A Case Study on Optimizing Accurate Half Precision Average

Kenny Peou[1,2,3]
kenny.peou@numscale.com

Alan Kelly[1], Joel Falcou[1,2,3]
NUMSCALE[1], LRI[2] and Université Paris-Saclay[3]

24/09/2018

numscale
Unlocked software performance

PARIS SACLAY
Communauté d'agglomération

A Very Average Presentation

- Average is fundamental component of some ML algorithms
  - k-means, meanshift, average pooling

A Very Average Presentation

- Average is fundamental component of some ML algorithms
  - k-means, meanshift, average pooling
- FP16 hardware support incoming
  - Pascal GPU, ARM SVE

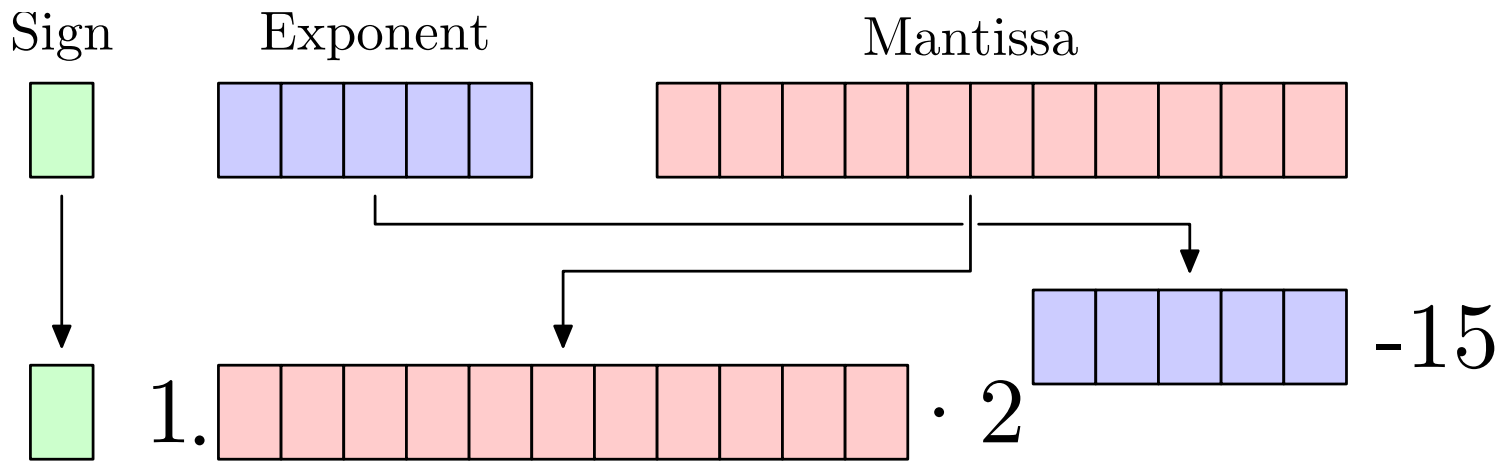
A Very Average Presentation

- Average is fundamental component of some ML algorithms
  - k-means, meanshift, average pooling
- FP16 hardware support incoming
  - Pascal GPU, ARM SVE
- FP16 precision imposes serious limitations

A Very Average Presentation

- Average is fundamental component of some ML algorithms
  - k-means, meanshift, average pooling
- FP16 hardware support incoming
  - Pascal GPU, ARM SVE
- FP16 precision imposes serious limitations
- Performance is important

Floating Point Format - Specifications



Precision	Exponent	Mantissa	Max	Min
Half	5	10	65504	$6.1 \cdot 10^{-5}$
Single	8	23	$3.4 \cdot 10^{38}$	$1.2 \cdot 10^{-38}$
Double	11	52	$1.8 \cdot 10^{308}$	$2.2 \cdot 10^{-308}$

Limitations

- Overflow - too big

  $65504 + 32 = \infty$

  $256 \cdot 256 = \infty$

Limitations

- Overflow - too big

  $65504 + 32 = \infty$

  $256 \cdot 256 = \infty$

- Underflow/Subnormals - too small

  $1 \div 66000 = 0$

  $0.0625/64992 = 9.53674e - 07$

Limitations

- Overflow - too big

  $65504 + 32 = \infty$

  $256 \cdot 256 = \infty$

- Underflow/Subnormals - too small

  $1 \div 66000 = 0$

  $0.0625/64992 = 9.53674e - 07$

- Exponent (mis)alignment - juuuuuuust wrong

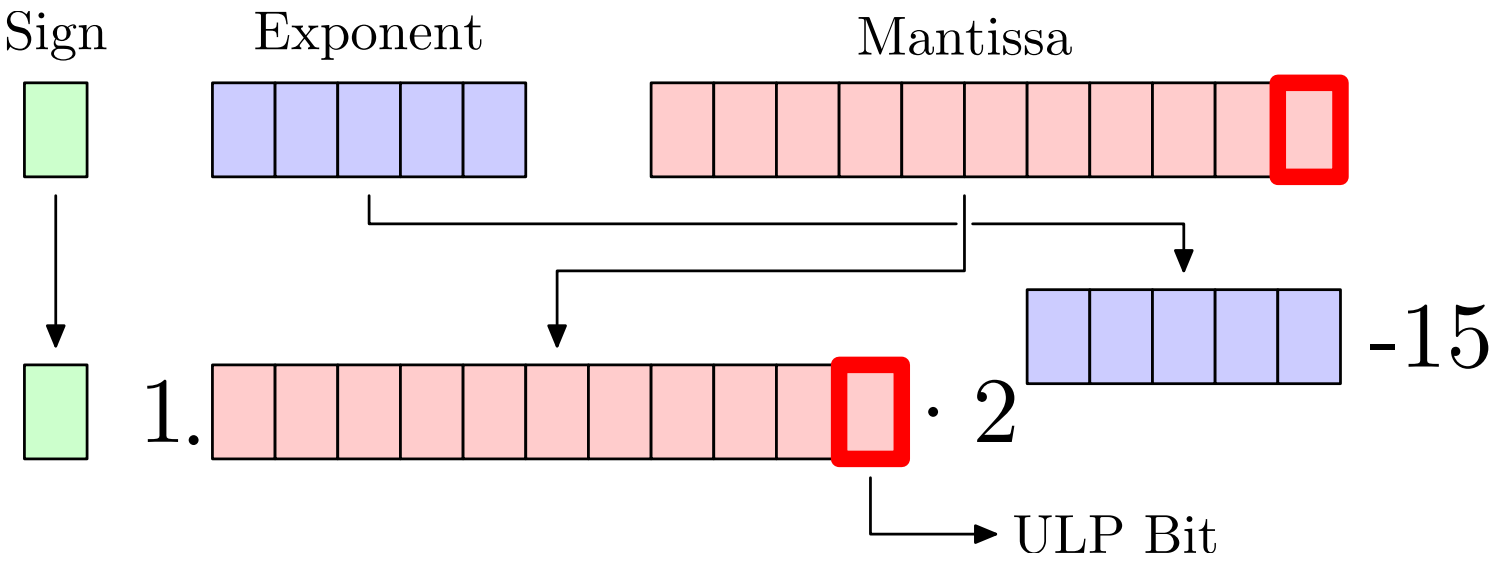
  $2048 + 1 = 2048$

  $2048 + 3.5 = 2050$

Limitations

- Overflow - too big
  $65504 + 32 = \infty$
  $256 \cdot 256 = \infty$
- Underflow/Subnormals - too small
  $1 \div 66000 = 0$
  $0.0625/64992 = 9.53674e - 07$
- Exponent (mis)alignment - juuuuuuust wrong
  $2048 + 1 = 2048$
  $2048 + 3.5 = 2050$
- Limited hardware support for FP16
  Emulated via `half` C++ library

Unit of Least Precision (ULP)

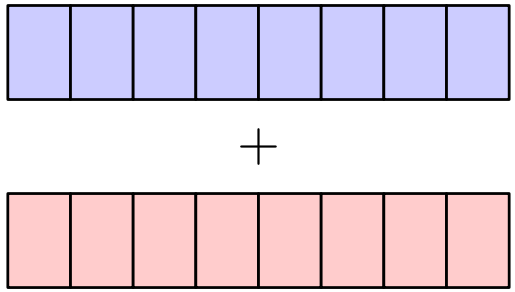


Base Value	Half ULP	Float ULP	Double ULP
2^{-10}	2^{-20}	2^{-33}	2^{-63}
1	2^{-10}	2^{-23}	2^{-53}
2^{10}	1	2^{-13}	2^{-43}

Table: Value of 1 ULP at different base values for half, single, and double precisions.

