FLOATING-POINT BIT-WIDTH OPTIMIZATION FOR LOW-POWER SIGNAL PROCESSING APPLICATIONS

Fang Fang, Tsuhan Chen, and Rob A. Rutenbar
Dept. of Electrical and Computer Engineering, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{ffang, tsuhan, rutenbar}@ece.cmu.edu

ABSTRACT

To enable floating-point (FP) signal processing applications in low-power mobile devices, we propose a lightweight FP design flow that can optimize the bit-width configuration. The optimization considers both the hardware cost and the numerical precision. Variable grouping is used to reduce the complexity of optimization by connecting software description and hardware implementation. The optimization algorithm is able to avoid local optima, and multiple-phase optimization helps to reduce the cost further. We apply the proposed design flow to the design of inverse discrete cosine transform (IDCT), and show that the power consumption of our lightweight FP IDCT is comparable to an optimized fixed-point design. In addition, promising results on some real-world applications such as video coding and speech recognition demonstrate that lightweight FP signal processing will find more and more applications in low-power devices.

1. INTRODUCTION

Digital signal processing has been widely used in mobile applications, such as video decoding, speech recognition. A lot of effort must be made to manage the complexity, power consumption and time-to-market of the modern signal processing system-on-chip (SoC) designs. However, some DSP algorithms are computationally intensive, rich in costly FP arithmetic operations rather than simple logic. FP hardware offers a wide dynamic range and high computation precision, yet occupies large fractions of total chip area and energy budget. Therefore, its usage in low-power applications is highly limited. Many embedded microprocessors such as the StrongARM do not even include a FP unit due to its unacceptable hardware cost.

In the real world, fixed-point hardware is used to reduce the complexity and power consumption at the cost of degraded precision. However, there is an obvious gap in the design flow: software designers prototype these algorithms using high-precision FP operations, while the silicon designers ultimately implement these algorithms using integer-like hardware, or fixed-point units. The translation from FP operations to fixed-point operations often distorts the natural form of the algorithm and even introduces perceptible artifacts. There is some previous work studying such translation and fixed-point bit-width optimization. In [1], it proposes an analytical approach to find the minimum fixed-point bit-width by the control-dataflow graph, but it needs the user to annotate the bit-widths of some variables. In [2], a simulation based optimization algorithm is developed to find the optimal bit-width configuration.

Even with such fixed-point optimization techniques, the translation is often a time-consuming procedure and becomes the bottleneck of the design flow. In order to speed up the design cycle and broaden FP signal processing applications, we propose a lightweight FP design flow, to enable the simulation, optimization and fast implementation of variable bit-width FP algorithms. As shown in Fig. 1, CMUfloat C++ class provides the support for variable bit-width (32 bits or less) FP operations so that the performance of the algorithm can be monitored during the software simulation stage. The optimization engine configures the bit-widths for both minimal hardware cost and acceptable algorithm performance. Then the algorithm can be easily developed into hardware with a library of variable bit-width FP arithmetic cores that we have developed, using the standard ASIC or FPGA flow.

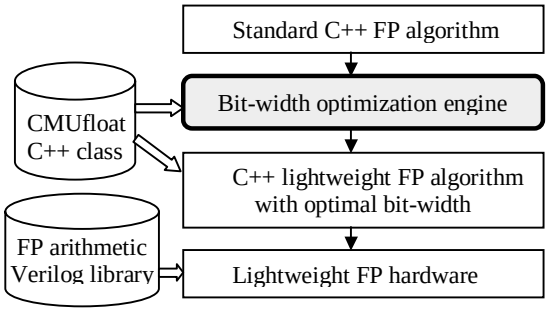

Fig. 1. Lightweight FP system design flow

In this paper, we focus on the bit-width optimization engine, as it is the core of such automated design flow, freeing the designer from the manual tuning of bit-widths. The cost function in this optimization engine is related to hardware cost and power consumption.

IDCT (Inverse Discrete Cosine Transform) is chosen to illustrate the effectiveness of the optimization engine and the whole design flow. As a result, our lightweight FP IDCT consumes comparable amount of power compared to a fixed-point implementation as configured in [3].

