SIMDizing Pairwise Sums

A summation algorithm balancing accuracy with throughput

Barnaby Dalton
IBM Canada
bdalton@ca.ibm.com

Amy Wang
IBM Canada
aktwang@ca.ibm.com

Bob Blainey
IBM Canada
blainey@ca.ibm.com

Abstract

Implementing summation when accuracy and throughput need to be balanced is a challenging endevour. We present experimental results that provide a sense when to start worrying and the expense of the various solutions that exist. We also present a new algorithm based on pairwise summation that achieves 89% of the throughput of the fastest summation algorithms when the data is not resident in L1 cache while eclipsing the accuracy of signifigantly slower compensated sums like Kahan summation and Kahan-Babuska that are typically used when accuracy is important.

Categories and Subject Descriptors G.1.0 [*General*]: Computer Arithmetic

Keywords accurate summation, pairwise summation, fast algorithms

1. Introduction

Finite-precision floating point summation is a very common underlying operation for many statistical and mathematical computations. It is a foundational building block used in other kernels which are in turn ubiquitous, such as variance, dot product and matrix multiply computations. One important need in kernels requiring summation is balancing the tradeoff between performance and accuracy.

Naïve summation, which simply adds each successive number in sequence to an accumulator, produces optimal performance that quickly saturates memory bandwidth on longer sequences. It is well-understood by modern compilers and is able to utilize maximal instruction-level parallelism even on short loops.

At the other extreme of the tradeoff, arbitrary precision floating point arithmetic affords the maximal accuracy that can be expressed in a finite-precision binary number but is several times slower. The GNU multi-precision arithmetic library [1] in an example of a mature implementation which provides this.

Instead of arbitrary precision, the extra precision may be fixed at some level higher than required of the original answer. Summing single-precision floats with a double-precision accumulator or using compensated sums is an example of this technique. Essentially the extra bits are maintained as satellite data after each term is summed and are remixed into the summation as course-corrections. Using a higher precision accumulator or Kahan summation [3] is an example where the correction is applied after each term. Kahan-Babuska [4] summation, on the other hand applies the correction once at the end of the summation. Compensated sums perform better than arbitrary-precision arithmetic summation and capture most of their accuracy.

Another form of summation which offers excellent accuracy but typically poor performance is pairwise summation. In pairwise summation, the summation is reassociated into a balanced binary expression tree prior to evaluation. It provides the best general error-bound to a summation which does not make use of compensated summation [2], or additional assumptions such as sorted data. In this paper we investigate optimization opportunities for pairwise sums and propose a new algorithm for computing pairwise sums with performance approaching that of naïve summation and accuracy near to compensated sums.

WPMVP '14, February 16, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2653-7/14/02. . . $15.00.
http://dx.doi.org/10.1145/2568058.2568070

2. Algorithm

In this section we examine pairwise summation in more detail and explore opportunities for performance tuning with SIMD instructions. Pairwise summation provides a compromise of accuracy between naïve summation and compensated sums such as Kahan and Kahan-Babuska, however, it is frequently overlooked because it typically has poorer performance than compensated sums.

Pairwise summation reassociates a summation into a balanced binary expression tree. It has a well known error bound [2] which increases logarithmically with the length of the sequence as opposed to linearly for naïve summation or a constant error bound for compensated sums.

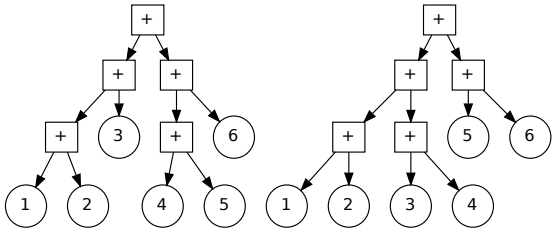

Figure 1. Two possible balanced expression trees for 1+2+3+4+5+6

To maintain the error bound, it is not essential that the path-length from the trees differs by at most 1; the path-length only

needs to be bounded by $\lceil log_2\ n \rceil$. This gives some freedom in selection of how to reassociate the terms. See figure 1 for an example of two possible balanced expression trees for the same sequence; the first typically created by recursive pairwise summation, the latter evaluated by our proposed iterative version.

