Montgomery reduction

유튜브 주소 : https://youtu.be/hkziOsdakx0

Montgomery reduction

- Montgomery Reduction
  - 나눗셈 없이 모듈러 연산을 수행하기 위한 알고리즘
- 모듈러 곱($a * b \, mod \, Q$) 의 나눗셈 연산을 효율적으로 수행
  - 전통적인 방식에선 곱셈 후 나눗셈을 통해 나머지를 계산 -> 비효율적
- Montgomery는 Q와 서로소인 값($R = 2^k$)을 선택 후 연산을 변환
  - $Mont(T) = T * R^{-1} mod \, Q$ 형태로 변환
- $R^{-1} mod \, Q$와 같은 상수는 사전에 계산 가능
  - 해당 과정을 통해 시프트 및 마스킹 연산만으로 연산 가능
  - 코드에서는 QINV 값으로 정의
- Montgomery reduction 수식: $Mont(T) = \frac{T+(T*\mu \, mod \, R)*Q}{R} \, mod \, Q$
  - T = a*b (곱셈 결과)
  - R = 2^k (시프트 연산을 위한 기준 값)
  - $\mu = -Q^{-1} \, mod \, R$ (보정 상수, QINV)

Montgomery 연산 단계

- Montgomery reductio은 일반 정수 도메인에서 직접 계산 X
  - Montgomery 도메인으로 변환 뒤 연산을 수행 후 다시 복원하는 방식으로 구성
  - 도메인: 계산 영역(숫자를 어떤 방식으로 표현하느냐에 따라 달라짐)
- 전체 연산 단계
  - 도메인 변환
    - 입력값 a,b를 Montgomery 도메인으로 변환
    - $\tilde{a} = a \ * R \ mod \ Q, \tilde{b} = b \ * R \ mod \ Q$
  - 도메인 내 곱셈
    - 변환된 값으로 연산 수행
    - $\tilde{c} = \ \tilde{a} \ * \ \tilde{b} \ * R^{-1} \ mod \ Q$
  - 결과 복원
    - 일반 도메인으로 복원
    - $c = \tilde{c} * R^{-1} \ mod \ Q$

NTT 내부에서의 Montgomery

- NTT는 NCC Sign의 핵심 연산
- Montgomery reduction은 NTT의 핵심 연산
- NCC Sign 내부에서 Montgomery Reduction이 사용되는 함수
  - NTT(): 정방향 변환
  - Invntt_toment(): 역변환
  - Pointwise_mul(): 계수별 곱셈
  - Base_Mul(): 다항식 곱셈

NTT 내부에서의 Montgomery

- NCC Sign의 핵심 연산 NTT
  - NTT radix2 & radix3 사용
  - Radix 3에선 한 루프 내 4번의 호출 수행
- NTT에선 Montgomery 연산이 전체 연산량의 큰 비중을 차지(70% 이상)
- 계수에 zeta값을 곱한 후 Montgomery_reduce 반복 수행

```
void ntt(int32_t * Out, int32_t * A){
    int32_t zeta1;
    int32_t t1;
    int len, start, j, k=0;

    if(Out!=A){
        memcpy(Out,A,sizeof(int32_t)*N);
    }

    zeta1 = zetas[k++];
    for(j = 0; j < N/2; j++)
    {
        t1 = montgomery_reduce((int64_t)zeta1 * Out[j + N/2]);

        Out[j + N/2] = Out[j] + Out[j + N/2] - t1;
        Out[j      ] = Out[j] + t1;
    }
```

```
#if NIMS_TRI_NTT_MODE != 3
    int32_t zeta2;
    int32_t t2,t3,t4;

    for (len = radix3_len; len >= 1; len = len / 3)
    { // radix-3
        for (start = 0; start < N; start += 3 * len)
        {
            zeta1 = zetas[k++];
            zeta2 = zetas[k++];

            for(j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)zeta1 * Out[j + len]);
                t2 = montgomery_reduce((int64_t)zeta2 * Out[j + 2*len]);
                t3 = montgomery_reduce((int64_t)Wmont * t1); //w
                t4 = montgomery_reduce((int64_t)W2mont * t2); //w^2
```

