Scaled Fixed-Point Frequency Selective Extrapolation for Fast Image Error Concealment

Nils Genser, Jürgen Seiler, and André Kaup
{ nils.genser, juergen.seiler, andre.kaup } @ fau.de
Multimedia Communications and Signal Processing

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Outline

- Motivation
- Goal of this Work
- Basics
- Proposed Modifications
- Evaluation: Floating-Point vs. Fixed-Point Algorithm
- Conclusion and Outlook

Motivation

Fast image signal extrapolation algorithms of particular interest for a wide number of applications, e.g.,

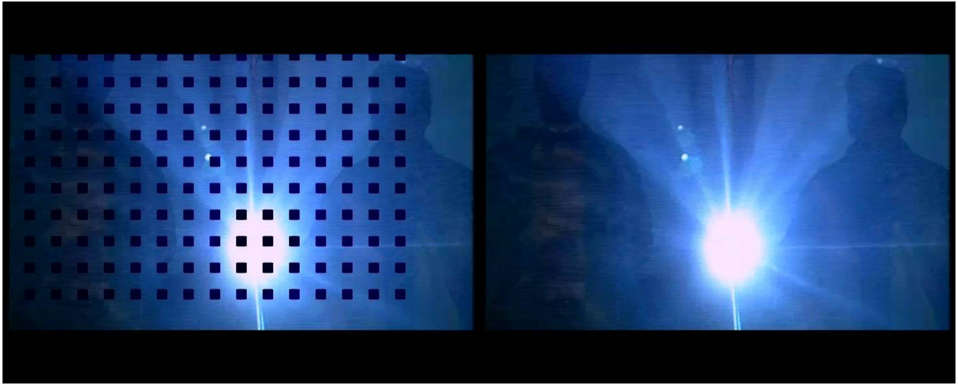
Image inpainting

Concealment of transmission errors in wireless video communication

Motivation: Image Inpainting


Original Inpainted

Motivation: Concealment of Transmission Errors



Damaged

Concealed

Goal of this Work

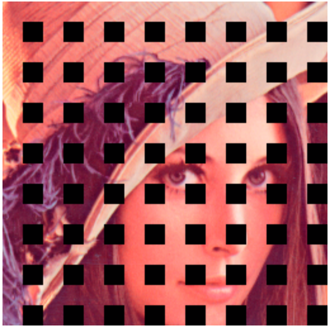
- Motivation
- Goal of this Work
- Basics
- Proposed Modifications
- Evaluation: Floating-Point vs. Fixed-Point Algorithm
- Conclusion and Outlook

Frequency Selective Extrapolation

Algorithm for high-quality extrapolation of image and video data

- Block-based, iterative method
- Model generation by superimposing weighted Fourier basis functions
- Residual error minimizing basis function selection in every step
- After finishing iterations, missing pixels extraction from model

(Seiler and Kaup 2010)



Goal of this Work

Problem:

1. Floating-point calculations computational expensive
2. Fixed-point arithmetic beneficial for several platforms, e.g., FPGAs

   (Ma, Najjar, and Roy-Chowdhury 2015)

Idea: Pure fixed-point implementation

⇒ Algorithmic adaptions required!

⇒ Adaptions usable for other algorithms as well!

Aim: Speed-up extrapolation, support new (hardware) platforms

But: Keep same reconstruction quality as state-of-the-art algorithm

Basics

- Motivation
- Goal of this Work
- **Basics**
- Proposed Modifications
- Evaluation: Floating-Point vs. Fixed-Point Algorithm
- Conclusion and Outlook

Integer and Decimal Numbers

Representation of integer numbers (e.g., as 8 bit binary numeral)

$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	0	$\mathbf{1}$	0	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

$\Rightarrow$ Adding up: $32_{10} + 8_{10} + 4_{10} + 2_{10} + 1_{10} = 47_{10}$

Representation of decimal numbers (e.g., as 8 bit fixed-point numeral)

$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	$2^{-1} = \frac{1}{2}$	$2^{-2} = \frac{1}{4}$	$2^{-3} = \frac{1}{8}$	$2^{-4} = \frac{1}{16}$
0	0	$\mathbf{1}$	0	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$

