Lightweight Arithmetic for Mobile Multimedia Devices

Tsuhan Chen
Carnegie Mellon University
tsuhan@cmu.edu

Thanks to Fang Fang and Rob Rutenbar

IEEE Transactions on Multimedia

EDICS
- Signal Processing for Multimedia Applications
- Components and Technologies for Multimedia Systems
- Human Factor, Interface and Interaction
- Multimedia Databases and File Systems
- Multimedia Communication and Networking
- System Integration
- Applications
- Standards and Related Issues
Multimedia Applications on Mobile Devices

- **Multimedia Processing**
  - More and more applications are ported from PCs to mobile devices
  - *Floating-point* computational intensive

- **Multimedia System Development**
  - Media designers use 32-64bit floats in C++ for algorithms
  - ASIC designers use 10-20bit fixed-point units in hardware
  - Serious design disconnect
Fixed-Point vs. Floating-Point

- **Fixed point**
  - 8 Limited dynamic range and precision
  - 8 Small, less power consumption
  - 8 From SW to HW: time-consuming and error-prone

- **Floating-point**
  - ✔ Wide dynamic range & high precision
  - ✔ Easy translation from SW to HW
  - 9 Big, power intensive

*How about make this lightweight? Don’t use more than necessary.*

What Does “Lightweight” Mean

- **Lightweight** ❍ Less bits

IEEE Standard

- FP Formats and ops for ordinary numbers
- Very small nums: denormals
- Delicate rounding modes
- ……

*Actually it’s more than this….*

*We can work on each dimension*
Lightweight Floating-Point Arithmetic

- Lightweight FP arithmetic is a *middle-ground* solution

![Diagram showing Lightweight FP between Fixed-Point and Floating-Point](image)

- Better numerical features than fixed-point
- Less complicated than IEEE FP
- Acceptable energy consumption
- Easy to prototype algorithms with
- Easy to implement into hardware

Software to Hardware Cycles

![Diagram showing media algorithms, parameters, lightweight arithmetic, C++ class, chip hardware](image)
Design Flow Comparison

Lightweight FP Design

1. C++ FP
   - 20 mins
2. Lightweight FP
3. SW simulation
   - Pass ?
4. HW design

Fixed-Point Design

1. C++ FP
   - 1 Day
2. Fixed-point
3. SW tuning
   - Pass ?
4. HW design

Design Flow Comparison

Lightweight FP Design

1. C++ FP
   - 20 mins
2. Lightweight FP
3. SW simulation
   - Pass ?
4. HW design

Fixed-Point Design

1. C++ FP
   - 1 Day
2. Fixed-point
3. SW tuning
   - Pass ?
4. HW design
IEEE Standard vs. Lightweight IP

IEEE FP Standard
- 32 / 64 bits
  - 8 / 11 bits exponent
  - 23 / 52 bits mantissa
  - 1 sign bit

Lightweight Arithmetic IP
- Fewer bits
  - Fewer bits of fraction
    - less numerical precision
  - Fewer bits of exponent
    - less dynamic range

- Specs
  - normal numbers as well as special values (infinity), edge cases (INF - INF), etc.

- Which of the special cases/numbers should be supported?

IEEE Floats vs. CMUfloats

IEEE Floats
- FP Formats and ops for ordinary numbers
- Very small nums: denormals
- Delicate rounding modes

CMUfloats
- Customizable format providing variable dynamic range and precision
  - Fraction [1, 23], exponent width [1,8]
- On-off switch for denormalization
- Multiple choices for rounding mode
  - Real-rounding / Jamming / Truncation
Rounding in CMUfloat

- We support not only IEEE rounding, but also two “quick & dirty” modes

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>Allowed Precision</th>
<th>Final, Rounded Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE Rounding</td>
<td>b2 b1 b0</td>
<td></td>
</tr>
<tr>
<td>Truncation</td>
<td>b2 b1 b0</td>
<td></td>
</tr>
<tr>
<td>Jamming</td>
<td>b2 b1 b0</td>
<td></td>
</tr>
</tbody>
</table>

- IEEE Rounding (Real rounding)
  - Achieves best results, but requires costly hardware
- Truncation
  - What most ASIC hardware designers do, for efficiency
- Jamming
  - Invented in 1940s, better than truncate, similar HW

C++ CMUfloat library

- Supported operators
  - Cmufloat double
  - float
  - int
  - short
  - +
  - ==
  - -
  - >=
  - >
  - *
  - <=
  - <
  - /
  - !=