The paper is organized as follows: Section 2 introduces briefly the CMUfloat C++ class we use in the simulator and the optimization engine. Section 3 describes the optimization algorithm, and an optimization example of IDCT. Section 4 demonstrates the hardware implementation of the lightweight FP IDCT. Comparisons are made among three implementations: fixed-point, lightweight FP, and IEEE standard FP IDCT. Section 5 shows the impact of lightweight FP on two applications: video decoding and speech recognition. Concluding remarks follow in Section 6.

2. CUSTOMIZABLE CMUfloat C++ LIBRARY

Since the standard C++ does not include variable bit-width FP data type, we create a customizable C++ library to support the configuration of the bit-width, rounding modes and exception handling as well. The data type is called 'CMU*float*' and implemented by overloading existing C++ arithmetic operators including +, -, *, /, etc. It allows direct operations, including assignment between '*CMUfloat*' and any C++ data types other than *char*, as shown in Fig. 2. All the typical C++ features, such

as pointer, reference, arrays, casting, and even I/O stream are also supported. The bit-width of a '*CMUfloat*' variable can vary from 3 to 32 including sign, fraction and exponent bits and can be easily specified during variable declaration. In Fig. 3, we show some examples of using '*CMUfloat*'.

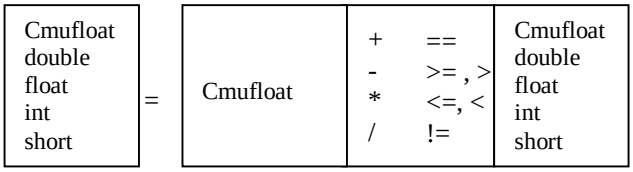

Fig. 2. Operators supported by CMUfloat

```
Cmufloat a(14,5,0.5);  //14 bit fraction and 5 bit exponent
Cmufloat b=1.5;   // Default Cmufloat is IEEE-standard float
Cmufloat c[2];    //  Define an array
float fa;

c[1] = a + b;       // Operation between Cmufloats
fa = a * b;         // Assign the result to float
c[2] = fa + b;      //  Operation between float and Cmufloat
cout << c[1] << c[2] << endl; // I/O stream
```

Fig. 3. Examples of CMUfloat

Such implementation of customizable FP C++ class offers two advantages. First, it provides a transparent mechanism to embed '*CMUfloat*' numbers in an algorithm. As shown in the example, designers can use '*CMUfloat*' just as a standard C++ data type. Therefore, the overall structure of the source codes can be preserve. Second, the arithmetic operators are implemented by bit-level manipulation that carefully emulates the hardware implementation. Therefore the correspondence between algorithm and final bit-level hardware is more exact than previous work [5,6], which appears to have implemented the operators by simply quantizing the result of standard FP operations into limited bit-width. Our approach guarantees better consistency with the hardware implementation. Hence, the numerical performance of the system during the early algorithm simulation is more trustworthy.

3. BIT-WIDTH OPTIMIZATION

With the support of CMUfloat, we can easily set the fraction and exponent bit-widths to be variables, and then let the optimization engine determine the configuration during the simulation. Since the exponent width is determined by the dynamic range, while the fraction width is related to the precision, we can configure them separately. We notice that if two operands have different exponent widths, there has to be an extra adder in hardware to convert the format before the computation. Therefore we simply choose a uniform exponent bit-width according to the overall dynamic range observed by the simulation. We then optimize the fraction width configuration as follows.

3.1 Problem description

The fraction width configuration can be formulated into an optimization problem.

minimize cost $(f_1, f_2, \ldots f_n)$

under the constraints :

$$\begin{cases} performance\,(f_1, f_2, \ldots f_n) >= requirement \\ f_1, f_2, \ldots f_n \in [1,23] \end{cases} \quad (1),$$

where $f_1, f_2, \ldots f_n$ are fraction widths of the variables used in the signal processing algorithm, and the cost function is a measure related to the power consumption.

$$cost = \sum_j power(op_j\_width, op_j\_type) \times execution\_count_j \quad (2),$$

where op_j_width and op_j_type represent the fraction width and the type of the *jth* operation, respectively, and $execution\_count_j$ is the number of times this operation is executed. The power consumption of j*th* FP operation is determined by op_j_type and op_j_width that is the larger value of two operands' fraction width. This power model can be built upon the power measurement of the FP hardware library. This term can be changed to any other form representing the hardware cost.