$\Rightarrow$ Adding up: $2_{10} + 0.5_{10} + 0.25_{10} + 0.125_{10} + 0.0625_{10} = 2.9375_{10}$

Fixed-Point Numbers

Fixed-point numbers understood as integers, but:

- Separation of binary numeral into pre-decimal and decimal places
- Fixed point position (location does not have to be stored)

Addition & subtraction: e.g., $1.5_{10} + 0.75_{10} = 2.25_{10}$ $\widehat{=}$

$$\begin{array}{rr} & 01.10 \\ + & 00.11 \\ \hline = & 10.01 \end{array}$$

⇒ Similar as for integers

Multiplication & division:

⇒ Additional shifts and word size casts necessary (Oberstar 2007)

Proposed Modifications

- Motivation
- Goal of this Work
- Basics
- Proposed Modifications
- Evaluation: Floating-Point vs. Fixed-Point Algorithm
- Conclusion and Outlook

Proposed Modifications

Target: Plain C algorithm

⇒ Design of fixed-point data types and operations by hand

Data types: Choice of word length ⇒ 32 vs. 64 bit?

⇒ Novel scaled algorithm to reduce required dynamic range

Operations: Addition, subtraction, multiplication, division

⇒ Own, fast fixed-point implementation

Functions: E.g., square-root and power approximation

⇒ Adaption of existing methods for problem statement

Fixed-Point Data Type

Modern PCs: Words with length of power of two and up to 64 bit (Intel 2016)

Requirement: E.g., worst-case value of basis function selection:

- $2 \cdot (2^8 \cdot \text{fft\_size})^2$, typically fft_size $= 64$
- $\log_2(2 \cdot (2^8 \cdot \text{fft\_size})^2) = 41$ bit for pre-decimal digits only

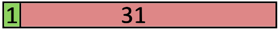
$\Rightarrow$ 64 bit words and 128 bit intermediate values for multiplication & division

Longer execution times for larger word lengths

How to decrease word length?

Fixed-Point Data Type

Solution: Scale signals to the range $[-1; +1)$



- Reduce required word length
- High accuracy instead of large dynamic range
- Multiplication can't overflow, addition and subtraction controllable

But: Algorithm and order of calculations change!

- Right order of additions, subtractions, multiplications and divisions
- Scale weighting function

$\Rightarrow$ Avoid overflows

Operations

An example: Multiplication

```
int32_t fp16_16_mul_round(int32_t a, int32_t b) {
    int64_t tmp = (int64_t)(a) * (int64_t)(b);
    return (int32_t) ((tmp + (1<<(16-1))) >> 16); // Round
}
```

Operations

An example: Multiplication

```c
int32_t fp16_16_mul_round(int32_t a, int32_t b) {
    int64_t tmp = (int64_t)(a) * (int64_t)(b);
    return (int32_t) ((tmp + (1<<(16-1))) >> 16); // Round
}
```

```c
int32_t fp16_16_mul_trunc(int32_t a, int32_t b) {
    int64_t tmp = (int64_t)(a) * (int64_t)(b);
    return (int32_t) (tmp >> 16); // Truncate
}
```

Operations

An example: Multiplication

```
int32_t fp16_16_mul_round(int32_t a, int32_t b) {
    int64_t tmp = (int64_t)(a) * (int64_t)(b);
    return (int32_t) ((tmp + (1<<(16-1))) >> 16); // Round
}
```

⇒ Slow

```
int32_t fp16_16_mul_trunc(int32_t a, int32_t b) {
    int64_t tmp = (int64_t)(a) * (int64_t)(b);
    return (int32_t) (tmp >> 16); // Truncate
}
```

⇒ Fast

```
int32_t fp16_16_mul_fast(int32_t a, int32_t b) {
    return (a >> 8) * (b >> 8); // Accept loss of accuracy
}
```

⇒ Faster

Operations: Significant amount of total calculation time

- Avoid rounding and stop word length casts

Functions

Problem: No fixed-point functions in C language

- Own implementations of suitable methods required

Power: $a^b \approx a^{\lfloor b \rfloor} = \prod_{1}^{\lfloor b \rfloor} a$, well suited as $a \ll b$ for this problem statement

