Work-Efficient Parallel Algorithms for Accurate Floating-Point Prefix Sums

Sean Fraser
Computer Science and Artificial Intelligence Laboratory
MIT
Cambridge, MA
sfraser@mit.edu

Helen Xu
Computer Science and Artificial Intelligence Laboratory
MIT
Cambridge, MA
hjxu@mit.edu

Charles E. Leiserson
Computer Science and Artificial Intelligence Laboratory
MIT
Cambridge, MA
cel@mit.edu

***Abstract*—Existing work-efficient parallel algorithms for floating-point prefix sums exhibit either good performance or good numerical accuracy, but not both. Consequently, prefix-sum algorithms cannot easily be used in scientific-computing applications that require both high performance and accuracy. We have designed and implemented two new algorithms, called `CAST_BLK` and `PAIR_BLK`, whose accuracy is significantly higher than that of the high-performing prefix-sum algorithm from the Problem Based Benchmark Suite, while running with comparable performance on modern multicore machines. Specifically, the root mean squared error of the PBBS code on a large array of uniformly distributed 64-bit floating-point numbers is 8 times higher than that of `CAST_BLK` and 5.8 times higher than that of `PAIR_BLK`. These two codes employ the PBBS three-stage strategy for performance, but they are designed to achieve high accuracy, both theoretically and in practice. A vectorization enhancement to these two scalar codes trades off a small amount of accuracy to match or outperform the PBBS code while still maintaining lower error.**

***Index Terms*—floating-point arithmetic, parallel algorithms, parallelism, prefix sums, span, summation, sum-depth, vectorization, work.**

I. Introduction

The ***prefix sum*** (also known as ***scan*** [2]) is a fundamental algorithmic building block for parallel computing, and consequently, it is often targeted for efficient implementation [2], [11]. In this paper, we shall study floating-point prefix sums, which underlie applications in scientific computing including summed-area table generation [7] and the fast multipole method [4]. For many floating-point calculations, numerical accuracy is as important, or often more important, than absolute performance. In the summed-area table problem, for example, practitioners sacrifice performance for accuracy [26]. Although floating-point prefix sums require both accuracy and high performance [8], traditional summation methods are usually optimized for performance. At the other extreme, compensated-summation algorithms significantly reduce round-off error by accounting for its propagation, but they tend to be unreasonably computationally expensive [1], [9], [14]. This paper presents algorithms for computing prefix sums of floating-point values that offer both accuracy and performance.

The ***prefix-sums operation*** computes the "running sum" of an array of n numbers.

Definition 1 (Prefix-sums operation): The prefix-sums operation takes an array $x = [x_0, x_1, \ldots, x_{n-1}]$ of n elements and returns the "running sum" $y = [y_0, y_1, \ldots, y_{n-1}]$, where

$$y_k = \begin{cases} x_0 & \text{if } k = 0, \\ x_k + y_{k-1} & \text{if } k \geq 1 \ . \end{cases} \quad (1)$$

Although our codes handle arbitrary n, to simplify our analysis, we shall generally assume that n is an exact power of 2.

Three fundamental prefix-sum algorithms, illustrated in Figure 1, have appeared in the literature. The naive **`FWD_SCAN`** algorithm directly implements the recursion in (1) and is illustrated in Figure 1(a). Although `FWD_SCAN` is serial and has low accuracy, it runs fast in practice, because it performs only $n-1$ floating-point additions, the minimum possible, and it takes advantage of architectural features, such as prefetching [25]. In contrast, the canonical ***pairwise*** prefix sum, shown in Figure 1(b), which we will call **`PAIR_SCAN`**, is parallelizable and achieves better accuracy, but it requires $2n - \lg n - 2$ additions, a constant-factor more overhead [2]. Moreover, its structure matches modern architectural features less well. Finally, the Kogge-Stone algorithm [15], shown in Figure 1(c), which we will call **`KS_SCAN`** (also described by Hillis and Steele [10]), achieves even higher accuracy than pairwise ordering requiring $n \lg n - n + 1 = \Theta(n \lg n)$ additions.

This research was sponsored in part by NSF Grant 1533644 and by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

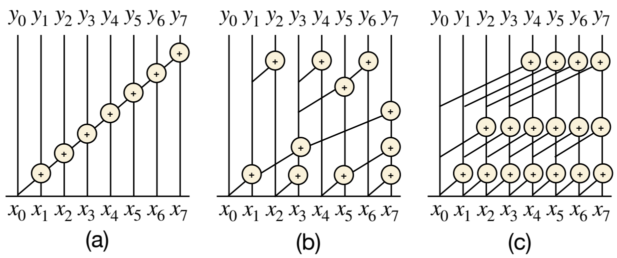

Figure 1: The canonical prefix-sum algorithms: **(a)** FWD_SCAN, **(b)** PAIR_SCAN, and **(c)** KS_SCAN. Each circle with a plus sign represents an addition operation taking as inputs two values below and outputting their sum above.

