Qualcomm Fixed-Point (qfxp) Library Training

Qualcomm Technologies, Inc.

80-VB419-107 Rev. A

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Contents (cont.)

The Mission

- Starting point – Reference code written in floating-point
- End goal – Optimized implementation running on DSP
- Many challenges to overcome at once:
  - Fixed-point arithmetic does not always have predictable effects on overall accuracy
  - Fixed-point arithmetic offers many tradeoffs, each impacting accuracy of the following:
    - 32-bit vs 6-bit vs 8-bit variables
    - Number of bits allocated to represent mantissa and exponent
    - High-precision, slow operations or low-precision, and fast operations
  - Parallelizing and optimizing code is complex, error-prone, and time-consuming

Goal

- Break down mission into small steps with tool that:
  - Works from C and C++
  - Runs on simulator and target
  - Helps user choose a fixed-point representation of each float variable
  - Helps in understanding impact of tradeoffs on overall accuracy
  - Provides bit-exact fixed-point model to assist with vectorization

Conversion Steps With Fixed-Point Library

1. Replace the floating-point variables with fixed-point variables.
2. Replace the floating-point operations with fixed-point operations.
3. Use the fixed-point library in the Floating-point mode to:
   - Ensure it still produces same output
   - Get suggestions from library on how to represent each variable in fixed-point
4. Transition progressively from the Floating-point mode to Fixed-point mode.
   - Monitor how overall accuracy is impacted by various tradeoffs
5. Generate the bit-exact reference test data after the fixed-point data path is validated.

Example: Floating-Point Code

```
extern float normFactor;
float normalize(float x, float y, float z) {
        float squareX = (x-CENTER_X)*(x-CENTER_X);
        float squareY = (y-CENTER_Y)*(y-CENTER_Y);
        float distance = sqrt(squareX+squareY)/(1<<DISTANCE_SHIFT);
        return normFactor*z/distance;
}
void normalizeArray(float* outputptr, float* xptr, float* yptr, float*
zptr) {
        for (int i=0;i<N-1;i++) {
                outputptr[i] = normalize(xptr[i],yptr[i],zptr[i]);
        }
}
```

Step 1: Declare Fixed-Point Variables

- Assume that all variables are in S.31 format (32-bit signed with 31 fractional bits)

```
qf_t x=S(32,0),y=S(32,0),z=S(32,0),ctrX=S(32,0),ctrY=S(32,0);
qf_t squareX=S(32,0),squareY=S(32,0),square=S(32,0);
qf_t distance=S(32,0), invDistance=S(32,0);
qf_t normZ=S(32,0),result=S(32,0);
qf_t CENTER_X_FP=S(32,0), CENTER_Y_FP=S(32,0),
normFactorFp=S(32,0);
```

Step 2: Set Fixed-Point Variables

- Convert floating-point variables to fixed-point

```
qf_floatToFix(&CENTER_X_FP,CENTER_X,1,QF_IDEAL);
qf_floatToFix(&CENTER_Y_FP,CENTER_Y,1,QF_IDEAL);
qf_floatToFix(&normFactorFp,normFactor,1,QF_IDEAL);

for (int i=0;i<N;i++) {
    qf_floatToFix(&x,xptr[i],1,QF_IDEAL);
    qf_floatToFix(&y,yptr[i],1,QF_IDEAL);
    qf_floatToFix(&z,zptr[i],1,QF_IDEAL);
    …
```

Use the most accurate conversion available

1 = Use rounding

Step 3: Use Fixed-Point Operations

- Replace floating-point operations with fixed-point operations
- For each operation, use highest-precision version available
  - Get functional first and compromise on precision later
  - Most accurate version is identified with flag QF_IDEAL
    For example:
    - Multiply uses 32 × 32 -> 64 full precision multiplier
    - Square root computes floating-point square-root internally

Fixed-Point Code v1

```
qf_sub(&ctrX,x,CENTER_X_FP);
qf_sub(&ctrY,y,CENTER_Y_FP);
qf_mult(&squareX,ctrX,ctrX,1,QF_IDEAL);
qf_mult(&squareY,ctrY,ctrY,1,QF_IDEAL);
qf_add(&square,squareX,squareY);
qf_sqrt(&distance,square,QF_IDEAL);
qf_multpwr2(&normDistance,distance,DISTANCE_SHIFT,1);
qf_invert(&invDistance,normDistance,QF_IDEAL);
qf_mult(&normZ,normFactorFp,z,1,QF_IDEAL);
qf_mult(&result,normZ,invDistance,1, QF_IDEAL);
```

Some operations are expected to always have full precision regardless of their implementations

Step 4: Validate Floating-Point Path

- Exercise floating-point path of fixed-point library
  - Passing test will validate that:
    - All floating-point operations have been converted properly
    - All variables are correctly set

```
outputptr[i]=qf_getFloatRef(result);
```

