

CO405H

Computing in Space with OpenSPL

Topic 11: Numerics

Oskar Mencer

Georgi Gaydadjiev

Department of Computing
Imperial College London

<http://www.doc.ic.ac.uk/~oskar/>

<http://www.doc.ic.ac.uk/~georgig/>

CO405H course page:

WebIDE:

OpenSPL consortium page:

<http://cc.doc.ic.ac.uk/openspl15/>

<http://openspl.doc.ic.ac.uk>

<http://www.openspl.org>

o.mencer@imperial.ac.uk

g.gaydadjiev@imperial.ac.uk

Lecture Overview

- Numerics: why we care
- Number representation and types for DFEs
- Rounding, Rounding, Rounding,...
- Arithmetic styles
- Error and other numerical issues
- Distribution of Variable Values
- Seismic Imaging and Weather Simulation examples
- Representation of Variables in Spatial Computing

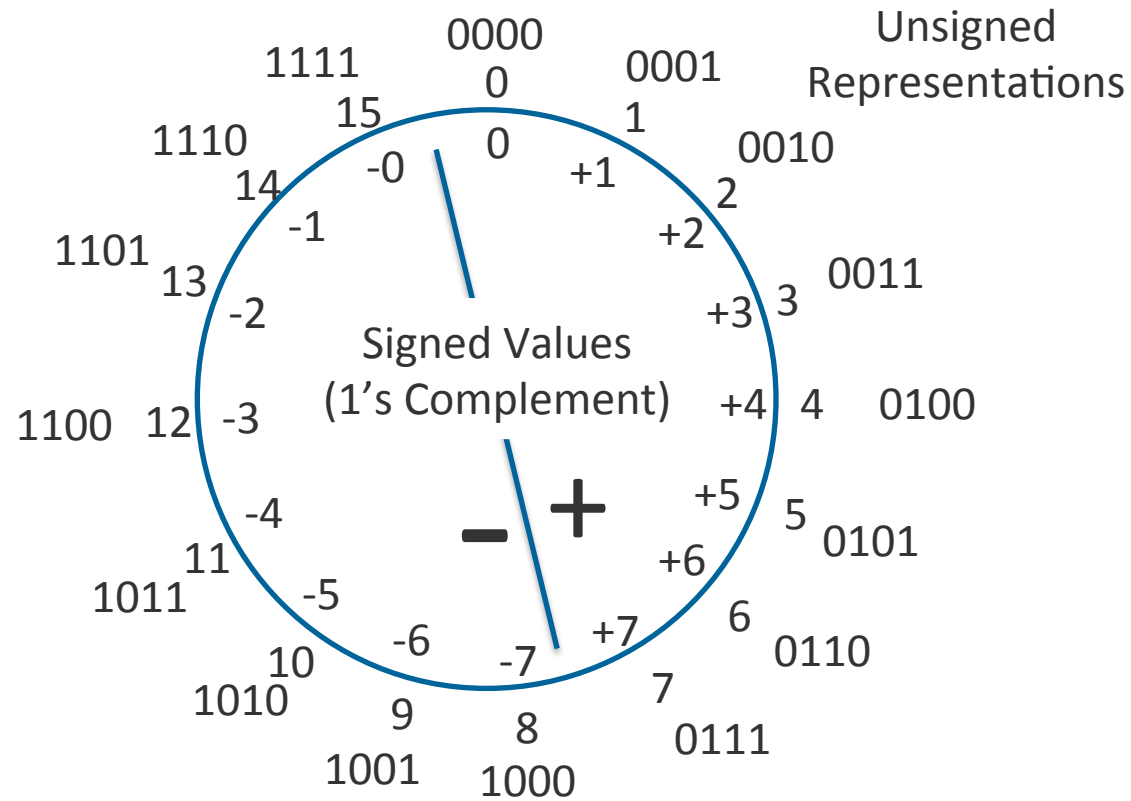
Why we care about Numerics

- Performance depends on the number of arithmetic units that fit in space on the DFE
→ lower precision → more units → higher performance
- Accuracy and performance may be competing objectives
 - DFE space depends on data representation and bitwidths
- DFEs give us control over number representation, to achieve just-enough accuracy in minimal space