Prefix sums are so ubiquitous that they have been included as primitives in some languages such as C++ [24], and more recently have been considered as a primitive for GPU computations in CUDA [6]. The fastest prefix sum on a CPU for large inputs is implemented in the Problem-Based Benchmark Suite (PBBS) Library [23]. The scan in PBBS, which we will call **FWD_BLK** due to its structure, achieves good performance but was not optimized for accuracy. The performance of the compensated-summation algorithm, which we call **COMP_SCAN**, is sufficiently slow that it is rarely used in practice, even though it has great accuracy (although COMP_SCAN is a useful benchmark for accuracy). In this paper, we introduce prefix-sum algorithms with comparable performance to FWD_BLK but with significantly better accuracy, although generally not attaining the levels of COMP_SCAN.

Analysis strategy

We shall analyze our prefix-sum algorithms using the ***work-span model*** [3, Chapter 27] for performance and the "sum-depth" which provides a useful proxy for accuracy. The ***work*** is the total time to execute the entire algorithm on a given input on one processor. We say that a parallel algorithm is (asymptotically) ***work-efficient*** if its work is within a constant factor of the work of the best serial algorithm for the problem. The ***span***[1] is the longest serial chain of dependencies in the computation (or the runtime on an ideal computer with no scheduling overhead and an infinite number of processors). The ***parallelism*** of an algorithm on a given input is the work divided by the span. Given a summation algorithm (e.g. reduction, prefix sum), the ***sum-depth*** is the longest chain of additions along any path from the inputs to the output(s). The worst-case backward error bound of a sum calculation is proportional to its sum-depth [1], [5], [8].

We can compare the three algorithms in terms of work, span, parallelism, and sum-depth in a task-parallel model, such as that which Cilk [12] provides. We generally analyze work, span, and parallelism asymptotically, because constant factors in these measures are often dominated by machine overheads. We express the sum-depth exactly, however, because accuracy is not influenced by machine performance. FWD_SCAN requires $\Theta(n)$ work, $\Theta(n)$ span, $\Theta(1)$ parallelism, and $n-1$ sum-depth. PAIR_SCAN can be implemented by a divide-and-conquer strategy involving $\Theta(n)$ work, $\Theta(\lg n)$ span, $\Theta(n/\lg n)$ parallelism, and $2\lg n - 2$ sum-depth (assuming, as we have mentioned, that n is an exact power of 2). Thus, it is work-efficient, as it is within a factor of 2 of the best-possible implementation. KS_SCAN requires $\Theta(n \lg n)$ work, $\Theta(lg^2 n)$ span, $\Theta(n/\lg n)$ parallelism, and $\lg n$ sum-depth. The reason that the span of KS_SCAN is $\Theta(\lg^2 n)$ rather than $\Theta(\lg n)$ is that its implementation involves $\Theta(\lg n)$ nested parallel loops over n iterations, and in the Cilk model, each parallel loop has span $\Theta(\lg n)$, resulting in a total span of $\Theta(\lg^2 n)$. The costs of these parallel prefix-sum algorithms are summarized in Table I.

When it comes to engineering a good parallel algorithm for prefix sum, constants matter. The parallel PAIR_SCAN algorithm, which has much better sum-depth (and hence accuracy) than FWD_SCAN, performs only double the number of floating-point additions and it can perform many of those operations in parallel. But a naive implementation of PAIR_SCAN is slower than FWD_SCAN in practice, because there are many other considerations, such as coping with limited memory bandwidth and processor-pipeline overheads. The PBBS implementation of FWD_BLK manages to overcome the performance limitations of the serial FWD_SCAN algorithm, and its sum-depth is a bit better, but it was not designed to minimize numerical round-off, making it unsuitable for use in numerical codes that require high accuracy.

Contributions

Our main contributions are two new algorithm implementations for floating-point prefix sum, called CAST_BLK and PAIR_BLK. These two algorithms achieve performance by adopting PBBS's three-stage blocked strategy, but within the stages, they are designed to be much more accurate, both in theory and in practice. Both CAST_BLK and PAIR_BLK are theoretically work-efficient and have small sum-depth. In practice, they both run fast on a modern multicore computer and exhibit high accuracy, achieving a good balance between the two concerns.

