arm

Arm Research Summit

Fast & Correct Floating-Point Addition with HPA

Neil Burgess

Cambridge

17 September 2018

What's the Problem?

Supercomputing in Science, Engineering and Finance: "Big Data"

The Need for Speed…

Parallelism in HPC important for:

- High-Performance
  - Large Problems
  - New Science
- Multi-core processing
  - Algorithm Partitioning
  - Opportunistic Processing

arm

Floating-point accumulation gives unreliable answers

Round-off errors

Experiment: add up array of n=16,320 positive FP32 numbers (exponent = $2^{-64} – 2^{+64}$)

Example: FP64 correct result = $1.11b58c2bf8e8c \times 2^{70}$ (“double-double” arithmetic)

```
res0 = +1.00010001101101011000110000101011111110001110100011100
```

Theory says #bits in error $\cong \log_2(\sqrt{n})$: n = 16,320 $\cong 2^{14}$ -> approx. 7 bits in error

Six accumulations into FP32 destination using different strides through array

```
err1 = -0.00000000000000000010010
err2 = -0.00000000000000000101010
err3 = -0.00000000000000000011010
err4 = -0.00000000000000000011000
err5 = -0.00000000000000000100010
err6 = -0.00000000000000000011100
avge = -0.00000000000000000011100
```

All errors negative (i.e. result is too small) due to many input numbers being too small to cause a rounding up

arm

Floating-point accumulation gives unreliable answers

Round-off errors & cancellations

Experiment: add up array of *n*=16,320 positive ***& negative*** FP32 numbers (ill-conditioned)

Cancellations cause round-off errors to shift leftwards

Six accumulations into FP32 destination using different strides through array

```
      err1 = -0.000000000100111110100100
      err2 = -0.000000000100011001100110
      err3 = +0.000000000101111110011000
      err4 = +0.000000000011111110000101
      err5 = -0.000000000100110101101100
      err6 = +0.000000000010101101101110
     |avge| = 0.000000000100011101101111
```

Half the bits are unreliable: at exascale, the full-length result would be nonsense

arm

Higher precision FP

Can minimize non-reproducibility & rounding error by using higher precision

- Motivation for e.g. “double double” arithmetic (in absence of h/w support for quad precision)

Six accumulations into ***FP64 destination*** using different strides through array

```
      err1 = -0.0000000000000000000000 0000000000000001100110101011000
      err2 = +0.0000000000000000000000 0000000000000001011001010101101
      err3 = -0.0000000000000000000000 0000000000000001100100010110
      err4 = +0.0000000000000000000000 0000000000000001101000000010101
      err5 = -0.0000000000000000000000 0000000000000001110110011100
      err6 = -0.0000000000000000000000 0000000000000001011101110011l1
     |avge| = 0.0000000000000000000000 0000000000000001001000110l011
```

Many more correct result bits: errors have been moved to lower significance

But… have not eliminated non-associativity & have used extra power for lower performance

arm

Double Double Accumulator

Adding FP64 numbers to a pair of non-overlapping FP64's

Widely used in math libraries (GMP, BLAS, UKMO, …)