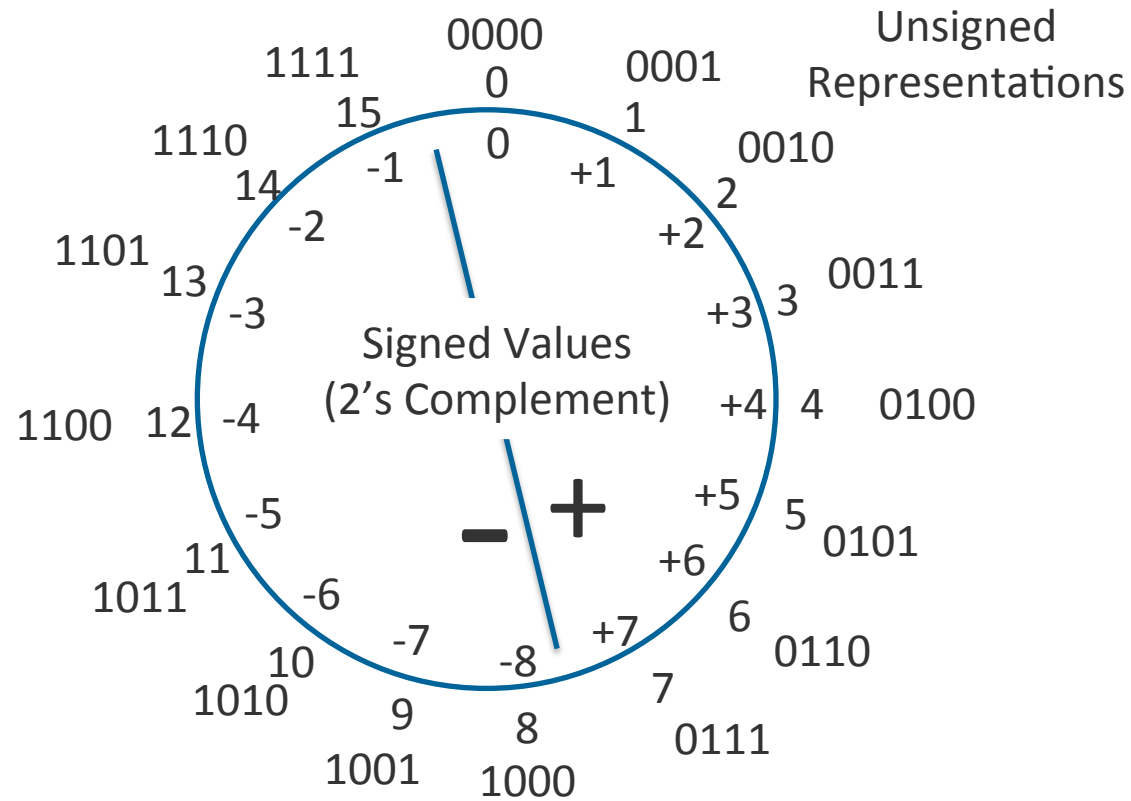
Number Representations

- **Microprocessors:**
 - Integer: unsigned, one's complement, two's complement,
 - Floating Point: IEEE single-precision, double-precision
- **Others:**
 - Fixed point
 - Logarithmic number representation
 - Redundant number systems: use more bits, compute faster
 - Signed-digit representation
 - Residue number system (modulo arithmetic)
 - Decimal: decimal floating point, binary coded decimal

One's Complement



Two's Complement

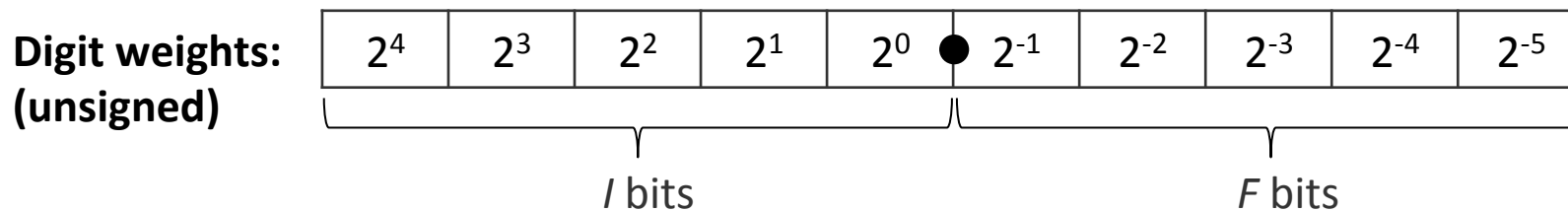


Signed N-bit Integers

- Sign-magnitude representation for integer x .
Most significant bit (msb) is the sign bit.
Advantages: symmetry around 0, easy access to $|x|$, simple overflow detection
Disadvantages: complexity of add/sub.
- One's complement numbers:
Represent $-x$ by inverting each bit.
Overflow=Sign_Carry_In ^ Sign_Carry_Cout
Advantages: fast negation
Disadvantages: add/sub correction: carry-out of sign bit is added to lsb
- Two's complement:
Represent $-x$ by inverting each bit and adding '1'.
Overflow=same as One's c.
Advantages: fast add/sub
Disadvantages: magnitude computation requires full addition

Fixed Point Numbers

- Generalisation of integers, with a 'radix point'
- Digits to the right of the radix point represent negative powers of 2



- F = number of fractional bits
 - Bits to the right of the 'radix point'
 - For integers, $F = 0$

Floating Point Representation

$$\text{sign} \cdot | \text{mantissa} | \cdot \text{base}^{\text{exponent}}$$

- regular mantissa = 1.xxxxxx
- denormal numbers get as close to zero as possible:
mantissa = 0.xxxxxx with min exponent
- IEEE FP Standard:
base=2, single, double, extended widths
- Computing in Space:
choose widths of fields + choose base
- Tradeoff:
 - **Performance**: small widths, larger base, truncation.
 - versus **Accuracy**: wide, base=2, round to even.
- Disadvantage: Floating Point arithmetic units tend to be very large compared to Integer/Fixed Point units.

Floating Point Maths

- Addition and subtraction:
 - Align exponents:
shift smaller mantissa to the larger number's exponent
 - Add mantissas
 - Normalize:
shift mantissa until starts with '1', adjust exponent
- Multiplication:
 - Multiply mantissas, add exponents
- Division:
 - Divide mantissas, subtract exponents

Number Representation for DFEs

- MaxCompiler has in-built support for floating point and fixed point/integer arithmetic
 - Depends on the *type* of the DFEVar
- Can type inputs, outputs and constants
- Or can *cast* DFEVars from one type to another
- Types are Java objects, just like DFEVars,

```
// Create an input of type t
DFEVar io.input(String name, DFEType t);

// Create an DFEVar of type t with constant value
DFEVar constant.var(DFEType t, double value);

// Cast DFEVar y to type t
DFEVar x = y.cast(DFEType t);
```

DFE Floating Point - dfeFloat

- Floating point numbers with base 2, flexible exponent and mantissa
- Compatible with IEEE floating point **except** does not support denormal numbers
 - When Computing in Space you can use a larger exponent

```
DFEType t = dfeFloat(int exponent_bits, int mantissa_bits);
```

- Examples:

↑
Including the sign bit

	Exponent bits	Mantissa bits
IEEE single precision	8	24
IEEE double precision	11	53
DFE optimized low precision	7	17

Why dfeFloat(7,17)...?

DFE Fixed Point – dfeFixOffset

- Fixed point numbers
- Flexible integer and fraction bits
- Flexible sign mode
 - SignMode.UNSIGNED or SignMode.TWOSCOMPLEMENT

```
DFEType t = dfeFixOffset(int num_bits, int offset, SignMode sm);
```

- Common cases have useful aliases

	Integer bits	Fraction bits	Sign mode
dfeInt(N)	N	0	TWOSCOMPLEMENT
dfeUInt(N)	N	0	UNSIGNED
dfeBool()	1	0	UNSIGNED

Mixed Types

- Can mix different types in a MaxCompiler kernel to use the most appropriate type for each operation
 - Type conversions costs area – must cast manually
- Types can be parameter to a kernel program
 - Can generate the same kernel with different types

```
class MyKernel extends Kernel {
    public MyKernel(KernelParameters k, DFEType t_in, DFEType t_out)
    {
        super(k);

        DFESVar p = io.input("p", dfeFloat(8,24));
        DFESVar q = io.input("q", t_in);

        DFESVar r = p * p;

        DFESVar s = r + q.cast(r.getType());
        io.output("s", s.cast(t_out), t_out);
    }
}
```

