SIAM J. SCI. COMPUT.
Vol. 31, No. 2, pp. 1156–1174


# REDUCING FLOATING POINT ERROR IN DOT PRODUCT USING THE SUPERBLOCK FAMILY OF ALGORITHMS*

ANTHONY M. CASTALDO†, R. CLINT WHALEY†, AND ANTHONY T. CHRONOPOULOS†

**Abstract.** This paper discusses both the theoretical and statistical errors obtained by various well-known dot products, from the `canonical` to `pairwise` algorithms, and introduces a new and more general framework that we have named `superblock` which subsumes them and permits a practitioner to make trade-offs between computational performance, memory usage, and error behavior. We show that algorithms with lower error bounds tend to behave noticeably better in practice. Unlike many such error-reducing algorithms, `superblock` requires no additional floating point operations and should be implementable with little to no performance loss, making it suitable for use as a performance-critical building block of a linear algebra kernel.

**Key words.** dot product, inner product, error analysis, BLAS, ATLAS

**AMS subject classifications.** 65G50, 65K05, 65K10, 65Y20, 68-04

**DOI.** 10.1137/070679946

## 1. Introduction.

A host of linear algebra methods derive their error behavior directly from dot product. In particular, most high performance dense systems derive their performance *and* error behavior overwhelmingly from matrix multiply, and matrix multiply's error behavior is almost wholly attributable to the underlying dot product that it is built from (sparse problems usually have a similar relationship with matrix-vector multiply, which can also be built from dot product). With the expansion of standard workstations to 64-bit memories and multicore processors, much larger calculations are possible on even simple desktop machines than ever before. Parallel machines built from these hugely expanded nodes can solve problems of almost unlimited size. The canonical dot product has a worst-case error bound that rises linearly with vector length. In the past this has not been deemed intolerable, but with problem sizes increasing it becomes important to examine the assumption that a linear rise in worst-case error is tolerable and to examine whether we can moderate it without a noticeable loss in performance.

Dot product is an important operation in its own right, but due to performance considerations linear algebra implementations only rarely call it directly. Instead, most large-scale linear algebra operations call matrix multiply (aka GEMM, for general matrix multiply) [1, 3], which can be made to run very near the theoretical peak of the architecture. High performance matrix multiply can in turn be implemented as a series of parallel dot products, and this is the case in our own ATLAS [31, 30] project, which uses GEMM as the building block of its high performance BLAS [15, 22, 10, 11, 9] implementation. Therefore, we are keenly interested in both the error bound of a given dot product algorithm and whether that algorithm is likely to allow for a high performance GEMM implementation. The implementation and performance of GEMM are not the focus of this paper, but we review them for

*Received by the editors January 11, 2007; accepted for publication (in revised form) July 25, 2008; published electronically December 17, 2008. This work was supported in part by National Science Foundation CRI grant CNS-0551504.
http://www.siam.org/journals/sisc/31-2/67994.html
†Department of Computer Science, University of Texas at San Antonio, 6900 N. Loop, 1604 West, San Antonio, TX 78249 (castaldo@cs.utsa.edu, whaley@cs.utsa.edu, atc@cs.utsa.edu).

each algorithm briefly, to explain why certain formulations seem more promising than others.

**1.1. Background and related work.** Because dot product is so important to the error analysis of linear algebra, it has been well studied; probably the main reference for linear algebra error analysis in general is Higham's excellent book [17], which extended the foundation provided by Stewart in [29]. We will therefore adopt and extend the notation from [17] for representing floating point rounding errors:

$$fl(x \circ y) = (x \circ y) \cdot (1 + \delta), \qquad \text{with } |\delta| \leq u, \tag{1.1}$$

where (i) $\circ$ is $x \oplus y$, $x \ominus y$, $x \odot y$, $x \oslash y$ for floating point (as opposed to exact) add, subtract, multiply, or divide operations; (ii) $u$ is the unit roundoff error, defined as $u = \frac{1}{2}\beta^{1-t}$; (iii) $\beta$ is the base of the numbers being used; and (iv) $t$ is the number of digits stored. Also, we assume that $|\delta| \leq u$ and that $\delta_{anything}$ is reserved notation for values such that $|\delta_{anything}| \leq u$. By the IEEE floating point standard, for single precision $u = 2^{-24} \approx 5.96\text{x}10^{-8}$, and for double precision $u = 2^{-53} \approx 1.11\text{x}10^{-16}$. This model presumes that a guard digit is used during subtraction, which is a required feature of IEEE floating point arithmetic.

