Fast, good, and repeatable: summations, vectorization, and reproducibility

Brett Neuman, Laura Monroe, Andy DuBois, Bob Robey

Los Alamos National Laboratory

11/17/2019

LA-UR-19-29774

Performance at exascale

- Exascale computing metrics are focused on performance (FLOPs)
- How do we achieve performance beyond Moore's law?
- Areas of focus for exascale performance:
  - Parallelism
  - Vectorization
  - Multithreading
  - Multicore
- Exascale will have a larger variety of precisions
- Are there any drawbacks to increasing performance through more parallelism?

Reproducibility at exascale

- Increased parallelism will also increase reproducibility issues
- The same parallel techniques for performance will lead to lower reproducibility
- Larger problem sizes are likely to lead to lower reproducibility
- Global summations run in parallel may be non-deterministic
- What are the major sources of reproducibility in scientific codes?

Global summations and rounding errors

- Rounding errors are a fundamental issue with reproducibility in large parallel codes
  - Global summations shown to be major sources of non-reproducible results
    - In a fluid dynamics simulation the global sum was a major source of inconsistency in mass and energy sums

      R. W. Robey, J. M. Robey, and R. Aulwes, "In search of numerical consistency in parallel programming," Parallel Computing, vol. 37, no. 4-5, pp. 217–229, 2011.
  - Currently, there are no community standards for acceptable reproducibility thresholds on exascale systems

    L. Pouchard, S. Baldwin, T. Elsethagen, J. Shantenu, B. Raju, E. Stephan, L. Tang, and K. Kleese Van Dam, "Computational reproducibility of scientific workflows at extreme scales," The International Journal of High Performance Computing Applications, pp. 1–14, 2019.

Improving reproducibility with minimal impact to performance

Global summation reproducibility is a concern..

- Enhanced-precision sum is a good solution for reproducibility
- Additional floating point operations are bad for performance
- Compiler can't automatically vectorize enhanced-precision which is bad
- Naïve vectorization is bad for reproducibility
- Combine vectorization and enhanced-precision sum - all good!
- Better reproducibility at little to no cost is good

Prior work on summations

- Prior work to reduce global summation reproducibility issues:
  - Serial efforts
    - Pairwise method work performed by MacCracken

      D. D. MacCracken and W. S. Dorn, Numerical methods and fortran programming: with applications in engineering and science. J. Wiley, 1964.
    - High precision libraries by David Bailey

      D. H. Bailey, "High-precision floating-point arithmetic in scientific computation," Computing in Science Engineering, vol. 7, no. 3, pp. 54–61, May 2005
    - Basic Linear Algebra Subprograms (BLAS)
      - ReproBLAS at UC Berkeley

        J. Demmel, H. D. Nguyen, and P. Ahrens, "Cost of floating-point reproducibility," https://www.nist.gov/sites/default/files/documents/itl/ssd/is/NRE-2015-07-Nguyen slides.pdf, Nov 2015.

        P. Ahrens, H. D. Nguyen, and J. Demmel, "Efficient reproducible floating point summation and BLAS," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-229, 2015
      - ExBLAS at KTH Royal Institute of Technology, Sweden

        R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS: Reproducible and accurate BLAS library," 2015.

        S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Numerical reproducibility for the parallel reduction on multi- and manycore architectures," Parallel Computing, vol. 49, pp. 83 – 97, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115001155

        R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS: Reproducible and accurate BLAS library," https://www.nist.gov/sites/default/files/documents/itl/ssd/is/NRE-2015-04-iakymchuk.pdf, Nov 2015.

Prior work on summations (cont.)

- Prior work to reduce global summation reproducibility issues:
  - Serial efforts
    - Compensated-summation techniques Kahan [1] and Knuth [2]
    - Carries a remainder value in a second variable
    - Includes the part of the number which cannot be represented in standard finite-precision
      - Kahan: Assumes one operand is larger in magnitude
      - Knuth: Computes correction term for both operands
    - **Kahan and Knuth require additional floating point operations..**

```
double do_kahan_sum(double* restrict var, long
    ncells)
{
   struct esum_type{
      double sum;
      double correction;
   } local;
   local.sum = 0.0;
   local.correction = 0.0;

   for (long i = 0; i < ncells; i++) {
      double corrected_next_term = var[i] +
          local.correction;
      double new_sum = local.sum + local.correction;
      local.correction = corrected_next_term -
          (new_sum - local.sum);
      local.sum = new_sum;
   }

   double sum = local.sum + local.correction;

   return(sum);
}
```

Exploiting parallelism for reproducibility?

- Let's apply some parallel techniques and exploit parallelism for reproducibility to offset the additional FLOPs of Kahan and Knuth summations. How about vectorization?
  - Vectorization can improve performance
    - Naive vectorization would hurt reproducibility
  - Enhanced-precision summation algorithms are ideal candidates for vectorization because the FLOPs increase but loads remain the same
- Can the compiler automatically handle vectorizing Kahan and Knuth algorithms?