Rounding

- When we remove bits from the RHS of a number we may want to perform *rounding*.
 - Casting / type conversion
 - Inside arithmetic operations
- Different possibilities
 - TRUNCATE: throw away unwanted bits
 - TONEAR: if ≥ 0.5 , round up (add 1)
 - TONEAREVEN: if > 0.5 round up, if < 0.5 round down, if $= 0.5$ then round to the nearest even number
- Lots of less common alternatives:
 - Towards zero, towards positive infinity, towards negative infinity, random....
- Very important in iterative calculations – may affect convergence behaviour

Rounding in MaxCompiler

- Floating point arithmetic uses TONEAREVEN
- Fixed point rounding is flexible, controlled by the *RoundingMode*
 - TRUNCATE, TONEAR and TONEAREVEN are in-built

```
DFEVar z;  
...  
optimization.pushRoundingMode (RoundingMode.TRUNCATE);  
  
z = z.cast(smaller_type);  
  
optimization.popRoundingMode ();
```

Arithmetic Styles

	Digit-Serial	Digit-Parallel
Sequential - loop x times		
Pipelined - loop unrolled		
Combinational - loop unrolled - no registers - logic min		

Arithmetic in MaxCompiler

- By default uses deeply pipelined arithmetic functions
 - Objective is high operating frequency
- Can reduce pipelining gradually to produce combinatorial functions, controlled by *pushing* and *popping* a “pipelining factor”
 - 1.0 = maximum pipelining ; 0.0 = no pipelining

```
DFEVar x, y, z; // floating point numbers
...
z = x * y; // fully pipelined
optimization.pushPipeliningFactor(0.5);
z += x; // half pipelined - lower latency
optimization.pushPipeliningFactor(0.0);
z += y; // no pipelining
optimization.popPipeliningFactor();
optimization.popPipeliningFactor();
z = z * 2; // fully pipelined again
```

Arithmetic takes Space on the DFE

- Addition/subtraction:
 - ~1 logic cell/bit for fixed point, while it takes hundreds of logic cells per floating point op
- Multiplication: Can use MULT blocks
 - 18x25bit multiply on Maxeler Vectis DFEs
 - Number of MULTs depends on total bits (fixed point) or mantissa bitwidth (floating point)

Approximate space cost models

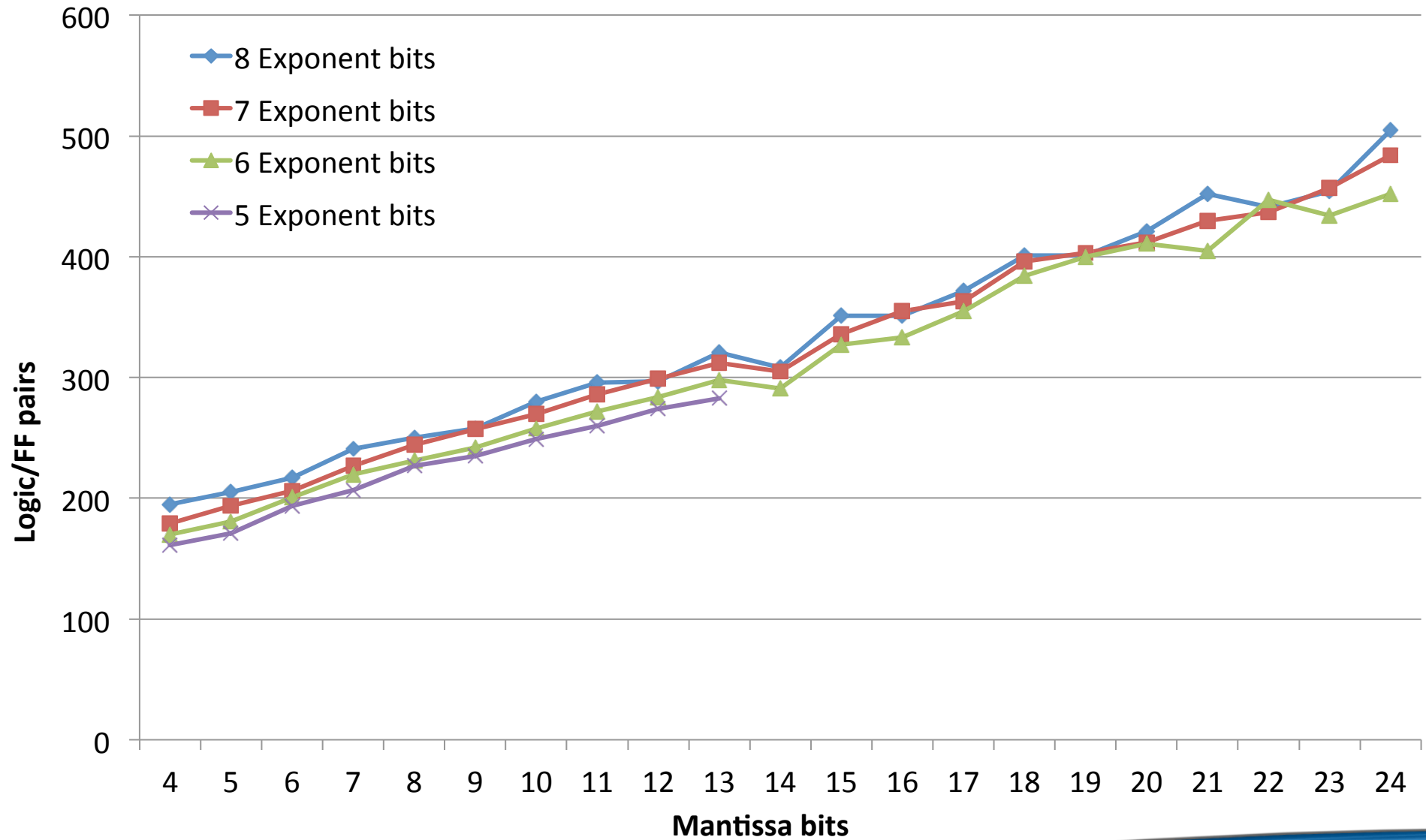
	Floating point: $dfeFloat(E, M)$		Fixed point: $dfeFix(I, F, TWOSCOMP)$	
	MULTs	LUTs	MULTs	LUTs
Add/subtract	0	$O(M \times \log_2(E))$	0	I+F
Multiply	$O(\text{ceil}(M/18)^2)$	$O(E)$	$O(\text{ceil}((I+F)/18)^2)$	0
Divide	0	$O(M^2)$	0	$O((I+F)^2)$