The floating point computations in dot product are multiplication and addition. We will see that the multiplicative error does not compound in dot product except through accumulation, and hence the main algorithmic opportunity for error reduction comes in strategies for summing the individual elementwise products. Therefore, the most closely related work is on reducing error in summations, as in [25, 13, 24, 28, 19, 12, 2, 6, 26, 14]. In this paper we use Stewart's $\langle k \rangle$ "error counter" notation [29]:

$$\langle n \rangle := \prod_{i=1}^{n} (1 + \delta_i)^{\rho_i}, \quad \text{with } \rho_i = \pm 1, \quad |\delta_i| \leq u.$$

Multiple $\langle k \rangle$ may represent distinct sets of $\delta_i$ and $\rho_i$ and are not necessarily equal to each other (thus cannot be multiplicatively factored out). These terms can be combined and manipulated as follows, where $x_1$ and $x_2$ are floating point numbers:

(1) $x_1 \langle p \rangle \odot x_2 \langle m \rangle = (x_1 \cdot x_2) \langle p + m + 1 \rangle$, and

(2) $x_1 \langle p \rangle \oplus x_2 \langle m \rangle = (x_1 + x_2) \langle \max(p, m) + 1 \rangle$

(i.e., given any $\langle p \rangle$ and $\langle m \rangle$ there exists a set of $\delta_i$ such that $|\delta_i| \leq u$ and $\rho_i = \pm 1$ forming a $\langle \max(p, m) + 1 \rangle$ that satisfies (2)). (2) is the key to algorithmic reduction in error: if we can evenly balance the size of $p$ and $m$ during addition, we can minimize the resulting error bound. Conversely, if $m = 1$ and $p = i$, as in canonical summation (where $i$ is the induction variable), then the error bound is maximized. We further show empirically that the algorithm with the least error bound usually produces the least error.

Higham (p. 69 of [17]) provides bounds for $\langle n \rangle$ (ignoring the possibility of overflow or underflow).

LEMMA 1.1.

$$|\delta_i| \leq u, \rho_i = \pm 1, nu < 1, \quad \prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \theta_n; \quad |\theta_n| \leq \frac{nu}{1 - nu} =: \gamma_n. \tag{1.2}$$

*Note that equality holds only for* $|\theta_1|$.

This paper is concerned only with algorithmic improvements for controlling error suitable to high performance implementation. A related and orthogonal approach is using extra and/or mixed precision arithmetic to reduce error, as in [6, 8, 23], but such approaches often present additional computational demands, and/or architecture-specific coding techniques, and so they are not the focus of this paper. The broader

effects of floating point error in linear algebra are also too large a pool of literature to survey in detail, but more information can be found in the overview texts [29, 17] and in [5, 21].

**1.2. Outline.** The remainder of this paper is organized in the following way: Section 2 surveys some known dot product implementations, including their error bounds, while section 3 introduces a general class of dot product algorithms we call `superblock`, which we believe is new. Section 4 then shows some results from our statistical studies of these algorithms, which will allow us to draw some conclusions about these techniques, the most important of which are summarized in section 5.

**2. Known dot products.** In this section we give an overview of several dot products of interest. Note that we are primarily interested in dot products that could likely be extended into high performance GEMM implementations. Since GEMM has $\mathcal{O}(N^3)$ floating point operations[1] (flops) and $\mathcal{O}(N^2)$ memory use, after tuning its performance is typically limited by the amount of computation to be done, and therefore we do not consider methods requiring any additional flops (e.g., compensated summation or the algorithms described in [27]). For performance reasons, we also avoid sorting the vectors of each individual dot product as discussed in [19, 7, 26]. Finally, we do not consider using extra or mixed precision, as both the performance and accuracy of such algorithms is strongly influenced by the architecture and compiler, and our focus here is on general algorithmic strategies.