2.1 Recursive Implementation

The simplest way to implement pairwise summation is to recursively subdivide the sequence in half and add the pairwise summation of each subsequence; The pairwise summation of a single element acts as a base case and is its own sum. This introduces at least as many function calls as there are elements. The function call overhead can not be eliminated and inhibits instruction-level parallelism of useful work. Furthermore, subdivision, even if stopped prematurely introduces misalignment which makes SIMDization challenging.

```
double pairwise_sum(double* x, size_t n) {
  if(n == 1) return *x;
  else return pairwise_sum(x, n/2)
            + pairwise_sum(x+n/2, n-n/2);
}
```

Figure 2. Pairwise summation

A viable tradeoff between performance and accuracy is possible by having a cut-off threshold where pairwise summation switches to naïve summation when the length of the sequence falls below a threshold k. Common values of k that yield performance improvements without large sacrifices in accuracy are 64 or 128. Because this reduces the number of function calls to $1/k$ of the original pairwise summation, the performance benefits are large.

Additionally, if the division into subsequences is rounded to the nearest alignment boundary, then the sum of k elements may be performed with an optimized SIMD loop without any prologue or epilogue.

```
// assume x is aligned to a SIMD boundary
// assume n is a multiple of the SIMD vector length
double pairwise_sum2(double* x, size_t n) {
  if(n <= 64) {
    // no prologue or epilogue necessary.
    return naive_aligned_simd_sum(x,n);
  } else {
    // midpoint is aligned to a SIMD boundary.
    size_t mid = (n/2) & ~0x03ul;
    return pairwise_sum2(x, mid)
         + pairwise_sum2(x+mid, n-mid);
  }
}
```

Figure 3. SIMDized pairwise summation with threshold

2.2 Iterative Implementation

We propose a bottom-up algorithm to compute pairwise sums. It has the following advantages over top-down pairwise sums:

- It is iterative, rather than recursive, hence avoids function call overhead and opens opportunities for instruction-level parallelism and loop-optimization techniques.
- It can be readily optimized with SIMD instructions without increasing the error bound.
- It is easier to unroll without paying the penalty for a branch.

We present performance and accuracy measurements of the shift-reduce summation algorithm in comparison with a compensated summation algorithm and the naive loop-based summation algorithm in the experimental section.

The inspiration for this technique is heavily influenced by the transition from creating top-down recursive descent parsers to bottom-up shift-reduce parsers. The idea is to treat terms at different levels of the balanced expression tree as distinct grammar rules. For example, the summation of $x_1 + x_2 + \cdots + x_7 + x_8$ has the following grammar rules:

- Two adjacent x terms may be replaced by an a term which is their sum. e.g., $a_1 = x_1 + x_2$.
- Two adjacent a terms may be replaced by a b term which is their sum. e.g., $b_2 = a_3 + a_4$.
- Two adjacent b terms may be replaced by a c term which is their sum.
- etc.

The summation iteratively goes through the following states:

sequence	stack	operation
$x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8$	$\emptyset$	shift $x1$
$x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8$	x_1	shift x_2
$x_3\ x_4\ x_5\ x_6\ x_7\ x_8$	$x_1\ x_2$	reduce $a_1 = x_1 + x_2$
$x_3\ x_4\ x_5\ x_6\ x_7\ x_8$	a_1	shift x_3
$x_4\ x_5\ x_6\ x_7\ x_8$	$a_1\ x_3$	shift x_4
$x_5\ x_6\ x_7\ x_8$	$a_1\ x_3\ x_4$	reduce $a_2 = x_3 + x_4$
$x_5\ x_6\ x_7\ x_8$	$a_1\ a_2$	reduce $b_1 = a_1 + a_2$
$x_5\ x_6\ x_7\ x_8$	b_1	shift x_5
$x_6\ x_7\ x_8$	$b_1\ x_5$	shift x_6
$x_7\ x_8$	$b_1\ x_5\ x_6$	reduce $a_3 = x_5 + x_6$
$x_7\ x_8$	$b_1\ a_3$	shift x_7
x_8	$b_1\ a_3\ x_7$	shift x_8
$\emptyset$	$b_1\ a_3\ x_7\ x_8$	reduce $a_4 = x_7 + x_8$
$\emptyset$	$b_1\ a_3\ a_4$	reduce $b_2 = a_3 + a_4$
$\emptyset$	$b_1\ b_2$	reduce $c_1 = b_1 + b_2$
$\emptyset$	c_1	done!

Figure 4. Steps in shift-reduce summation

If the sequence length is not a power of two, then there will be some remaining terms on the stack that are irreducible upon completion. To handle this case while maintaining the path-length invariant of the balanced tree, we simply sum all terms on the stack in reverse-order as soon as all terms of the original sequence have been shifted.