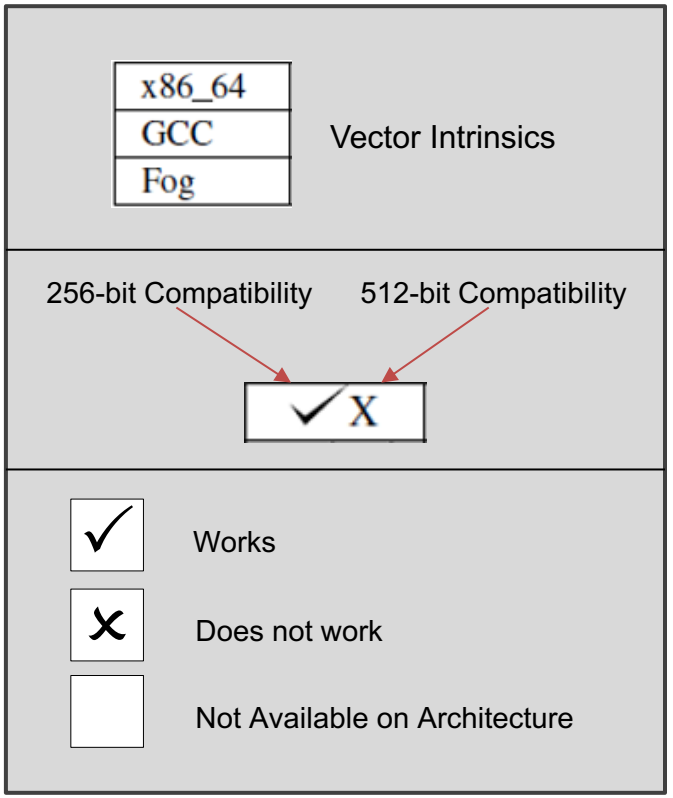
Vectorizing enhanced-precision algorithms

- No.. enhanced-precision sum has a loop-carried dependency that can never be automatically vectorized by a compiler
- Can we vectorize the operations within the algorithm manually?
- Yes, using vector intrinsics:
  - Architecture and compiler determines vector intrinsics that can be used
  - Vector intrinsics:
    - Intel x86 (x86-64)
      - Run on both Intel and AMD
    - GCC vector extensions
      - Using GCC compiler on a variety of architectures
    - Agner Fog vector class library
      - Implementations in C++

Vector Intrinsics and portability

- Vector intrinsics provide performance benefits at the cost of portability
  - Designer should plan for the minimum set of vector intrinsics based on performance and portability needs for their platform

Platform	Clang 8.0.1	GCC 9.2.0	Intel 19.0.5	PGI 19.7	XLC 16.1.1.1.3
Skylake-Gold					
x86_64	✓ X	✓ X	✓ X	X	X
GCC	✓ ✓	✓ ✓	✓ ✓	X	X
Fog	✓ ✓	✓ ✓	✓ ✓	X	X



Portability chart

Platform	Clang 8.0.1	GCC 9.2.0	Intel 19.0.5	PGI 19.7	XLC 16.1.1.1.3
Skylake-Gold					
x86_64	✓X	✓X	✓X	X X	
GCC	✓✓	✓✓	✓✓	X X	
Fog	✓✓	✓✓	✓✓	X X	
AMD 7551 32 core					
x86_64	✓X	✓X	✓X	X X	
GCC	✓✓	✓✓	✓✓	X X	
Fog	✓✓	✓✓	✓✓	X X	
ARM ThunderX2-B0					
x86_64	X X	X X			
GCC	✓✓	✓✓			
Fog	X X	X X			
Power 9					
x86_64	X X	X X		X X	X X
GCC	✓✓	✓✓		X X	X X
Fog	X X	X X		X X	X X

TABLE I: Portability of vector intrinsic implementations. The first of the two marks in each cell indicates the 256-bit implementation and the second is the 512-bit version.

A check mark indicates the vector intrinsics work and the X means that it does not.

Blank cells indicate that the compiler is not available for that architecture.

256-bit implementations are supported by more platforms but 512-bit offers best performance.

Implementation

Vector Implementations of Reproducible Sums

1. Load four values from a standard array into a vector variable
2. The standard Kahan or Knuth operation is done on all four-wide vector variables
3. Store the four vector lanes into a regular, aligned array of four values
4. Sum the four sums from the four vector lanes using scalar variables

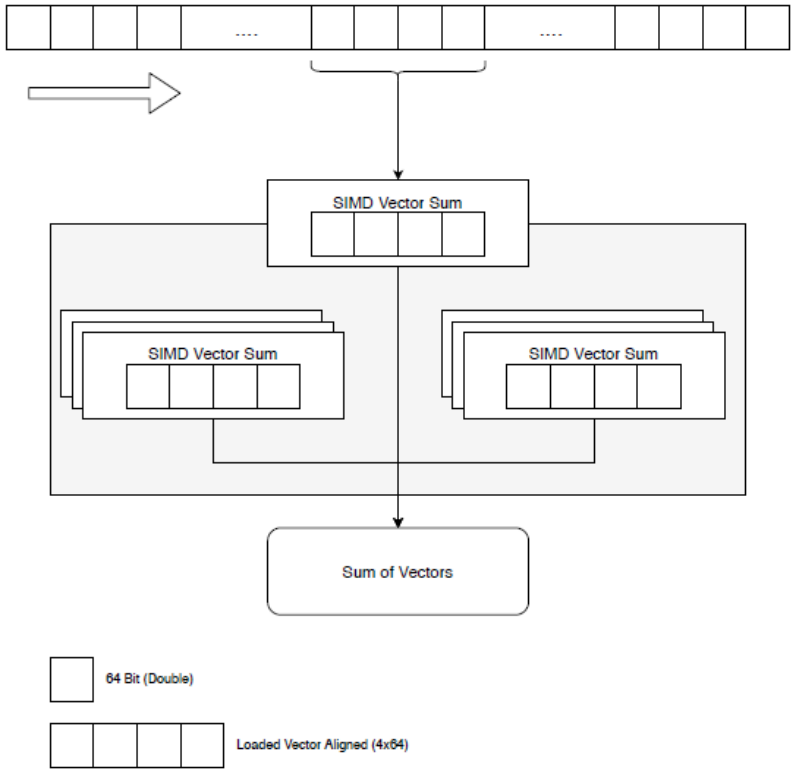


Fig. 1: Vector aligned summation

Platforms – Can we achieve more from 512-bit vector units

- Two clusters were used to test scalability up to 512-bit vector unit supported architecture:
  - Potatohead: Heterogeneous experimental cluster
    - Intel Xeon E5-2650 Sandy Bridge
    - AVX2
    - Used for 256-bit vector units
  - Darwin: Experimental cluster with various CPUs and GPUs
    - Skylake-Gold 6152
    - AVX-512
    - Used for 512-bit vector units