Therefore, we present and analyze three known methods in this section, including comments indicating their suitability as a building block for high performance GEMM implementation. Section 2.1 discusses `canonical` dot product, section 2.2 surveys two versions of the blocked dot product, and section 2.3 presents `pairwise` dot product.

**2.1. Canonical dot product.** The `canonical` algorithm is

```
for (dot=0.0,i=0; i < N; i++) dot += X[i] * Y[i];
```

which calculates the dot product for two $n$-dimensional vectors, $\mathbf{x} = \{x_i\}_{i=1}^n$ and $\mathbf{y} = \{y_i\}_{i=1}^n$, in the order

$$\begin{aligned}&((\ldots(((x_1 \odot y_1) \oplus x_2 \odot y_2) \oplus x_3 \odot y_3) \oplus \ldots) \oplus x_n \odot y_n)\\ &\Longleftrightarrow (x_1 \cdot y_1)\langle n\rangle + (x_2 \cdot y_2)\langle n\rangle + (x_3 \cdot y_3)\langle n-1\rangle + \cdots + (x_n \cdot y_n)\langle 2\rangle .\end{aligned}$$

In [17] and much of the summation literature, this method is called the recursive algorithm. Since the `pairwise` algorithm (surveyed in section 2.3) is naturally implemented using recursion and this method is naturally implemented using a simple iterative loop, we avoid this name and refer to this algorithm, which is certainly the most widely used in practice, as `canonical`. The forward error bound is given on p. 69 of [17] as follows: For $s = \mathbf{x}^\mathsf{T}\mathbf{y}$ and $\hat{s} = \mathbf{x}^\mathsf{T} \odot \mathbf{y}$,

$$|s - \hat{s}| \leq \gamma_n |\mathbf{x}|^\mathsf{T} |\mathbf{y}|. \tag{2.1}$$

[1] Strassen's partitioning can reduce the order of the floating point operations to $\approx\ \mathcal{O}(N^{2.807})$, and various refinements can reduce the exponent even further, but at the cost of *increasing* the error bound significantly. The forward error bound for $N \times N$ matrix multiply using a canonical dot product is $\mathcal{O}(N)$. In Higham's numerical stability analysis (p. 359 of [16]), he arrives at a bound (depending upon the level where recursion stops) of $\mathcal{O}(3N^2 + 25N)$ (for one level of recursion) to $\approx\ \mathcal{O}(6N^{3.585} - 5N)$ (for full recursion). Thus Strassen's algorithm and refinements thereof are not considered in this paper, since our aim is reducing error. The `superblock` technique described herein may well reduce these error bounds to make Strassen's algorithm more acceptable, but that is a line of inquiry we do not address here.

For insight into this result, notice that the subscript on the $\gamma$ term is the greatest number of flops to which any given input is exposed; this is the maximum error counter $\langle\cdot\rangle$. This leads us to the observation that different algorithms distribute flops differently over their inputs, and thus the more uniformly an algorithm distributes flops over inputs the better it will be on worst-case error. Dot product forward error bounds are all of this general form; for expositional simplicity we shall abuse notation by referring to the error bound by the $\gamma$ element that changes from algorithm to algorithm. Thus we shall say the `canonical` algorithm has a $\gamma_n$ error bound or, in a further abuse of notation to ease exposition, an $\mathcal{O}(n)$ error bound.

*Implementation notes.* Canonical dot product is the usual starting point for optimized block products. A host of transformations can easily be performed on canonical product (unrolling, pipelining, peeling, vectorization, prefetching, etc.). SIMD vectorization (as is used in architectural extensions like SSE or 3DNow!) in particular is equivalent to a "postload" blocked dot product, as discussed next in section 2.2, and will approximately divide the error bound by the vector length; e.g., if $n$ is a multiple of the vector length of 4, a typical vectorization will produce an error bound of $\gamma_{\left(\frac{n}{4}+3\right)}$. Canonical dot product is almost never used to directly build a high performance GEMM, since it fails to efficiently use the memory hierarchy. Dot product has no opportunity for cache reuse, but GEMM does. Therefore, when parallel dot products are used to implement matrix multiply, they are typically blocked to encourage cache reuse, and this type of dot product is discussed in section 2.2.