$Performance(f_1, f_2, \ldots f_n)$ is an objective measurement of the numerical precision of the signal processing algorithm, usually in terms of SNR (signal-to-noise ratio), or MSE (mean square error).

3.2 Variable grouping

There are usually hundreds of FP variables in a signal processing algorithm. If we assign a different fraction width to every single variable, the complexity of the optimization problem will be unacceptably enormous. Also the large variation among the resulting fraction widths will make the hardware design more complex. Signal processing applications usually run on embedded DSPs or custom designed ASICs, both of which can only afford one or very few types of FP formats. Therefore variable grouping according to the hardware implementation topology can help reduce the number of different fraction widths in the optimization procedure. In this case, $f_1, f_2 \ldots f_n$ in (1) represent the fraction widths of the variable groups. A variable grouping example will be shown in 3.4.

3.3 Algorithm description

(a) For each f_i , reduce f_i to its minimal value that satisfy the performance requirement, while keeping all the other variables to be the full width, i.e., 23. Then set each f_i at such minimal, which gives the starting point for the following optimization procedure.

(b) Check the performance requirement. If it is satisfied, then we are done.

(c) Starting from the lower bound configuration obtained in (a), choose the variable to increase the fraction width by one, according to

$$\begin{cases} \dfrac{\Delta performance}{\Delta \cos t} \text{ is the biggest among all } f_i \\ \Delta performance > 0 \end{cases} \quad (3)$$

If none of f_i satisfies (3), that means we have entered a local optimum. No matter which f_i we choose to increase by one, $\Delta performance$ is always negative. In this case, we take a big step by increasing all the f_i by one. Repeat (c), until the performance requirement is satisfied.

(d) Starting from the result of (c), choose variable to decrease the fraction width by one, according to

$$\begin{cases} |\Delta cost| \text{ is the biggest among all } f_i \\ \Delta cost < 0 \\ \text{performanc} >= \text{requirement} \end{cases} \quad (4)$$

If none of f_i satisfies (4), we are done. Otherwise, repeat (d).

We have studied some fixed-point bit-width optimization techniques, as indicated in [3]. The best algorithm they have proved by experiments is called "min+1bit," which is integrated into Steps (a), (b), (c) of our algorithm as shown above. Compared to the algorithm in [3], Step (c) considers both the hardware cost and the precision performance when choosing the best f_i, instead of only the precision. Also it has some mechanism to avoid local optima. Step (d) helps to reduce the cost further while maintaining the performance.

3.4 Optimization of fraction widths on IDCT

To be comparable with previous work [4] on fixed-point optimization, we choose a multiplier-adder-based topology for IDCT. As shown in Fig. 4, the diagram is a straightforward mapping from the source code. In the source code, there are 56 FP variables that can be divided into 10 groups according to the implementation diagram.

```
for (i = 0; i < 4; i++ ) {     // upper half
        dout[i] = 0 ;
        for (j = 0; j< 4; j++ ) {
          dout[i] += Coeff[i][j] * din[j];
        }
    }
    for (i = 0; i < 4; i++ ) {   // lower half
        dout[4+i] = 0 ;
        for (j = 0; j< 4; j++ ) {
          dout[4+i] += Coeff[4+i][j] * din[4+j];
        }
    }
    // post-processing
      for( i = 0 ; i < 4 ; i++ )
        y[i] = (dout[i] + dout[i+4])/2 ;
    for( i = 0 ; i < 4 ; i++ )
        y[7-i] = (dout[i] - dout[i+4])/2 ;
```

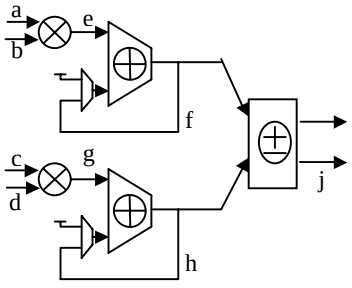

Fig. 4. Source code and diagram of IDCT

In the optimization algorithm, we use the MSE (mean square error) as a performance measure. The maximum allowable MSE is 0.02, according to the IEEE Standard 1180-1990 [7]. The cost of each operator is simplified to be *type_factor* * *operator_fraction_width* * *execution_count*, where the *type_factor* is ADD_F for an adder, and MUL_F for a multiplier, respectively. We will show how the variable grouping and *type_factor* affect the optimal configuration.