- If code does not pass test, you likely:
  - Forgot to express an operation from original floating-point code
  - Forgot to initialize a fixed-point variable
  - Introduced global fixed-point variables and forgot that code was multithreaded

Step 5: Analyze Statistics

- Library gathers statistics for all fixed-point variables
- Analyze statistics to decide how to represent floating-point variables

x:(S0.31) Variable name and fixed-point format in use

Code version: v1

Statistics on the float variable

Avg: -1.167. Avg abs: 193.7. [-402.592;401.782]

Fixed-point format for no overflow

(S9)

Statistics on the fixed-point errors

Avg err: 1.112. Avg |err|: 192.7. Max err: 401.6

Data sample size

on 256 iteration

- Bottom line for variable x:
  - S0.31 is inappropriate because it overflows
  - S9.22 is a better fit, or S9.6 for 16-bit representation

Step 6: Adjust Data Formats and Retry

- Modify fixed-point data formats

```
qf_t x=S(32,9),y=S(32,8),z=S(32,0),ctrX=S(32,9),ctrY=S(32,8);
…
```

- Run code again

Code version: v2

x:(S9.22) — Variable name and fixed-point format in use

Statistics on the float variable — Fixed-point format for no overflow

Avg: -1.167. Avg abs: 193.7. [-402.592;401.782]

Stats on fixed-point errors — Data sample size

Avg err: 0.000. Avg |err|: 0.000. Max err: 0.000 — on 256 iteration

- Better!

Step 7: Exercise Fixed-Point Path

- We now know that variables will not overflow
- Assess accuracy of fixed-point simulation

- Change

```
outputptr[i]=qf_getFloatRef(result);
```

(Returns the float variable value from the floating-point path implemented in the library)

- To

```
outputptr[i]=qf_getFloatApprox(result);
```

(Returns the float value represented by the int variable from the fixed-point path)

Step 8: Confirm if the Tests are Passed

- What if we do not pass the test?
  - Overflow error?
    - Check variable statistics again
  - Accuracy error?
    - Does floating-point path have too much dynamic range to be expressed in fixed-point?
    - Can internal precision of some variables be increased?
    - Can we reallocate a few bits from mantissa to fractional part?
- In our case, we pass the test:

```
PASSED. error=0.000001
```

Code version: v3

- Great! Are we done yet?

Step 9: Optimize the Fixed-Point Path

- Initial fixed-point implementation has maximum accuracy on all operations, however, it will be very slow
  - Internal arithmetic for some operations uses floating-point arithmetic
  - High-precision operations might not have an assembly counterpart
- Replace each operation progressively with faster and less accurate counterpart
  - For example:
    - Use fixed-point implementation of invert and square root (sqrt)
    - Use 32 × 16 -> 32 multiply-shift instead of 32 × 32 -> 64 multiply followed by shift-reduce
    - Use formats that remove need for shifts
      - S10.21 + S10.21 -> S10.21 instead of S8.23 + S9.22 -> S10.21
      - S1.30 * S.31 -> S1.30 instead of S1.30 * S.31 -> S.31
      - sqrt (S3.28) = sqrt(2).sqrt(S4.26)
        That is, ensure that mantissa + sign bits are even
      - multpwr2fp(S3.28,2) -> S1.30 is no-op; multpwr2fp(S3.28,2) -> S.31 is a shift instruction

Realistic Invert Operation

- First baby step

Code version: v4

```
qf_sub(&ctrX,x,CENTER_X_FP);
qf_sub(&ctrY,y,CENTER_Y_FP);
qf_mult(&squareX,ctrX,ctrX,1,QF_IDEAL);
qf_mult(&squareY,ctrY,ctrY,1,QF_IDEAL);
qf_add(&square,squareX,squareY);
qf_sqrt(&distance,square,QF_IDEAL);
qf_multpwr2(&normDistance,distance,DISTANCE_SHIFT,1);
qf_invert (&invDistance,normDistance,QF_HVX);
qf_mult(&normZ,normFactorFp,z,1,QF_IDEAL);
qf_mult(&result,normZ,invDistance,1,QF_IDEAL);
```

Note: HVX stands for Qualcomm® Hexagon™ Vector eXtensions.

Debug!

`FAILED. error=19.9799676`

- Does not look like an accuracy issue, does it? What is it?
- Look at code and variable stats again:

```
        invertQF_hvx(&invDistance,normDistance);
```

normDistance:(S5.26)… (U5).... Avg |err|: 1.956e-07. Max err: 1.907e-06 on 256 iter

invDistance:(S0.31)... (U-1)... Avg |err|: 5.542e-01. Max err: 1.439e+00 on 256 iter

- Good input to invert function; bad output… Why?
- HVX invert implementation expects 16-bit input data; we are passing 32-bit
- Change `normDistance =S5_26;` to `normDistance=S21_10;` and retry