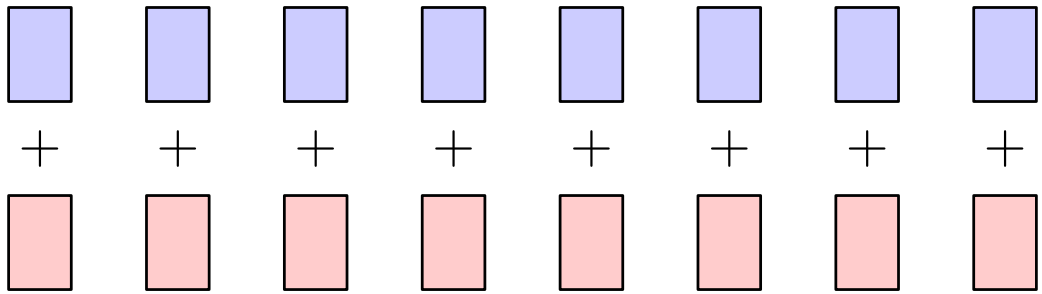
Parallelization (SIMD)

Single Instruction Single Data



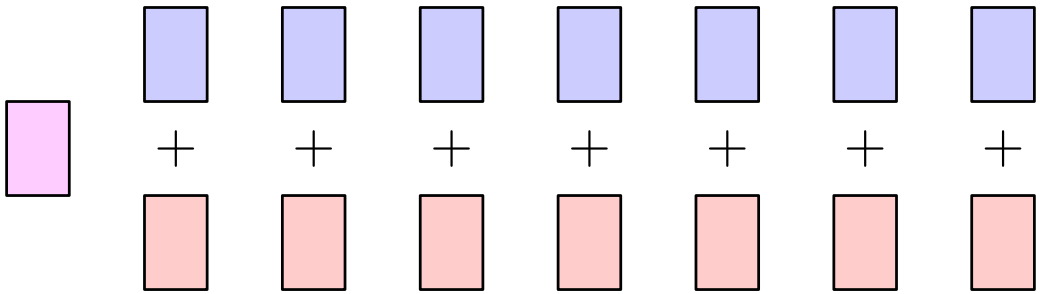
Parallelization (SIMD)

Single Instruction Single Data



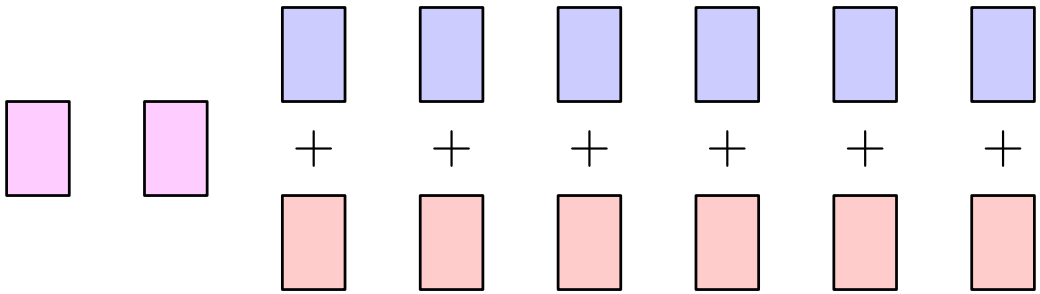
Parallelization (SIMD)

Single Instruction Single Data



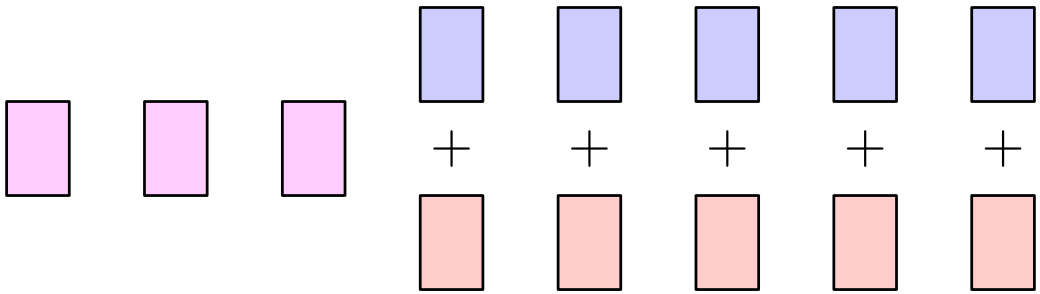
Parallelization (SIMD)

Single Instruction Single Data



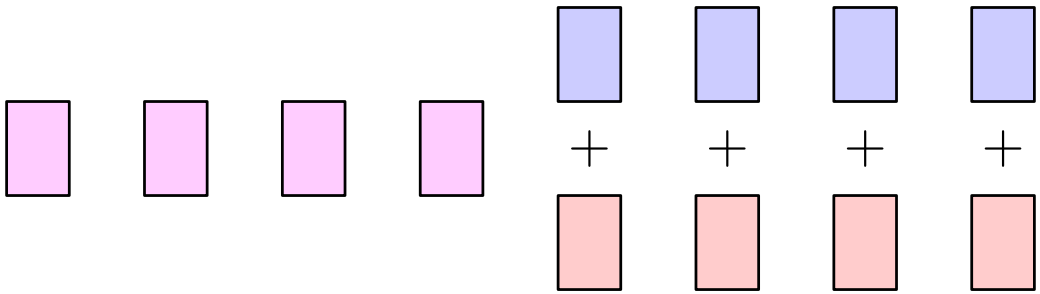
Parallelization (SIMD)

Single Instruction Single Data



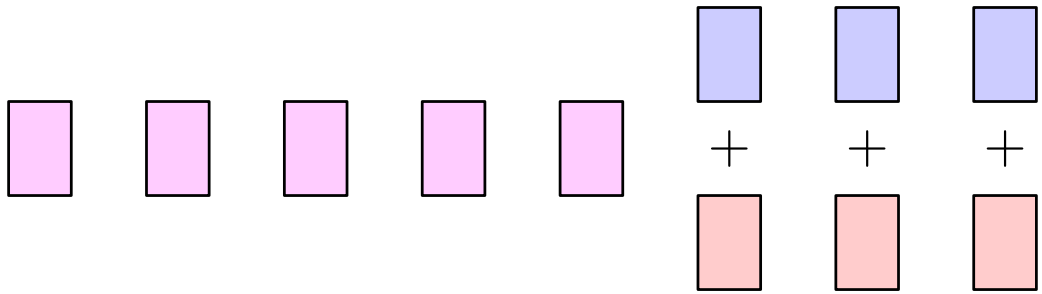
Parallelization (SIMD)

Single Instruction Single Data



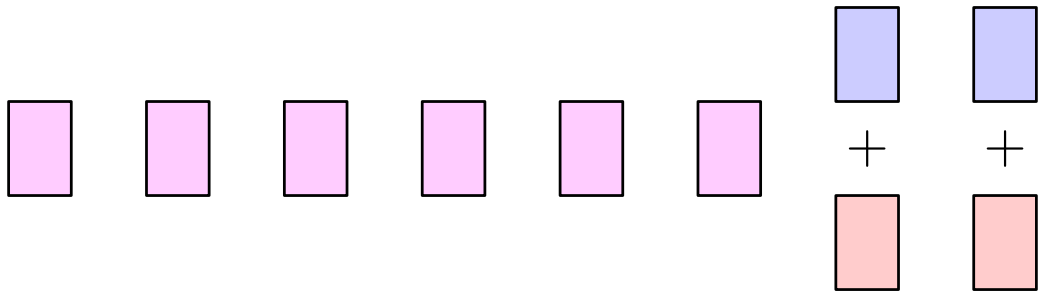
Parallelization (SIMD)

Single Instruction Single Data



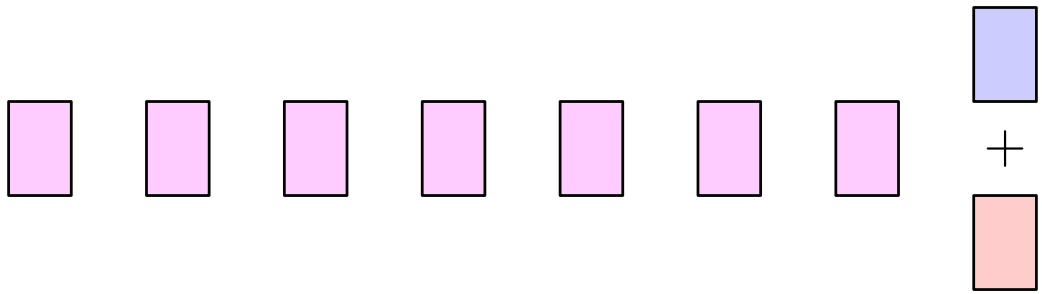
Parallelization (SIMD)