I = Integer bits, F = Fraction bits. E = Exponent bits, M = Mantissa Bits

MULT usage for N x M multiplication

		M →																			
N ↓	Bits	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	
	18	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3
	20	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3
	22	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3
	24	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3
	26	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6
	28	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6
	30	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6
	32	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6
	34	2	2	2	2	4	4	4	4	4	4	4	4	4	6	6	6	6	6	6	6
	36	2	3	3	3	4	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
	38	2	3	3	3	4	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
	40	2	3	3	3	4	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
	42	2	3	3	3	4	4	4	4	4	4	5	5	5	5	6	6	6	6	6	7
	44	3	3	3	3	6	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
	46	3	3	3	3	6	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
	48	3	3	3	3	6	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
	50	3	3	3	3	6	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
	52	3	3	3	3	6	6	6	6	6	6	6	6	6	6	9	9	9	9	9	9
	54	3	4	4	4	6	6	6	6	6	6	7	7	7	7	9	9	9	9	9	10

Logic usage for floating point addition



Example: Quad Precision Floating Point

Quad Precision (112 bit mantissa, 15 bit exponent)

One Maxeler Maia DFE = 21.4 GFLOP/s

1U MPC-X2000 with 8 Maias = **171.2 GFLOP/s**.

1U server node of Sandybridge 16-core is estimated to run at **1.05 GFLOP/s**.

1U to 1U: ~160x.

Benefits of Fixed Point

- Consider fixed point compared to single precision floating point
- If range is tightly confined, we could use *24-bit* fixed point
- If data has a wider range, may need *32-bit* fixed point

	dfeFloat(8,24)	dfeFixOffset(24,..)	dfeFixOffset(32,..)
Add	500 logic cells	24 logic cells	32 logic cells
Multiply	2 MULTs	2 MULTs	4 MULTs

- Arithmetic is not 100% of the chip. In practice, often see ~5x performance boost from fixed point.

Error

- $\forall A, B: \mathfrak{R}. A \text{ (op) } B = \text{result} + \text{error}$
- Floating point *introduces (dynamic) relative error*
 - Error = f(exponent of result) \rightarrow relative error
- Fixed point *introduces (static) absolute error*
 - Error = f(rightmost bit position of result) \rightarrow static error
- Error is minimized by thoughtful rounding

Other numerics issues

- **Overflow**
 - Number is too large (positive or negative) to be represented
 - Usually catastrophic – important data is lost/invalid
- **Underflow**
 - Number is too small to be represented and rounds to zero
 - With fixed point, happens gradually
 - With floating point without denormals, happens suddenly
 - Usually underflowing data is not so important (numbers are very small)
- **Bias**
 - If errors do not have a mean of zero, they will grow with repeated computation.
 - Big issue in iterative calculations
 - numbers gradually get more and more wrong!
 - **TONEAREVEN** rounding mode minimizes bias

Numerics II: Variables, Values and Distribution

STEP 1: For all variables in the application which are moving to the DFE, record all values of each particular variable.

STEP 2: look at histogram, distribution, average, min, max, variance, etc.

STEP 3: pick representation to match range and distribution of values for the variable.

Ex 1: Number Representations in Geoscience

Case Study: Complex Exponential in Downward Continued Based Migration

$$U(w, k_s, k_g, z + \Delta z) = \exp[-i w v (\sqrt{1 - \frac{v k_g}{w}} + \sqrt{1 - \frac{v k_s}{w}})] \cdot U(w, k_s, k_g, z)$$

Double Square Root (DSR) condition (the complex exponential step) with frequency w

- a small table holds the vk/w values
- next an approximate value of the square root is looked up

The design consists of three parts:

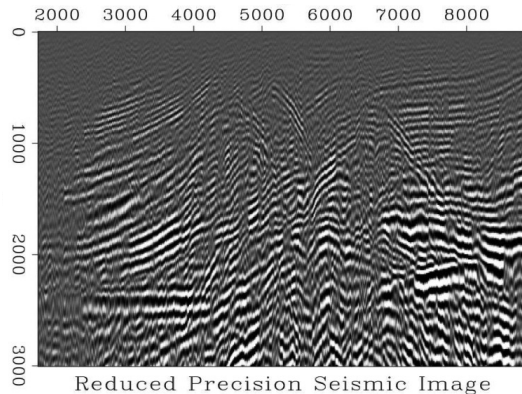
- *square root calculation*
- *sine/cosine evaluation, and*
- *complex multiplication.*

The above three parts have quite different range of variables → different bit-widths in each part: SQRT (square root), SINE (sine/cosine evaluation) and WMUL (wave-field complex multiplication) bit-width.

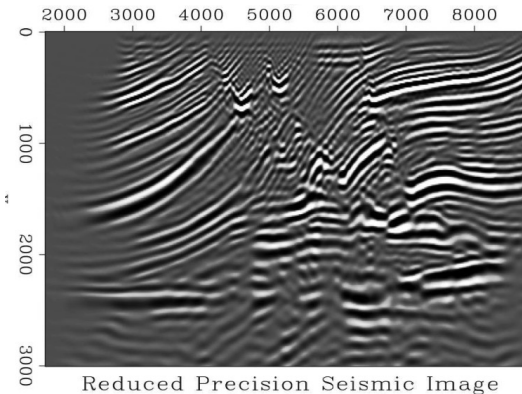
[H.Fu, W. Osborne, R. G. Clapp, O. Pell, Accelerating Seismic Computations [], 70th EAGE Conference, Italy, 2008]