256-bit performance and reproducibility – Sandy Bridge

X-Axis: Serial (non-vectorized), Serial OpenMP is pragma simd, then 256 or 512 bit vector lengths for Kahan or Knuth

- *If the graph label has (GCC / Fog) then the results of that vector intrinsic differ from the others. If there is no label, then the results were the same across all vector intrinsics

Y-Axis: Total runtime for summation (**lower is better**)

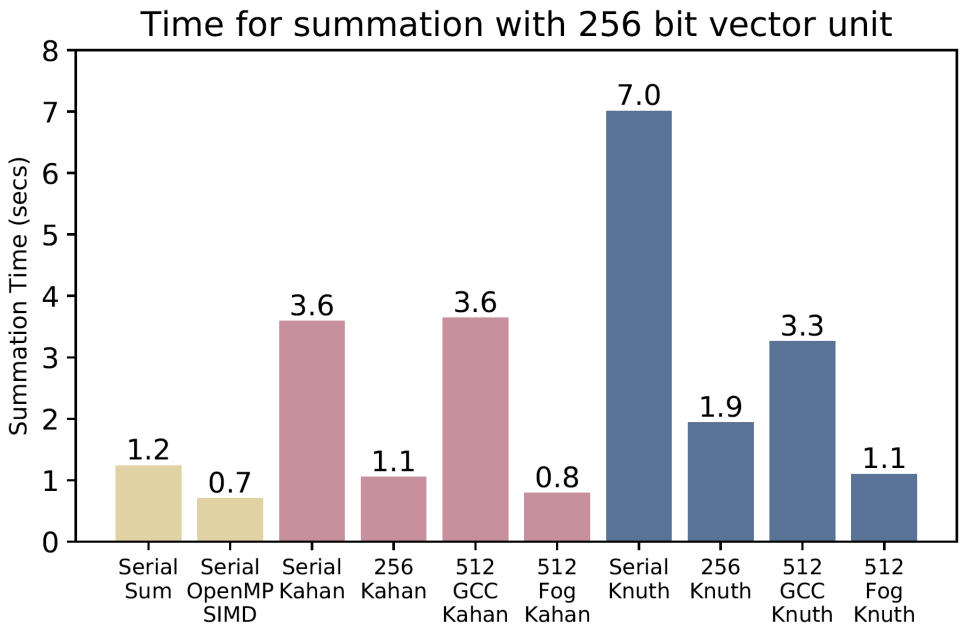


Fig. 2: For the Sandy Bridge CPU, the vectorized Kahan and Knuth summations speedup the enhanced precision methods by over 3x to almost the runtimes of the serial sums.

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.242
Serial Sum OpenMP SIMD	-3.356e-09	0.706
Kahan Sum	0	3.590
256 bit vector Kahan (All)	0	1.052
512 bit vector Kahan (GCC)	-1.388e-16	3.643
512 bit vector Kahan (Fog)	-1.388e-16	0.792
Knuth Sum	0	7.010
256 bit vector Knuth (All)	0	1.942
512 bit vector Knuth (GCC)	0	3.260
512 bit vector Knuth (Fog)	0	1.100

TABLE II: Results of different summation methods on a 256-bit vector unit (Sandy Bridge)

256-bit performance and reproducibility – Sandy Bridge

Kahan implementation has 3.4x speedup compared to serial Kahan

Faster than non-vectorized serial sum

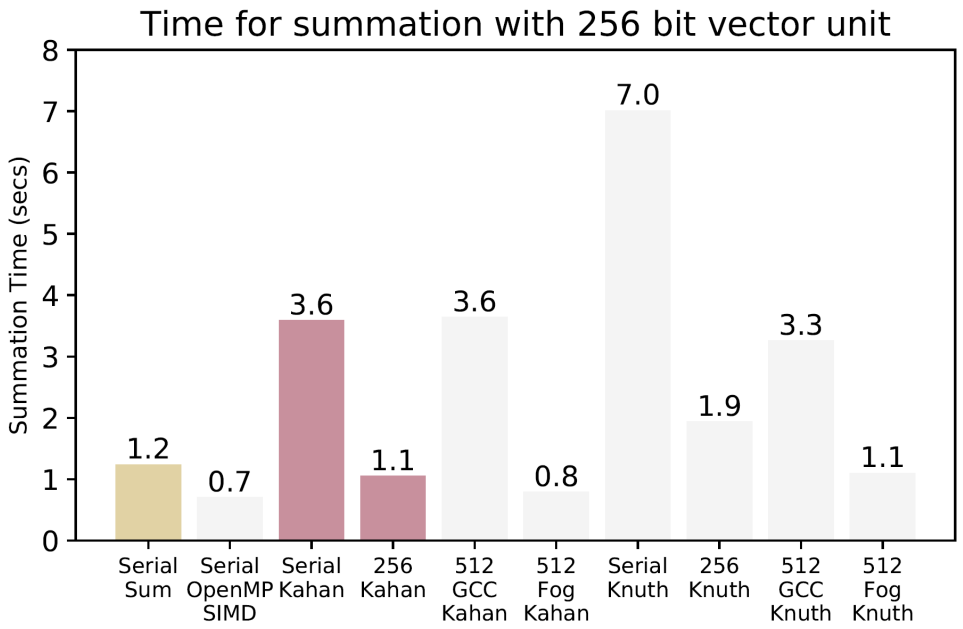


Fig. 2: For the Sandy Bridge CPU, the vectorized Kahan and Knuth summations speedup the enhanced precision methods by over 3x to almost the runtimes of the serial sums.