**2.2. Blocked dot product.** For some optimizations it is necessary to block operations into chunks that make good use of the various levels of local memory that exist on computers. For a dot product of two vectors of large dimension $N$, this implies breaking up the vectors into $N_b$-sized subvector chunks that are computed separately and then added together. There are two obvious algorithms for blocked dot product, which we call *preload* and *postload*; we show that postload is strongly preferable to preload due to error growth. Figure 2.1 gives pseudocode for both versions of the algorithm (we assume $N$ is a multiple of $N_b$ for expositional simplicity throughout this section).

```
s = 0.0
blocks = N/N_b
for(b = 0; b < blocks; b++)
{
   for(i = 0; i < N_b; i++)
      s = s ⊕ (x[i] ⊙ y[i])
   x += N_b; y += N_b
}
return(s)
```

**(a) Preload blocked dot product**

```
s = 0.0
blocks = N/N_b
for(b = 0; b < blocks; b++)
{ sb = (x[0] ⊙ y[0])
   for(i = 1; i < N_b; i++)
      sb = sb ⊕ (x[i] ⊙ y[i])
   s = s ⊕ sb
   x += N_b; y += N_b
}
return(s)
```

**(b) Postload blocked dot product**

FIG. 2.1. *Pseudocode for blocked dot products.*

The preload algorithm of Figure 2.1(a) is probably the most obvious implementation. However, it is not optimal errorwise. The term $s$ is used in every intermediate computation, so the error term on $s$ will dominate the total error. The first add to $s$ is an add to zero that does not cause error. So there are $N-1$ adds to $s$, along with

the $\gamma_1$ error bound from the multiply, which means that preload blocked dot product has the same error bound as `canonical`.

Now consider a slight alteration to this algorithm, as shown in Figure 2.1(b). Instead of accumulating on a single value throughout the computation, we accumulate the dot product for each block separately and then add that result to $s$. So the blocked dot product consists of $\frac{N}{N_b}$ canonical dot products each of size $N_b$, each of which then adds to the total sum. In [17, p. 70] Higham notes the forward error bound for this algorithm is now

$$|s - \hat{s}| \leq \gamma_{\left(\frac{N}{N_b}+N_b-1\right)} |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}|. \tag{2.2}$$

If we assume a fixed $N_b$, postload reduces the error bound by a constant factor which depends on $N_b$. The minimum value is found by looking at the $\gamma$ subscript as a function of $N_b$, say $f(N_b) = N_b + \frac{N}{N_b} - 1$. As Higham notes, the minimum occurs at $N_b = \sqrt{N}$, yielding an error bound of $\gamma_{\left(2\sqrt{N}-1\right)}$, thus changing the *order* of the error. Extending this procedure to an arbitrary number of levels of blocking is the key idea behind superblocking.

*Implementation notes.* Most high performance GEMM implementations use one of these blocked algorithms, where the $N_b$ value is chosen based on the size of one of the caches and other architectural features (such as the number of floating point registers, instruction cache size, etc.). It is perhaps not obvious, but the postload algorithm requires no extra storage when used in GEMM. When extended to GEMM, the scalar accumulators used by the dot product algorithm naturally become output *matrices*. However, these output matrices must be loaded to registers to be operated on, and thus the architecture provides a set of temporaries that are not present in storage. This is indeed where the algorithms get their names: in preload, the summation-so-far is loaded to the registers before beginning the loop indexing the common dimension of the input matrices, but in postload the summation is not loaded until that loop is complete. Therefore, whether the preload or postload algorithm is used varies by library (ATLAS mostly uses the preload algorithm at present); indirect experience with vendor-supplied BLAS seems to indicate that many use the preload version, but it is impossible to say for sure what a closed-source library does algorithmically. However, personal communication with Fred Gustavson (who is strongly involved in IBM's computational libraries) indicates that at least some in the industry are aware of the error reductions from postload and have at least historically used it.