Figure 2 summarizes the accuracy and performance of the two algorithms. As shown in the figure, CAST_BLK and PAIR_BLK dominate FWD_BLK on medium-sized inputs. On large inputs (Figure 2(c)), FWD_BLK exhibits the best performance, but CAST_BLK and PAIR_BLK perform competitively and are much more accurate.

To be specific, our contributions are as follows:

- The design and Cilk [12] implementation of two low-sum-depth, high-performance algorithms for prefix sums, called CAST_BLK and PAIR_BLK.

[1]Sometimes called **critical-path length** or **computational depth**.

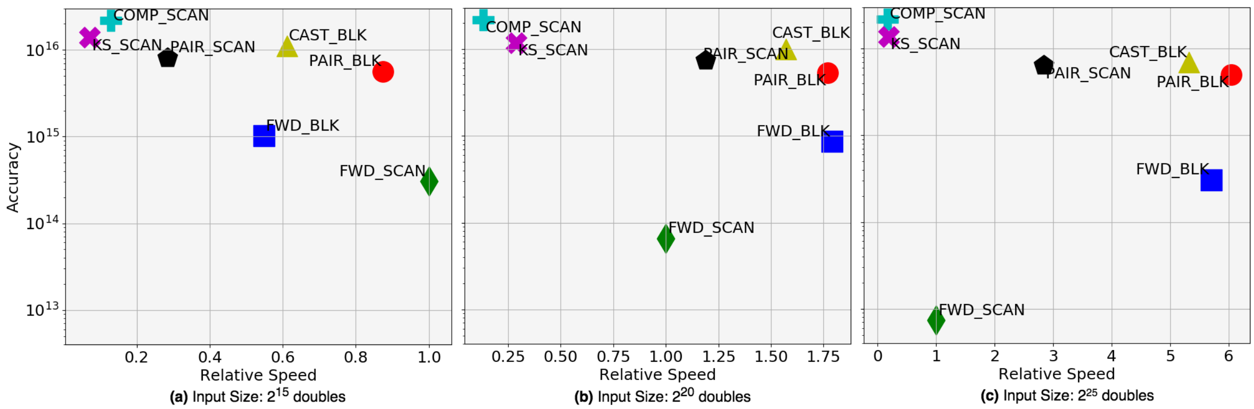

Figure 2: A comparison of the numerical accuracy and performance of PAIR_BLK and CAST_BLK with five other prefix-sum algorithm implementations. All algorithms were run on three different input sizes of 64-bit floating-point values (doubles) uniformly distributed on the interval $[0, 1]$ using the multicore computer described in Section IV. The horizontal axis in each graph shows the ratio of the running time of each algorithm to the naive FWD_SCAN algorithm (right is better). The vertical axis shows the reciprocal root mean square relative error of the output (up is better).

- An experimental study of CAST_BLK and PAIR_BLK and five other prefix-sum algorithms that demonstrates that high performance and numerical accuracy can be achieved simultaneously.
- A vectorization enhancement to CAST_BLK, called CAST_BLK_SIMD, and a corresponding vectorization enhancement to PAIR_BLK, called PAIR_BLK_SIMD, which trades off a small amount of accuracy for improvements in performance, especially for small input sizes.

Outline

The rest of the paper is organized as follows. Section II provides a taxonomy of building blocks for prefix sum algorithms that we will use to exactly specify the more complicated optimized prefix sums in this paper. Section III describes and analyzes CAST_BLK and PAIR_BLK. Section IV presents an experimental evaluation of prefix-sum algorithms. Section V describes how to further optimize the two prefix-sum algorithms with vectorization. Finally, we provide concluding remarks in Section VI.

II. Characterizing Prefix-Sum Algorithms

In this section we define building blocks for prefix sums in order to organize and specify more complicated algorithms in later sections. The building blocks are composed of the summation ***kernel*** (either a scan or reduce) and the ***ordering*** that it follows. As described in Section I, scan computations can have forward, pairwise, or Kogge-Stone orderings. Reductions can also have a forward or pairwise ordering.

We will use S and R to denote scans and reductions, respectively, and prepend them with f, p, or k for forward, pairwise, or Kogge-Stone, respectively, to specify an ordering. For example, the naive forward scan FWD_SCAN is exactly the building block fS.

Prefix sums in stages

More complex blocked algorithms such as FWD_BLK may compose these primitives sequentially in ***stages*** by dividing the input into blocks and running kernels on each block in parallel. A blocked scan may run a different summation algorithm in each stage, or even a broadcast (denoted by C). Blocking coarsens parallel implementations by processing the blocks in parallel but doing the work of each block in serial. Furthermore, blocking decreases the sum-depth by decreasing the length of the longest chain of additions.