A fairly clear pattern emerges that we use to simplify our implementation greatly. An a-reduction occurs after every second shift. A b-reduction occurs after every four shifts. A c reduction after every 8.

Alternatively, we can formulate the pattern differently. After n shifts, exactly $ntz(n)$ reductions occur, where ntz is the number of trailing zeros of the binary representation of n. This leads to the implementation of shift-reduce summation in figure 5.

2.3 Tuning shift-reduce summation

Observing that reductions only occur every second shift, we can unroll the loop once, shifting two elements which we immediately reduce. In fact, we may unroll the loop any power of two iterations and perform the balanced binary-tree explicitly. We show the algorithm unrolled 4 times in figure 8. We must also handle residual elements leftover if the sequence is not a multiple of the unrolling factor in an epilogue loop.

```
double shift_reduce_sum(double* x, size_t n) {
  double stack[64], v;
  size_t p = 0;
  for(size_t i=0; i<n; ++i) {
    v = x[i]; // shift
    for(size_t b=1; i & b; b<<=1, --p) // reduce
      v += stack[p-1];
    stack[p++] = v;
  }
  double sum=0.0;
  while(p) sum += stack[--p];
  return sum;
}
```

Figure 5. Shift-reduce summation

```
double shift_reduce_sum_unrolled(double* x,
                                 size_t n)
{
  double stack[64], v;
  size_t p = 0;
  for(size_t i=0; i<n; i+=4) {
    // shift
    v = x[i] + x[i+1];
    w = x[i+2] + x[i+3];
    v = v + w;
    // reduce
    for(size_t b=4; i & b; b<<=1, --p)
      v += stack[p-1];
    stack[p++] = v;
  }
  double sum=0.0;
  size_t epilogue = n & ~3ul;
  for (size_t i=epilogue; i<n; ++i)
    sum += x[i];
  while(p) sum += stack[--p];
  return sum;
}
```

Figure 6. Shift Reduce Algorithm Unrolled by 4

To further improve on the performance, the shift-reduce kernel is SIMD optimized. A SIMD optimized example, assuming a 2-way SIMD with 8 numbers to be reduced, is shown in Figure 7. SIMD optimization should be advantageously applied to the unrolled version of the shift-reduce algorithm. The SIMD optimized algorithm follows similar process as explained above except each SIMD addition achieves effectively 2 scalar additions. For example, b1 and b2 are added onto b3 and b4, giving a1 and a2 all in SIMD registers. This process can be easily extended to handle 4-way, or 8-way SIMD if the architecture supports it. Two important aspects to note is that the final SIMD result which contains 2 accumulated sums, $\sigma 1$ and $\sigma 2$, in this example, needs to be further reduced down to one result. In the case of a 4-way SIMD, there would be four sub-sums, e.g. $\sigma 1$, $\sigma 2$, $\sigma 3$, $\sigma 4$, that need to be reduced down to one result via a tree like reduction step such as $((\sigma 1 + \sigma 2) + (\sigma 3, \sigma 4))$. Intel provides a horizontal-add SIMD instruction which performs this within a register. When necessary, the loop is peeled to the nearest SIMD alignment boundary before the shift-reduce loop begins. Furthermore, when applying SIMD optimization to the unrolled shift-reduce algorithm, it is important to also apply SIMD to the epilogue loop.

It is important to note that the SIMDized shift-reduce loop is not reassociated identically to the non-SIMDized version. A two-way SIMDized shift-reduce sum is equivalent to two parallel SIMD sums; one summing elements with odd indices, and one with even indices. However, the error bound measurements remain the same as with pairwise summation.