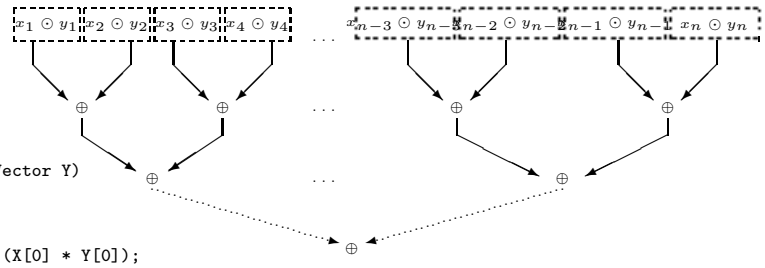
In a high performance library $N_b$ is chosen to optimize performance and can be heavily architecture dependent, and so it cannot typically be varied exactly as called for to get the $\gamma_{\left(2\sqrt{n}-1\right)}$ bound. In our own ATLAS (and we suspect in other libraries as well), $N_b$ is either completely fixed or selectable from a small set of values that obtained the best performance. However, even libraries with a fixed $N_b$ can produce lower-order worst-case errors for a reasonable range of problems sizes, and those that can choose amongst several candidate $N_b$'s can do even better, as we outline next in section 2.2.1.

**2.2.1. Optimal and near-optimal blocking for postload dot product.** Postload blocked dot product has the error bound given in (2.2), but to achieve this bound $N_b$ must be variable, and we believe in most implementations it is not. What we show here is that there is wide latitude in choosing $N_b$, so a few fixed block sizes can achieve the $\mathcal{O}(\sqrt{N})$ error bounds across an expansive range of problem sizes. More formally, suppose, for some small constant $c$, that $N_b = c \cdot \sqrt{N}$. Then the error

factor on each term will be

$$\gamma_{\left(c\cdot\sqrt{N}+\frac{N}{c\cdot\sqrt{N}}-1\right)} = \gamma_{\left(c\cdot\sqrt{N}+\frac{1}{c}\cdot\sqrt{N}-1\right)} = \gamma_{\left((c+\frac{1}{c})\sqrt{N}-1\right)}. \tag{2.3}$$

*Implementation notes.* The utility of this modified bound becomes clear with an example. Suppose we have a GEMM with an $N_b = 60$, a typical value used by ATLAS. This is the perfect block size for $N = 3600$. But it achieves nearly the same error bound for all $N \in [900, 14400]$: If $N = 900$, the optimal blocking factor is 30, so at this extremity, $N_b = \frac{1}{2}\sqrt{N}$, and so by (2.3) we have an error bound of $\gamma_{\left(2.5\sqrt{N}-1\right)}$. At the other extremity, the optimal blocking factor for $N = 14400$ is 120. Then $N_b = 2\sqrt{N}$, so by (2.3) we again have an error bound of $\gamma_{\left(2.5\sqrt{N}-1\right)}$. This is only about 25% higher than if we were able to use the optimal values of 30 and 120, and no $N \in [900, 14400]$ has a worse bound. If we wanted $\mathcal{O}(\sqrt{N})$ bounds on much larger $N$, a second choice of $N_b' = 240$ would seamlessly cover the adjacent range $N = 14400$ to $N = 230400$ with the same bound, $\gamma_{\left(2.5\sqrt{N}-1\right)}$. Any $N_b' \in (60, 240]$ would overlap the range $[900, 14400]$; in such cases $N_b$ or $N_b'$ could be chosen, either the one producing the lesser bound or by performance related criteria. Because of this wide latitude in choosing $N_b$, one may imagine a GEMM implementation in which a small handful of block sizes with overlapping ranges are chosen in order to allow $\mathcal{O}(\sqrt{N})$ error bounds across the full range of practical problem sizes, with little to no impact on even the most highly tuned performance.




```
scalar dotProduct
   (int n, Vector X, Vector Y)
{  scalar sum;
   int n1 = n/2;
   int n2 = n - n1;
   if (n == 1) return (X[0] * Y[0]);
   sum =  dotProduct(n1, X[0:n1-1], Y[0:n1-1]);
   sum += dotProduct(n2, X[n1:n-1],Y[n1:n-1]);
   return (sum);
} // END *** dotProduct ***
```