Square root: $x = \sqrt{a} \quad \rightarrow \quad x_{k+1} = \frac{1}{2} \cdot \left(x_k + \frac{a}{x_k} \right)$, with $a, x_0 \in \mathbb{R}^+$

- Known as Heron's method (Kosheleva 2009)
- Typically: $x_0 = \frac{a+1}{2}$, here: $k = 6$ to achieve similar reconstruction quality

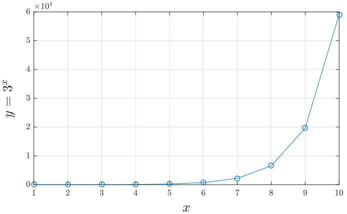
Summary

Fixed-point data types: Word length estimation, design of scaled algorithm

Operations: Own, fast implementation of addition, subtraction, etc.

⊕ ⊖ ⊙ ⊘

Functions: High-speed power and square root methods



Evaluation: Floating-Point vs. Fixed-Point Algorithm

Evaluation: Test Systems

Desktop	
CPU	i7-6700K
Speed	4.00 GHz
Cores	4
RAM	16 GB

Netbook	
CPU	Pentium N3540
Speed	2.16 GHz
Cores	4
RAM	4 GB

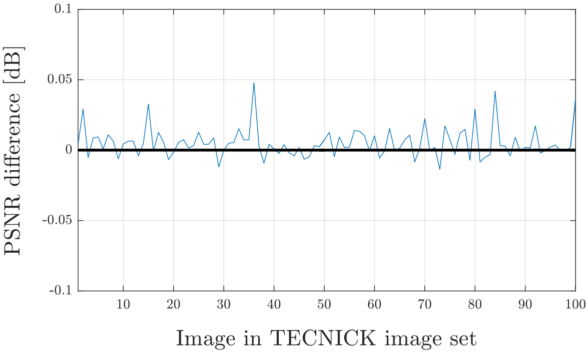
Notebook	
CPU	i7-6700HQ
Speed	2.60 GHz
Cores	4
RAM	8 GB

Zybo Zynq	
CPU	ARM Cortex-A9
Speed	1 GHz
Cores	2
RAM	512 MB

Evaluation: Reconstruction Quality

Evaluation of the state-of-the-art and the proposed algorithm:

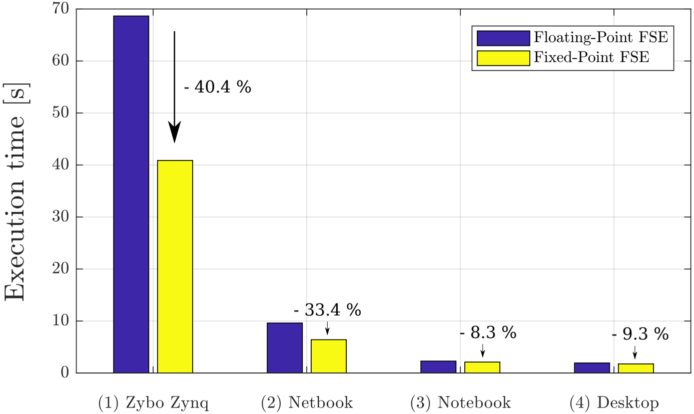
⇒ TECNICK data set, 100 images, 1200 × 1200 pixels



- PSNR deviation very small
- Slightly higher reconstruction quality as state-of-the-art approach

State-of-the-art algorithm: (Seiler and Kaup 2010)

Evaluation: Execution Time



Execution times averaged over TECNICK data set

- Speed-up platform dependent
- Up to 40.45 % faster than state-of-the-art method

State-of-the-art algorithm: (Seiler and Kaup 2010)

Conclusion and Outlook

- Motivation
- Goal of this Work
- Basics
- Proposed Modifications
- Evaluation: Floating-Point vs. Fixed-Point Algorithm
- Conclusion and Outlook

Conclusion and Outlook

Scaled fixed-point Frequency Selective Extrapolation:

- Fast and high quality method for error concealment of image & video data
- Use of fixed-point arithmetic to fasten extensive calculations
- Execution time decreases on average by 22.84 %, at best by up to 40.25 %
- Same reconstruction quality as floating-point algorithm

Outlook: Explore new (hardware) platforms

Fixed-point method required for implementations, e.g, on FPGAs

⇒ Investigate the suitability of the proposed algorithm!

