마이크로프로세서응용

<16조, final>

조원 : 201810528 고려욱

201810845 박종혁

1. LDPC

1) 동작사진

```
<LDPC decoder>

Measured Accuracy : NSR(dB) = -inf

-----Benchmarking Start-----
Case 0: LDPC Reference
Nr,          Max,          Min,      Average,    Fltr Avg,  Fltr_Avg(ms)
 5,     41537052,     41526981,     41532189,     41532305,      124.597
Case 1: LDPC Optimization
Nr,          Max,          Min,      Average,    Fltr Avg,  Fltr_Avg(ms)
 5,      7628820,      7620078,      7623049,      7622116,       22.866
----Benchmarking Complete----

Optimized LDPC Decoder is x5.44 faster than Reference
```

2) 전체 코드

```
void ldpcd_opt(float(*NLLR)[N], float** opt_out) {

    ////////////////////////
    // Edit code below!! //

    //동적할당은 정확성만 낮아지
    float * Zxn = (float *)calloc((Z*Mp)*(Z*Np),sizeof(float));
    float * Zn = (float *)calloc((Z * Np),sizeof(float));
    float * Lxn = (float *)calloc((Z*Mp)*(Z*Np),sizeof(float));

    float Zn[Z * Np] = { 0, };  // final Value L
    int tp[Z * Mp][Z * Np] = { 0, };
    float a = 0;
    int count[Z * Np] = { 0, };
    int ct;

    for (int p = 0; p < WC; p++) {
        for (int j = 0; j < Z * Np; j++) {
            Zn[j] = NLLR[p][j];
        }
        float Zxn[Z * Mp][Z * Np] = { 0, };      //초기화 시마다 배열 새로 선언
        float Lxn[Z * Mp][Z * Np] = { 0, };      //초기화 시마다 배열 새로 선언

        for (int k = 0; k < iter; k++) {
            for (int j = 0; j < M; j++) {
                float min1 = 500;  //가장 최소값
                float min2 = 500;  //두번째 최소값
                int minx = 0;        //첫번째 최소값 위치 저장
                int sgn = 1;         //부호저장
                int t = 0;           //H배열이 1인 배열의 i값 저장
                if (k == 0) {
                    ct = 0;
                    for (int i = 0; i < N; i++) {
                        if (H[j][i] == 1)
                            tp[j][ct++] = i;
                    }
                    count[j] = ct;
                }
                for (int r = 0; r < count[j]; r++) {
                    t = tp[j][r];
                    Zxn[j][t] = Zn[t] - Lxn[j][t];
                    if (Zxn[j][t] < 0) {
                        sgn = sgn * -1;
                    }
                    a = fabs(Zxn[j][t]);
                    if (min1 > a) {
                        min2 = min1;
```

```
                    min1 = a;
                    minx = t;
                }
                else if (min2 > a)
                    min2 = a;
            }

            //sel step
            for (int r = 0; r < count[j]; r++) {
                t = tp[j][r];
                if (minx == t) {
                    if (min2 > Offset) {
                        Lxn[j][t] = min2 - Offset;
                    }
                    else {
                        Lxn[j][t] = 0;
                    }
                }
                else {
                    if (min1 > Offset) {
                        Lxn[j][t] = min1 - Offset;
                    }
                    else {
                        Lxn[j][t] = 0;
                    }
                }
                if (Zxn[j][t] >= 0) {
                    Lxn[j][t] = Lxn[j][t] * sgn;
                }
                else
                    Lxn[j][t] = Lxn[j][t] * sgn * -1;

                Zn[t] = Zxn[j][t] + Lxn[j][t];
            }
        }
    }

    /* output before Hard Decision */
    for (int i = 0; i < MAX_LENGTH; i++)
        opt_out[p][i] = Zn[i];
}

// Edit code above!! //
//////////////////////
}
```

3) 적용 아이디어 설명

3-1) 적용 아이디어

(1) Loop Optimization 적용

```
for (int r = 0; r < count[j]; r++) {
    t = tp[j][r];
    if (minx == t) {
        if (min2 > Offset) {
            Lxn[j][t] = min2 - Offset;
        }
        else {
            Lxn[j][t] = 0;
        }
    }
    else {
        if (min1 > Offset) {
            Lxn[j][t] = min1 - Offset;
        }
        else {
            Lxn[j][t] = 0;
        }
    }
    if (Zxn[j][t] >= 0) {
        Lxn[j][t] = Lxn[j][t] * sgn;
    }
    else
        Lxn[j][t] = Lxn[j][t] * sgn * -1;

    Zn[t] = Zxn[j][t] + Lxn[j][t];
}
```