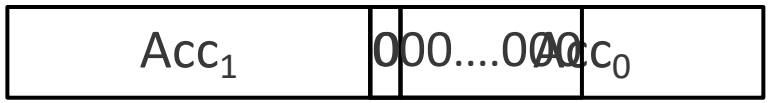
10 FAdd/FSub's needed

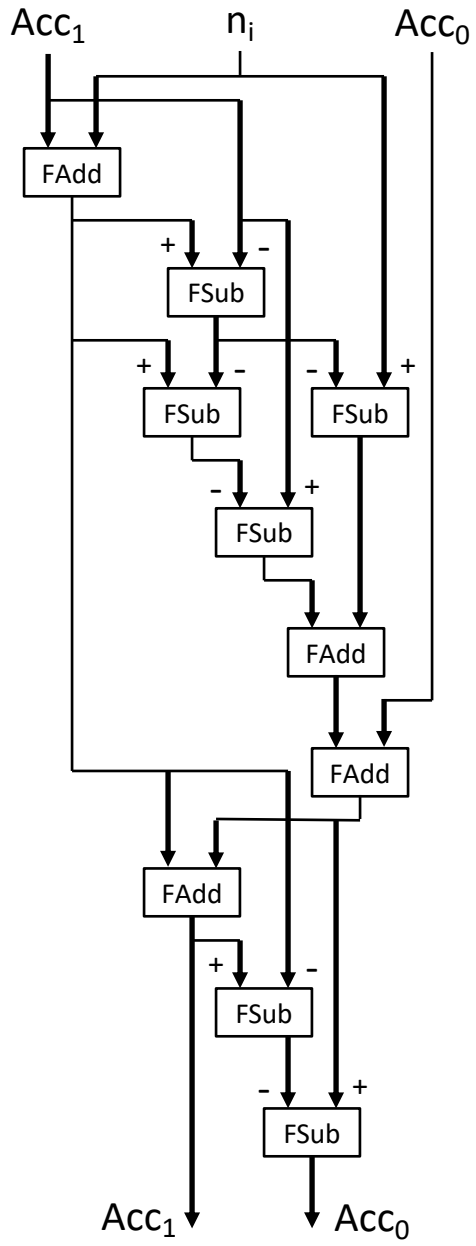
Lots of dependencies

Not bitwise reproducible (Acc1 *probably* reproducible)

Slow & wasteful of power

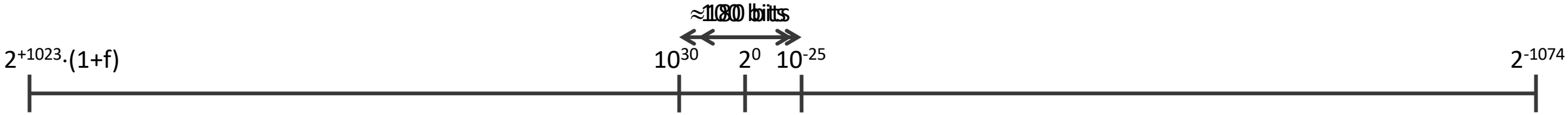
Can start next accumulation before previous one finishes





arm

Observation: Reduced FP Exponent Range Is Enough



IEEE FP64: 11 bit exponent: $2^{-1074} \leq$ fp64 value $\leq 2^{+1023} \cdot (1+f)$

"100 bits [i.e. quad precision mantissa] suffices for many HPC applications", D. Bailey, IEEE ARITH keynote, Austin, 2013

Confirmed by US National Labs & others at recent SuperComputing conferences:

- "most problems fit in the range 10^{-25} to 10^{30}, a span of about 180 bits"
- "in most cases, we're around the 10^{-15} tolerance ($\approx 2^{-50}$) because of machine epilson, compiler rounding/optimization etc "

arm

Wanted: A Practical Associative High-Precision Accumulator

Most problems do not require the full exponent range

- many programmers use FP64 for convenience
- small values may well be unimportant

ARM has introduced SVE

- Scalable (128 – 2048 bit) Vector Extension to ARM's ISA for HPC

Proposal: Use ***High-Precision Anchored (HPA)*** numbers and exploit features of SVE to enable fast (i.e. parallel) reproducible FP summation

What is a High-Precision Anchored (HPA) Number?

An HPA number comprises:

- a long 2's-complement integer, comprising 100-200 or even more bits
  - more precision than available in binary64 (or even binary128 if wanted)
  - length of the long integer provides a range over which to accumulate exactly
- an "anchor" that says how to interpret the long integer
  - anchor is analogous to a block floating-point FP exponent, i.e. fixed for a given problem
  - anchor represents the least significant exponent value we are interested in

HPA accumulation is fixed-point → associative

A programmer can pick the range for the application area or problem…

- e.g. subatomic values could be in the range 2^-100 to 2^-1 (anchor = -100, range = 100 bits)
- e.g. astronomical values could be in the range 2^20 to 2^199 (anchor = +20, range = 180 bits)