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.242
Serial Sum OpenMP SIMD	-3.356e-09	0.706
Kahan Sum	0	3.590
256 bit vector Kahan (All)	0	1.052
512 bit vector Kahan (GCC)	-1.388e-16	3.643
512 bit vector Kahan (Fog)	-1.388e-16	0.792
Knuth Sum	0	7.010
256 bit vector Knuth (All)	0	1.942
512 bit vector Knuth (GCC)	0	3.260
512 bit vector Knuth (Fog)	0	1.100

TABLE II: Results of different summation methods on a 256-bit vector unit (Sandy Bridge)

256-bit performance and reproducibility – Sandy Bridge

Knuth implementation has 3.6x speedup compared to serial Knuth

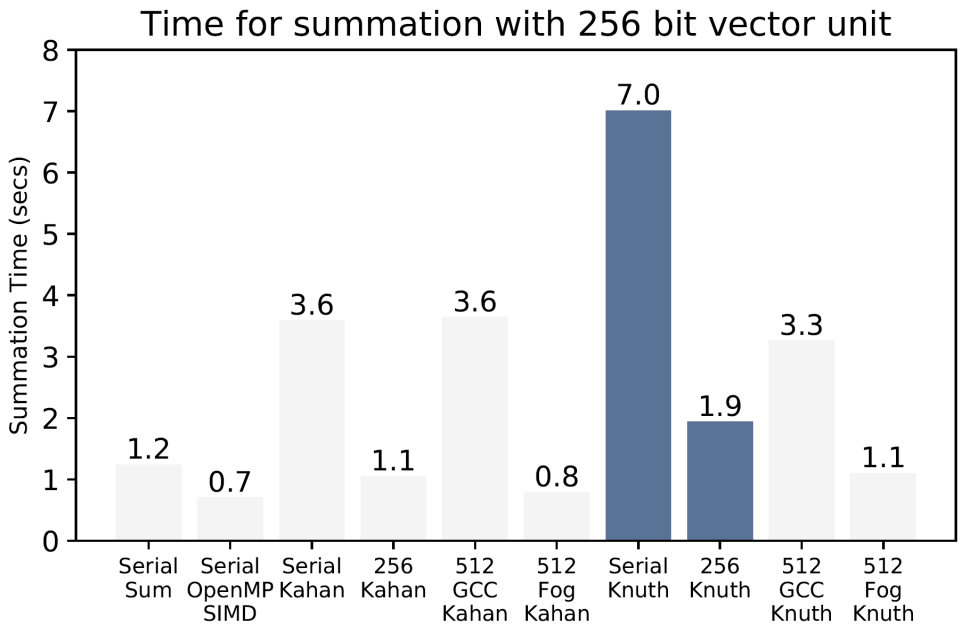


Fig. 2: For the Sandy Bridge CPU, the vectorized Kahan and Knuth summations speedup the enhanced precision methods by over 3x to almost the runtimes of the serial sums.

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.242
Serial Sum OpenMP SIMD	-3.356e-09	0.706
Kahan Sum	0	3.590
256 bit vector Kahan (All)	0	1.052
512 bit vector Kahan (GCC)	-1.388e-16	3.643
512 bit vector Kahan (Fog)	-1.388e-16	0.792
Knuth Sum	0	7.010
256 bit vector Knuth (All)	0	1.942
512 bit vector Knuth (GCC)	0	3.260
512 bit vector Knuth (Fog)	0	1.100

TABLE II: Results of different summation methods on a 256-bit vector unit (Sandy Bridge)

256-bit performance and reproducibility – Sandy Bridge

GCC and Fog handle 256-bit conversion to 512-bit vector units. Fog shows improvement (4.5x Kahan, 6.3x Knuth), GCC not so much (0x Kahan, 2.1x Knuth)

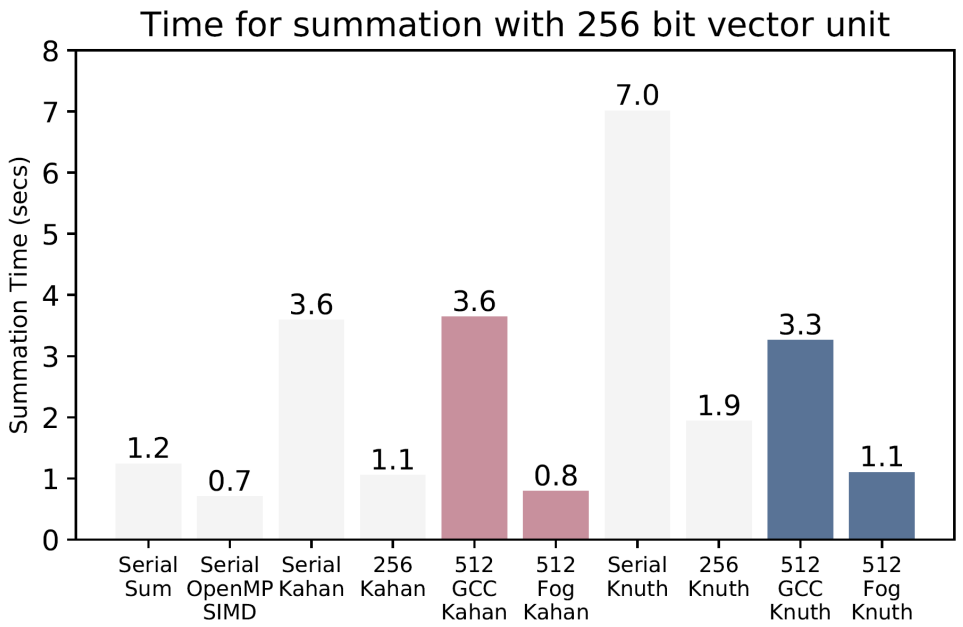


Fig. 2: For the Sandy Bridge CPU, the vectorized Kahan and Knuth summations speedup the enhanced precision methods by over 3x to almost the runtimes of the serial sums.