We use the building blocks to specify stages of algorithms by listing the primitive in each stage. For example, FWD_BLK divides the input into blocks and executes in three stages. In the first stage, it runs a forward reduce on each block. In the second stage, it runs FWD_SCAN on the results of the first stage. In the third stage, it runs FWD_SCAN to propagate the results of the the second stage to each block. Therefore, FWD_BLK is exactly specified with the building blocks fRfSfS.

III. Low Sum-Depth Prefix Sums

In this section we will describe CAST_BLK and PAIR_BLK, two new blocked prefix-sum algorithms optimized for low sum-depth as well as for performance. We illustrate the difference between CAST_BLK and PAIR_BLK in Figure 3, specify them according to the building blocks in Section II and summarize the theoretical bounds on all discussed algorithms in Table I.

Reducing sum-depth via broadcast

The first algorithm, which we will call CAST_BLK, reduces the sum-depth by replacing one of the summation stages in FWD_BLK with a broadcast. Specifically, it replaces the reduction in stage 1 and the forward scan

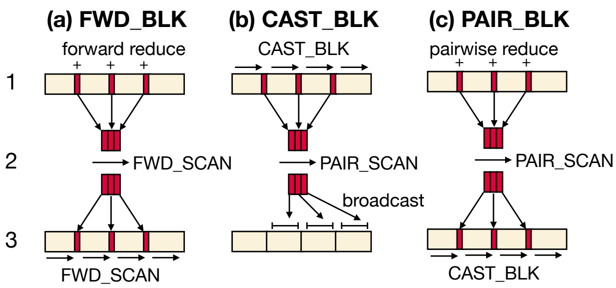

Figure 3: Blocked prefix-sum algorithms in stages.

in stage 2 with the `PAIR_SCAN` subroutine. In order to compute the prefix sum, `CAST_BLK` only needs to broadcast the end of each block to every entry in the next block. In our implementation of `CAST_BLK`, we replace stage 1 recursively with a second level of blocking and run `CAST_BLK` again, which reduces the sum-depth and does not affect the asymptotic work and span.

Analysis: `CAST_BLK` is work-efficient and achieves lower span and sum-depth than `FWD_BLK`. Given block sizes B, B' for the first and second level of blocking (respectively), `CAST_BLK` has $\Theta(\lg n)$ span and $2\lg n - 4$ sum-depth. We omit the proofs of the theoretical bounds for space, but they are all generated by aggregating the bounds on the building blocks from Table I.

Pairwise summation

The next algorithm, which we will call `PAIR_BLK`, replaces the forward summation subroutines in `FWD_BLK` with low sum-depth prefix sums. `PAIR_BLK` also divides the input into blocks of size B and proceeds in stages. Specifically, it runs a pairwise reduction in the first stage and `PAIR_SCAN` in the second stage. The last stage runs `CAST_BLK` on blocks of size $B' < B$.

The `PAIR_BLK` algorithm can be parallelized blockwise in the same way as `FWD_BLK`.

Analysis: `PAIR_BLK` is work-efficient and achieves lower sum-depth than `FWD_BLK`. Given a first-level block size B, `PAIR_BLK` has $\Theta(\lg n)$ span and $2\lg n + \lg B - 5$ sum-depth.

IV. Evaluation

In this section we present an experimental evaluation of prefix sum algorithms on a CPU in terms of both performance and accuracy. As we will see, `CAST_BLK` and `PAIR_BLK` achieve competitive performance with `FWD_BLK` but are up to an order of magnitude more accurate.

Experimental setup

We used a general-purpose multicore from MIT Supercloud [20] with 20 physical cores (with 2-way hyperthreading) and 2 Intel Xeon Gold 6248 @ 2.50GHz processors.

We implemented all algorithms in C++ using Cilk [12] for fork-join parallelism. We used the Tapir/LLVM [22] branch of the LLVM [16], [17] compiler (version 8) with the `-O3` and `-march=native` flags.