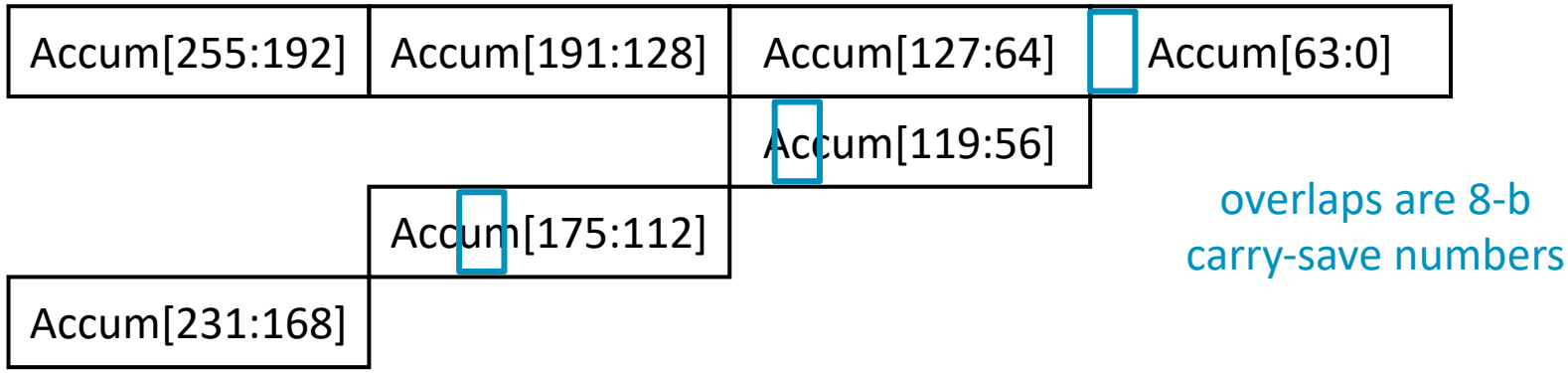
… or can set up default range in a library that dynamically adjusts during an accumulation

Redundant Long Integer Arithmetic

100- or 200-bit adders are impractical, so allow vector elements to “overlap”

- Provide headroom in each lane to accommodate carries

For example, allowing 8 bits’ overlap between lanes:



arm

High-Precision Anchored (HPA) format

New datatype, denoted (Zi, Zm)

- vector of 64-bit integers considered as one long redundantly-represented integer
- metadata (either vector or scalar) interpreting long integer – anchor point(s) & overlap

Example: 232-bit HPA number, with bit-weightings from 155 to -76

- long long i[3:0]; // four 64-bit values
- long long m[3:0] = {92, 36, -20, -76}; // boundaries, with 8-b overlaps
- the high order bit of i[3] represents 2^{155}
- the low order bit of i[0] represents 2^{-76}
- could sum FP values corresponding to that range exactly and *reproducibly*

In CPU terms, need two instructions:

- convert FP to HPA & accumulate
- convert HPA to FP

Converting FP to HPA & accumulate

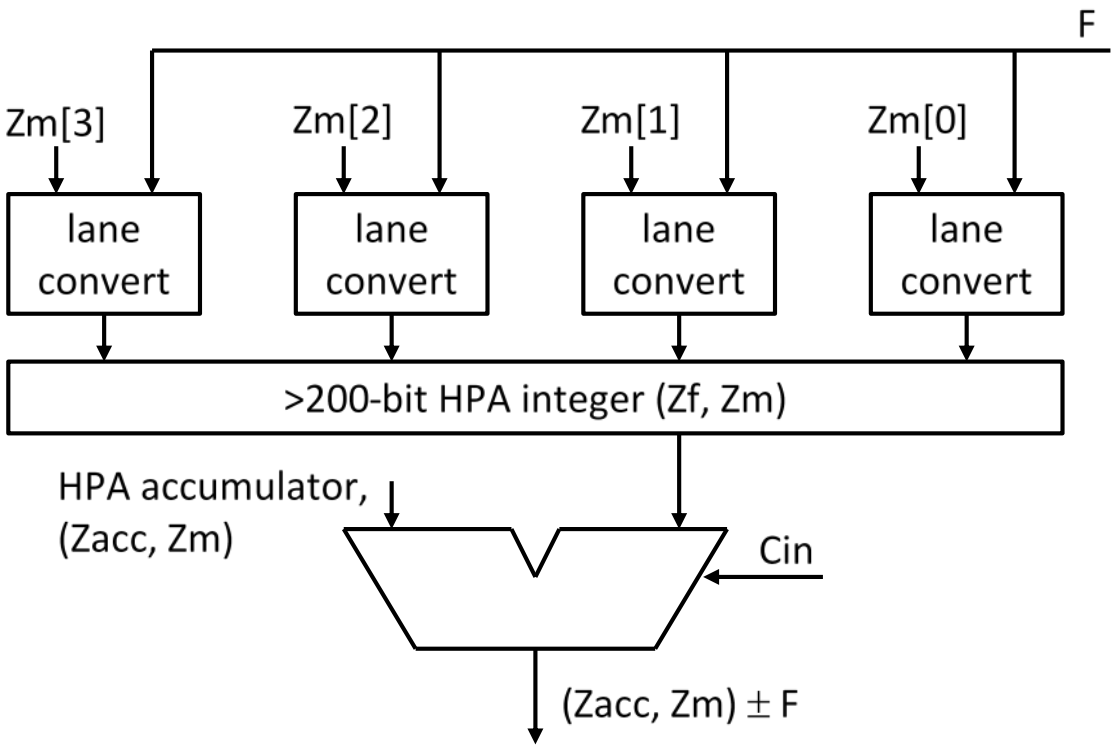
(Conceptually)

Broadcast FP number, F, to all lanes

- Each lane compares exponent of F with Zm[i] & picks significand bits for that lane (shift and mask)
- If F is –ve, take 2's-complement in every lane *independently* (redundancy allows this)
- DP number (53-b mantissa) extends across three 64-b lanes if #overlap bits ≥ 13b
- Note: F could be unrounded FP32 product

Accumulate in Zacc, keeping carries in overlap portion of each lane

- Long wordlength integer addition reduced to 64-b vector integer addition

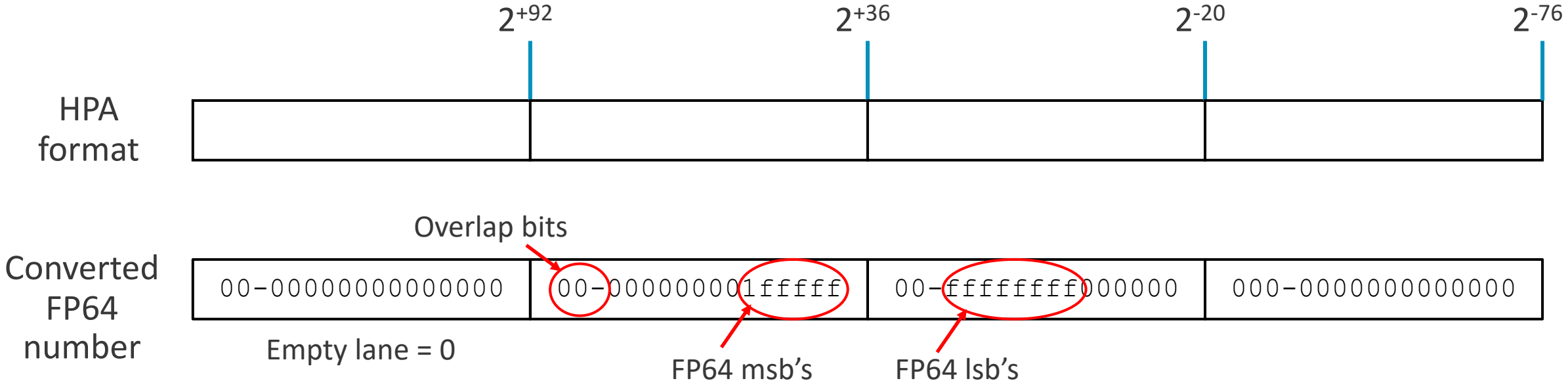


arm

Example FP64 -> HPA conversion

Convert FP64 number to HPA with anchor = {92, 36, -20, -76}, overlap = 8

FP64 number = $+2^{56}$ x 1.fffffffffffff (fraction = all 1's; msb = 2^{+56}, lsb = 2^{+4})