Single Instruction Single Data



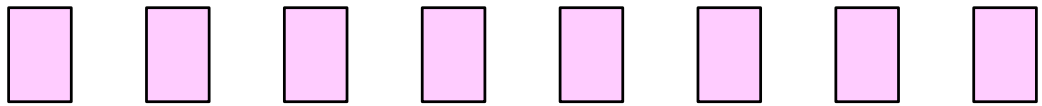
Parallelization (SIMD)

Single Instruction Single Data



Parallelization (SIMD)

Single Instruction Single Data

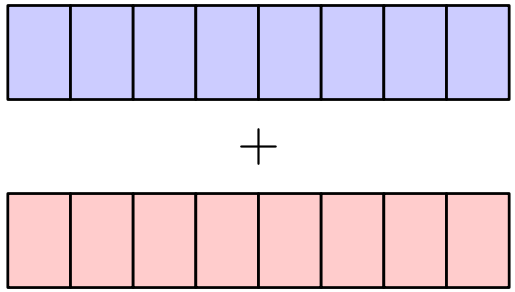


Parallelization (SIMD)

Single Instruction Single Data

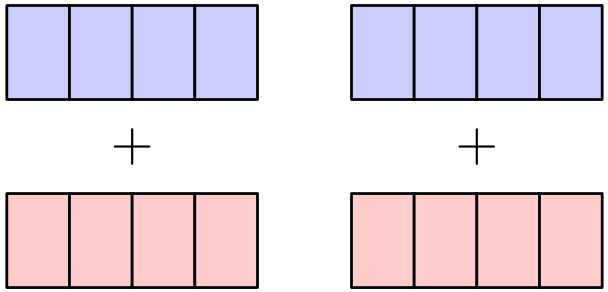
Parallelization (SIMD)

Single Instruction Multiple Data



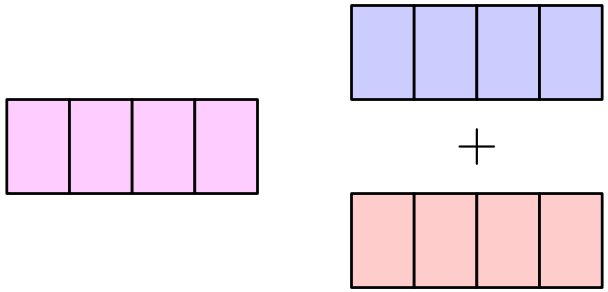
Parallelization (SIMD)

Single Instruction Multiple Data



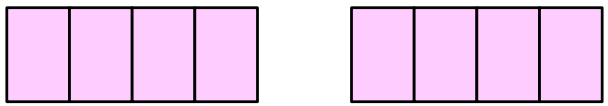
Parallelization (SIMD)

Single Instruction Multiple Data



Parallelization (SIMD)

Single Instruction Multiple Data



Parallelization (SIMD)

Single Instruction Multiple Data

SIMD Technologies

- GPU - CUDA, OpenCL
- Multithread - OpenMP, MPI
- CPU Vectorization - AVX, NEON, PPC
  - No communication/transfer
  - Fixed register sizes
  - Not portable (without NSIMD)

Vectorization - NSIMD

```
// AVX
__m256 a = _mm256_load_ps( &(A[i]) );
__m256 b = _mm256_load_ps( &(B[i]) );
__m256 c = _mm256_add_ps( a , b );
_mm256_store_ps( &(C[i]) , c );
```

Vectorization - NSIMD

```
// NEON
float32x4_t a = vld1q_s32( &(A[i]) );
float32x4_t b = vld1q_s32( &(B[i]) );
float32x4_t c = vaddq_f32( a , b );
vst1q_f32( &(C[i]) , c );
```

Vectorization - NSIMD

```
// VMX
__vector float a = vec_ld( 0 , &(A[i]) );
__vector float b = vec_ld( 0 , &(B[i]) );
__vector float c = vec_add( a , b );
vec_st( c , 0 , &(C[i]) );
```

Vectorization - NSIMD

```
// NSIMD (All architectures)
nsimd::pack<float> b = nsimd::load( &(B[i]) );
nsimd::pack<float> c = nsimd::load( &(C[i]) );
nsimd::pack<float> a = b * c;
nsimd::store( &(A[i]) , a );
```

Performance Considerations

- Number of Operations/Instructions
- Instruction Latency
- Data Types (SIMD consideration)

Operation	Integer	Float	SIMD Float	SIMD Double
Load/Store	1	1	1	1
Addition	1	3	3	3
Subtraction	1	3	3	3
Multiplication	3	5	5	5
Division	22	10	10	10

Table: Minimum cycles required to perform basic arithmetic operations on an Intel Haswell CPU. (Agner Fog)

Calculating the Average

Seems simple, right?

$$\text{Avg}(X) = \frac{1}{N} \sum_{i=1}^{N} x_i$$

Not with half precision!

Precision Benchmarks

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	$\text{Input}[i] = \text{s} \cdot i$					
$\text{s} = 1$	100 1000 10000					
$\text{s} = 0.001$	100 1000 10000					
Fixed Ratio	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
$\text{r} = \text{N}/2$	100 1000 10000					
Fixed Diff	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
$\text{d} = \text{N}/2$	100 1000 10000					
Random	$\text{Input}[i] = \mathit{rand}()$					
$[0, 1]$ $[0, 10]$	1000 1000					
Fixed	$\text{Input}[i] = \text{c}$					
$\text{c} = 10$	1000 10000 1000000					
Repeating	$\text{Input}[i] = 10 + (i\%3)$					
10 - 12	300 3000 30000					
Image1 Image2	2073600 2073600					

Performance Benchmarks

Processor Used	Clock Speed	RAM
Intel i7-4790S	3.20GHz	16GB
Power8 8348-21C	2.061GHz	64GB
ARM Cortex-A57r1	1.91GHz	4GB

Benchmark	Compiler Used	Compilation Flags
Scalar	All of below	`-O3`
SSE	GCC 6.3.1	`-O3 -msse4.2`
AVX	GCC 6.3.1	`-O3 -mavx2`
NEON	GCC 5.4.1	`-O3 -march=armv8-a+simd`
Altivec	GCC 6.3.0	`-O3 -maltivec`