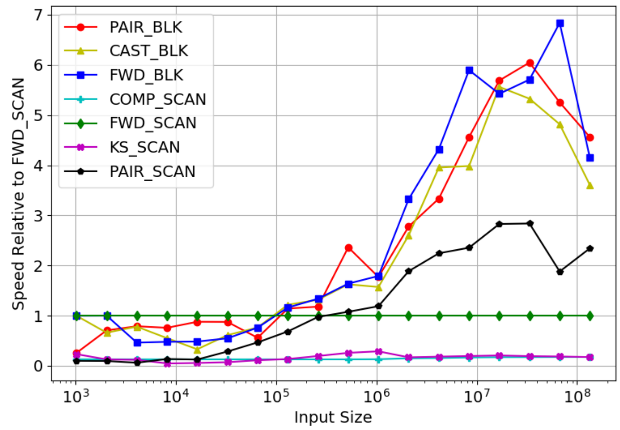

Figure 4: A comparison of the performance of `PAIR_BLK` and `CAST_BLK` with five other prefix-sum algorithms implementations on uniformly distributed doubles on the interval $[0, 1]$. On this plot, up is better.

Our data set consists of `IEEE754` double-precision[2] 64-bit floats randomly generated with the Mersenne Twister 19937 generator [19].

For the blocked algorithms, we set $B = 1024$ to match `FWD_BLK` for the fairest comparison and set $B' = 16$, although different block sizes may result in lower sum-depths or better performance in practice.

Performance

Figure 4 shows the speedup[3] obtained for the different algorithms over serial `FWD_SCAN`. For small inputs, `PAIR_BLK`, `CAST_BLK`, and `FWD_BLK` exhibit similar performance. Since `FWD_BLK` is optimized for larger inputs[4] where memory bandwidth is the bottleneck, it performs up to $1.4\times$ better than `PAIR_BLK` and `CAST_BLK`.

As shown in Figure 4, the speedup for all parallel prefix sum algorithms is relatively small compared to the number of physical cores. This limited scalability is due to the memory bandwidth because the actual computation involved in a scan (one addition per element) is small compared to the cost of data movement. Therefore, prefix sum algorithms are often memory-bound on CPUs and can experience performance variability due to data transfer on large inputs.

Accuracy

We measured the numerical error of the prefix sum algorithms on doubles under distributions from Higham's methodology [8]. Specifically, we drew numbers according to `Unif`$(0, 1)$ (the uniform distribution between 0 and

[2]The results are the same for single-precision floats given no overflow.
[3]We measured runtime as the median of 7 trials.
[4]In these experiments, about 4 million doubles fit in cache.

Table I: Prefix-sum algorithms, their descriptions according to the taxonomy in Section II, and their theoretical work, span, and sum-depth on inputs of size $n \geq 4$. For blocked algorithms, we denote the block size at the first level of blocking with B, where $B, n/B \geq 4$.

Algorithm	*Description*	*Source*	*Work*	*Span*	*Parallelism*	*Sum-Depth*
FWD_SCAN	fS	[8]	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$n-1$
PAIR_SCAN	pS	[2]	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n/\lg n)$	$2\lg n - 2$
KS_SCAN	kS	[15]	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n/\lg n)$	$\lg n$
FWD_BLK	fRfSfS	[23]	$\Theta(n)$	$\Theta(B + n/B)$	$\Theta(B + n/B)$	$2B + n/B + 1$
CAST_BLK	(pSpSC)pSC	[this work]	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n/\lg n)$	$2\lg n - 4$
PAIR_BLK	pRpS(pSpSC)	[this work]	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n/\lg n)$	$2\lg n + \lg B - 5$

1), Exp(1) (the exponential distribution with $\lambda = 1$), and Norm$(0, 1)$ (the standard normal distribution).

Since worst-case floating-point rounding error bounds tend to be pessimistic, we follow the methodology described by Higham [8]. We experimentally evaluate the accuracy of summations as follows:

- We use higher-precision floating point values[5] [21] as a reference point to compare relative error.
- We draw random inputs from uniform, exponential and normal distributions.
- We use the compensated summation algorithm[6] COMP_SCAN [14] as an accuracy benchmark.
- We quantify error as the root mean square relative error.

In floating-point arithmetic, the summation ordering determines the computed sum. For all $k = 0, 1, \ldots, n-1$, let S_k be the ***real value*** of the scan at index k ($S_k = \sum_{i=0}^{k} x_i$), and let $\hat{S}_k$ be the computed sum. The ***relative error*** of $\hat{S}_k$ is defined as $E_k = \hat{S}_k - S_k$. Given n summation results $\hat{S}_0, \ldots, \hat{S}_{n-1}$ and real values $S_0, \ldots, S_{n-1}$, the ***root mean square relative error*** is as follows:

$$\mathrm{RMSE} = \left(\frac{1}{n} \sum_{k=0}^{n-1} E_k^2 \right)^{1/2}.$$

We measure error on the different distributions as the RMSE. The machine epsilon ($\epsilon = 2.22 \times 10^{-16}$ for doubles) is an upper bound on the relative error of any single summation due to rounding [9].