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.242
Serial Sum OpenMP SIMD	-3.356e-09	0.706
Kahan Sum	0	3.590
256 bit vector Kahan (All)	0	1.052
512 bit vector Kahan (GCC)	-1.388e-16	3.643
512 bit vector Kahan (Fog)	-1.388e-16	0.792
Knuth Sum	0	7.010
256 bit vector Knuth (All)	0	1.942
512 bit vector Knuth (GCC)	0	3.260
512 bit vector Knuth (Fog)	0	1.100

TABLE II: Results of different summation methods on a 256-bit vector unit (Sandy Bridge)

256-bit performance and reproducibility – Sandy Bridge

Relative difference of zero between analytical and calculated value

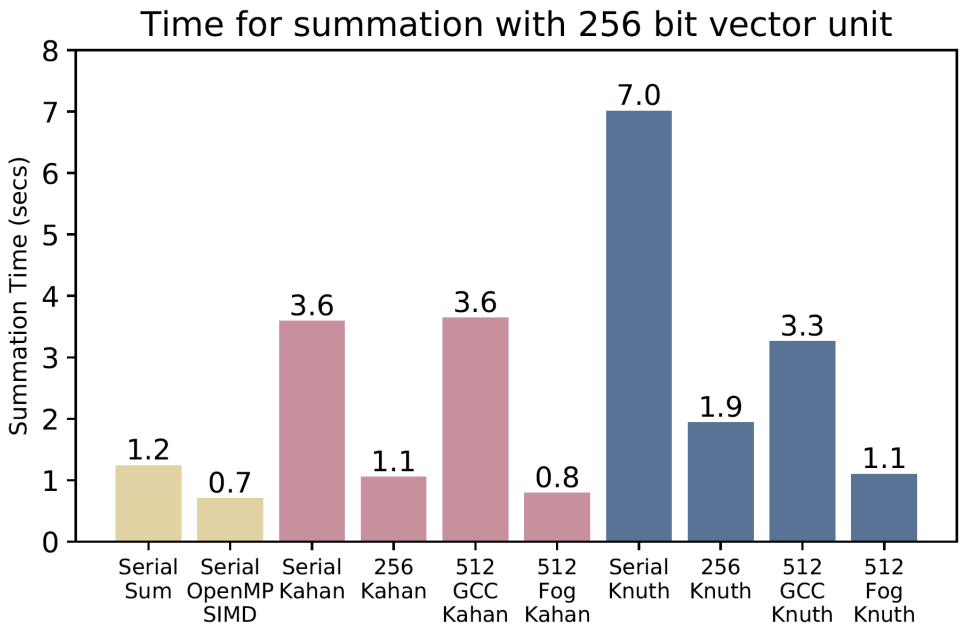


Fig. 2: For the Sandy Bridge CPU, the vectorized Kahan and Knuth summations speedup the enhanced precision methods by over 3x to almost the runtimes of the serial sums.

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.242
Serial Sum OpenMP SIMD	-3.356e-09	0.706
Kahan Sum	0	3.590
256 bit vector Kahan (All)	0	1.052
512 bit vector Kahan (GCC)	-1.388e-16	3.643
512 bit vector Kahan (Fog)	-1.388e-16	0.792
Knuth Sum	0	7.010
256 bit vector Knuth (All)	0	1.942
512 bit vector Knuth (GCC)	0	3.260
512 bit vector Knuth (Fog)	0	1.100

TABLE II: Results of different summation methods on a 256-bit vector unit (Sandy Bridge)

512-bit performance and reproducibility - Skylake

X-Axis: Serial (non-vectorized), Serial OpenMP is pragma simd, then 256 or 512 bit vector lengths for Kahan/Knuth

- *If the graph label has (GCC / Fog) then the results of that vector intrinsic differ from the others. If there is no label, then the results were the same across all vector intrinsics

Y-Axis: Total runtime for summation (**lower is better**)