Fixed-point bit-width exploration

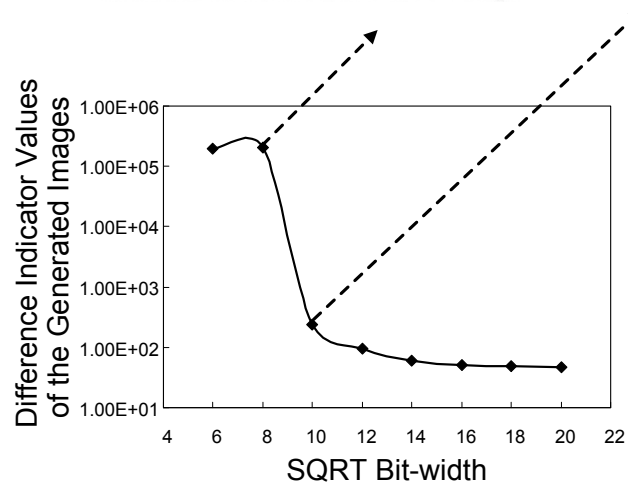
8-bit fixed-point



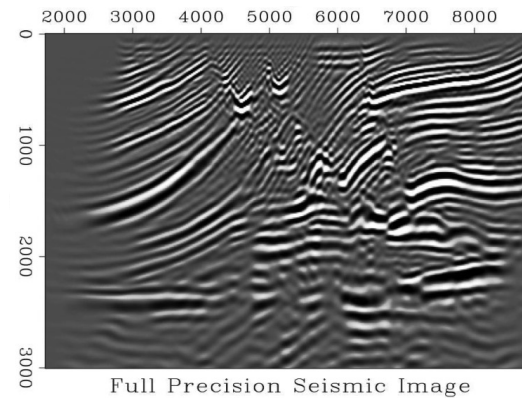
10-bit fixed-point



Different parts are explored *separately*, i.e., when we investigate one part, we keep the bit-widths in other parts a constant high value



'true' image: single-precision floating-point

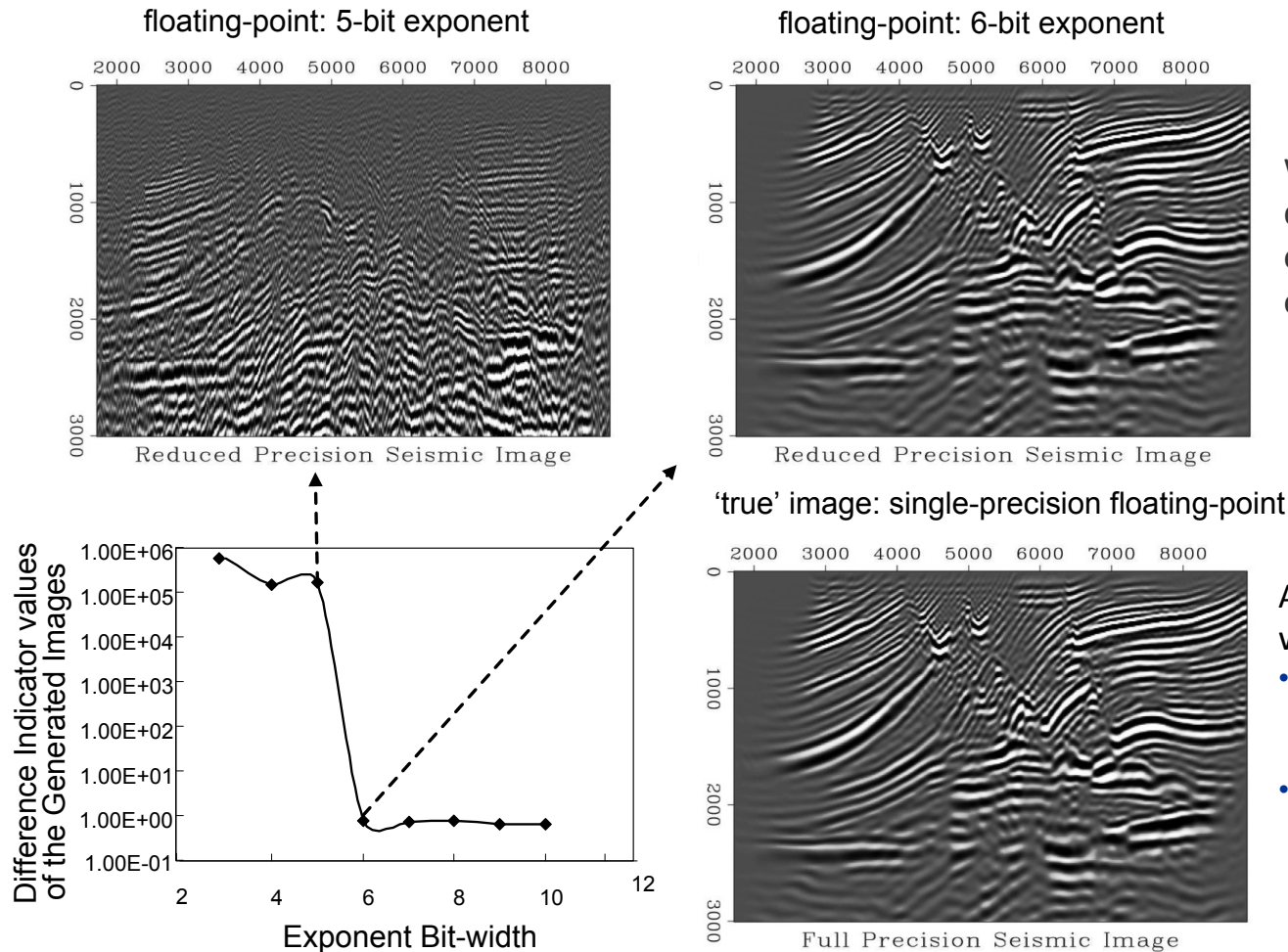


Similarly, we observe a significant drop of the error when the SQRT bit-width increases from 8 to 10

Similar precision thresholds observed in both synthetic and field results. This behavior enables an automatic tool to determine the minimum precision that still keeps the result *good enough*

[H.Fu, W. Osborne, R. G. Clapp, O. Pell, Accelerating Seismic Computations [], 70th EAGE Conference, Italy, 2008]

Floating-point bit-width exploration



We use the Marmousi synthetic data set as the test data, and explore different combinations of exponent and mantissa bit-width

A precision threshold at exponent width of 6 bits:

- The error drops significantly when we increase the exponent width from 5 bits to 6 bits
- The image also turns from nearly random noise at 5 bits, to almost identical to the 32-bit image at 6 bits

[H.Fu, W. Osborne, R. G. Clapp, O. Pell, Accelerating Seismic Computations [], 70th EAGE Conference, Italy, 2008]

Acceleration results

- Based on exploration results, we use 12, 16, and 16-bit fixed-point numbers for the SQRT, SINE and WMUL parts