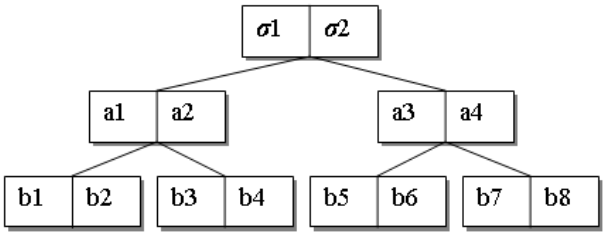

Figure 7. Shift Reduce Algorithm Example with SIMD

```
// assume x is 256-bit aligned
// assume n is a multiple of 16
float shift_reduce_avx(float* x, size_t n)
{
  __m256 stack[60];
  size_t p = 0;
  for(size_t i=0; i+16 <= n; i += 16)
  {
    // unrolled shift+reduce
    __m256 v = _mm256_add_ps( _mm256_load_ps(x+i),
                              _mm256_load_ps(x+i+4));
    __m256 w = _mm256_add_ps( _mm256_load_ps(x+i+8),
                              _mm256_load_ps(x+i+12));
    v = _mm256_add_ps(v,w);

    // reduce
    for(size_t bitmask=16;
        i & bitmask;
        bitmask <<= 1, --p)
    {
      v = _mm256_add_ps(v, stack[p-1]);
      stack[p++] = v;
    }
  }
  // collapse irreducible stack entries
  __m256 vsum = _mm256_setzero_ps();
  for(size_t i=p; i>0; --i)
    vsum = _mm256_add_ps(vsum, stack[i-1]);

  // horizontal sum
  vsum = _mm256_hadd_ps(vsum, vsum);
  vsum = _mm256_hadd_ps(vsum, vsum);
  return _mm_cvtss_f32(
           _mm_add_ss( _mm256_castps256_ps128(vsum),
                       _mm256_extractf128_ps(s, 1) ));
}
```

Figure 8. AVX implementation of shift-reduce with prologue and epilogue loops elided

3. Results

3.1 Experimental Setup

The performance numbers are benchmarked on a 2.2 GHz Sandy-Bridge EP system running RHEL 6 Linux. The system consists of two 8-core sockets with each core being capable of 2-way SMT. Each core has a private L1 (32kB) and L2 (256kB) cache. The L3

cache is shared among all cores of a socket socket and is 20MB in size. All kernels are compiled with Intel 13 compilers.

The input data is a precomputed uniform random sequence of floats or doubles stored in memory. To simulate cache-residency of various working set sizes the elements of each working set size were summed repeatedly with the first iteration omitted. We show several working set sizes to exhibit performance characterisics for data residency in different levels of the cache hierarchy.

3.2 Summation Kernel

We are interested in two metrics in the summation algorithms: accuracy and throughput performance. The first is measured as the base-2 logarithm of the recipricol relative error. This measurement was used because it gives an approximation of the number of bits correct in the mantissa of the floating point value and comparisons of a perfectly accurate result are comparable to minute errors.

The formula used for error calculations is:

$$\log_2 \left| \frac{\chi}{\chi - \sigma} \right|$$

where χ is the summation result performed using arbitrary-precision arithmetic using the GNU multi-precision library [1] and σ is the sum from the respective kernel. When σ and χ agree exactly, we cap the answer at the number of bits the respective mantissa can represent: 24 bits for floats, 53 for doubles.

Note that the formula simplifies to $\log_2 |\chi| - \log_2 |\chi - \sigma|$ which can be internalized as approximately counting the number of bits to which σ and χ agree on.

The kernels evaluated include:

- several implementations of naïve summation; with and without SIMD.
- compensated sums: Kahan and Kahan-Babuska; with and without SIMD.
- shift-reduce summation.

3.3 Accuracy

The relative error remains constant as the range of uniform random numbers is varied. The only exception to this was that compensated sums may produce NaNs, even with very short sequences when the distribution is very large. We encountered this with both Kahan summation and Kahan-Babuska summation both with and without SIMD when we computed a sum-of-squares computation on data distributed between $\pm 10^{21}$.

All accuracy test measurements reported in the graphs used uniform random floats between ± 1000.

An interesting observation is the consistent improvement to accuracy SIMDization yields for Naive summation in figure 9. This is because k-way simd is equivalent to summing k shorter sequences in parallel dampening the resulting error. Both Kahan and Kahan-Babuska get near-perfect results as expected and shift-reduce summation occupies a medium, albeit a consistent one.

When we square the inputs as we sum in figure 10, the error increases dramatically and naive summation is not even able to claim three decimal digits of accuracy on sequences of 4 million elements. Interestingly, the relative-error of shift-reduce summation approaches that of compensated sums in this case.

3.4 Throughput

Figure 11 shows the single thread performance of three different summation algorithms using 4-byte floating point input data. All three algorithms have been SIMD vectorized by hand using AVX or SSE intrinsics. Further, the loop body of each algorithm has been unrolled to achieve better instruction level parallelism.