**(a) Pairwise pseudocode** **(b) Pairwise summation diagram**

FIG. 2.2. *Pseudocode and flop exposure of* `pairwise` *dot product.*

**2.3. Stability of the pairwise dot product.** Finally, we consider the `pairwise` algorithm, which can be naturally implemented using recursion, as shown in Figure 2.2(a). This algorithm performs precisely as many flops as the canonical form, but, instead of accumulating the products one by one, it constructs a $\lceil \log_2(n) \rceil$ deep binary tree of them, as shown in Figure 2.2(b). Thus this algorithm has the property of distributing the flop load exactly equally among its inputs and thus minimizing the worst-case error. `Pairwise` is discussed in [17, p. 70], where Higham produces an error bound for the forward error for this algorithm:

$$|s_n - \widehat{s_n}| \leq \gamma_{(\lceil \log_2 n \rceil + 1)} |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}|. \tag{2.4}$$

`Pairwise` demonstrates an instance where recursion, by distributing usage of prior results uniformly, inherently improves the error bound of the result. In general, this is a powerful principle: The fewer flops elements are subjected to, the lower the

worst-case error. We will demonstrate in section 4 that these lower worst-case error algorithms do indeed produce lower actual errors on average than `canonical`.

*Implementation notes.* Despite its superior error bound, this algorithm has several drawbacks that prevent it from being the default dot product for performance-aware applications. First, the general recursive overhead can be too expensive for most applications. Second, the smaller sizes found towards the bottom of the recursion prevent effective use of optimizations such as unrolling, pipelining, and prefetch. These optimizations often must be amortized over reasonable length vectors, and, for optimizations such as prefetch, we must be able to predict the future access pattern. Straightforward recursive implementation will limit or completely remove the freedom to perform these optimizations, and so it is generally much less optimizable than a loop-based implementation, even when the recursive overhead can be minimized. We note that the naive version of this algorithm requires $\frac{n}{2}$ workspaces (stored on the stack in the recursive formulation) to store the partial results. Generally, extra workspace usage results in greater cache pollution, which tends to degrade performance even further. With smarter accumulator management, we may reduce the worskpace requirements to $(1 + \log_2(n))$ (see [4] for details). We will derive a similar result using our `superblock` algorithm in section 3. However, since in matrix multiply these workspaces must be matrices (as opposed to scalars for dot product), `pairwise` is usually not practical due to memory usage, even if the performance considerations highlighted above do not discourage its use.

**3. Superblocked dot product.** The error bound on a dot product is primarily due to summation; the initial multiplication of elements adds only one to the final $\gamma$ subscript. Thus we begin both our analysis and introduction with `superblock` summation: Given $t$ temporaries we can accumulate a sum in $t$ levels; so our initial idea was to find optimal blocking factors for each level to minimize the upper bound of the floating point error. We discovered the ideal blocking factor is identical for all levels, which also simplifies the error bound. The code for this generalized `superblock` dot product is given in Figure 3.1(a), and an illustration of the summation part of the algorithm is shown in Figure 3.2. In Figure 3.2 note that each level needs only one temporary, so all shown summations are using the same workspace. Therefore, after level $t-1$ receives its $N_b$th addition into the first running sum shown on the left of level $t-1$ of Figure 3.2, the $t-1$ sum is added into the current $t-2$ sum and zeroed, and the next summation shown to the right is begun using the same workspace. The principle is straightforward; on each level the blocking factor $N_b$ is identical,[2] and for $t$-level summation this requires $N_b = N^{1/t}$.

When there is only one level ($t = 1$) this algorithm becomes `canonical`, for $t = 2$ it becomes `postload blocked`, and for $t = \log_2(N)$ it becomes space-optimal `pairwise` (as shown in Proposition 3.3). Therefore, all the algorithms surveyed in section 2 may be viewed as special cases of the `superblock` class of dot products. In Proposition 3.1 we first prove the `superblock` summation result and its error bound; then in Proposition 3.2 we extend that result to the `superblock` dot product.