Discussion

As shown in Figure 5, both the CAST_BLK and PAIR_BLK algorithm exhibit up to $10\times$ more error than compensated summation. Although the compensated summation algorithm has the highest accuracy, it is at about $20\times$ slower than CAST_BLK and PAIR_BLK.

Overall, CAST_BLK and PAIR_BLK are much more accurate than forward summation-based algorithms such as FWD_BLK and FWD_SCAN. The CAST_BLK algorithm achieves up to $8\times$ less error than FWD_BLK and up to $103\times$ less error than FWD_SCAN on large inputs. Similarly, PAIR_BLK achieves up to $5.8\times$ less error than FWD_BLK and up to $76\times$ less error than FWD_SCAN. Therefore, CAST_BLK and PAIR_BLK attain much better accuracy with comparable performance to FWD_BLK.

V. Vectorizing Prefix Sums

This section describes a vectorized forward scan algorithm called SCAN_SIMD. We evaluate SCAN_SIMD as a subroutine in blocked scan algorithms and show that it strictly improves FWD_BLK. In pairwise blocked algorithms, vectorization trades off accuracy for improved performance.

The vectorized prefix-sum subroutine SCAN_SIMD divides the input array into chunks of size vector width V (e.g. 256 bits in Intel AVX2 [18]), performs a vectorized version of KS_SCAN on each chunk, and processes the chunks serially from left to right. Although KS_SCAN is not work-efficient, it is well-suited to SIMD operations because it has high data-level parallelism. Figure 6 contains an example of SCAN_SIMD on one vector. In general, given a vector width V, SCAN_SIMD requires $2 \log V + 4$ vector operations to compute a scan on one block, while the scalar FWD_SCAN requires $3V$ scalar operations [5].

Evaluation

We implemented SCAN_SIMD with Intel Intrinsics [13] and use it as a subroutine in FWD_BLK, CAST_BLK, and PAIR_BLK. We call the resulting algorithms FWD_BLK_SIMD, CAST_BLK_SIMD, and PAIR_BLK_SIMD, respectively. All experiments were run on the same setup from Section IV.

As shown in Figure 7, SCAN_SIMD and FWD_BLK_SIMD strictly dominate their scalar counterparts FWD_SCAN and FWD_BLK in both performance and accuracy because SCAN_SIMD improves the throughput and lowers the sum-depth over FWD_SCAN. Specifically, SCAN_SIMD is up to $2.2\times$ faster and up to $2.5\times$ more accurate than FWD_SCAN. Furthermore, FWD_BLK_SIMD is up to $2\times$ faster when

[5] We used 100-digit precision floating-point values via Boost.

[6] Compensated summation is sometimes called Kahan summation.