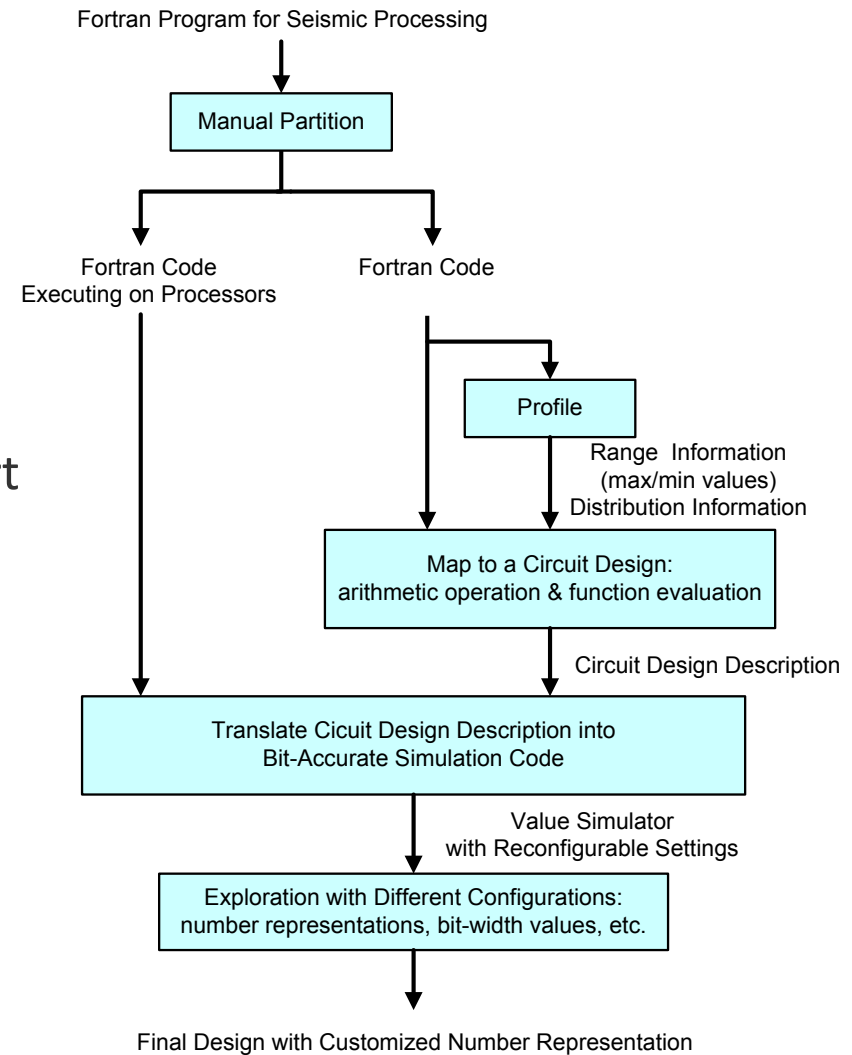
size of data set	processing time		speedup
	software	DFE	
43056	5.32 ms	0.84 ms	6.3
216504	26.1 ms	3.77 ms	6.9

- Implemented on Maxeler Vectis DFE with 192 MULTs
- The design consumes 28% of logic elements, 15% of FMEM, 10% of MULTs. Mapping 6 cores into the card (with additional effort) would provide up to 40x of node-to-node acceleration.

[H.Fu, W. Osborne, R. G. Clapp, O. Pell, Accelerating Seismic Computations [], 70th EAGE Conference, Italy, 2008]

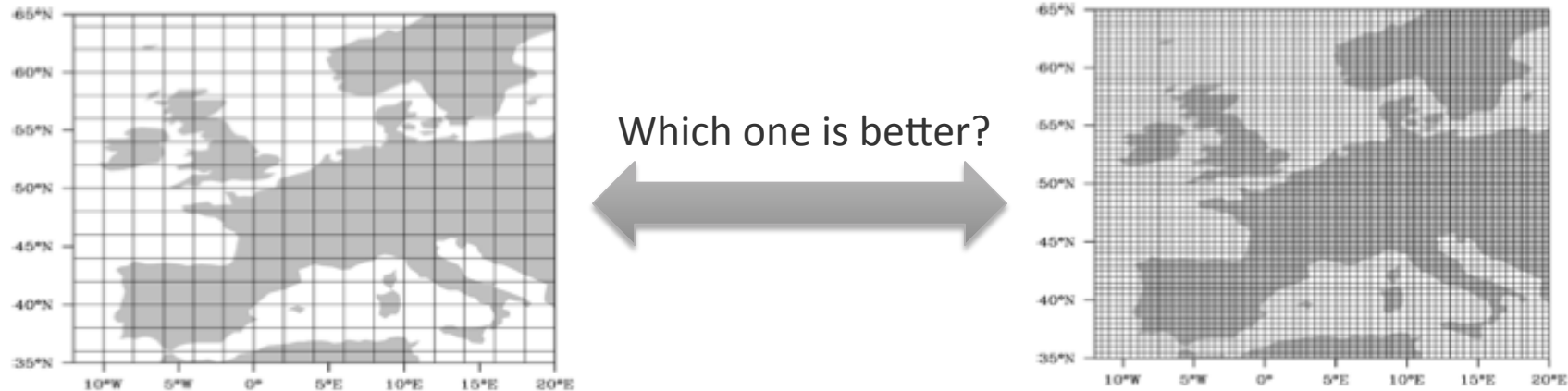
Tool for adapting number presentations

- Partition the program into software and hardware part
- Profile the variables in the hardware part
- Map the dataflow part to the DFE
- Explore the design space with different precisions



[H.Fu, W. Osborne, R. G. Clapp, O. Pell, Accelerating Seismic Computations [], 70th EAGE Conference, Italy, 2008]

Example 2: Weather / climate models on DFEs



Finer grid and higher precision are obviously preferred but the computational requirements will increase → Power usage → \$\$

What about using reduced precision? (15 bits instead of 64 double precision FP)