- Other supported C++ features
  - Pointer
  - Reference
  - Array
  - Argument passing
  - I/O stream

```cpp
Cmufloat * a;
Cmufloat & a;
Cmufloat a[10][10];
func ( Cmufloat a )
cout << a;
```
C++ Cmufloat Library

- **Supported operators**

<table>
<thead>
<tr>
<th>Cmufloat</th>
<th>double</th>
<th>float</th>
<th>int</th>
<th>short</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>==</td>
<td>&gt;=</td>
<td>&gt;</td>
<td>Cmufloat</td>
</tr>
<tr>
<td>-</td>
<td>&gt;,</td>
<td>&lt;=</td>
<td>&lt;</td>
<td>float</td>
</tr>
<tr>
<td>*</td>
<td>&lt;=,</td>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td>/</td>
<td>!</td>
<td></td>
<td></td>
<td>short</td>
</tr>
</tbody>
</table>

- **Other supported C++ features**
  - Pointer
    
    ```
    Cmufloat * a;
    ```
  - Reference
    
    ```
    Cmufloat & a;
    ```
  - Array
    
    ```
    Cmufloat a[10][10];
    ```
  - Argument passing
    
    ```
    func ( Cmufloat a )
    ```
  - I/O stream
    
    ```
    cout << a;
    ```

Software Library: Advantages

- **Transparent mechanism to embed ‘Cmufloat’ in the algorithm**
  - The overall structure of the source code can be preserved
  - Minimal effort in translating standard FP to lightweight FP

```cpp
Cmufloat <14,5> a = 0.5;  // 14 bit fraction and 5 bit exponent
Cmufloat <> b= 1.5;       // Default Cmufloat is IEEE float
Cmufloat <18,6> c[2];    // Define an array
float fa;

c[1] = a + b;              // Assign the result to float
c[2] = fa + b;             // Operation between float and Cmufloat
```
Software Library: Advantages (Cont.)

- Arithmetic operators are implemented by bit-level manipulation: more precise

Our approach:
Emulates the hardware implementation exactly

```c
Add(b, c) {
    a' = b + c;
    a = round(a');
}
```

Previous approach

Built-in FP operator
Round to limited bit-width

Summary: Features Supported

- Bit widths
  - Variable from 2 bits (1 sign + 1 exp + 0 man) to 32 bits (IEEE std)

- Rounding
  - Use jamming (1.00011 rounds to 1.01)
  - Experiments show jamming is nearly as good as full IEEE rounding, always superior to truncation, yet same complexity as truncation

- Denormalized numbers
  - Not supported—our experiments on video/audio codecs suggest that denormal numbers do not improve the performance

- Exceptions
  - Support only the exceptional values for infinity, zero and NAN
  - Helps make the smaller FP sizes more robust
Hardware Library: ASIC Design Flow

- Verilog to layout flow
- Timing & area analysis
- Power analysis

![Diagram showing the ASIC Design Flow]

Lightweight FP Adders/Multipliers

- **Feature Supported**
  - Bit widths:
    - Variable from 3 bits (1 sign + 1 exp + 1 frac) to 32 bits (IEEE std)
  - Rounding:
    - Jamming / Truncation

- **Design Issues**
  - Design method
  - Subcomponent structures
    - Core integer adder structure?
    - Core shifter structure?
    - Core integer multiplier structure?
Floating Pt Adder

Blue modules have large area and/or delay

Floating Pt Multiplier

Blue modules have large area and/or delay

We see that the multiplier has less ‘over-head’ than the adder
Design Examples: Adders

<table>
<thead>
<tr>
<th></th>
<th>32-bit FP</th>
<th>20-bit FIX</th>
<th>14-bit FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area( um^2) - post layout</td>
<td>26634</td>
<td>4866</td>
<td>10096</td>
</tr>
<tr>
<td>Delay(ns) - post synthesis</td>
<td>48.95</td>
<td>2.44</td>
<td>25.77</td>
</tr>
</tbody>
</table>

Design Examples: Multipliers

<table>
<thead>
<tr>
<th></th>
<th>32-bit FP</th>
<th>20-bit FIX</th>
<th>14-bit FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area( um^2) - post layout</td>
<td>60713</td>
<td>40738</td>
<td>8851</td>
</tr>
<tr>
<td>Delay(ns) - post synthesis</td>
<td>24.14</td>
<td>22.82</td>
<td>15.89</td>
</tr>
</tbody>
</table>
Power Analysis