PROPOSITION 3.1. *For a $t$ temporary `superblock` summation, the blocking factor that minimizes the worst-case error counter is $N^{\frac{1}{t}}$, which produces a worst-case error counter of $\langle t(N^{\frac{1}{t}} - 1)\rangle$, corresponding to an error bound of $\gamma_{(t(N^{\frac{1}{t}}-1))}$.*

*Proof.* The proof is inductive on $t$. The proposition is trivially true for $t = 1$, producing a worst-case error counter of $\langle 1(N^{\frac{1}{1}} - 1)\rangle = \langle N - 1\rangle$.

[2]For clarity we assume the computation produces an integer blocking factor.

```
scalar dotProd(Vec X, Vec Y, int t)
{  int n = X.length;
   int nb = pow(n, 1/t);
   scalar tmp[t] = {0.0};
   int cnt[t] = {0};

   for (i=0; i < n; i++)
   {
      tmp[t-1] += X[i] * Y[i];
      if (++cnt[t-1] == nb)
      {
         for (j=t-2; j; j--)
         {  tmp[j] += tmp[j+1];
            tmp[j+1] = 0.0;
            cnt[j+1] = 0;
            if (++cnt[j] < nb) break;
         }
      }
   }
   return(tmp[0]);
}
```

**(a) $t$-level superblock**

```
scalar dotProd(Vec X, Vec Y, int nb)
{  int n = X.length;
   int nblks = n/nb;
   int nsblks = sqrt(nblks);
   int blksInSblk = nblks/nsblks;
   scalar dot=0.0, sdot, cdot;

   for (s=0; s < nsblks; s++)
   {  sdot = 0.0;
      for (b=0; b < blksInSblk; b++)
      {
         cdot = X[0] * Y[0];
         for (i=1; i < nb; i++)
            cdot += X[i] * Y[i];
         sdot += cdot;
         X += nb; Y += nb;
      }
      dot += sdot;
   }
   return(dot);
}
```

**(b) 3-level fixed-$N_b$ superblock**

FIG. 3.1. *Pseudocode for* `superblock` *algorithms.*

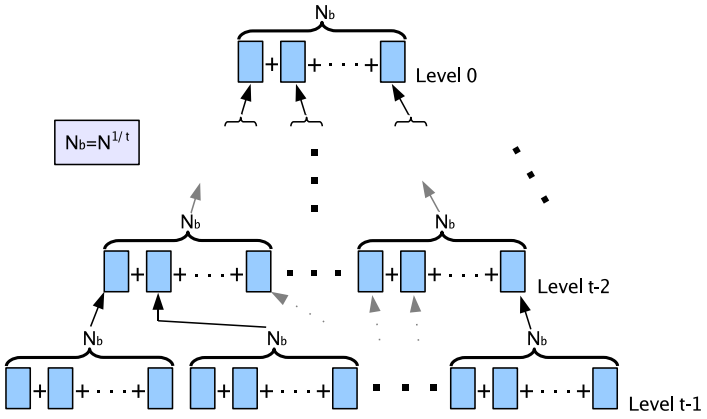



FIG. 3.2. *"t"-level* `superblock` *summation.*

Therefore, assume the proposition is true for $t$. For $t+1$ level blocking we choose $N_b$ as the lowest level blocking factor. This will produce a worst-case error counter of $\langle N_b - 1\rangle$ on each block summation, with $\frac{N}{N_b}$ elements (the block sums) to be added in the subsequent $t$ levels of the addition. By the proposition, the ideal blocking factor for these remaining $t$ levels is $(\frac{N}{N_b})^{\frac{1}{t}}$, and will produce a worst-case error counter of $\langle t((\frac{N}{N_b})^{\frac{1}{t}} - 1)\rangle$, which shall be added to $\langle N_b - 1\rangle$ to find the worst-case error counter of a $(t+1)$-level `superblock` summation. This can be expressed as $\langle f(N_b)\rangle$ with $f(N_b)$ shown in (3.1a). We minimize this and solve for $N_b$, as shown in (3.1b).