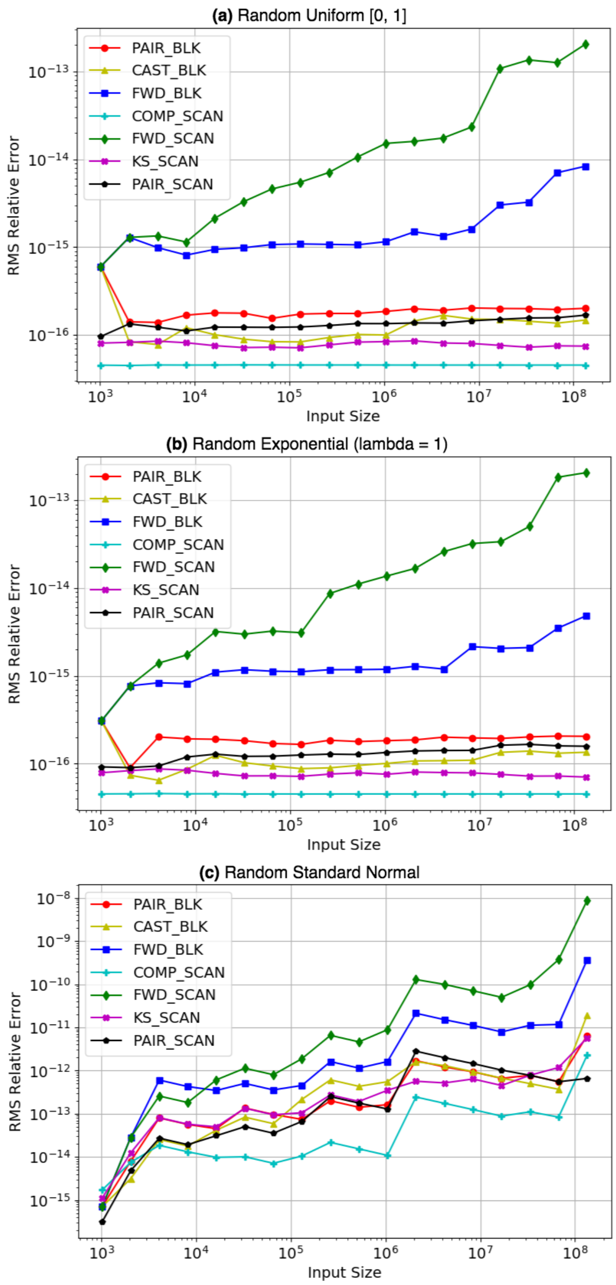

Figure 5: A comparison of the numerical error of `PAIR_BLK` and `CAST_BLK` with five other prefix-sum algorithms. On this plot, down is better.

inputs fit in cache and comparable on larger inputs while achieving 2× less error than `FWD_BLK`.

Algorithm description

Vectorizing scans in `CAST_BLK` and `PAIR_BLK` trades off accuracy for performance. `CAST_BLK_SIMD` and `PAIR_BLK_SIMD` are up to 2× faster than `FWD_BLK` when inputs fit in the cache, and they are competitive with `FWD_BLK` when the inputs are large. Finally,

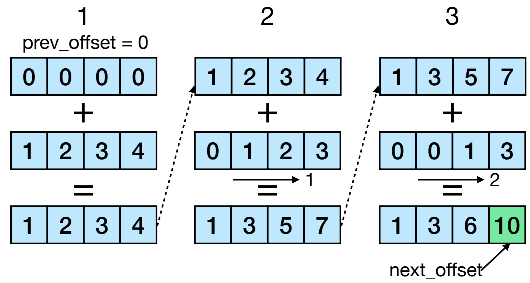

Figure 6: An example of `SCAN_SIMD` on one vector ($V = 4$). A solid arrow means a vector shift by the number next to it, additions are vector adds, and a dotted arrow denotes use of an output at a previous step as input to the next step.

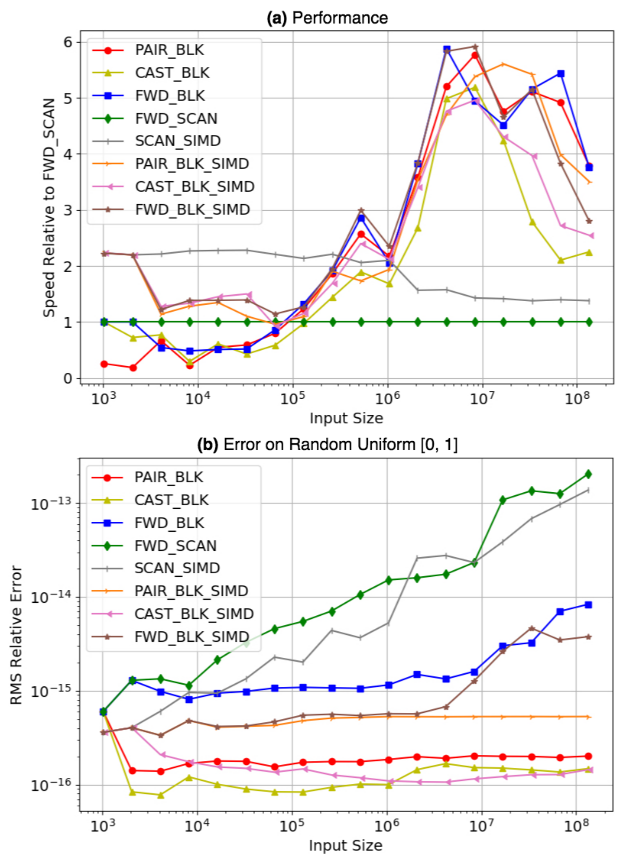

Figure 7: A comparison of the performance and error between `FWD_SCAN`, `CAST_BLK`, `PAIR_BLK`, and `FWD_BLK` and their vectorized counterparts.

`CAST_BLK_SIMD` and `PAIR_BLK_SIMD` are both about 2× more accurate than `FWD_BLK`.

VI. Conclusion

In scientific computing, floating-point prefix sums require both high accuracy and performance. We have introduced two new algorithms, `CAST_BLK` and `PAIR_BLK`, which achieve competitive performance and much better accuracy than the state-of-the-art CPU parallel scan. Furthermore, we showed that augmenting parallel-prefix sums with vectorization improves performance. Since many applications are implemented on CPUs, a faster and more accurate prefix-sum library for general-purpose multicores has the potential to speed up a wide variety of programs while providing numerical precision. We conclude with an avenue for future research and a brief discussion of the role of GPUs in computing scans.