- Ref 코드의 For loop를 합치고 순서를 바꿔서 최소한의 for loop를 동작하게 만듬. For-loop를 줄이면서 branch instruction을 줄일 수 있음.
  <prilimary 기준으로 약 69ms 성능개선>

(2) 초기화 및 변수 선언을 for loop안쪽에서 동작

```
for (int p = 0; p < WC; p++) {
    for (int j = 0; j < Z * Np; j++) {
        Zn[j] = NLLR[p][j];
    }
    float Zxn[Z * Mp][Z * Np] = { 0, };     //초기화 시마다 배열 새로 선언
    float Lxn[Z * Mp][Z * Np] = { 0, };     //초기화 시마다 배열 새로 선언

    for (int k = 0; k < iter; k++) {
        for (int j = 0; j < M; j++) {
            float min1 = 500;  //가장 최소값
            float min2 = 500;  //두번째 최소값
            int minx = 0;       //첫번째 최소값 위치 저장
            int sgn = 1;        //부호저장
            int t = 0;          //H배열이 1인 배열의 i값 저장
```

미리 밖에서 선언한 변수들을 가져오는 것이 아니라 새로 생성하여 저장
이를 통해 초기화에 대한 코드를 없애고 assembly상에서 변수 위치에 접근하는데 들어가는 시간을 줄일 수 있음.
<약 3ms 성능개선>

(3) 외부 글로벌 H배열에 대한 접근을 행 loop안에서 한번 만 동작하여 조건 지정한 것 최소한으로 동작할 수 있도록 함

```
if (k == 0) {
    ct = 0;
    for (int i = 0; i < N; i++) {
        if (H[j][i] == 1)
            tp[j][ct++] = i;
    }
    count[j] = ct;
```

Tp2차원배열에 H배열에서 지정한 행 안에 값이 1인 열의 값들을 담아낸다. K가 0일때만 동작하여 계속 반복시에 똑 같은 행의 값에 대한 접근을 줄여주는 역할을 한다. 반복문을 돌며 Ct변수에 총 행의 수를 담고, count배열에 그 열의 총 개수를 담아 뒤에서 참조 할 때 편하게 적용한다.

이를 통해 기존의 ref코드에서 계속해서 해당 cell이 H matrix에서 1인지 검사하는데 들어가는 시간을 최소한으로 줄일 수 있다.

```
for (int r = 0; r < count[j]; r++) {
    t = tp[j][r];
```

접근은 해당열에서 H배열의 값이 1을 가지는 행들만 순서대로 가져와 지정한 LDPC decoding을 실행한다.

<약 35ms 성능개선>

(4) 임시변수 활용

```
for (int r = 0; r < count[j]; r++) {
    t = tp[j][r];
    a = Lxn[j][t];
    if (minx == t) {
        if (min2 > Offset) {
            a = min2 - Offset;
        }
        else {
            a = 0;
        }
    }
    else {
        if (min1 > Offset) {
            a = min1 - Offset;
        }
        else {
            a = 0;
        }
    }
    if (Zxn[j][t] >= 0) {
        a = a * sgn;
    }
    else
        a = a * sgn * -1;
    Lxn[j][t] = a;
    Zn[t] = Zxn[j][t] + Lxn[j][t];
}
```

- 임시변수에 계산 시 활용할 배열의 값을 저장하여 계산마다 배열에 접근하지 않도록 함.

<약 1ms성능개선>

> Preliminary

- 약 16ms 속도 개선
- Temporary variable (tmp_r, tmp_i 활용, 처음에 적용되었던 곳과 다른 곳에 추가로 적용)
- Dave Eberly's Fast square Algorithm 적용
- Loop unrolling, loop order change 적용
- Loop tiling 적용했으나, 성능 개선 미비

```
<QR-Decomposition>
No. of Tx Antennas: 8
No. of Rx Antennas: 8
No. of subcarriers : 1960

Measured Accuracy : NSR(dB) = -57.682

-----Benchmarking Start-----
Case 0: QRD Reference
Nr,        Max,        Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    19747862,    19742055,    19746014,    19746278,       59.239
Case 1: QRD Optimization
Nr,        Max,        Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    14437062,    14433660,    14435736,    14435830,       43.307
----Benchmarking Complete----

Optimized FFT is x1.37 faster than Reference
```

1. 모든 multiply, division 연산을 shift 연산으로 변환

- Preliminary 기준으로 약 3ms 속도 개선
- Shift 연산이 *, /, % 연산보다 빠르다.