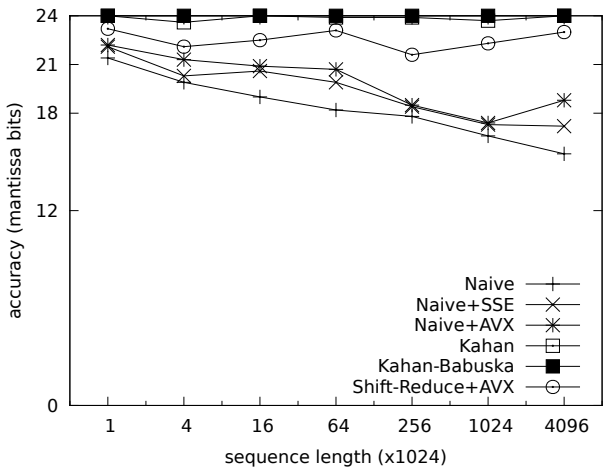

Figure 9. Accuracies with Various Algorithms (single precision float input)

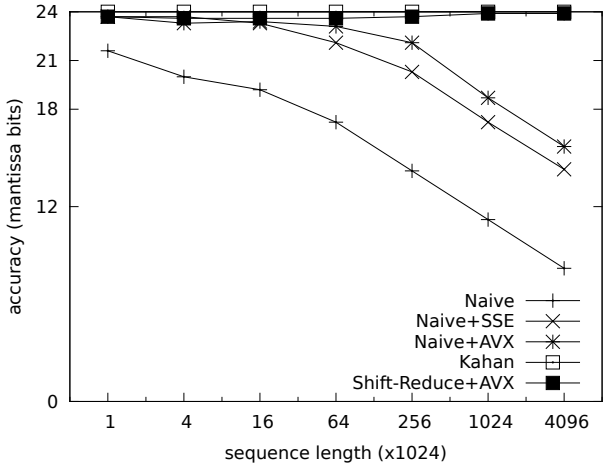

Figure 10. Accuracies with Various Sum of Square Algorithms (single precision float input)

There is no surprise that naïve summation achieves the highest throughput. When the data is resident in L1 cache (32Kb or 8K elements), it is faster by a wide margin. It reaches its peak throughput with AVX of 22.5 billion elements per second with 8K element sequences. A second, smaller, dropoff near 32K elements occurs when the L2 cache is fully utilized; naïve summation claims about 8.5 billion elements per second here. It stabilizes at about 6.5 billion elements per second streaming from memory.

In comparison, Shift-reduce only achieves about 10 billion elements per second on 8K element sequences and Kahan-Babuska is short of 4 billion elements per second here. At the L2-threshold, Shift-reduce achives just short of 7.5 billion elements per second and Kahan-Babuska holds stable shy of 4 billion elements per second. When streaming from memory, Shift-reduce maintains about 5.9 billion elements per second and Kahan-Babuska about 3.3 billion elements per second.

Shift-reduce summation enjoys 86% of the throughput of naïve summation when streaming from L2 and 90% when streaming from memory. However, when sequences are small enough to fit in L2,

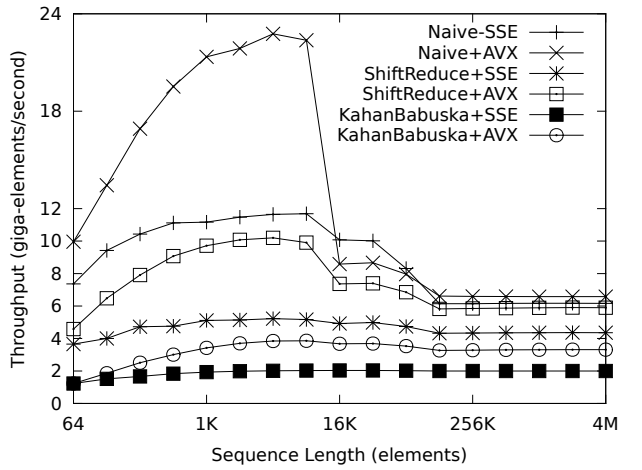

Figure 11. Throughput Comparisons between Summation Algorithms

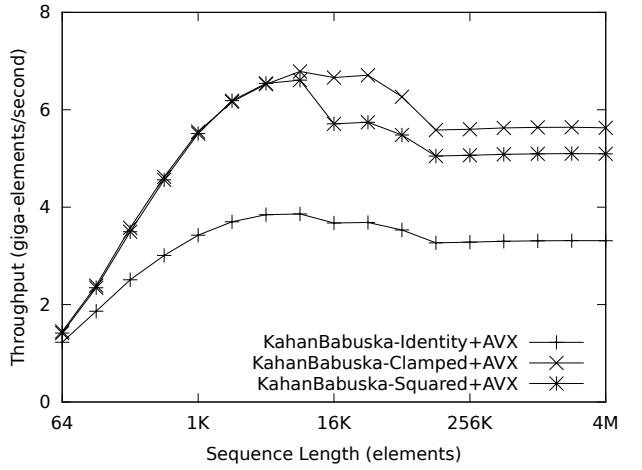

Figure 12. Performance of Kahan Babuska Algorithms with Various Input Transformations

the loss of accuracy to using naïve summation, as seen in figure 9 is negligible if SIMD is used. When naïve summation begins to degrade in accuracy, near about 128K elements, the performance gap is narrow.