A standard practice for enhancing the precision of dot products and other computations that involve summing a large number of floating-point values is to maintain the internal sums with extended precision. The two fast-and-accurate algorithms we have studied, `CAST_BLK` and `PAIR_BLK`, would seem to fare differently if intermediate values can be kept with extended precision. The `CAST_BLK` algorithm would require the extended precision values resulting from the first stage to be maintained until they can be used in the third stage, whereas the `PAIR_BLK` algorithm would require only the intermediate stage to manage extended precision. Consequently, for situations where extended precision is available, we believe that `PAIR_BLK` would likely show a performance advantage over `CAST_BLK`, but we leave this study to future research.

What role might GPUs play in fast-and-accurate scans? After all, GPUs provide considerably more floating-point capability than does a typical CPU. Unfortunately, for general-purpose computations, transferring data from a multicore to an attached GPU accelerator is so slow that a computation such as a floating-point scan cannot avail itself of the faster computational capability. GPUs can effectively perform scans within a GPU computation (for example, NVIDIA provides such a library [6]). But since they are unsuitable for performing scans as a subroutine within a general-purpose program, multicores need their own fast-and-accurate parallel algorithms, such as `CAST_BLK` and `PAIR_BLK`.

Acknowledgments

We would like to thank Guy Blelloch of Carnegie Mellon, Julian Shun of MIT, and Yan Gu of UC Riverside for providing technical benchmarks used in this work. We thank Alexandros-Stavros Iliopoulos of MIT for helpful discussions and feedback. The MIT Supercloud provided an invaluable environment for our experiments.

References

[1] Pierre Blanchard, Nicholas J. Higham, and Theo Mary. A class of fast and accurate summation algorithms. *SIAM Journal on Scientific Computing*, 42(3):A1541–A1557, 2020.

[2] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[4] E. D. Demaine, M. L. Demaine, A. Edelman, C. E. Leiserson, and P. Persson. Building blocks and excluded sums. *SIAM News*, 38(4):1–5, 2005.

[5] Sean Fraser. Computing included and excluded sums using parallel prefix. Master's thesis, Massachusetts Institute of Technology, 2020.

[6] Mark Harris, Shubhabrata Sengupta, and John Owens. Parallel prefix sum (scan) with CUDA. *GPU Gems*, 39(39):851–876, 08 2007.

[7] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24, pages 547–555. Wiley Online Library, 2005.

[8] Nicholas J. Higham. The accuracy of floating point summation. *SIAM J. Scientific Computing*, 14:783–799, 1993.

[9] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, 2nd edition, 2002.

[10] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *CACM*, 29(12):1170–1183, December 1986.

[11] D. Horn. Stream reduction operations for GPGPU applications. *GPU Gems*, 2, 01 2005.

[12] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available at `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[13] Intel Corporation. Intel Intrinsics Guide. Available at `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, 2020.

[14] William Kahan. Further remarks on reducing truncation errors. *CACM*, 8(1):40, 1965.

[15] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, 100(8):786–793, 1973.

[16] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

[17] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, page 75, Palo Alto, California, March 2004.

[18] Quoc-Thai V Le. How Intel Advanced Vector Extensions 2 improves performance on server applications. Available at `https://software.intel.com/content/www/us/en/develop/articles/how-intel-avx2-improves-performance-on-server-applications.html?language=en`, 2014.

[19] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[20] MIT Supercloud. Available at `https://supercloud.mit.edu/`, 2020.

[21] Boost Organization. Boost C++ libraries: Multiprecision. Available at `https://www.boost.org/doc/libs/1_66_0/libs/multiprecision/doc/html/boost_multiprecision/tut/floats/cpp_bin_float.html`, 2020.

[22] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. *SIGPLAN Not.*, 52(8):249–265, January 2017.

[23] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The Problem Based Benchmark Suite. In *24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 68–70, New York, NY, USA, 2012. ACM.

[24] `cppreference.com`. `std::inclusive_scan`. Available at `https://en.cppreference.com/w/cpp/algorithm/inclusive_scan`, 2020.

[25] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[26] Gernot Ziegler. Summed area ripmaps. GPU Technology Conference (talk). Available at `https://on-demand.gputechconf.com/gtc/2012/presentations/S0096-Summed-Area-Ripmaps.pdf`, 2012.