The first set of experiments is for the variable grouping. We set ADD_F = ADD_M = 1. Three types of variable grouping are compared:

Grouping_A : No variable grouping, i.e., each variable can have a different bit-width.

Grouping_B : 6 groups {a, b}, {c, d}, {e},{g}, {f, h}, {i, j}

Grouping_C: 4 groups {a, b}, {c, d}, {e, g, f, h}, {i, j}

For Grouping_A, fraction widths are distributed from 8 to 13. The distribution histogram is shown in Fig. 5. It is really hard to configure the hardware design given such a wide distribution.

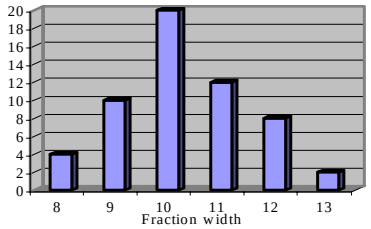

Fig. 5. Histogram of fraction width distribution

The results for Grouping_B and Grouping_C are reasonably distinguished. Grouping_C has more uniform bit-width than Grouping_B because it tries to group more variables together.

Grouping_B:	{a, b},	{c, d},	{e},	{g},	{f, h},	{i, j}
Bit-width :	10	11	10	11	12	11
Grouping_C:	{a, b},	{c, d},	{e, g, f, h},		{i, j}	
Bit-width :	11	11	11		11	

Table 1. Results of Variable grouping

So the point of variable grouping is to give some hardware topology information to the optimization algorithm. If the hardware has lots of resource reusing, then more variables should be grouped together.

The second set of experiments is regarding the *type_factor*. This time we choose Grouping_C as the variable grouping and ADD_F = 1. The comparison of two MUL_F's is shown in Table 2. When the hardware cost of a multiplier is three times of an adder of the same bit-width, then it tries to use fewer bits for the multiplier at the cost of more bits for the adder.

MUL_F = 1 :	{a, b},	{c, d},	{e, g, f, h},	{i, j}
Bit-width :	11	11	11	11
MUL_F = 3 :	{a, b},	{c, d},	{e, g, f, h},	{i, j}
Bit-width :	10	11	12	11

Table 2. Type factor

The variable grouping and the power consumption related cost function enable the designer to optimize the hardware cost at the early design stage. Since the simulation is much faster on the software level, it is beneficial to determine the optimum bit-width configuration during the software simulation.

4. HARDWARE IMPLEMENTATION OF LIGHTWEIGHT FP IDCT

In order to demonstrate the efficiency of our lightweight FP design flow, and the power reduction by narrowing the bit-width, we design a IDCT according to the diagram and the bit-width configuration in Section 3 (5 exponent bits + 11 fraction bits for

all the variables). It is synthesized with the plug-in parametric FP arithmetic cores in our Verilog library and is measured the area and power consumption. We also implement an IEEE standard 32-bit FP IDCT and a fixed-point IDCT for comparison. The bit-width for fixed-point implementation is according to the optimization results in [4]. The area and power are compared in Table 3.

	Area (um^2)	Power (mw)
Fixed-point	36905	12.6
Lightweight FP	66797	18.5
Standard FP	200905	54.3

Table 3. Comparison of three IDCTs

As we can see from the table, there is significant savings (about 70%) in area and power when using lightweight FP units, instead of standard floating units. The power consumption of lightweight FP IDCT is only 1.5 times of that of the fixed-point IDCT, which makes FP IDCT possible for low-power mobile applications.

5. REAL-WORLD APPLICATIONS

We have also done experiments on some real-world signal processing applications. We notice that applications dealing with the modest-resolution human sensory data can actually tolerate some computation error in the intermediate or even final results, while giving the similar results. A H.263 video decoder and the Sphinx speech recognizer, both developed in CMU, are chosen to be our benchmarks. In these applications, the variable grouping is done in a naïve way that all variables are put in one group, so that we can plot the trends of precision degradation with bit-width reduction.

