Qualcomm Technologies, Inc.

Qualcomm® Fixed-Point (qfxp) Library

80-VB419-69 A

December 7, 2017

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

Contents

1 Introduction

The Qualcomm® Fixed Point (qfxp) library provides assistance with converting floating-point operations to fixed-point operations. Its intent is to convert a floating-point algorithm reference implementation into a fixed-point equivalent ready to be coded on the Qualcomm Hexagon™ DSP, with or without Qualcomm Hexagon Vector eXtensions (HVX).

The output of the library is a bit-exact model of a realistic fixed-point implementation that will run on the DSP and meet the precision and dynamic range requirements acceptable to the user. The output of the library is not an optimized DSP implementation; rather, an intermediary step toward such an implementation.

1.1 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Motivations

Many reference implementations rely on floating-point arithmetic internally since it allows the developer to focus on the functionality of the algorithm being designed without having to worry about the dynamic range of the signals being processed.

Floating-point code can readily run on either the CPU or the DSP. However, this approach presents several disadvantages.

- Running code on the CPU consumes significantly more power than on the DSP.
- Running floating-point code instead of fixed-point code on the DSP is usually slower for various reasons:
    - Reduced thread-level parallelism.

      The floating-point unit, which allows for the parallel execution of up to two floating-point operations each cycle, is shared across all DSP hardware threads. This means that if different threads request the execution of more than two floating-point operations at the same time, some threads will stall.
    - Reduced packet-level parallelism.

      The DSP can execute up to four instructions per packet. However, only two of these instructions can be floating-point instructions.
    - Reduced instruction-level parallelism.

      The scalar instruction set v5x of the DSP supports many SIMD instructions capable of performing two operations on 16-bit quantities or four operations on 8-bit quantities. HVX, the fixed-point wide-vector engine for the Hexagon DSP, offers a much greater level of parallelism since in 128-byte mode, one HVX instruction can perform respectively 32, 64, and 128 operations on 32-bit, 16-bit, or 8-bit quantities.
    - Added latencies. V5x floating-point instructions have a two-cycle latency.

3 Challenges addressed by the library

Converting a floating-point implementation to an optimized, parallel, fixed-point implementation presents multiple challenges:

- Introducing fixed-point arithmetic does not always have predictable effects on overall accuracy.
- Fixed-point operations come in many flavors to pick from, each with their own impacts on performance and accuracy. For example:
    - 32-bit vs. 16-bit vs. 8-bit data types
    - Number of bits allocated to represent the integer and fractional parts of each fixed-point variable
    - High-precision but slow operations, or fast but low-precision operations
- Once the code is expressed in fixed-point arithmetic, parallelizing it is complex, error-prone, and time-consuming.

Attempting to transition directly from a scalar floating-point implementation to a parallel, optimized, vectorized implementation is like eating an apple in one bite: it will work if the apple is small; otherwise it will be hard to chew.

The library allows you to break up these challenging tasks into smaller, more manageable steps with tools assisting with each of these elementary steps.

4 Library components

4.1 Data paths

At the heart of the qfxp library is the data structure used to represent each variable. Each variable is represented as both a floating-point and a fixed-point value, thus allowing at any point in time and for any given variable to understand how the fixed-point approximation deviates from the floating-point reference. In other words, the qfxp library runs in parallel a floating-point and a fixed-point implementation of the same data path. This allows you to understand the impact of implementation tradeoffs on the accuracy of each variable.

4.2 Operations

The qfxp library supports basic arithmetic operations: add, sub, mult, mac, invert, sqrt, and rescaling by a power of two. Support for each of these operation means that each operation is supported in floating-point and in one or more fixed-point implementation favors. For example, in addition to the reference floating-point multiplication, the multiply operation supports the following fixed-point variants:

- QF_IDEAL: signed 32-bit x 32-bit 64-bit followed by shift and optional rounding, back to 32-bit.
- QF_DSP: signed fractional 32-bit x 32-bit 32-bit. The 32-bit output is further shifted as needed with optional rounding.
- QF_HVX_MULT_32_32: signed fractional 32-bit x 32-bit 32-bit made of two 16-bit x 32-bit multiplications. The 32-bit output is further shifted as needed with optional rounding.
- QF_HVX_MULT_32_16: signed fractional 32-bit x 16-bit 32-bit. The 32-bit output is further shifted as needed with optional rounding.

You can build on the existing source of the library to expand the operations supported by the library to suit your needs, either by adding more flavors to an already supported operation, or by creating support for a new operation.

4.3 Statistics

After each operation, the library accumulates statistics on how far the fixed-point output diverge from the floating-point output.

At any point in time, you can display a report on such statistics for any variable you choose. The report includes:

- Variable name
- Fixed-point format currently used to represent the variable
- Average value, average absolute value, min value, and max value of the floating-point variable
- Minimum number of bits required to represent the integer part of the variable without overflowing
- Average value, average absolute value, and max value of the error between the floating-point variable and its fixed-point representation
- Number of iterations on which the statistics were gathered

5 Steps for converting floating-point code to fixed-point code

To obtain a bit-exact reference model of a realistic fixed-point implementation using the qfxp library, follow these steps:

1. Convert the floating-point code to using the floating-point library:
    a. Replace floating-point variables with the library fixed-point variables.
    b. Replace floating-point operations with the library fixed-point operations.
    c. Verify that the floating-point output from the library matches the original floating-point data by passing any tests you have set up.

    At this point, you have validated that you correctly replaced all floating-point operations with the equivalent operations available from the library. Also, by displaying statistics on any variable of your choice, you will get suggestions on which fixed-point format is appropriate to use to guarantee that no overflow will occur.
2. Using the most-accurate fixed-point flavor of all operations, verify that the fixed-point output from the library still passes your tests.

    If you do not pass your tests, verify first that all operations are using the most accurate flavor available. Also, ensure that for operations that have an underlying software-implementation, you meet the requirements of that implementation. For example, if you perform an invert operation, make sure that the fixed-point format you provide does not lead to generating numbers with a range that exceeds what is supported by the invert library.
3. Transition progressively from slow, high-accuracy flavors of each fixed-point operation to faster, less accurate flavors.

    Monitor how the overall accuracy is impacted by the various tradeoffs you pick. When you have reached an appropriate tradeoff of speed vs accuracy for your implementation and still pass all your tests, the resulting implementation is a bit-exact model of the fixed-point implementation you should implement next.
4. Generate bit-exact reference test data to assist with the creation of your optimized fixed-point implementation

These steps are illustrated in the `qfxp_sample` project example, and they are covered in the PowerPoint presentation contained in that project.

6 And if all fails…

Some code exhibits such high dynamic range variations that it might not be possible to convert to a full fixed-point implementation. However, it might still be preferable to find other ways to port the floating-point code to fixed-point.

Two techniques are worth considering:

1. Use block-floating-point arithmetic.

   This approach consists of assigning one exponent to a group of values and normalizing all values with respect to that exponent. It is then possible to perform fixed-point arithmetic on the normalized values, updating the exponent shared by all values as needed, and de-normalizing the values at the end of a sequence of block-floating-point operations.

2. Use the HVX floating-point emulation library included in qmath and delivered as part of the SDK.

   Using this library, you can convert floating-point data to a proprietary Qualcomm floating-point format; perform basic operations such as add, sub, mult, mac, and divide; and transition back to a true floating-point format at the end.