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 pracy na drugim etapie projektu zespołowego z implementacji metody indeksu w architekturach GPU. Przedstawiono w nim przygotowane przez zespół projekty i rysunki koncepcyjne wymaganych do zaimplementowania algorytmów.

Spis treści

1 Mnożenie modularne dużych liczb

W celu wykonania mnożenia dużych liczb $a, b \in \mathbb{F}_p$ wykorzystany zostanie 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
$r_p = r_p(ab) = r_p((x_1 \cdot 2^{32} + y_1)(x_2 \cdot 2^{32} + y_2)) = r_p(x_1x_2 \cdot_p 2^{64}) +_p r_p(x_1y_2 \cdot_p 2^{32}) +_p r_p(x_2y_1 \cdot_p 2^{32}) +_p r_p(y_1y_2)$.
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

1 $x_1 \leftarrow a >> 16$
2 $y_1 \leftarrow a \;\&\&\; 0xffff$
3 $x_2 \leftarrow b >> 16$
4 $y_2 \leftarrow b \;\&\&\; 0xffff$
5 $half_a \leftarrow (((x_1x_2)\%p) \cdot r_p(2^{32}))\%p$
6 $half_b \leftarrow (((x_1y_2)\%p) \cdot r_p(2^{16}))\%p$
7 $half_c \leftarrow (((x_2y_1)\%p) \cdot r_p(2^{16}))\%p$
8 $half_d \leftarrow (y_1y_2)\%p$
9 **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

1 $x_1 \leftarrow a >> 32$
2 $y_1 \leftarrow a \;\&\&\; 0xffffffff$
3 $x_2 \leftarrow b >> 32$
4 $y_2 \leftarrow b \;\&\&\; 0xffffffff$
5 $half_a \leftarrow (halfMult(x_1, x_2) \cdot r_p(2^{64}))\%p$
6 $half_b \leftarrow (halfMult(x_1, y_2) \cdot r_p(2^{32}))\%p$
7 $half_c \leftarrow (halfMult(x_2, y_1) \cdot r_p(2^{32}))\%p$
8 $half_a \leftarrow halfMult(y_1, y_2)$
9 **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: $a^b \ mod \ n$

```
1  bits ← to_bin(b)
2  nbits ← length(bits)
3  a ← a%n
4  result ← 1
5  x ← a
6  for i ← 0 to nbits do
7      if bits[i]==1 then
8          result ← result * x
9          result ← result%n
10     x ← x * x
11     x ← x%n
12 return result
```

2.2 Faktoryzacja w bazie

Dana jest baza $\mathcal{N} = \{2, 3, \dots p_k\}$, gdzie $p_i \in \mathcal{P}$ i p_k jest największą liczbą pierwszą mniejszą B. W celu faktoryzacji wykorzystany zostanie 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

```
1 for i ← 0 to k do
2     counter ← 0
3     while a%N[i] do
4         a ← a/N[i]
5         counter ← counter + 1
6     end
7     result[i] ← counter
8 end
9 return result
```

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

Input: a - faktoryzowana liczba, $\mathcal{N}$ - baza rozkładu
Output: $result = [e_1, e_2, \ldots, e_k]$ - 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 $\mathcal{R}$ generowanych jest l liczb losowych e_i. Następnie, przy wykorzystaniu algorytmu 3, wyznaczane są wartości a^{e_i}.
Niech e będzie tablicą l- elementową, jako $fastPow(a, e, p)$ rozumiane jest jednoczesne wykonanie $fastPow(a, e_i, p)$ dla każdego elementu tablicy z wykorzystaniem procesora graficznego.

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

Algorithm 7: Algorytm Euklidesa, `isInversible`

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

```
while b ≠ 0 do
    temp ← b
    b ← a mod b
    a ← temp
if a == 1 then
    return True
else
    return False
```

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, otrzymuje się

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

czyli t jest elementem odwrotnym a w pierścieniu Z_p.
Wykorzystując poniższy algorytm możemy znaleźć odwrotność w dowolnym pierścieniu liczbowym.

Algorithm 8: Rozszerzony Algorytm Euklidesa, `inverse`

Input: a, p - liczby naturalne
Output: $x = a^{-1} \bmod p$ lub informacja, że taka liczba nie istnieje

```
1  u ← 1, w ← a, x ← 0, ← p
2  while w ≠ 0 do
3  |  if w < z then
4  |  |  swap(u, x)
5  |  |  swap(w, z)
6  |  q ← w/z − (w%z)
7  |  u ← u − q · x  w ← w − q · z
8  if z ≠ 1 then
9  |  return None
10 if x < 0 then
11 |  x ← x + p
12 return x
```

4 Eliminacja Gaussa

Niech

$$\mathbb{A}_{l,k} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,k} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,k} \\ \vdots & \vdots & & \vdots \\ a_{l,0} & a_{l,1} & \cdots & a_{l,k} \end{bmatrix}.$$

Jako $\mathbb{A}_i = [a_{i,0}, a_{i,1}, \ldots, a_{i,k}]$ rozumiany jest i-wiersz macierzy. Wtedy, następujący zapis $\mathbb{C} = \mathbb{A}_n - \mathbb{A}_m$, gdzie $\mathbb{C} = [c_0, c_1, \ldots, c_k]$ oznacza jednoczesne wykonanie działania $c_j = \mathbb{A}_{n,j} - \mathbb{A}_{m,j}$ dla każdego elementu wiersza z wykorzystaniem procesora graficznego. Natomiast $\mathbb{C} = mult(\mathbb{A}_m, b)$ rozumiany jest jako jednoczesne wykonanie $mult(a_{m,j}, b)$.

Algorithm 9: Algorytm eliminacji Gaussa, `Gauss`

Input: $\mathbb{A}_{l,k} = \mathbb{A}_{l-1,k} \| \mathbb{E}_{1,k}$
Output: $\mathbb{X}_{1,k}$

```
1  for j ← 0 to l do
2      for i ← 0 to k do
3          if i ≠ j then
4              if isInversible(a_{j,j}, p − 1) then
5                  b ← mult(a_{i,j}, inverse(a_{j,j}, p − 1), p − 1)
6                  tmp ← mult(A_i, b, p − 1)
7                  A_i ← A_i − tmp
8              end
9              else
10                 swap(A_i, A_{i+1})
11                 i ← i − 1
12             end
13         end
14     end
15 end
16 for i ← 0 to k do
17     if isInversible(a_{i,i}, p − 1) then
18         b ← inverse(a_{i,i}, p − 1)
19         A_i ← mult(A_i, b, p − 1)
20     end
21     else
22         return None
23     end
24 end
25 return A_k
```