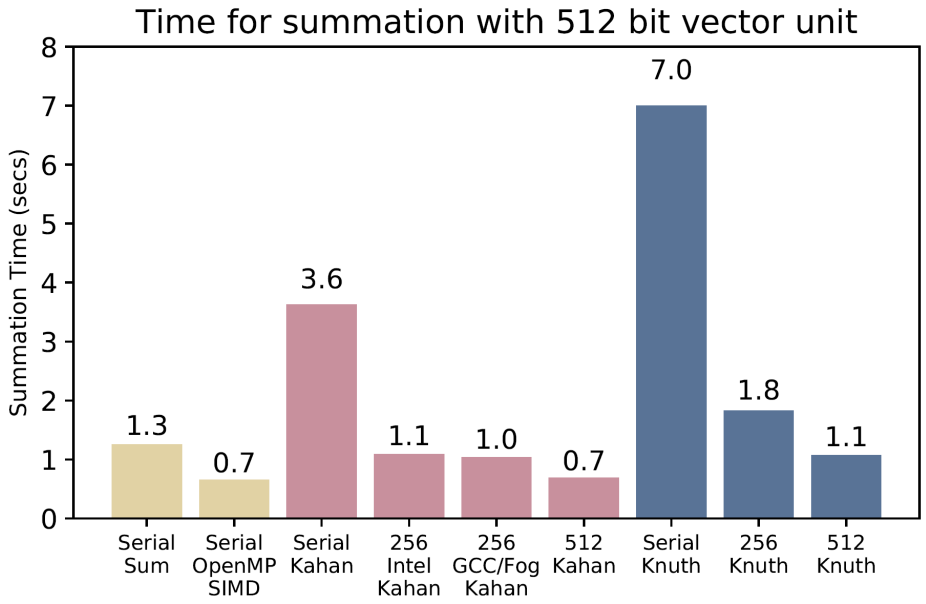


Fig. 3: On the Skylake CPU, the 512-bit vectorized Kahan implementations are as fast as the regular serial summation!

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.260
Serial Sum OpenMP SIMD	-1.986e-09	0.654
Kahan Sum	0.0	3.625
256 bit vector Kahan (Intel)	0.0	1.089
256 bit vector Kahan (GCC/Fog)	0.0	1.034
512 bit vector Kahan (All)	-1.388e-16	0.688
Knuth Sum	0.0	6.998
256 bit vector Knuth (All)	0.0	1.831
512 bit vector Knuth (All)	0.0	1.076

TABLE II: Results of different summation methods on a 512-bit vector unit (Skylake-Gold 6152)

512-bit performance and reproducibility - Skylake

Kahan implementation is as fast as the regular serial summation

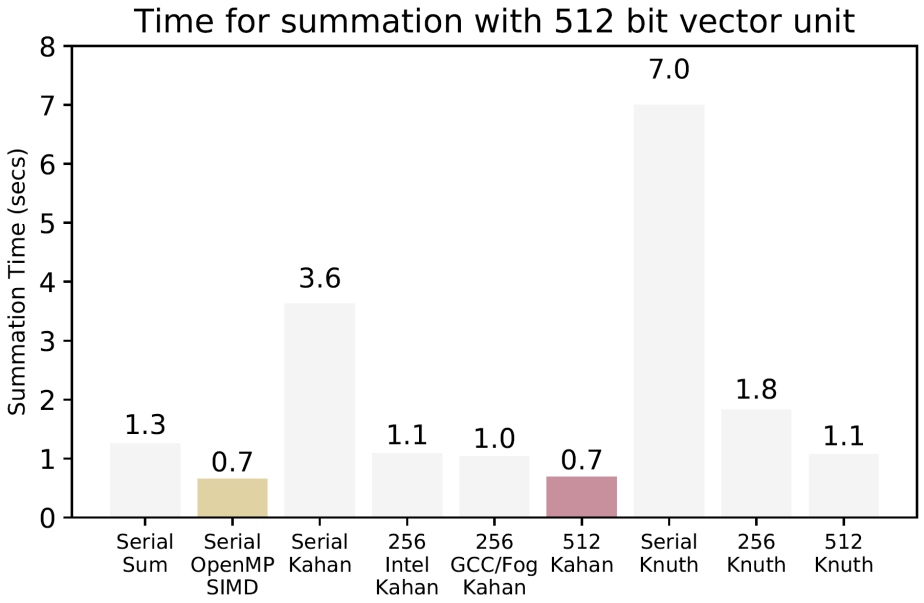


Fig. 3: On the Skylake CPU, the 512-bit vectorized Kahan implementations are as fast as the regular serial summation!

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.260
Serial Sum OpenMP SIMD	-1.986e-09	0.654
Kahan Sum	0.0	3.625
256 bit vector Kahan (Intel)	0.0	1.089
256 bit vector Kahan (GCC/Fog)	0.0	1.034
512 bit vector Kahan (All)	-1.388e-16	0.688
Knuth Sum	0.0	6.998
256 bit vector Knuth (All)	0.0	1.831
512 bit vector Knuth (All)	0.0	1.076

TABLE II: Results of different summation methods on a 512-bit vector unit (Skylake-Gold 6152)

512-bit performance and reproducibility - Skylake

More consistent performance across all compilers

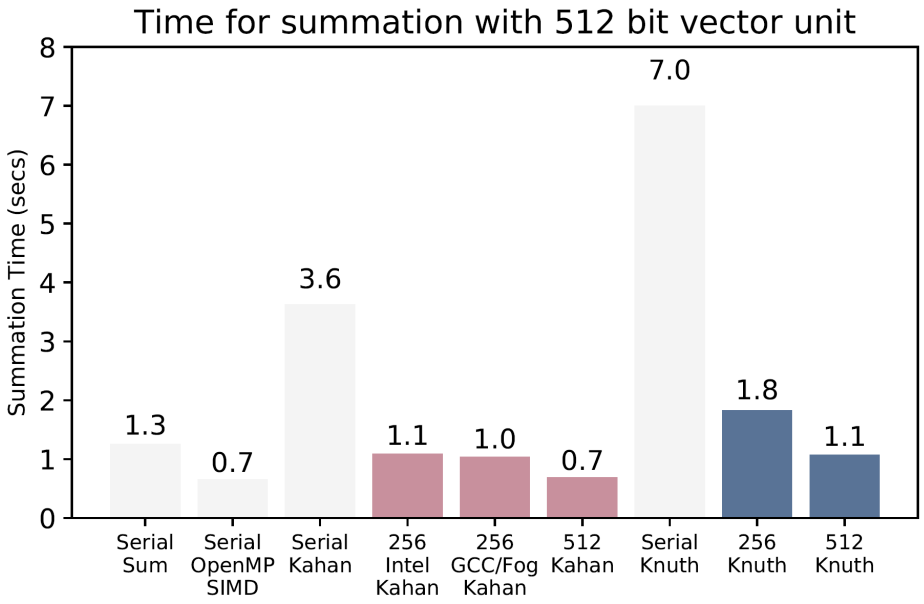


Fig. 3: On the Skylake CPU, the 512-bit vectorized Kahan implementations are as fast as the regular serial summation!