Performance measured using float instead of half

Naive Average

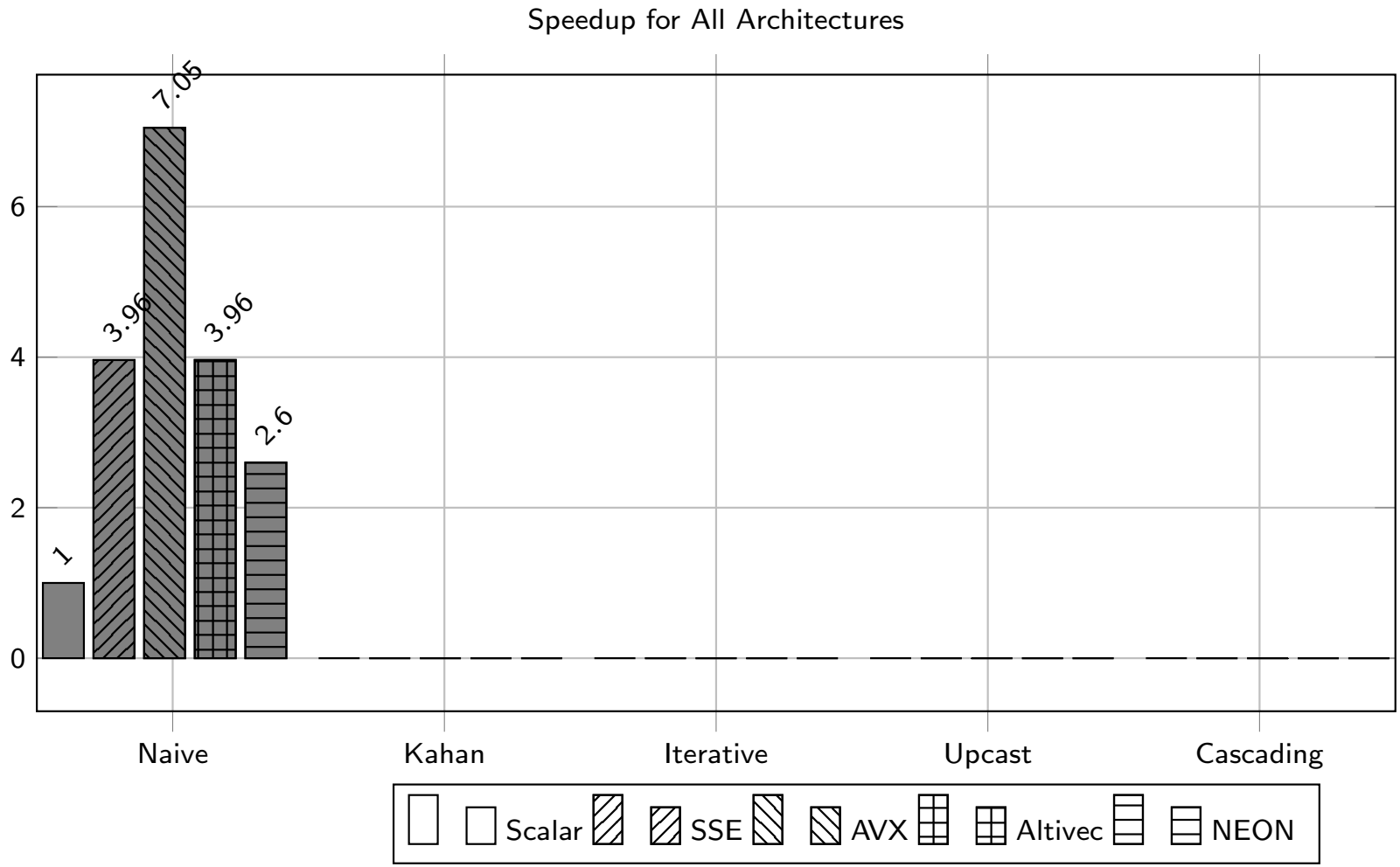
```
1: function NAIVE AVERAGE(Array)
2:     sum = 0
3:     for a in Array do
4:         sum += a
5:     end for
6:     avg = sum / length(Array)
7: end function
```

- Rounding errors gets worse and worse
- Susceptible to overflow
- Computationally simple

Naive Average

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	$\text{Input}[i] = \text{s} \cdot i$					
s = 1	100	9				
	1000	fail				
	10000	fail				
s = 0.001	100	16				
	1000	198				
	10000	fail				
Fixed Ratio	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
r = N/2	100	1				
	1000	fail				
	10000	fail				
Fixed Diff	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
d = N/2	100	13				
	1000	fail				
	10000	fail				
Random	$\text{Input}[i] = rand()$					
[0, 1]	1000	271				
[0, 10]	1000	fail				
Fixed	$\text{Input}[i] = \text{c}$					
c = 10	1000	152				
	10000	1070				
	1000000	1277				
Repeating	$\text{Input}[i] = 10 + (i\%3)$					
10 - 12	300	17				
	3000	709				
	30000	1998				
Image1	2073600	fail				
Image2	2073600	1761				

Naive Average



Average Using Kahan Sum

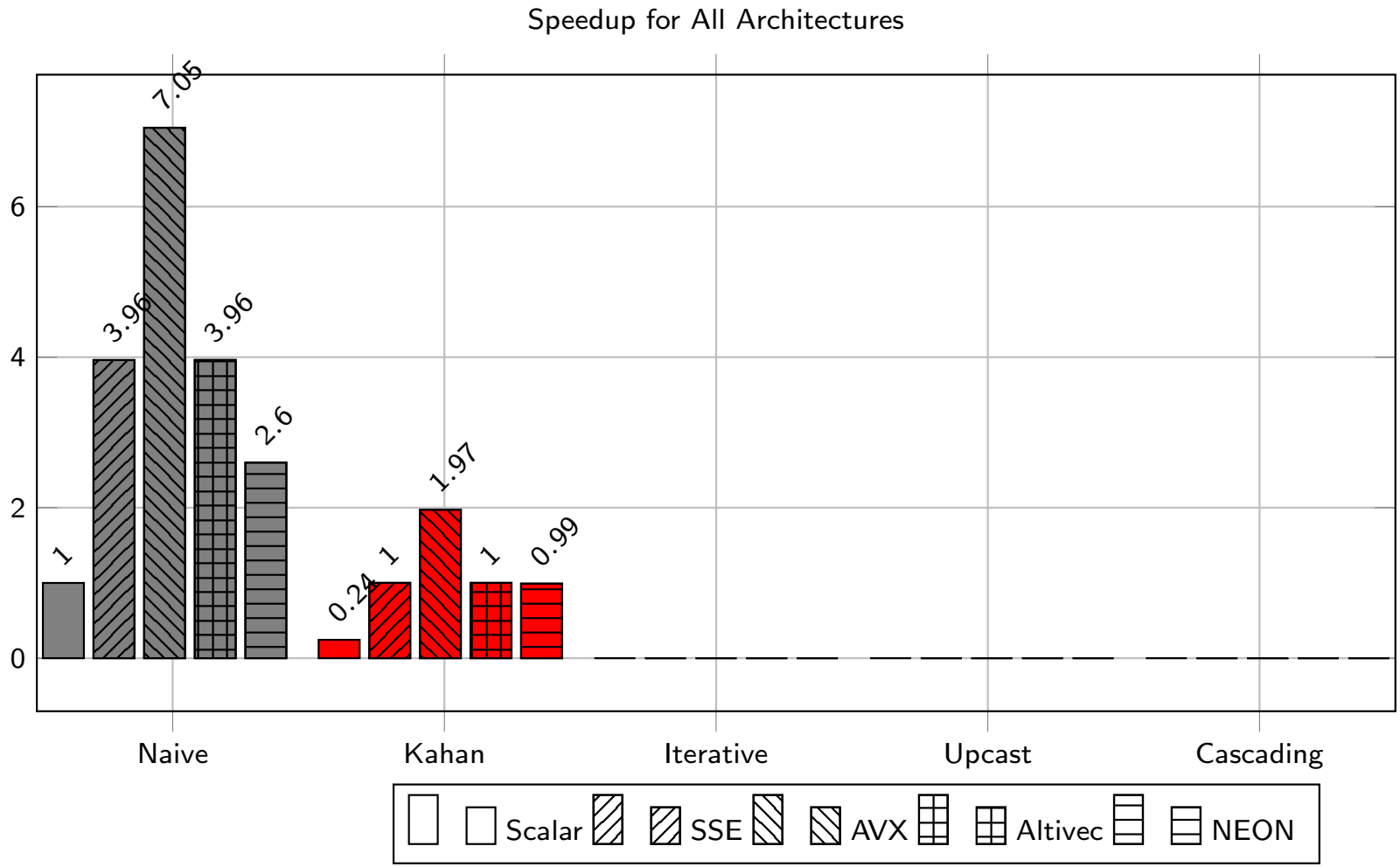
```
 1: function KAHAN AVERAGE(Array)
 2:     sum = 0
 3:     rem = 0
 4:     for a in Array do
 5:         y = a - rem
 6:         t = sum + y
 7:         rem = ( t - sum ) - y
 8:         sum = t
 9:     end for
10:     avg = sum / length(Array)
11: end function
```

- Mostly compensates rounding errors
- Still susceptible to overflow and misalignments

Average Using Kahan Sum

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	$\text{Input}[i] = \text{s} \cdot i$					
s = 1	100	9	1			
	1000	fail	fail			
	10000	fail	fail			
s = 0.001	100	16	1			
	1000	198	1			
	10000	fail	fail			
Fixed Ratio	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
r = N/2	100	1	1			
	1000	fail	fail			
	10000	fail	fail			
Fixed Diff	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
d = N/2	100	13	1			
	1000	fail	fail			
	10000	fail	fail			
Random	$\text{Input}[i] = \textit{rand}()$					
[0, 1]	1000	271	0			
[0, 10]	1000	fail	fail			
Fixed	$\text{Input}[i] = \text{c}$					
c = 10	1000	152	0			
	10000	1070	fail			
	1000000	1277	fail			
Repeating	$\text{Input}[i] = 10 + (i\%3)$					
10 - 12	300	17	0			
	3000	709	1			
	30000	1998	fail			
Image1	2073600	fail	fail			
Image2	2073600	1761	391			