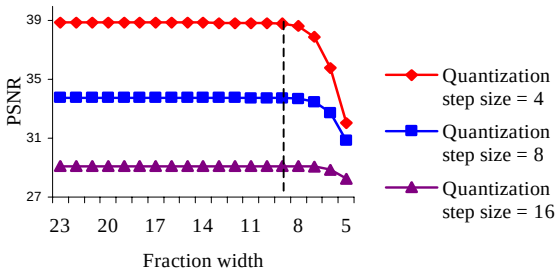

Fig. 6. Bit-width reduction for video decoder

In a H.263 video decoder, FP operations only appear in the IDCT core. We use PSNR (Peak-Signal-to-Noise-Ratio) as an objective measure for video quality. As shown in Fig. 6, the fraction bit-width can be reduced down to 9 bits without noticeable degradation of PSNR. We also note that such reduction depends on the quantization step size. The larger the step size, the fewer fraction bits are needed, because more computational error can be hiden in the quantization noise.

In the Sphinx speech recognizer, FP has much more occurrence than in the video decoder. The word recognition error rate is chosen to be the performance measure. As we can see from Fig. 7, the fraction bit-width can be cut down to 4 bits without much negative impact on the recognition rate.

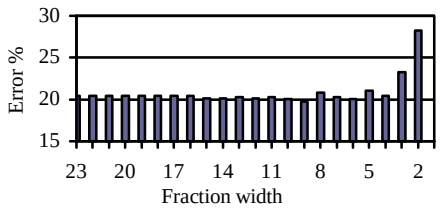

Fig. 7. Bit-width reduction for Sphinx speech recognizer

6. CONCLUSION

We proposed our lightweight FP system design flow of which the bit-width optimization engine is a core component. It takes both the hardware cost and the numerical performance into account, and finds the optimal bit-width configuration. Variable grouping enables the designer to give some hardware topology information to the optimizer so that the configuration result is easier to be mapped onto hardware design.

Experiments on IDCT demonstrate the proposed design flow, including the optimization and hardware implementation. The power consumption of our lightweight IDCT is comparable to an optimized fixed-point design.

With preliminary results in real-world applications such as video decoding and speech recognition, we believe that our lightweight FP design flow will bring more and more signal processing algorithms into low-power mobile devices. To apply variable grouping and the whole optimization algorithm to other applications is our future direction.

7. REFERENCES

[1] M.Willems, V. Bursgens, H. Keding, T. Grotker and H. Meyr, "System level fixed-point design based on an interpolative approach", DAC, 1997. Proceedings of the 34th, pp 293-298

[2] K. Kum, J. Kang and W. Sung, "AUTOSCALER for C: An optimizing FP to integer C program converter for fixed-point digital signal processors," IEEE Trans. Circuits. Syst., vol. 47, Sep.2000, pp 840-848

[3] M-A. Cantin, Y. Savaria, D. Prodanos and P. Lavoie, "An automatic word length determination method", Circuits and Systems, 2001. ISCAS, vol. 5, pp 53 -56

[4] S. Kim; K. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs",
Circuits and Systems II: Analog and Digital Signal Processing, IEEE Trans., vol 45, Nov. 1998, pp 1455-1464

[5] D.M. Samanj, J. Ellinger, E.J. Powers, E.E. Swartzlander, "Simulation of variable precision IEEE floating point using C++ and its application in digital signal processor design," Circuits and Systems, Proceedings of the 36th Midwest Symposium on, 1993, pp.1509-1514

[6] R. Ignatowski, E.E. Swartzlander, "Creating new algorithm and modifying old algorithms to use the variable precision floating point simulator," Signals, Systems and Computers, 1994 Conference Record of the Twenty-Eighth Asilomar Conference on , vol. 1 , 1994, pp152-156

[7] "IEEE-standard specifications for the implementations of 8X8 inverse discrete cosine transform," IEEE Std 1180-1990, Institute of Electrical and Electronics Engineers, Inc, 1990