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 Oznaczenia

Tablica 1: Oznaczenia funkcji i operatorów. *Źródło: opracowanie własne*

Symbol	**Opis**
>>	przesunięcie bitowe
&&	koniunkcja bitowa
%	reszta z dzielenia
`to_bin()`	zwraca postać binarną argumentu
`length()`	zwraca długość argumentu
`randomInt()`	zwraca losową liczbę całkowitą
`swap(a,b)`	zmiennej a przypisuje wartość zmiennej b i odwrotnie

2 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

$x_1 \leftarrow a >> 16$
$y_1 \leftarrow a \,\&\&\, 0xffff$
$x_2 \leftarrow b >> 16$
$y_2 \leftarrow b \,\&\&\, 0xffff$
$half_a \leftarrow (((x_1x_2)\%p) \cdot r_p(2^{32}))\%p$
$half_b \leftarrow (((x_1y_2)\%p) \cdot r_p(2^{16}))\%p$
$half_c \leftarrow (((x_2y_1)\%p) \cdot r_p(2^{16}))\%p$
$half_d \leftarrow (y_1y_2)\%p$
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 $(half_a + half_b + half_c + half_d)\%p$

3 Poszukiwanie relacji i faktoryzacja w bazie

3.1 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

$bits \leftarrow to\_bin(b)$
$nbits \leftarrow length(bits)$
$a \leftarrow a\%n$
$result \leftarrow 1$
$x \leftarrow a$
for $i \leftarrow$ *0* ***to*** *nbits* **do**
  if *bits[i]==1* **then**
    $result \leftarrow result * x$
    $result \leftarrow result\%n$
  $x \leftarrow x * x$
  $x \leftarrow x\%n$
return *result*

3.2 Faktoryzacja w bazie

Dana jest baza $\mathcal{N} = \{2, 3, \ldots p_k\}$, 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: $result = [e_1, e_2, \ldots, e_k]$ - czynniki

for $i \leftarrow 0$ ***to*** k **do**
  $counter \leftarrow 0$
  while $a\%N[i]$ **do**
    $a \leftarrow \frac{a}{N[i]}$
    $counter \leftarrow counter + 1$
  end
  $result[i] \leftarrow counter$
end
return *result*

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

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
```

3.3 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
```

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

4.1 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 $b \neq 0$ **do**
  $temp \leftarrow b$
  $b \leftarrow a \bmod b$
  $a \leftarrow temp$
if $a == 1$ **then**
  return $True$
else
  return $False$

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

$u \leftarrow 1, w \leftarrow a, x \leftarrow 0, z \leftarrow p$
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**
    return $None$
if $x < 0$ **then**
    $x \leftarrow x + p$
return x

5 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: $\mathbb{X}_{1,k}$

for $j \leftarrow 0$ ***to*** l **do**
    for $i \leftarrow 0$ ***to*** k **do**
        if $i \neq j$ **then**
            if $isInversible(a_{j,j}, p-1)$ **then**
                $b \leftarrow mult(a_{i,j}, inverse(a_{j,j}, p-1), p-1)$
                $tmp \leftarrow mult(\mathbb{A}_i, b, p-1)$
                $\mathbb{A}_i \leftarrow \mathbb{A}_i - tmp$
            end
            else
                $swap(\mathbb{A}_i, \mathbb{A}_{i+1})$
                $i \leftarrow i - 1$
            end
        end
    end
end
for $i \leftarrow 0$ ***to*** k **do**
    if $isInversible(a_{i,i}, p-1)$ **then**
        $b \leftarrow inverse(a_{i,i}, p-1)$
        $\mathbb{A}_i \leftarrow mult(\mathbb{A}_i, b, p-1)$
    end
    else
        return $None$
    end
end
return $\mathbb{A}_k$