arm

SVE

32 scalable vector registers

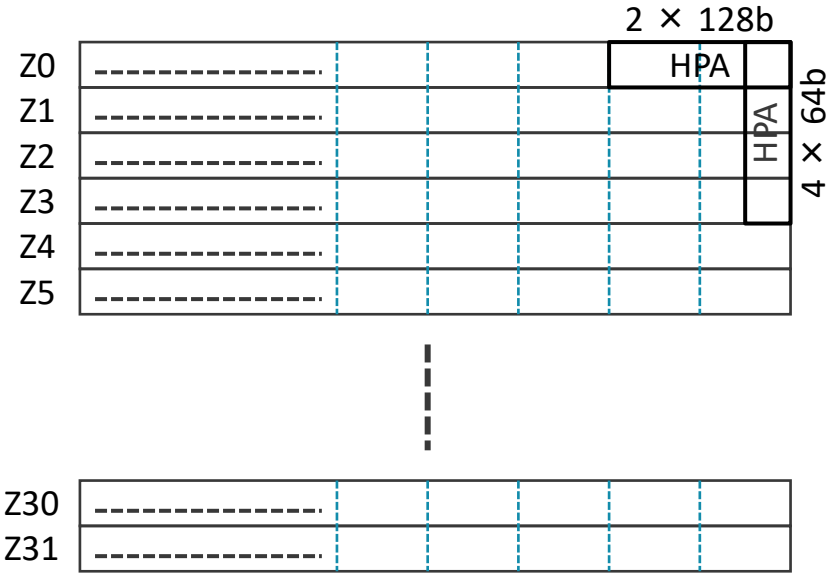
- 128·k bits: k = 1...16

“Vector Length Agnostic”

- Software reads VL & “self-adjusts” degree of parallel processing accordingly

Mitigates against HPA?

- HPA explicitly ***not*** vector length agnostic!
- Difficulties arise if SVE vector length ≠ HPA size



HPA processing in SVE

FP numbers handled in vectors, Zn

- Contiguous load of data

What about metadata, Vm?

- anchor (16 bits), overlap (5 bits)
- also need lane “type” (2 bits)
- Hold 2 sets of metadata in 64 bits of Zm, repeat across vector & index required value

FCVTHA (Zda, Zm[index], Zn)

- Only 3 vector registers in SVE opcodes
- Logic similar to FCVTS

	Lane 3	Lane 2	Lane 1	Lane 0
Zn	FP[3]	FP[2]	FP[1]	FP[0]
Zn+1	FP[7]	FP[6]	FP[5]	FP[4]
Zn+2	FP[11]	FP[10]	FP[9]	FP[8]

	Lane 3	Lane 2	Lane 1	Lane 0
Zm,index	Vm[i]	Vm[i]	Vm[i]	Vm[i]

arm

Example HPA in SVE: add & convert

e.g. 2-lane 100-bit HPA (ovlp = 14)

- Vm[1:0] stored in Zm[1:0]
- Select Vm[i] by index

- FCVTHA (Zd, Zm[0], Zn)
- // change index
- FCVTHA (Zd+1, Zm[1], Zn)
- // load next FP data register
- FCVTHA (Zd, Zm[0], Zn+1)
- FCVTHA (Zd+1, Zm[1], Zn+1)
- …

4 HPA accumulators A[1:0]

Vector-length agnostic

	Lane 3	Lane 2	Lane 1	Lane 0
Index = 0	Vm[0]	Vm[0]	Vm[0]	Vm[0]