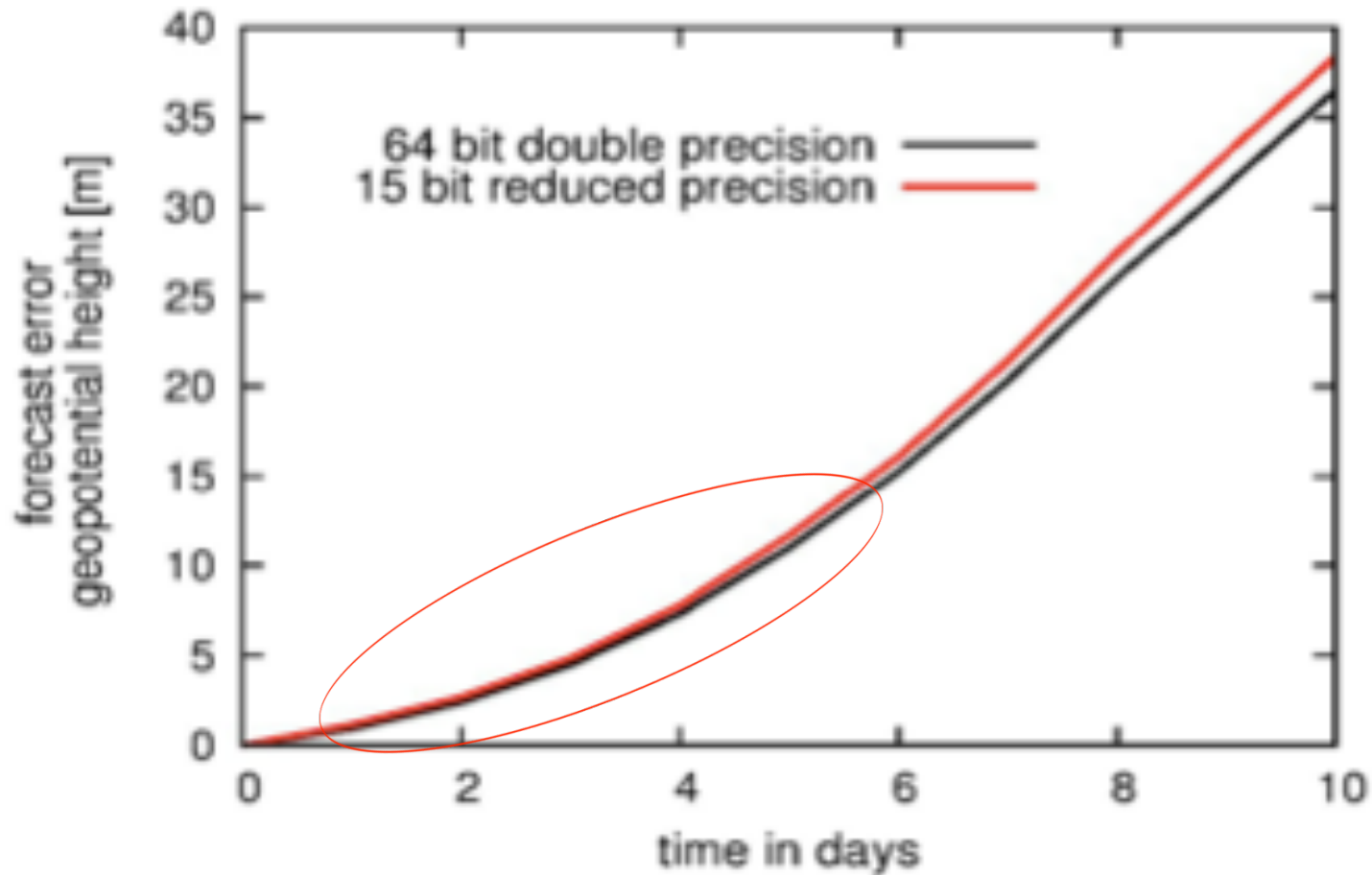


We use only **15 bits** for 98% of the computation:



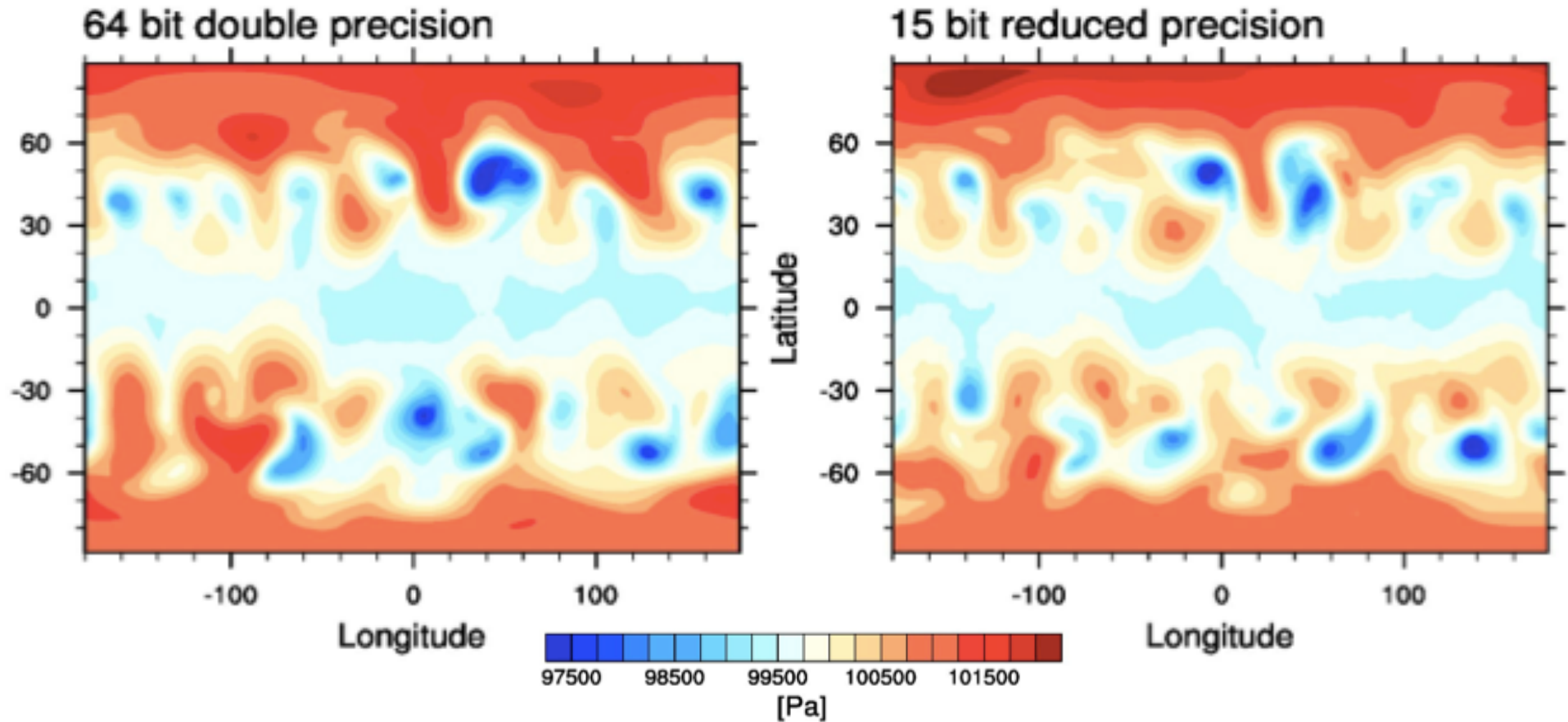
[P. Düben, T. Palmer, Oxford, OCCAM, "Weather and climate models on dataflow engines," CeBIT, 2013]