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.260
Serial Sum OpenMP SIMD	-1.986e-09	0.654
Kahan Sum	0.0	3.625
256 bit vector Kahan (Intel)	0.0	1.089
256 bit vector Kahan (GCC/Fog)	0.0	1.034
512 bit vector Kahan (All)	-1.388e-16	0.688
Knuth Sum	0.0	6.998
256 bit vector Knuth (All)	0.0	1.831
512 bit vector Knuth (All)	0.0	1.076

TABLE II: Results of different summation methods on a 512-bit vector unit (Skylake-Gold 6152)

512-bit performance and reproducibility - Skylake

Kahan and Knuth performance improved for GCC

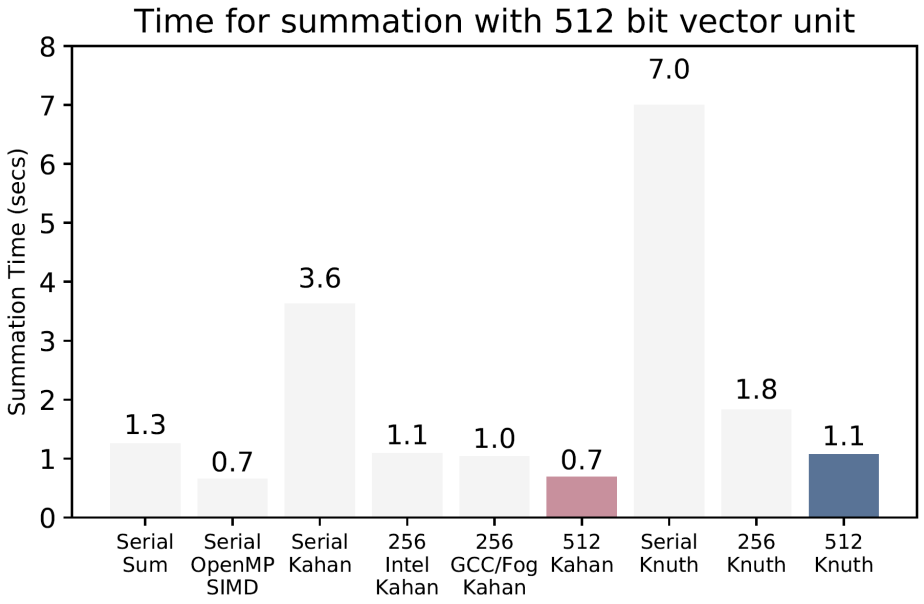


Fig. 3: On the Skylake CPU, the 512-bit vectorized Kahan implementations are as fast as the regular serial summation!

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.260
Serial Sum OpenMP SIMD	-1.986e-09	0.654
Kahan Sum	0.0	3.625
256 bit vector Kahan (Intel)	0.0	1.089
256 bit vector Kahan (GCC/Fog)	0.0	1.034
512 bit vector Kahan (All)	-1.388e-16	0.688
Knuth Sum	0.0	6.998
256 bit vector Knuth (All)	0.0	1.831
512 bit vector Knuth (All)	0.0	1.076

TABLE II: Results of different summation methods on a 512-bit vector unit (Skylake-Gold 6152)

512-bit performance and reproducibility - Skylake

GCC would need different vectorized versions for 256 and 512 bit. Fog would be best as single 512-bit vector version on both 256-bit and 512-bit hardware

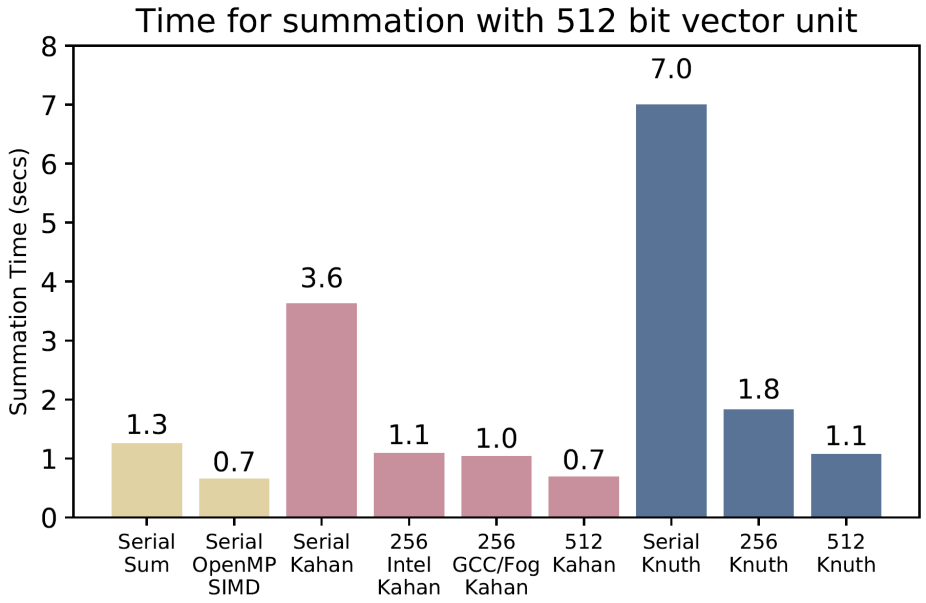


Fig. 3: On the Skylake CPU, the 512-bit vectorized Kahan implementations are as fast as the regular serial summation!

