Lightweight Arithmetic for Mobile Multimedia Devices

Tsuhan Chen 陳祖翰
Carnegie Mellon University
tsuhan@cmu.edu

Thanks to Fang Fang and Rob Rutenbar

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 for hardware
  - Serious design disconnect
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

<table>
<thead>
<tr>
<th></th>
<th>Fixed point</th>
<th>Floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td>exp</td>
</tr>
<tr>
<td>Fraction</td>
<td></td>
<td>fraction</td>
</tr>
<tr>
<td>× Limited dynamic range and precision</td>
<td></td>
<td>✓ Wide dynamic range &amp; high precision</td>
</tr>
<tr>
<td>✓ Small, less power consumption</td>
<td>✓ Big, power intensive</td>
<td></td>
</tr>
<tr>
<td>× From SW to HW: time-consuming and error-prone</td>
<td>✓ Easy translation from SW to HW</td>
<td></td>
</tr>
</tbody>
</table>

How about make this lightweight?

Don’t use more than necessary.
What Does “Lightweight” Mean

Lightweight \( \equiv \) Less bits

Actually it’s more than this....

IEEE Standard

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

We can work on each dimension

Lightweight Floating-Point Arithmetic

- Lightweight FP arithmetic is a middle-ground solution

- 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

Media algorithms
- Parameters
- Lightweight Arithmetic
- C++ class

Chip hardware
- Lightweight FP Op
- Synthesizable Verilog
- Lightweight FP Op

Design Flow Comparison

Lightweight FP Design
- C++ FP
- Lightweight FP
- SW simulation
- Pass ?
- HW design

Fixed-Point Design
- C++ FP
- Fixed-point
- SW tuning
- Pass ?
- HW design
Design Flow Comparison

Lightweight FP Design

- C++ FP
- 20 mins
- Lightweight FP
- SW simulation
- Pass?
- HW design

Fixed-Point Design

- C++ FP
- 1 Day
- Fixed-point
- SW tuning
- Pass?
- HW design

IEEE Standard vs. Lightweight IP

IEEE FP Standard
- 32 / 64 bits
  - 8 / 11 bits exponent
  - 23 / 52 bits mantissa
  - 1 sign bit
  - Specs
    normal numbers as well as special values (infinity), edge cases (INF - INF), etc.

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

- 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

  - **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

<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>-</td>
<td>&gt;=,</td>
<td>&gt;</td>
</tr>
<tr>
<td>-</td>
<td>*</td>
<td>&lt;=,</td>
<td>&lt;</td>
<td>!=</td>
</tr>
<tr>
<td>*</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### CMUfloat double float int short

### 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;`

## 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>-</td>
<td>&gt;=,</td>
<td>&gt;</td>
</tr>
<tr>
<td>-</td>
<td>*</td>
<td>&lt;=,</td>
<td>&lt;</td>
<td>!=</td>
</tr>
<tr>
<td>*</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td></td>
<td></td>
<td></td>
<td></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;
fa = 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
```

Previous approach

```
Add( b, c) {
  a' = b + c;
  a = round (a');
}
```
Summary: Features Supported

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

- **Rounding**
  - Use jamming (1.00\texttt{11} 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

**LOGIC:**
- Synopsys DesignCompiler

**SIMULATION:**
- ModelSim

**LAYOUT:**
- Cadence Silicon Ensemble

**POWER:**
- Synopsys DesignPower

**Technology Library**
- Standard Cells, Logical, physical, and timing views

**Synopsys DesignWare**
- Basic blocks: Integer arithmetic, multiplexors, etc
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

32-bit Floating-point

14-bit Floating-pt

<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(um²) - post layout</td>
<td>26634</td>
<td>4866</td>
<td>10096</td>
</tr>
<tr>
<td>Delay(ns) - post synthesys</td>
<td>48.95</td>
<td>2.44</td>
<td>25.77</td>
</tr>
</tbody>
</table>
Adder Analysis

Fixed-point adder is very simple; floating point adder is complex

**Fixed Point**

\[
\begin{align*}
\text{aaaa} \cdot \text{aaaa} \\
\text{bbbb} \cdot \text{bbbb}
\end{align*}
\]

No rounding

If not 0, then overflow

\[
\text{pppp} \cdot \text{pppp}
\]

Final answer

**Floating Point**

\[
\begin{align*}
\text{aa} \times 2^{\text{aa}} \\
\text{bb} \times 2^{\text{bb}}
\end{align*}
\]

\[
\begin{align*}
1.\text{aa} \times 2^{\text{MAX(aa,bb)}} \\
0.00\text{bb} \times 2^{\text{MAX(aa,bb)}} \\
0.0\text{ppp} \times 2^{\text{MAX(aa,bb)}}
\end{align*}
\]

Alignments are expensive

Extra add & normalize steps

\[
\text{pp} \times 2^{\text{pp}}
\]

Final answer

Design Examples: Multipliers

32-bit Floating-point

20-bit Fixed-point

14-bit Floating-pt

<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>
Multiplier Analysis

Fixed-pt needs *more* bits to get the same dynamic range, *increasing* the size of the multiplier unit.

**Fixed Point**

\[
\begin{array}{c}
\text{aaaa} . \text{aaaa} \\
\text{bbbb} . \text{bbbb} \\
\text{0000} \text{pppp} . \text{pppppppp} \\
\end{array}
\]

If not 0, then overflow
Must round these away

**Floating Point**

\[
\begin{array}{c}
\text{aa} \times 2^{\text{aa}} \\
\text{bb} \times 2^{\text{bb}} \\
\frac{(\text{aa} \times \text{bb}) \times 2^{(\text{aa}+\text{bb})}}{\text{pp} \times 2^{\text{pp}}} \\
\end{array}
\]

Must round & normalize
Extra adder

**Radix-2 vs. Radix-16**

- Higher radix has less complexity in the shifter

  Radix-2 FP (8-bit fraction + 1 leading bit)

  Radix-16 FP (11-bit fraction)

- More fraction width increases the complexity of multiplier
- If the number of adders is much more than the number of multipliers, then radix-16 is preferred

<table>
<thead>
<tr>
<th>Radix</th>
<th>Area (um²)</th>
<th>Delay (ns)</th>
<th>Radix</th>
<th>Area (um²)</th>
<th>Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8401</td>
<td>10.21</td>
<td>2</td>
<td>7893</td>
<td>5.8</td>
</tr>
<tr>
<td>16</td>
<td>7389 (-12%)</td>
<td>8.48 (-17%)</td>
<td>16</td>
<td>11284 (+43%)</td>
<td>6.62 (+14%)</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

At the decoding end, we have a choice as to how accurately we wish to decode the data for playback.
Video Codec

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

IDCT requires floating point, and has an IEEE quality spec (1180-1990) that requires comparison against a 64-bit IEEE double implementation.

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](chart.png)

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

![Video Demo](images.png)

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

![Area vs. Delay vs. Power Comparison Graph]

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:

![Low-Resolution Display Diagram]

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

- 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.

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”

Ongoing Work: Automatic Design Flow

- Standard C++ FP algorithm
  - `main() { double x, y; x = 2*x + y; }`
- Bit-width optimization engine
  - `CMUfloat C++ class`
- C++ lightweight FP algorithm with optimal bit-width
  - `main() { CMUfloat x, y; x = 2*x + y; }`
- Lightweight FP hardware design
  - `FP arithmetic Verilog library`
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 low-power mobile devices**

---

**Advanced Multimedia Processing Lab**

Please visit us at:
http://amp.ece.cmu.edu