References I

Intel (2016). *6th Generation Intel Processor Datasheet for S-Platforms*. Datasheet, Volume 1 of 2. Intel.

Kosheleva, O. (2009). "Babylonian method of computing the square root: Justifications based on fuzzy techniques and on computational complexity". In: *NAFIPS 2009 - 2009 Annual Meeting of the North American Fuzzy Information Processing Society*, pp. 1–6. DOI: 10.1109/NAFIPS.2009.5156463.

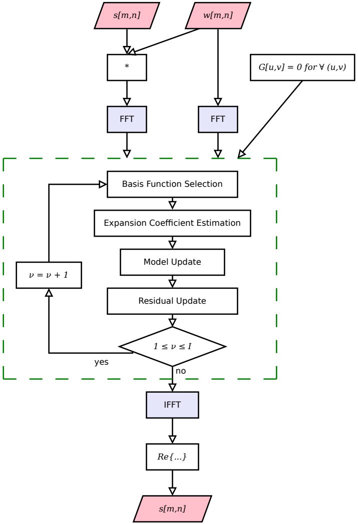
Ma, X., W. A. Najjar, and A. K. Roy-Chowdhury (2015). "Evaluation and Acceleration of High-Throughput Fixed-Point Object Detection on FPGAs". In: *IEEE Transactions on Circuits and Systems for Video Technology* 25.6, pp. 1051–1062. ISSN: 1051-8215. DOI: 10.1109/TCSVT.2014.2360030.

Oberstar, E. L. (2007). *Fixed-Point Representation & Fractional Math*. Oberstar Consulting.

References II

Seiler, J. and A. Kaup (2010). “Complex-Valued Frequency Selective Extrapolation for Fast Image and Video Signal Extrapolation”. In: *IEEE Signal Processing Letters* 17.11, pp. 949–952.

Overview of Frequency Selective Extrapolation



Evaluation Parameters

Parameter	**Value**
Number of iterations	100
FFT size	64
Block size	16
Border width	8
Decay $\hat{\rho}$	0.8
Orthogonality compensation γ	0.5
Weight of already concealed pixels	0.2

Fixed-Point Multiplication

Multiplication and division: Additional operations necessary

Example: `1.5 * 2.0 = 3.0`

$1.5_{10} \mathrel{\widehat{=}} 01.10$ resp. $2.0_{10} \mathrel{\widehat{=}} 01.00$

$$
\begin{array}{rcl}
0110 & * & 1000 \\
\hline
011 & & 0 \\
00 & & 00 \\
0 & & 000 \\
 & & 0000 \\
\hline
0011 & & 0000
\end{array}
$$

Result: Twice the desired width, value $\widehat{=}\ 48_{10}$

⇒ Shift two digits to right and cut first ones: $11.00 \mathrel{\widehat{=}} 3.0_{10}$

Fixed-Point Multiplication

Multiplication and division: Additional operations necessary

Example: `1.5 * 2.0 = 3.0`

$1.5_{10} \mathrel{\widehat{=}} 01.10$ resp. $2.0_{10} \mathrel{\widehat{=}} 01.00$

$$\begin{array}{r l}
0110 & * \; 1000 \\
\hline
011 & 0 \\
00 & 00 \\
0 & 000 \\
 & 0000 \\
\hline
0011 & 0000
\end{array}$$

Result: Twice the desired width, value $\widehat{=} 48_{10}$

⇒ Shift two digits to right and cut first ones: $11.00 \mathrel{\widehat{=}} 3.0_{10}$

Explanation:

(Here: Four bits, unsigned, two pre-decimal digits, two decimal digits)

1. Multiplication of digits (increase word length)
2. Addition of intermediate values
3. Shift to right (by the number of decimal digits)
4. Cut to original word length (from ahead)

⇒ **Additional shifts and word length casts**

⇒ **Increased computation time compared to integer multiplication and division**