Zn	FP[3]	FP[2]	FP[1]	FP[0]
Zn+1	FP[7]	FP[6]	FP[5]	FP[4]

Zd	A[0][3]	A[0][2]	A[0][1]	A[0][0]

arm

HPA in SVE: other operations

These occur much less frequently than HPA convert-&-adds

1. Convert HPA number to canonic form (eliminate carries)

- add upper bits of Zda (with sign extension) to bottom of Zda+1 & then reset to zero
- all overlap bits set to 0, except in most significant HPA register
- opportunity to check for overflow

2. HPA vector reduction

- add all elements across a vector together, assuming same metadata for all elements (predicated if not)

3. Convert HPA back to FP

- FP64 result is exact if ovlp ≥ 11 (disallow smaller overlaps for this reason)
- logic similar to 64-b $\rightarrow$ DP SCVTF

Performance

Model of A-class CPU with 2 Vector units, 2 Integer units, 2 Load/Store units

All loops contain: `INCD,CMP,B  LD1D,FADD/FSUB,FCVTHA  MOV`

Accumulation Scheme	Result bits	Insts / vector	FP op / vector	Cycle count / element VL=128	VL=256	VL=512	Mean cycles / vector
Dbl	e.g. 40	5	1	1.00	0.50	0.25	1.99
Dbl (unroll)	e.g. 40	7	2	0.50	0.25	0.13	1.01
Dbldbl	e.g. 94	20	10	10.01	4.76	2.27	19.08

Summary

"the fast drives out the slow even if the fast is wrong." - W. Kahan

- but what if the fast is right?

why deal with the irreproducibility & incorrectness of FP accumulation?

- for reasonable exponent ranges, high-precision HPA accumulation is order of magnitude faster than e.g. double-double accumulation
- for larger exponent ranges, still ok

HPA enables high-precision FP accumulation that is reproducible AND parallelizable

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!

Demmel & Nguyen (2015)

"Fast Reproducible Summation"

Software solution

Initialized accumulator

2^{K_2}	2^{K_1}	2^{K_0}
1.100…00	1.100…00	1.100…00

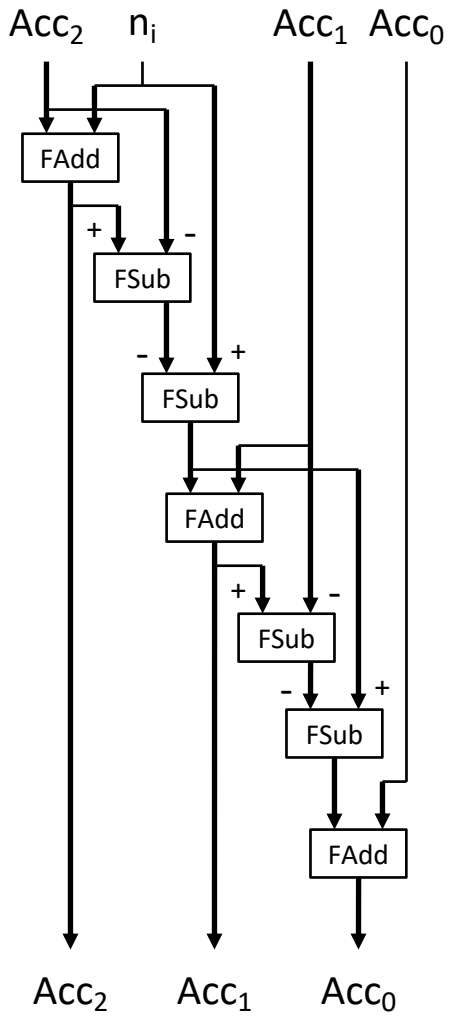
$(K_2 - K_1) = (K_1 - K_0) = 40$

$K_2 \geq \exp(\max(\mathrm{FP64}(n_i))) + 11$

Add 2^{11} numbers with *constant* K's, OR lsb's(Jam1)

Fixed-point accumulator ∴ reproducible

U.C.Berkeley: ReproBLAS



arm