`PASSED. error=0.000708  // was 0.000001`

Toward Optimized Fixed-Point

- Use fixed-point implementation of sqrt

```
  qf_sqrt(&distance,square,QF_HVX);
PASSED. error=0.001076;  // was 0.000708
```

- Use 1-cycle 32 × 16 multiplies

```
  qf_mult(&squareX,ctrX,ctrX,1,QF_HVX_MULT_32_16);
  qf_mult(&squareY,ctrY,ctrY,1,QF_HVX_MULT_32_16);
  …
  qf_mult(&normZ,normFactorFp,z,1,QF_HVX_MULT_32_16);
  qf_mult(&result,normZ,invDistance,1,QF_HVX_MULT_32_16);
PASSED. error=0.001813;  // was 0.001076
```

Getting Faster Still

- Use fractional types that minimize number of shifts

Code version: v5

```
qf_t x=S(32,9),y=S(32,9),z=S(32,0),ctrX=S(32,9),ctrY=S(32,9);
qf_t squareX=S18_13,squareY=S15_16,square=S18_13;
qf_t distance=S(32,9), invDistance=S(32,0),normDistance=S21_10;
qf_t normZ=S7_24,result=S6_25;
qf_t CENTER_X_FP=S(32,9), CENTER_Y_FP=S(32,9), normFactorFp=S7_24;
PASSED. error=0.001813;  // unchanged
```

Faster Still?

- After 32-bit path is validated and optimized, attempt 16-bit:

```
qf_t x=S(16,9),y=S(16,9),z=S(16,0),ctrX=S(16,9),ctrY=S(16,9);
qf_t squareX=S(16,18),squareY=S(16,18),square=S(16,18);
qf_t distance=S(16,9), invDistance=S(16,0),normDistance=S(16,5);
qf_t normZ=S(16,7),result=S(32,7);
qf_t CENTER_X_FP=S(16,9), CENTER_Y_FP=S(16,9),
normFactor_FP=S(16,7);
FAILED. error=8.436691
```

Code version: v6

- Full 16-bit path is not sustainable

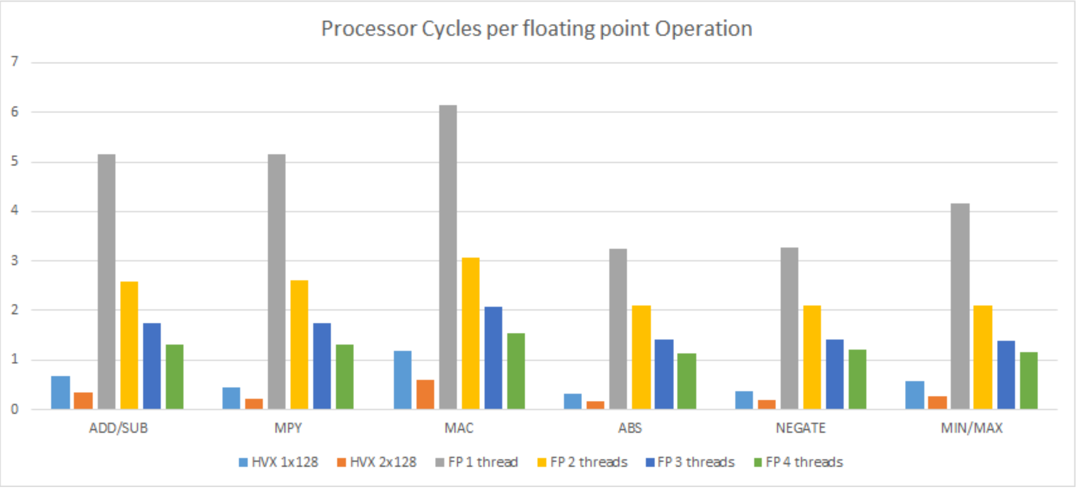
```
qf_t squareX=S(32,18),squareY=S(32,18),square=S(32,18);
PASSED. error=0.001787;  // was 0.001813
```

Code version: v7

- Better than before (pure luck)

What if Fixed-Point is Not Good Enough?

- We developed support for pseudo float arithmetic
- Each number is expressed as pair of numbers
  - Fraction is expressed as S31
  - Mantissa is expressed as S1.30
- Basic operations are already supported
  - pseudo-float <--> float or double
  - add, sub
  - max, min
  - mpy, mac
  - negate, abs
  - comparisons
- Invert and sqrt are in work



Assisting With Actual Implementation

- Fixed-point library also offers functions that can be used to write or debug actual optimized implementation
- `int qf_getRaw(FixedPoint fp)`
  - Returns actual integer value representing fixed-point value
  - Allows comparison of variables from actual and reference implementation
- `qf_getFxpApprox (float ref, int s, int m, int f, int rnd)`
  - Returns integer representing floating-point in fixed-point format
- `QF(ref, s, m, f)`
  - Same as `qf_getFxpApprox`, but as macro, it allows for setting constants

Questions?

https://createpoint.qti.qualcomm.com