są względnie pierwsze.

Algorithm 7: Algorytm Euklidesa

Input: a, b - liczby naturalne
Output: True, jeśli gcd$(a, b) == 1$, False w przeciwnym przypadku.

while $b \neq 0$ **do**
    $temp := b$
    $b := a \bmod b$
    $a := temp$
if $a == 1$ **then**
    $output := True$
else
    $output := False$

Result: output

3.2 Rozszerzony algorytm Euklidesa

Tożsamość Bezout mówi, że liczby a i p są względnie pierwsze i wtedy i tylko wtedy, gdy istnieją takie liczby s i t, że

$$ps + at = 1,$$

Wówczas po zredukowaniu tej równości modulo p otrzymujemy

$$at \equiv 1 \bmod p,$$

czyli t jest elementem odwrotnym a w pierścieniu Z_p.

Poniższy algorytm odnajduje element odwrotny do elementu a w pierścieniu Z_p.

Algorithm 8: Rozszerzony Algorytm Euklidesa

Input: a, p - liczby naturalne

Output: $x = a^{-1}$ mod p lub informacja, że taka liczba nie istnieje

$u \leftarrow 1, w \leftarrow a, x \leftarrow 0, \leftarrow p$

while $w \neq 0$ **do**

  if $w < z$ **then**

    $swap(u, x)$

    $swap(w, z)$

  $q \leftarrow w/z - (w\%z)$

  $u \leftarrow u - q \cdot x \; w \leftarrow w - q \cdot z$

if $z \neq 1$ **then**

  Result None

if $x < 0$ **then**

  $x \leftarrow x + p$

Result: x