The two cliffs are less obvious with the shift reduce algorithm using SSE and with the Kahan-Babuska algorithm. This is because the core is not drawing the data fast enough given the extra computations that need to be performed. Being the algorithm with highest accuracy, Kahan-Babuska is the most computationally intensive algorithm. The if-converted Kahan-Babuska algorithm which makes SIMD vectorization possible is shown in Figure 13. The accuracy improvement of Kahan-Babuska over Shift-Reduce summation is marginal while getting only half of the throughput. This suggests Shift-Reduce summation as a serious candidate when accurate sums are desired.

Kahan-Babuska is a well known algorithm which exhibits the same accuracy as the original Kahan algorithm. By doing the error compensation outside of the accumulation loop, it avoids the loop carried dependence that is in the original Kahan algorithm. As such,

```
float kahan_babuska_sum(float* x, size_t n) {
  float sum=0.0, error=0.0;
  for(size_t i=0; i<n; ++i) {
    float a = (fabs(sum)>fabs(x[i]) ? sum : x[i];
    float b = (fabs(sum)>fabs(x[i]) ? x[i] : sum;
    sum   += x[i];
    error += (a - sum) + b;
  }
  return sum + error;
}
```

Figure 13. The if-converted Kahan-Babuska Algorithm for SIMD Vectorization.

it is amenable for SIMD vectorization. One important advantage we take is in realizing that if the input **x[i]** in Figure 13 is always positive, the compare and select statements (computing a and b) can be turned into usages of a max and a min intrinsics.

For example, if results are clamped to zero or sequared, we see a performance improvement in figure 12. When no transformation or assertion of positive data is made we see performance roughly halved.

All three curves make use of AVX on single precision floating point input and include the transformation overhead. The **identify-float-avx** is performance when no transformation is done on the input data and hence no additional overhead. The **clamped-float-avx** is performance when when negative input values are clamped to zero followed by summation. The **squared-float-avx** is performance when input values are squared followed by summation. All three curves shows cliffs at the drop off of each cache level. However, it is clear that squared and clamped summation achieve much better performance than identity summation despite having the need to do the extra computation to first squaring or clamping the input data. This is attributed to the effective use of min and max intrinsics in the loop body.

One important point is that the shift-reduce algorithm and the Kahan-Babuska algorithm both achieve near theoretical accuracy of 24 bits as shown in Figure 9.

4. Conclusion

The first point that needs to be made is to know when to reach for a different summation algorithm from the iconic C accumulator loop. This is often a difficult judgement call to make, because unlike poor performance, poor accuracy can be difficult to spot.

A good rule of thumb appears to be that you lose about half a bit of accuracy every time you double the length of your sequence. Whether this means you are losing accuracy depends on the range of signifigant bits in your input. The results may be more extreme if you are transforming your data such as in a sum of squares operation where the accuracy deteriorates rapidly. A good threshold to consider shift-reduce summation at is about 100K elements.

A second takeaway from this study is that reassociation towards a balanced binary tree, even by SIMDizing a naïve sum, very quickly improves accuracy.

Finally, the proposed shift-reduce algorithm provides a balanced tradeoff between naïve summation and compensated sums. It matches naïve sums in performance except when they are so short that accuracy is not at risk in the first place. It is easily twice as fast as our best tuned compensated sums and at least 4 or 5 times faster than a reference implementation. More importantly on average it rarely differs in more than the least signifigant bit.

The results scale to doubles as well which were not discussed in great detail in this paper, but the loss of accuracy of naïve summation is less signifigant overall.

References

[1] The gnu multiple precision arithmetic library.

[2] N. J. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. 2002.

[3] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM*, 8:40, January 1965.

[4] A. Neumaier. Rundungsfehleranalyse einiger verfahren zur summation endlicher summen. *ZAMM Journal of Applied Mathematics and Mechanics*, 54:39–51, November 1974.