Average Using Kahan Sum



Iterative Average

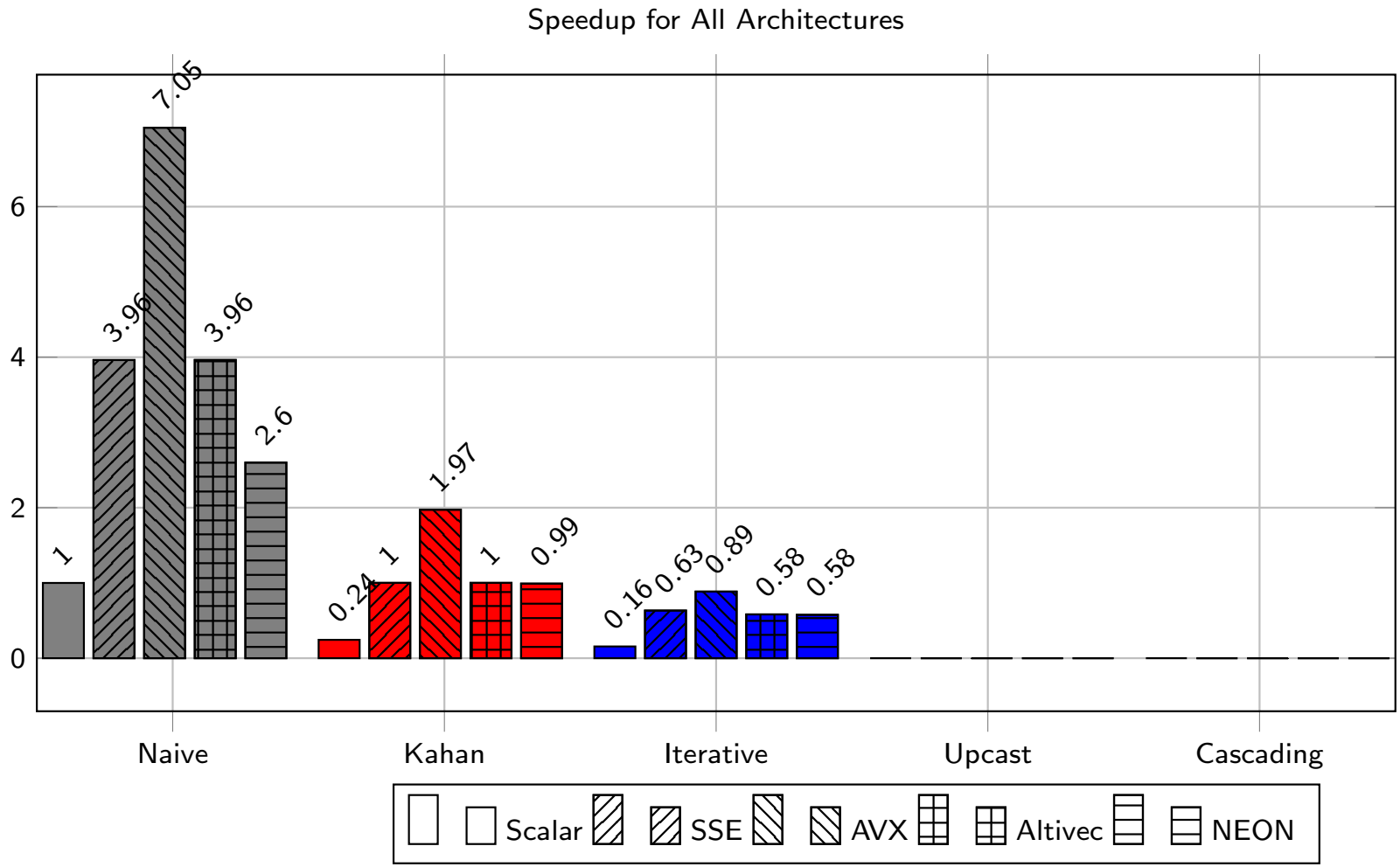
```
1: function ITERATIVE AVERAGE(Array)
2:     avg = 0
3:     for i in 1 → length(Array) do
4:         avg += (avg - Array[i]) / i
5:     end for
6: end function
```

- Removes risk of overflow
- Misalignment, underflow, and subnormals get progressively worse
- $$\lim_{i \to \infty} \frac{(x_i - \text{Avg}_i)}{i} = 0$$

Iterative Average

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	$\text{Input}[i] = \text{s} \cdot i$					
$\text{s} = 1$	100	9	1	0		
	1000	fail	fail	0		
	10000	fail	fail	993		
$\text{s} = 0.001$	100	16	1	20		
	1000	198	1	249		
	10000	fail	fail	1486		
Fixed Ratio	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
$\text{r} = \text{N}/2$	100	1	1	17		
	1000	fail	fail	3		
	10000	fail	fail	994		
Fixed Diff	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
$\text{d} = \text{N}/2$	100	13	1	23		
	1000	fail	fail	227		
	10000	fail	fail	737		
Random	$\text{Input}[i] = \mathit{rand}()$					
$[0, 1]$	1000	271	0	249		
$[0, 10]$	1000	fail	fail	261		
Fixed	$\text{Input}[i] = \text{c}$					
$\text{c} = 10$	1000	152	0	0		
	10000	1070	fail	0		
	1000000	1277	fail	0		
Repeating	$\text{Input}[i] = 10 + (i\%3)$					
10 - 12	300	17	0	74		
	3000	709	1	128		
	30000	1998	fail	128		
Image1	2073600	fail	fail	1037		
Image2	2073600	1761	391	1701		

Iterative Average



Increased Precision

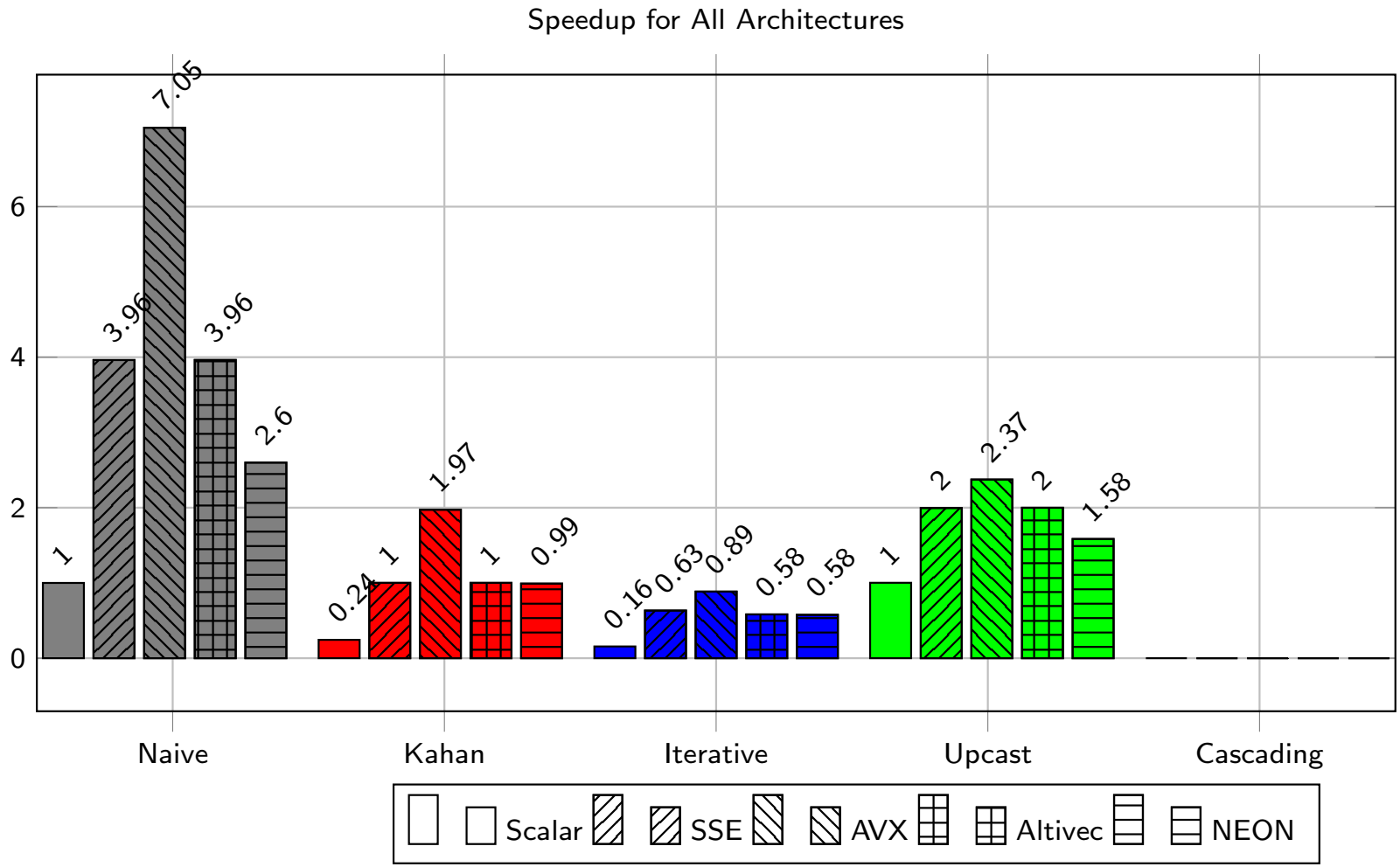
```
1: function UPCAST AVERAGE(Array)
2:     sum = (upcast)0
3:     for a in Array do
4:         sum += (upcast)a
5:     end for
6:     avg = sum / length(Array)
7: end function
```