COM6 - Tera Term VT

메뉴(F) 수정(E) 설정(S) 제어(O) 창(W) 도움말(H)

```
<QR-Decomposition>
No. of Tx Antennas: 8
No. of Rx Antennas: 8
No. of subcarriers : 1960

Measured Accuracy : NSR(dB) = -57.682

-----Benchmarking Start-----
Case 0: QRD Reference
Nr,        Max,        Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    19746285,    19744182,    19745512,    19745582,       59.237
Case 1: QRD Optimization
Nr,        Max,        Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    13354811,    13351965,    13353365,    13353359,       40.060
----Benchmarking Complete----

Optimized FFT is x1.48 faster than Reference
```

2. Float square root approximation algorithm 적용

- math.h의 sqrt() 보다 속도 빠르다.
- 약 1~2ms 속도 개선
- Approximation 으로 인한, accuracy 감소 (-29.864dB)
- https://bits.stephan-brumme.com/squareRoot.html

```
//Dependant on IEEE representation and only works for 32 bits
 t = *(unsigned int*) &sq;
 t += 127 << 23;            // adjust bias
 t >>= 1;                   // approximation of square root
 tmp_r = *(float*) &t;
```

COM6 - Tera Term VT

메뉴(F) 수정(E) 설정(S) 제어(O) 창(W) 도움말(H)

```
----Benchmarking Complete----

Optimized FFT is x1.57 faster than Reference


<QR-Decomposition>
No. of Tx Antennas: 8
No. of Rx Antennas: 8
No. of subcarriers : 1960

Measured Accuracy : NSR(dB) = -29.864

-----Benchmarking Start-----
Case 0: QRD Reference
Nr,        Max,        Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    19746454,    19743148,    19744610,    19744563,       59.234
Case 1: QRD Optimization
Nr,        Max,        Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    12988189,    12985202,    12986548,    12986512,       38.960
----Benchmarking Complete----

Optimized FFT is x1.52 faster than Reference
```

3. tmp_r 역수 취해줘, for loop 에서 나눗셈 대신 곱셈으로 연산

- 약 2ms 속도 개선
- 곱셈 연산이 cycle 더 작다.

```
tmp_r = 1 / tmp_r;       // 역수를 취하고 곱셈으로 바꿔준다.

for(k = 0; k < NRX; k++)
{
   tmp_col[k].real *= tmp_r;
   tmp_col[k].img *= tmp_r;

   Q[(i << 6) + (k << 3) + j] = tmp_col[k];
}
```

```
<QR-Decomposition>
No. of Tx Antennas: 8
No. of Rx Antennas: 8
No. of subcarriers : 1960

Measured Accuracy : NSR(dB) = -29.864

-----Benchmarking Start-----
Case 0: QRD Reference
Nr,         Max,         Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    19750447,    19747288,    19748777,    19748755,       59.246
Case 1: QRD Optimization
Nr,         Max,         Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    12345514,    12344649,    12345270,    12345317,       37.036
----Benchmarking Complete----

Optimized FFT is x1.60 faster than Reference
```

4. Temporary variable 적용

- tmp_r, tmp_i
- my_complex tmp_c, tmp_col[8]
- matrix의 하나의 원소를 저장하는 tmp_c
- 8x8 matrix의 하나의 열을 저장하는 tmp_col
- 약 10ms 속도 개선

*** Final result**

```
<QR-Decomposition>
No. of Tx Antennas: 8
No. of Rx Antennas: 8
No. of subcarriers : 1960

Measured Accuracy : NSR(dB) = -29.864

-----Benchmarking Start-----
Case 0: QRD Reference
Nr,         Max,         Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    19743676,    19740576,    19741582,    19741446,       59.224
Case 1: QRD Optimization
Nr,         Max,         Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,     9140273,     9134313,     9135778,     9135400,       27.406
----Benchmarking Complete----

Optimized FFT is x2.16 faster than Reference
```

```
void qrd_opt(my_complex* H, my_complex* Q, my_complex* R)
{
   ////////////////////
   // Edit code below //
   int i, j, k, l;
   unsigned int t;
   float sq, tmp_r, tmp_i;
   my_complex tmp_c, tmp_col[8]={0, }; // temporary variables
```

```
for(k = 0; k < 8; k++)
{
   tmp_col[k] = Q[(i << 6) + (k << 3) + j];
   sq += tmp_col[k].real * tmp_col[k].real
      + tmp_col[k].img * tmp_col[k].img;
}
```

```
for(k = 0; k < NRX; k++)
{
   tmp_col[k].real *= tmp_r;
   tmp_col[k].img *= tmp_r;

   Q[(i << 6) + (k << 3) + j] = tmp_col[k];
}
```

```
// Compute Row
for(k = j + 1; k < NTX; k++)
{
   tmp_r = 0, tmp_i = 0;       // Temporary variables
   for(l = 0; l < NRX; l++)
   {
      tmp_r += (tmp_col[l].real * Q[(i << 6) + (l << 3) + k].real
               + tmp_col[l].img * Q[(i << 6) + (l << 3) + k].img);
      tmp_i += (tmp_col[l].real * Q[(i << 6) + (l << 3) + k].img
               - tmp_col[l].img * Q[(i << 6) + (l << 3) + k].real);
   }
      R[(i << 6) + (j << 3) + k].real = tmp_r;
      R[(i << 6) + (j << 3) + k].img = tmp_i;
}
```

```
// Update Column - Temporary variable & loop order change
for(l = 0; l < NRX; l++)
{
   tmp_r = Q[(i << 6) + (l << 3) + j].real;       // Temporary variables
   tmp_i = Q[(i << 6) + (l << 3) + j].img;

   for(k = (j + 1); k < 8; k++)
   {

      tmp_c.real = Q[(i << 6) + (l << 3) + k].real
                  - (R[(i << 6) + (j << 3) + k].real * tmp_r
                    - R[(i << 6) + (j << 3) + k].img * tmp_i);
      tmp_c.img = Q[(i << 6) + (l << 3) + k].img
                  - (R[(i << 6) + (j << 3) + k].real * tmp_i
                    + R[(i << 6) + (j << 3) + k].img * tmp_r);

      Q[(i << 6) + (l << 3) + k] = tmp_c;
      }
   }
}
```