Weather models precision comparison



[P. Düben, T. Palmer, Oxford, OCCAM, "Weather and climate models on dataflow engines," CeBIT, 2013]

What about 15 days of simulation?



Surface pressure after **15 days** of simulation for the double precision and the reduced precision simulations (quality of the simulation hardly reduced)

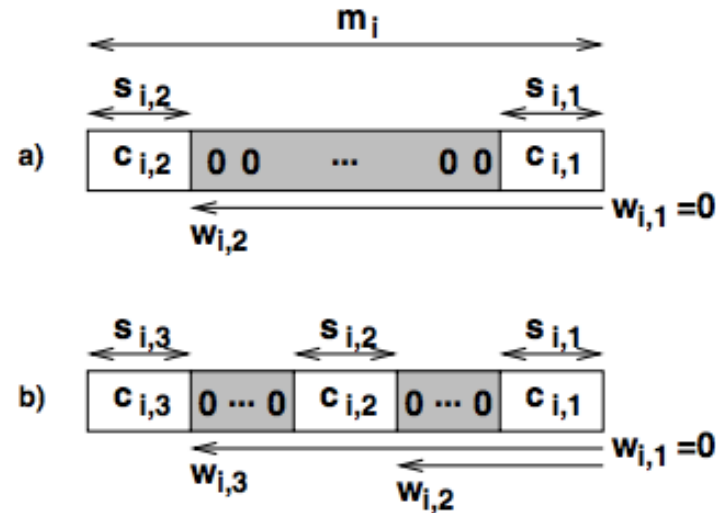
[P. Düben, T. Palmer, Oxford, OCCAM, "Weather and climate models on dataflow engines," CeBIT, 2013]

More Advanced Number Representations

- Integer: unsigned, signed, one's/ two's complement
- Floating Point: single, double precision (also custom)
- Fixed point: (any custom radices / bitwidths)
- Logarithmic number representation
- Redundant number systems: use more bits
 - Signed-digit representation
 - Residue number systems (modulo arithmetic)
 - Decimal: decimal floating point, binary coded decimal
- Decimal: decimal floating point, binary coded decimal
- *(all more exotic application specific representations)*
- Static and dynamic adaptations
- OpenSPL / MaxJ supports all of the above

Advanced Optimisation for Number Representation

Minimise '1's in Polynomial Coefficients



$$p = \frac{32799}{32768} - \frac{609}{32768}x - \frac{14881}{32768}x^2.$$

$$p = -\frac{75}{32768} + \frac{34538}{32768}x - \frac{6169}{32768}x^2.$$

$$p = \frac{32793}{32768} + \frac{31836}{32768}x + \frac{21146}{32768}x^2.$$

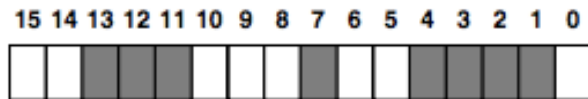


Fig. 2. Target format for cos function.

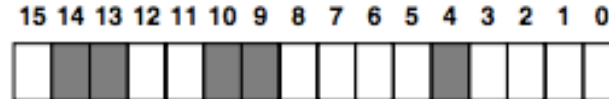


Fig. 3. Target format for sin function.

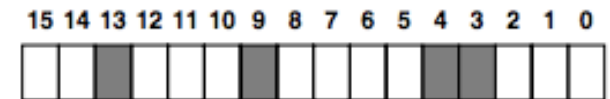


Fig. 4. Target format for exp function.

Nicolas Brisebarre, Jean-Michel Muller and Arnaud Tisserand
[Sparse Coefficient Polynomial Approximations for Hardware Implementation](#),
 Asilomar Conference, 2004.

Conclusions

- Understand your data requirements
- Know what fixed or floating point you need
- Mind rounding
- Number representations affect the space of computation as well as time!

Further Reading on Computer Arithmetic

- Recommended reading:

- Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, ACM Computing Surveys, March 1991

- Textbooks:

- Koren, “Computer Arithmetic Algorithms,” 1998.
- Pahrani, “Computer Arithmetic: Algorithms and Hardware Designs,” Oxford University Press, 2000.
- Waser, Flynn, “Introduction for Arithmetic for Digital Systems Designers,” Holt, Rinehard & Winston, 1982.
- Omondi, “Computer Arithmetic Systems,” Prentice Hall, 1994.
- Hwang, “Computer Arithmetic: Principles, Architectures and Design,” Wiley, 1978.

Practice Exercises

1. Write a MaxCompiler kernel that takes one `dfeFloat(8, 24)` input stream and adds it to a `dfeFloat(11, 53)` input stream to produce a `dfeFloat(11, 53)` result.
2. What will be the result of trying to represent $X=2^{32}$ and $Y=2^{-2}$ in each of the following number types:

```
dfeFixOffset(32, 0, SignMode.UNSIGNED)
dfeFixOffset(32, 0, SignMode.TWOSCOMPLEMENT)
dfeFixOffset(28, 4, SignMode.UNSIGNED)
dfeFixOffset(32, 4, SignMode.UNSIGNED)
dfeFloat(11, 53)
dfeFloat(8, 24)
dfeFloat(8, 32)
dfeFloat(8, 33)
```

3. Construct a test to show the difference between rounding modes on a multiplication operation of two `dfeFixOffset(4, 4, SignMode.TWOSCOMPLEMENT)` numbers. Vary the number of fraction bits – what is the impact on the bias difference between TONEAR and TONEAREVEN and why?