- Same fundamental problems as naive average
- Limitations less problematic

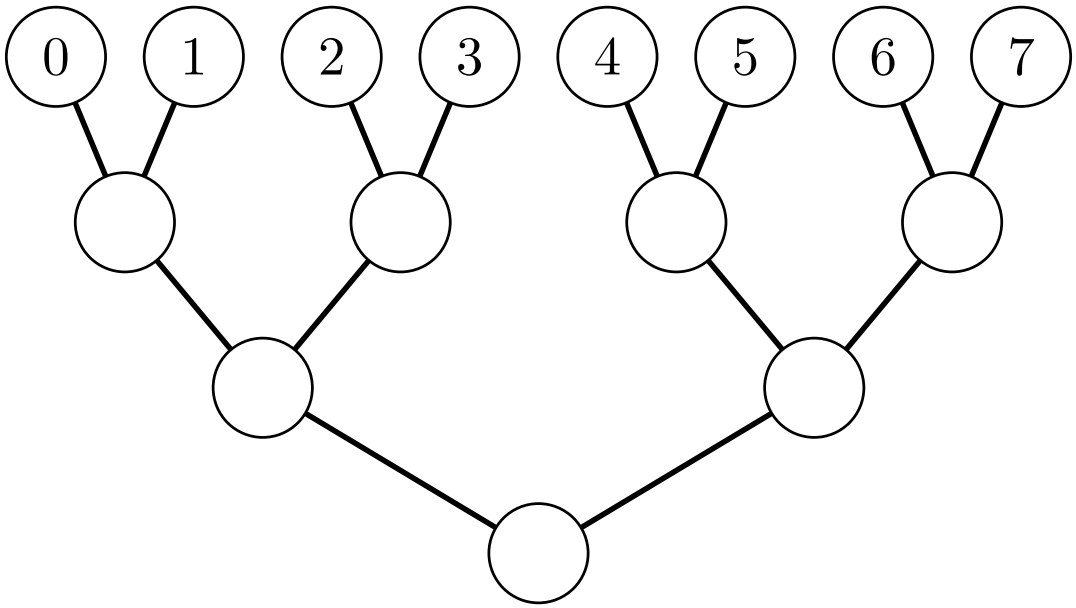
Increased Precision

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	$\text{Input}[i] = \text{s} \cdot i$					
s = 1	100	9	1	0	0	
	1000	fail	fail	0	0	
	10000	fail	fail	993	0	
s = 0.001	100	16	1	20	0	
	1000	198	1	249	0	
	10000	fail	fail	1486	0	
Fixed Ratio	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
r = N/2	100	1	1	17	0	
	1000	fail	fail	3	0	
	10000	fail	fail	994	0	
Fixed Diff	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
d = N/2	100	13	1	23	0	
	1000	fail	fail	227	0	
	10000	fail	fail	737	0	
Random	$\text{Input}[i] = rand()$					
[0, 1]	1000	271	0	249	0	
[0, 10]	1000	fail	fail	261	0	
Fixed	$\text{Input}[i] = \text{c}$					
c = 10	1000	152	0	0	0	
	10000	1070	fail	0	0	
	1000000	1277	fail	0	0	
Repeating	$\text{Input}[i] = 10 + (i\%3)$					
10 - 12	300	17	0	74	0	
	3000	709	1	128	0	
	30000	1998	fail	128	0	
Image1	2073600	fail	fail	1037	1	
Image2	2073600	1761	391	1701	12	