Method	Relative Difference	Runtime (secs)
Serial Sum	8.423e-09	1.260
Serial Sum OpenMP SIMD	-1.986e-09	0.654
Kahan Sum	0.0	3.625
256 bit vector Kahan (Intel)	0.0	1.089
256 bit vector Kahan (GCC/Fog)	0.0	1.034
512 bit vector Kahan (All)	-1.388e-16	0.688
Knuth Sum	0.0	6.998
256 bit vector Knuth (All)	0.0	1.831
512 bit vector Knuth (All)	0.0	1.076

TABLE II: Results of different summation methods on a 512-bit vector unit (Skylake-Gold 6152)

Error in summation

- The errors of the enhanced precision sums appear to be zero but are not perfect
- But there is a reduction in error by five to six orders of magnitude which will be extremely helpful at improving reproducibility
- Two example source codes are available to see how various vector intrinsics run on your set of architectures and compilers
  - https://github.com/LANL/GlobalSums [23]
  - https://github.com/EssentialsofParallelComputing/Chapter6 [24]

Conclusions

- Vector intrinsics can be used for global sums to offset additional floating point operations of Kahan and Knuth algorithms while retaining the rounding error resilience of these algorithms
- Portability is the trade-off for the performance and reproducibility gains
- Our implementation provides additional parallelism that gives designers another tool to use when balancing reproducibility and performance

Future Work

- Determining the best precision to use throughout the application
- Higher precision with reproducible global sums opens up possibilities for lowering precision in other parts of the application
  - Exascale architecture may have additional lower precision capabilities
- Best ways to vectorize for improved GPU single precision capabilities and improved CPU vector units
- Better code profiling to identify areas within codes that benefit from vectorization

Questions

References

[1] W. Kahan, “Further remarks on reducing truncation errors,” Communications of the ACM, vol. Vol. 8, no. 1, p. 40, 1965.

[2] D. E. Knuth, The Art of Computer Programming. Addison-Wesley Press, 1969, vol. 2, chap. 4.

[5] L. Pouchard, S. Baldwin, T. Elsethagen, J. Shantenu, B. Raju, E. Stephan, L. Tang, and K. Kleese Van Dam, “Computational reproducibility of scientific workflows at extreme scales,” The International Journal of High Performance Computing Applications, pp. 1–14, 2019.

[10] D. D. MacCracken and W. S. Dorn, Numerical methods and fortran programming: with applications in engineering and science. J. Wiley, 1964.

[11] D. H. Bailey, “High-precision floating-point arithmetic in scientific computation,” Computing in Science Engineering, vol. 7, no. 3, pp. 54–61, May 2005.

[17] J. Demmel, H. D. Nguyen, and P. Ahrens, “Cost of floating-point reproducibility,” https://www.nist.gov/sites/default/files/documents/itl/ssd/is/NRE-2015-07-Nguyen slides.pdf, Nov 2015.

[18] P. Ahrens, H. D. Nguyen, and J. Demmel, “Efficient reproducible floating point summation and BLAS,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-229, 2015.

[19] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, “ExBLAS: Reproducible and accurate BLAS library,” 2015.

[20] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, “Numerical reproducibility for the parallel reduction on multi- and manycore architectures,” Parallel Computing, vol. 49, pp. 83 – 97, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115001155

[21] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, “ExBLAS: Reproducible and accurate BLAS library,” https://www.nist.gov/sites/default/files/documents/itl/ssd/is/NRE-2015-04-iakymchuk.pdf, Nov 2015.

[22] R. W. Robey, J. M. Robey, and R. Aulwes, “In search of numerical consistency in parallel programming,” Parallel Computing, vol. 37, no. 4-5, pp. 217–229, 2011.

[23] R. Robey, “Global sum examples,” https://github.com/LANL/GlobalSums, 2019.

[24] R. Robey and Y. Zamora, “Vectorization examples,” https://github.com/EssentialsofParallelComputing/Chapter6, 2019.

Kahan Vectorized

Listing 2: The Kahan sum using the Intel x86 vector intrinsics

```
#include <x86intrin.h>
static double sum[4] __attribute__ ((aligned (64)));

double do_kahan_sum_v(double *var, long ncells)
{
   double const zero = 0.0;
   __m256d local_sum = _mm256_broadcast_sd((double
       const*) &zero);
   __m256d local_correction =
       _mm256_broadcast_sd((double const*) &zero);

   #pragma simd
   #pragma vector aligned
   for (long i = 0; i < ncells; i+=4) {
       __m256d var_v = _mm256_load_pd(&var[i]);
       __m256d corrected_next_term = var_v +
           local_correction;
       __m256d new_sum = local_sum +
           local_correction;
       local_correction = corrected_next_term -
           (new_sum - local_sum);
       local_sum = new_sum;
   }
   __m256d sum_v;
   sum_v  = local_correction;
   sum_v += local_sum;
   _mm256_store_pd(sum, sum_v);

   struct esum_type{
      double sum;
      double correction;
   } local;
   local.sum = 0.0;
   local.correction = 0.0;

   for (long i = 0; i < 4; i++) {
      double corrected_next_term_s = sum[i] +
          local.correction;
      double new_sum_s = local.sum +
          local.correction;
      local.correction = corrected_next_term_s -
          (new_sum_s - local.sum);
      local.sum = new_sum_s;
   }
   double final_sum = local.sum + local.correction;
   return(final_sum);
}
```