- IDCT in
  - 32-bit IEEE FP
  - 15-bit radix-16 lightweight FP
  - Fixed-point implementation
    - 12-bit accuracy for constants
    - Widest bit-width is 24 in the whole algorithm (not fine tuned)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Area(um²)</th>
<th>Delay(ns)</th>
<th>Power(mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE FP</td>
<td>926810</td>
<td>111</td>
<td>1360</td>
</tr>
<tr>
<td>Lightweight FP</td>
<td>216236</td>
<td>46.75</td>
<td>143</td>
</tr>
<tr>
<td>Fixed-point</td>
<td>106598</td>
<td>36.11</td>
<td>110</td>
</tr>
</tbody>
</table>

Multimedia Encoding/Decoding

We can choose how accurately we wish to decode the data
Video Codec

- H.261/263, MPEG-1/2/4, and even JPEG

Video Quality vs. Bit-width

- Use PSNR (Peak-Signal-to-Noise) to measure perceptual video quality

- CMUfloat can go very small, ~14bits
  (5 exponent + 8 fraction + 1 sign bits = 14 total bits)

Yellow pts show where PSNR decreases by 0.2dB from asymptotic value
Rounding Modes

- Compare 3 rounding modes using IDCT video streams

Comparison of Rounding Methods

Jamming is nearly as good as real rounding in precision, but as simple as truncation in hardware.

Video Demo

- IEEE double vs. variable-precision CMUfloats

Decoded with 64-bit "double" IDCT
Decoded with 14-bit "lightweight" IDCT
Decoded with 11-bit "lightweight" IDCT
Hardware Reduction Using Lightweight FP

- Comparison in Area/Delay/Power
  - 32-bit IEEE FP IDCT / 14-bit lightweight FP IDCT with Jamming rounding / 20-bit fixed point IDCT

![Bar chart comparing area, delay, and power consumption for different IDCT implementations.]

Low-Resolution Display

- Media software commonly done in full precision (32–64 bits)
  - Why do this if the display cannot handle it?
  - On a portable video player:

![Diagram showing the process of decoding and displaying a bitstream on a low-resolution display.]

- Can’t we do better than this, with smarter operators?
Low-Resolution Display (cont.)

- Simplest: needs ~20-bit lightweight floats to work
- Better: needs 16-bit lightweight floats; even just 11-bits looks decent
- Best: needs just 9-bit floats (4 fraction bits) to work just fine.

Results

- Simplest: needs ~20-bit lightweight floats to work
- Better: needs 16-bit lightweight floats; even just 11-bits looks decent
- Best: needs just 9-bit floats (4 fraction bits) to work just fine.

Encoder

Decoder

Bitstream

0–255

0–255

Simplest (dumbest):
just encode/decode as usual,
let the display "figure it out"

Better:
decode with just enough precision
so a quantizer can retrieve right
2-bit pixel values

Best:
Encode and decode with
min precision needed so
quantizer gets right pixels

Video Demo

- Full Precision (64 bit)

Using 23 bits (IEEE 1180 passed)

Using 11 bits (IEEE 1180 failed)
How About Audio?

- MPEG-1/2 Layer 3 (MP3)

- No standard tests for quality

Audio Quality

- Need to rely on subjective testing on perceptual quality
  - Mean Opinion Score (MOS)
    - From 5 “imperceptible difference” to 1 “really annoying”

- Results
  - 8 subjects. 6-bit exponent and 3~7 bit fraction
Conclusion

- Tradeoff between the “lightweight FP” and the “fixed-point”

![Diagram showing tradeoff between lightweight FP and fixed-point]

- Design time
- Hardware cost
- Numerical performance

Ongoing Work: Automatic Design Flow

- Standard C++ FP algorithm
- Bit-width optimization engine
- C++ lightweight FP algorithm with optimal bit-width
- Lightweight FP hardware design

```plaintext
main( ) {
    double x,y;
    x = 2*x + y;
    ...
}
```

```plaintext
main( ) {
    CMUfloat x,y;
    x = 2*x + y;
    ...
}
```
Recap…

- **Accomplishments**
  - C++ lightweight FP arithmetic library
  - Verilog lightweight FP arithmetic library
  - Extensive experiments on video/audio/speech

- **Is the lightweight FP solution universal?**
  - No, tradeoff between fixed-point solution and lightweight FP solution

- **Ongoing work**
  - Automatic design flow

- **Important for multimedia on low-power mobile devices**

---

**Advanced Multimedia Processing Lab**

*Please visit us at:*

http://amp.ece.cmu.edu