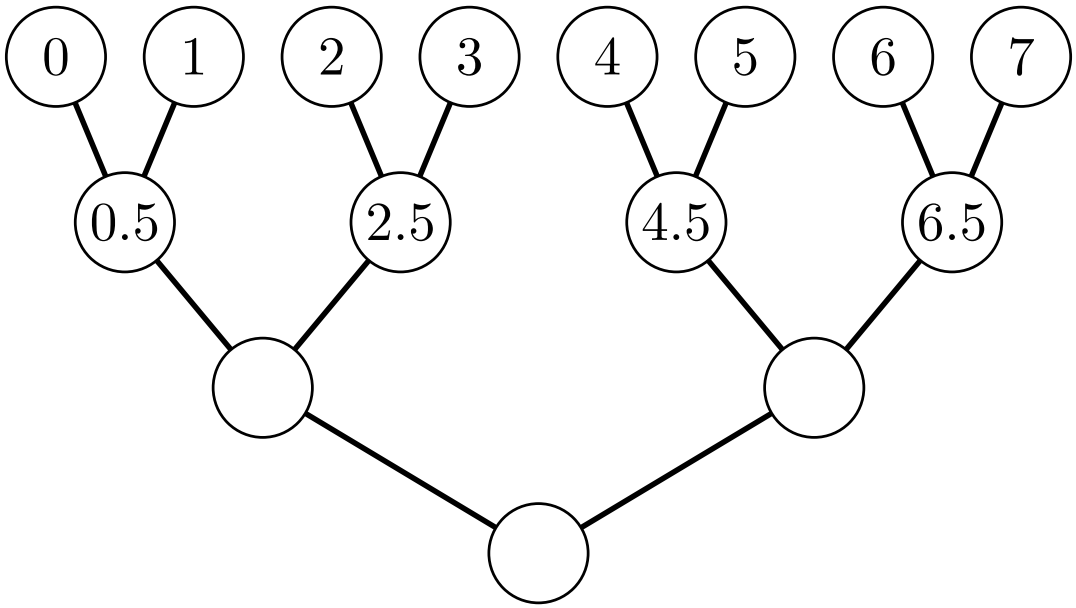
Increased Precision



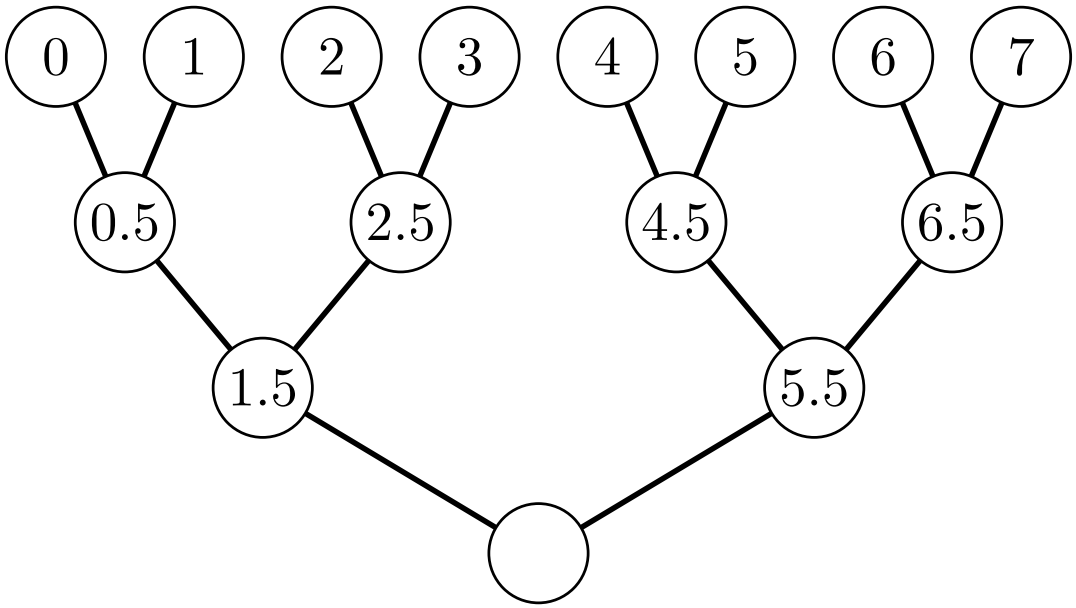
Cascading Average



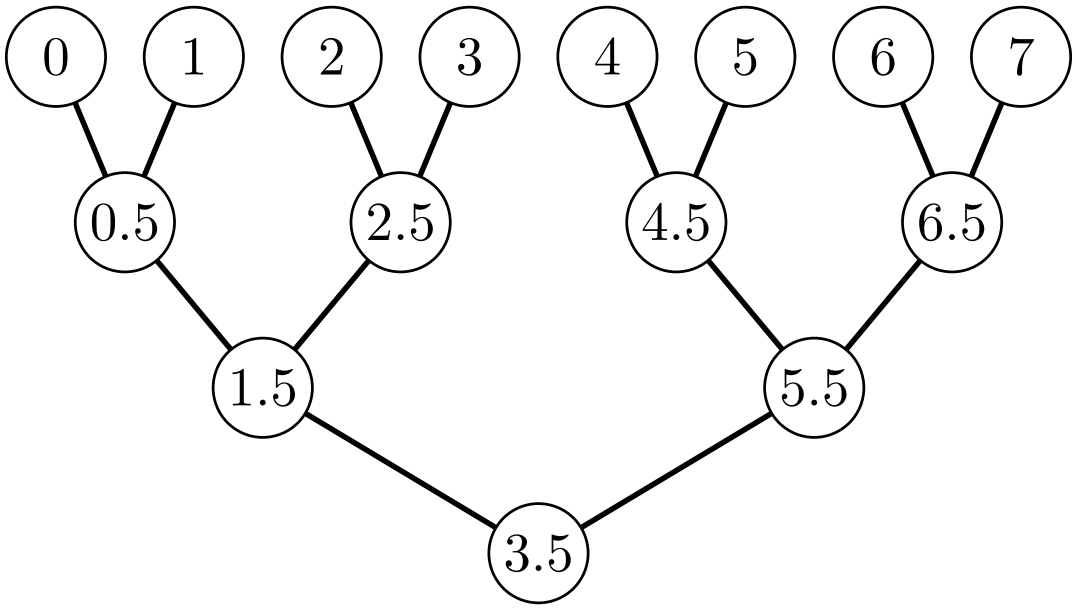
Cascading Average



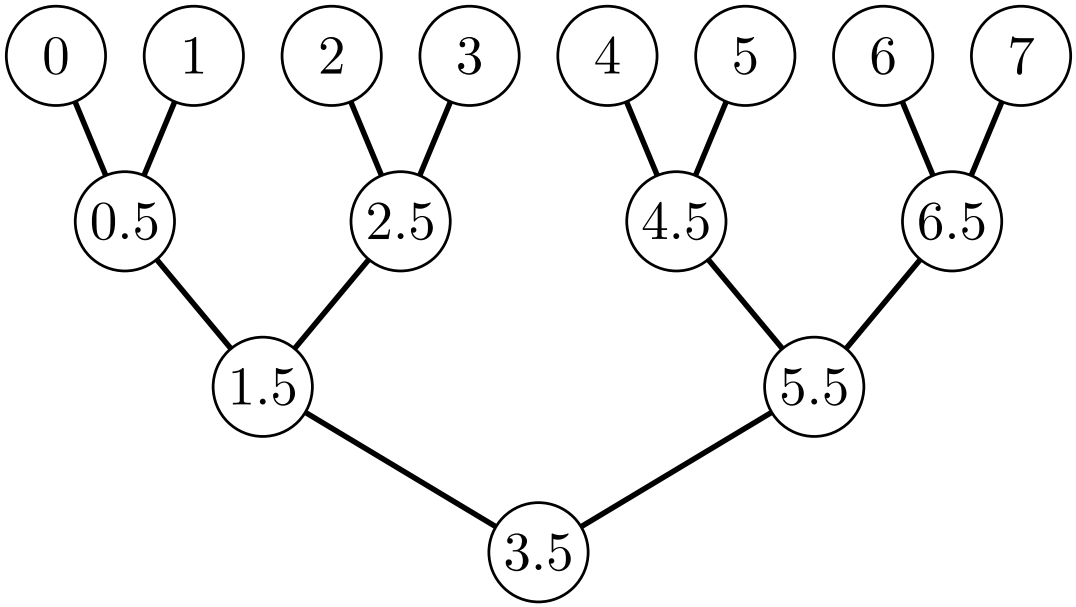
Cascading Average



Cascading Average



Cascading Average



- Pairwise operations
- No accumulator = no overflow
- Rounding errors from input variation
- $\leq 2N$ operations

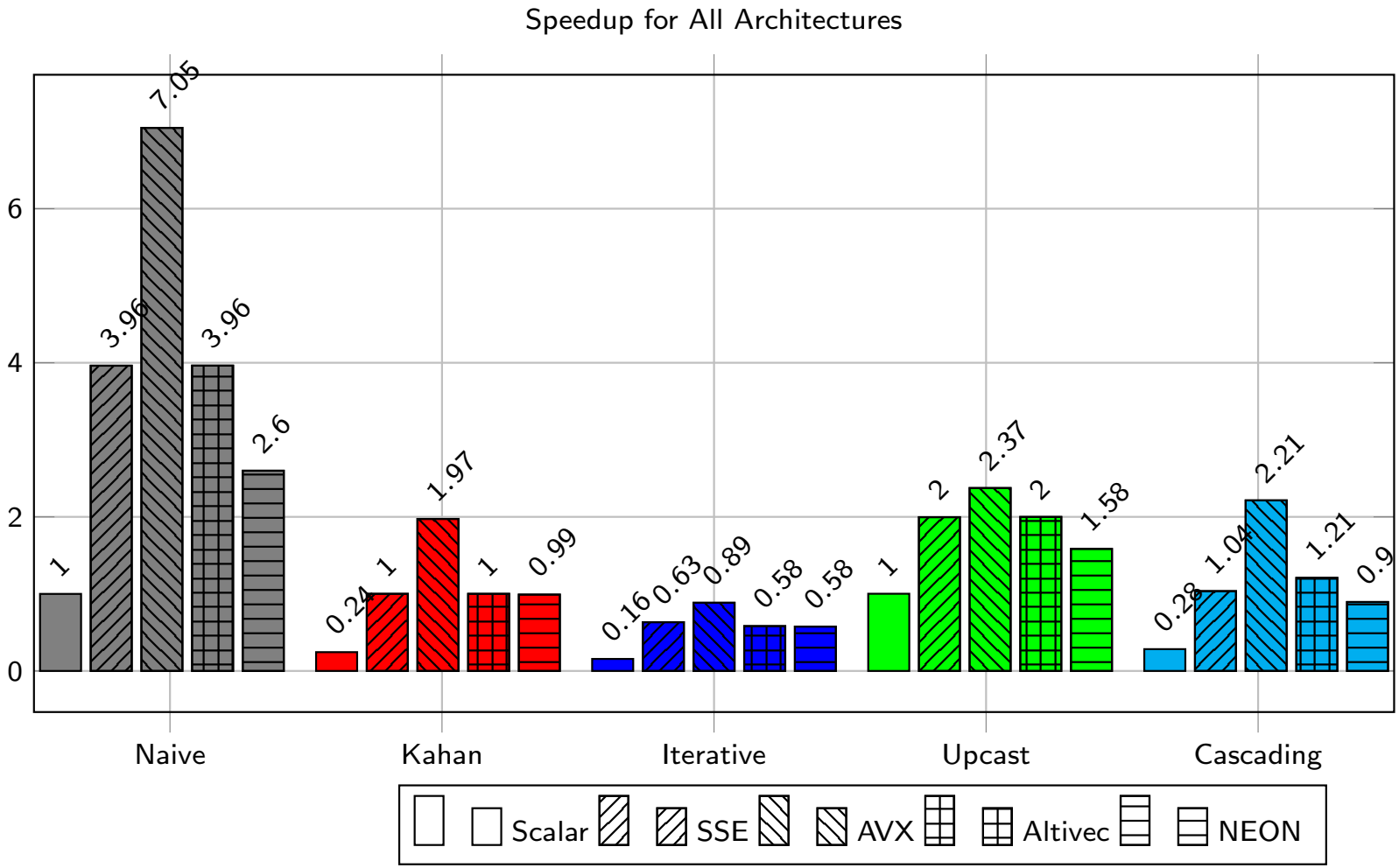
Cascading Average

```
1: function CASCADING AVERAGE(Array)
2:     n = length(Array)
3:     if n == 1 then
4:         return Array[0]
5:     else if n == 2 then
6:         return ( Array[0] + Array[1] ) / 2
7:     else
8:         return ( Cascading Average( Array[1:n/2] ) + Cascading
   Average( Array[n/2:n] ) ) / 2
9:     end if
10: end function
```