NTT 내부에서의 Montgomery

- Invntt_toment(): NTT 역변환 함수
  - Radix 3 루프 안에서 4회 호출
- Pointwise_mul: 각 계수에 리덕션
  - 루프 내에서 N회 반복
- Base_mul: 2차식 블록 단위 곱셈
  - 각 항마다 2~4회 호출
- 4가지 함수에서 Montgomery_reduce 호출 횟수는 약 2000~3000번

```c
void pointwise_mul(int32_t* C, int32_t* A, int32_t* B){
    for(int i=0;i<N;i++){
        C[i] = montgomery_reduce((int64_t)A[i] * B[i]);
    }
}
void base_mul(int32_t* C, int32_t* A, int32_t* B, int32_t zeta){ //2차식 곱셈
    C[0]=montgomery_reduce((int64_t)A[2]*B[1]);
    C[0]+=montgomery_reduce((int64_t)A[1]*B[2]);
    C[0]=montgomery_reduce((int64_t)C[0]*zeta);
    C[0]+=montgomery_reduce((int64_t)A[0]*B[0]);

    C[1]  = montgomery_reduce((int64_t)A[2]* B[2]);
    C[1]  = montgomery_reduce((int64_t)C[1]* zeta);
    C[1] += montgomery_reduce((int64_t)A[0]* B[1]);
    C[1] += montgomery_reduce((int64_t)A[1]* B[0]);

    C[2]  = montgomery_reduce((int64_t)A[2]* B[0]);
    C[2] += montgomery_reduce((int64_t)A[1]* B[1]);
    C[2] += montgomery_reduce((int64_t)A[0]* B[2]);
}
```

```c
void invntt_tomont(int32_t * Out, int32_t * A){
    int32_t zeta1;
    int32_t t1,t2;
    int len, start, j, k=0;

    if(Out!=A) memcpy(Out,A,sizeof(int32_t)*N);

#if NIMS_TRI_NTT_MODE != 3
    int32_t zeta2;

    for(len = 1; len <= radix3_len ; len = 3*len)
    {//radix-3
        for(start = 0; start < N; start += 3*len)
        {
            zeta1 = zetas_inv[k++];
            zeta2 = zetas_inv[k++];
            for(j = start; j < start + len; j++)
            {
                t1 = montgomery_reduce((int64_t)W2mont * Out[j + len]) + montgomery_reduce((int64_t)Wmont * Out[j + 2*len]);
                t2 = Out[j + len] + Out[j + 2*len];

                Out[j + 2*len] =  montgomery_reduce((int64_t)zeta2 * (Out[j] - (t1 + t2)));
                Out[j + len] = montgomery_reduce((int64_t)zeta1 * (Out[j] + t1));
                Out[j     ] = Out[j] + t2;
            }
```

Montgomery reduce c코드

- Montgomery reduce C 구현 레퍼런스 코드
- 곱셈 후 보정값을 계산하여 나눗셈 없이 결과 도출
- 연산 과정
  - 입력값 a에 대해 t 값 계산(a * QINV)
    - 하위 32비트만 사용
  - T * Q값을 빼서 보정값 계산
  - 최종 결과는 (a – t * Q) >> 32
    - 2^{32}로 나누는 효과
  - 계수를 하나씩 처리하는 구조

```c
////////// ref /////////
int32_t montgomery_reduce(int64_t a) {
  int32_t t;

  t = (int64_t)(int32_t)a*QINV;
  t = (a - (int64_t)t*Q) >> 32;
  return t;
}
```

Montgomery reduction 최적 구현 코드

- ARMv8 어셈블리를 사용한 최적 구현 코드
- 4계수씩 벡터 레지스터에 로드하여 병렬 연산
- 핵심 명령어
  - Sqdmulh: 상위 32비트 곱셈
  - Mul: 32비트 정수 곱셈, 하위 32비트 저장
    - 모듈러 연산을 해야하므로 상위 비트 필요 X
  - Shsub: 비트 간 뺄셈 후 ½
- C에서 루프 4번 반복해야 할 연산을 한 번에 처리

```
  입력:
    x0: out    (int32_t *out)      ← 결과 저장 위치
    x1: a      (int32_t *a)        ← 첫 번째 입력 배열 (길이 4)
    x2: b      (int32_t *b)        ← 두 번째 입력 배열 (길이 4)
    x3: qval   (int32_t[2])        ← {Q, QINV}

  벡터 레지스터:
    v0: a                         ← 입력 a
    v2: b                         ← 입력 b
    v3: Q                         ← 모듈러 계수 Q
    v4: QINV                      ← -Q^{-1} mod 2^32 (몽고메리 상수)
    v6: high(a * b)               ← a * b >> 32
    v27: b * QINV                 ← 중간 곱
    v7: a * (b * QINV)            ← 전체 곱
    v16: high(a * b * QINV * Q)→ 전체 보정값의 상위 32비트
    v6: 최종 결과              ← high(a*b) - high(a*b*QINV*Q)
*/


montgomery_reduce:
_montgomery_reduce:
    // [1] Q 값 설정 (v3에 Q 값 저장)
    // 0xfc = 252; rev16 처리하여 0xfc00; 또는 직접 상수를
    // [3] 입력 a, b 로드
    ld1     {v0.4s}, [x1]         // v0: 4개의 int32_t 값 from a
    ld1     {v2.4s}, [x2]         // v1: 4개의 int32_t 값 from b
ld1R     {v3.4s}, [x3],#4         // v1: 4개의 int32_t 값 from b
ld1R    {v4.4s}, [x3]

sqdmulh v6.4s, v0.4s, v2.4s
mul.4s  v27, v2, v4
mul.4s  v7, v0, v27
sqdmulh v16.4s, v7.4s, v3.4s
shsub.4s    v6, v6, v16

    // 결과를 out 배열에 저장
    st1     {v6.4s}, [x0]

    ret
```

최적 구현 성능 측정

- Apple m2 Xcode 상에서 성능 측정
- 반복 1,000,000회 수행
- 약 6.2배 성능 향상

```
✅ PASS: All values matched!

[Performance Test - 1000000 repetitions]
Ref time: 0.015922 sec
Asm time: 0.002572 sec
Speedup:  6.19x
Program ended with exit code: 0
```

Q & A