> Fastest, but high dB (-17.430)

- 24.975 ms, -17.430dB
- 구조체 임시 변수 tmp_c1, tmp_c2 선언
- tmp_c1, tmp_c2를 활용해 loop unrolling 적용

```
<QR-Decomposition>
No. of Tx Antennas: 8
No. of Rx Antennas: 8
No. of subcarriers : 1960

Measured Accuracy : NSR(dB) = -17.430

-----Benchmarking Start-----
Case 0: QRD Reference
Nr,         Max,         Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,    19744885,    19742590,    19743250,    19743128,       59.229
Case 1: QRD Optimization
Nr,         Max,         Min,     Average,    Fltr Avg,  Fltr_Avg(ms)
10,     8326180,     8323911,     8325098,     8325111,       24.975
----Benchmarking Complete----

Optinized FFT is x2.37 faster than Reference
```

```
void qrd_opt(my_complex* H, my_complex* Q, my_complex* R)
{
   //////////////////////
   // Edit code below //
   int i, j, k, l;
   unsigned int t;
   float sq, tmp_r, tmp_i;
   my_complex tmp_c1, tmp_c2, tmp_col[8]={0, };
```

```
// Update Column - Temporary variable & loop order change
for(l = 0; l < NRX; l++)
{
   tmp_r = Q[(i << 6) + (l << 3) + j].real;        // Frequency Reduction
   tmp_i = Q[(i << 6) + (l << 3) + j].img;

   for(k = (j + 1); k < 7; k+=2)
   {

      tmp_c1.real = Q[(i << 6) + (l << 3) + k].real
                                  - (R[(i << 6) + (j << 3) + k].real * tmp_r
                                    - R[(i << 6) + (j << 3) + k].img * tmp_i);
      tmp_c1.img = Q[(i << 6) + (l << 3) + k].img
                                   - (R[(i << 6) + (j << 3) + k].real * tmp_i
                                     + R[(i << 6) + (j << 3) + k].img * tmp_r);

      tmp_c2.real = Q[(i << 6) + (l << 3) + (k+1)].real
                                  - (R[(i << 6) + (j << 3) + (k+1)].real * tmp_r
                                    - R[(i << 6) + (j << 3) + (k+1)].img * tmp_i);
      tmp_c2.img = Q[(i << 6) + (l << 3) + (k+1)].img
                                   - (R[(i << 6) + (j << 3) + (k+1)].real * tmp_i
                                     + R[(i << 6) + (j << 3) + (k+1)].img * tmp_r);

      Q[(i << 6) + (l << 3) + k] = tmp_c1;
      Q[(i << 6) + (l << 3) + (k+1)] = tmp_c2;
      }
   if(~j & 1){
      Q[(i << 6) + (l << 3) + 7].real = Q[(i << 6) + (l << 3) + 7].real
                                                  - (R[(i << 6) + (j << 3) + 7].real * tmp_r
                                                   - R[(i << 6) + (j << 3) + 7].img * tmp_i);
      Q[(i << 6) + (l << 3) + 7].img = Q[(i << 6) + (l << 3) + 7].img
                                                  - (R[(i << 6) + (j << 3) + 7].real * tmp_i
                                                   - R[(i << 6) + (j << 3) + 7].img * tmp_r);
   }

   }
}
```