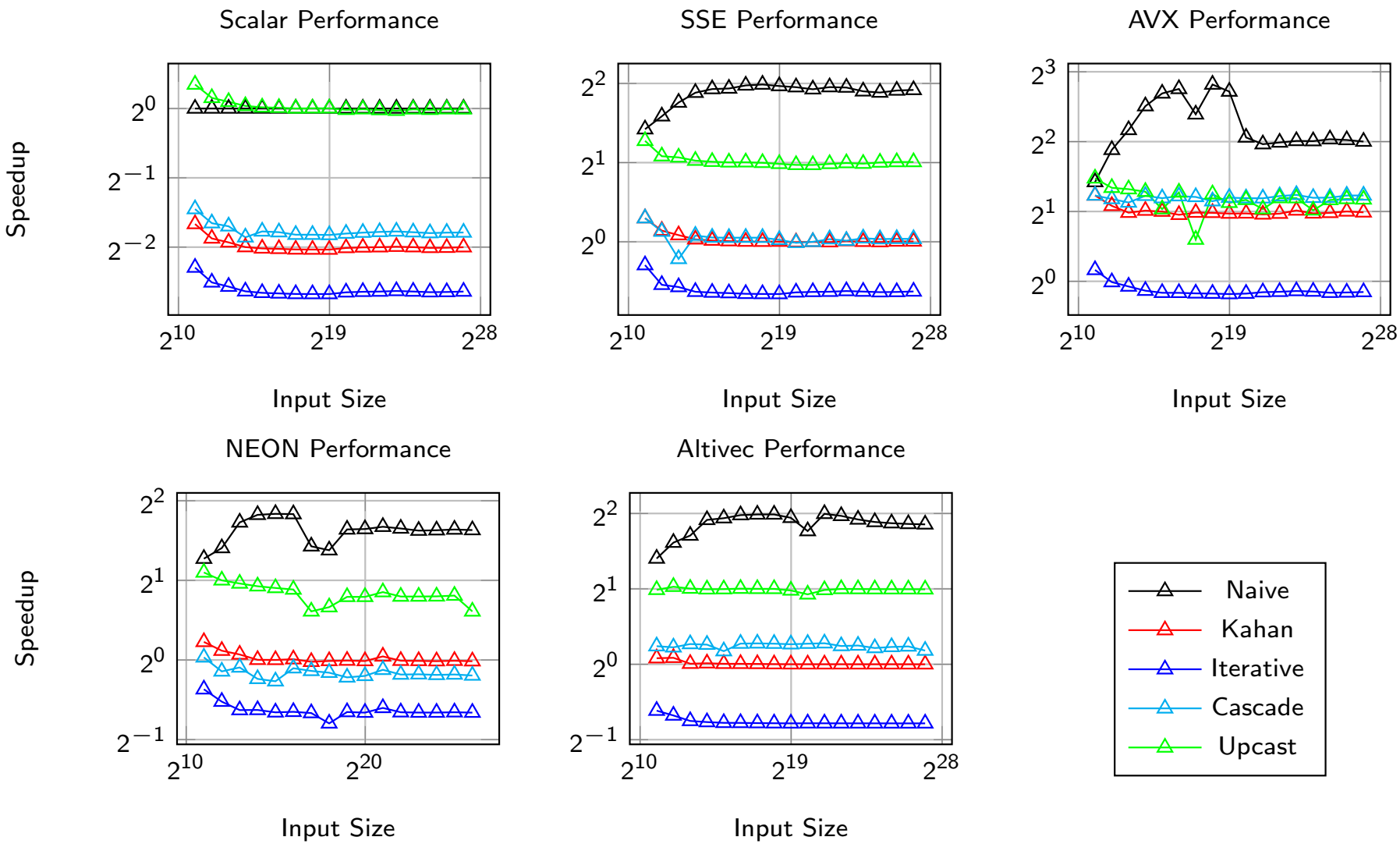
Cascading Average

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	$\text{Input}[i] = \text{s} \cdot i$					
s = 1	100	9	1	0	0	0
	1000	fail	fail	0	0	0
	10000	fail	fail	993	0	1
s = 0.001	100	16	1	20	0	2
	1000	198	1	249	0	4
	10000	fail	fail	1486	0	4
Fixed Ratio	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
r = N/2	100	1	1	17	0	0
	1000	fail	fail	3	0	0
	10000	fail	fail	994	0	1
Fixed Diff	$\text{Input}[i] = \begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
d = N/2	100	13	1	23	0	0
	1000	fail	fail	227	0	0
	10000	fail	fail	737	0	0
Random	$\text{Input}[i] = rand()$					
[0, 1]	1000	271	0	249	0	3
[0, 10]	1000	fail	fail	261	0	4
Fixed	$\text{Input}[i] = \text{c}$					
c = 10	1000	152	0	0	0	0
	10000	1070	fail	0	0	0
	1000000	1277	fail	0	0	0
Repeating	$\text{Input}[i] = 10 + (i\%3)$					
10 - 12	300	17	0	74	0	1
	3000	709	1	128	0	1
	30000	1998	fail	128	0	1
Image1	2073600	fail	fail	1037	1	3
Image2	2073600	1761	391	1701	12	7

Cascading Average



Performance - Overall



Analysis

- Sum then divide methods overflow easily
- Iterative average method fails for large N
- Kahan sum (with divide when necessary) is the most precise when it works
- Naive method is fast enough to increase precision and still be faster (with SIMD)
- Cascading Average is never a bad choice thanks to its robustness

Conclusion

- Numerical precision is complicated
- Performance requires effort
- Even simple things can need research

Future Works

- Similar study using 8 bit floats
- Similar study with exotic numerical formats
- Mix algorithms intelligently
- Look into other "solved" algorithms under half precision

THE END

Full Precision Results

Input	N	Naive	Kahan	Iterative	Upcast	Cascade
Sequential	Input[i] = s $\cdot$ i					
s = 1	100	9	1	0	0	0
	1000	fail	fail	0	0	0
	10000	fail	fail	993	0	1
s = 0.001	100	16	1	20	0	2
	1000	198	1	249	0	4
	10000	fail	fail	1486	0	4
s = -1	100	9	1	0	0	0
	1000	fail	fail	0	0	0
	10000	fail	fail	993	0	1
Fixed Ratio	Input[i] = $\begin{cases} \text{even: } i \div 2 \\ \text{odd: } i \div (2 \cdot \text{r}) \end{cases}$					
r = N/2	100	1	1	17	0	0
	1000	fail	fail	3	0	0
	10000	fail	fail	994	0	1
Fixed Diff	Input[i] = $\begin{cases} \text{even: } i \div 2 \\ \text{odd: } (i \div 2) + \text{d} \end{cases}$					
d = N/2	100	13	1	23	0	0
	1000	fail	fail	227	0	0
	10000	fail	fail	737	0	0
Random	Input[i] = $rand()$					
[0, 1]	1000	271	0	249	0	3
[0, 10]	1000	fail	fail	261	0	4
[0, 100]	1000	fail	fail	246	0	2
[0, 1000]	1000	fail	fail	253	0	3
Fixed	Input[i] = c					
	1000	152	0	0	0	0
c = 10	10000	1070	fail	0	0	0
	100000	1259	fail	0	0	0
	1000000	1277	fail	0	0	0
Repeating	Input[i] = 10 + (i%3)					
10 - 12	300	17	0	74	0	1
	3000	709	1	128	0	1
	30000	1998	fail	128	0	1
	300000	1401	fail	128	0	1
Image1	2073600	fail	fail	1037	1	3
Image2	2073600	1761	391	1701	12	7