$$f(N_b) = N_b - 1 + t\left(\left(\frac{N}{N_b}\right)^{\frac{1}{t}} - 1\right), \tag{3.1a}$$

$$f'(N_b) = 1 + t\left(N^{\frac{1}{t}} \times \frac{-1}{t} \times N_b^{\frac{-1}{t}-1}\right) = 0 \Rightarrow 1 = N^{\frac{1}{t}} \times N_b^{\frac{-(t+1)}{t}} \Rightarrow N_b = N^{\frac{1}{t+1}}. \tag{3.1b}$$

Thus the optimal $N_b$ for level $t+1$ is $N^{\frac{1}{t+1}}$, leaving $\frac{N}{N_b} = \frac{N}{N^{\frac{1}{t}}} = N^{\frac{t}{t+1}}$ elements to be summed in $t$ levels. By our assumption the optimal blocking factor for these elements is $(N^{\frac{t}{t+1}})^{\frac{1}{t}} = N^{\frac{1}{t+1}}$, so the same blocking factor is used on all $t+1$ levels. Substituting this $N_b$ into (3.1a) gives the worst-case error counter of

$$\left\langle N^{\frac{1}{t+1}} - 1 + t\left(\left(N^{\frac{t}{t+1}}\right)^{\frac{1}{t}} - 1\right)\right\rangle \iff \left\langle (t+1)(N^{\frac{1}{t+1}} - 1)\right\rangle$$

which implies an error bound of $\gamma_{((t+1)(N^{\frac{1}{t+1}}-1))}$, as desired. ∎

PROPOSITION 3.2. *The following bound holds true on the forward error for the t temporary* `superblock` *dot product computation:*

$$|s_n - \widehat{s_n}| \leq \left(\gamma_{t\left(\sqrt[t]{N}-1\right)+1}\right)|\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|. \tag{3.2}$$

*Proof.* `Superblock` dot product differs from summation only in that it has 1 additional error factor of $(1+\delta)$ due to the multiply, and we note that adding 1 to the subscript of the result proven in Proposition 3.1 yields (3.2). ∎

PROPOSITION 3.3. *An $N = 2^t$, t-level* `superblock` *dot product is equivalent to the space-efficient* `pairwise` *dot product.*

*Proof.* Replacing $N$ with $2^t$ initially, and later $t$ with $\log_2(N)$, and applying Proposition 3.2 tells us the worst-case error counter must be

$$\left\langle t((2^t)^{\frac{1}{t}} - 1) + 1\right\rangle = \left\langle t(2^{\frac{t}{t}} - 1) + 1\right\rangle = \langle t+1\rangle = \langle \log_2(N) + 1\rangle,$$

which implies the error bound

$$|s - \hat{s}| \leq \gamma_{(\log_2(N)+1)}|\mathbf{x}|^{\mathsf{T}}|\mathbf{y}|$$

identical to the `pairwise` result (2.4), which is accomplished in $t = \log_2(N)$ workspaces. ∎

*Storage note.* Caprani [4] showed space-efficient `pairwise` takes $\lfloor \log_2(N) \rfloor + 1$ storage locations, which disagrees with our count of $t = \log_2(N)$ (where we assume $\lfloor \log_2(N) \rfloor = \lceil \log_2(N) \rceil$) as just shown in Proposition 3.3. Caprani uses a stack-based scheme almost identical to ours, but his algorithm assumes a separate temporary storage area for the final result; instead we assume the result is being computed in place. This accounts for the one unit difference, and thus we do not claim to require less storage despite the differing counts.

**3.1. Fixed-$N_b$ superblock.** As so far presented, `superblock` is interesting mainly from a theoretical standpoint, since its implementation would probably be only a little more practical than `pairwise`. However, we can make a straightforward adaptation to this algorithm which makes it a practical algorithm for building a high performance GEMM (at least in the way we perform GEMM in ATLAS) in those cases where the problem size is too great for `postload blocked` GEMM alone to give the lower-order worst-case error term. As previously mentioned, in implementation $N_b$ is either fixed or at most variable across a relatively narrow range. Therefore, we assume $N_b$ is not variable when deriving our practical `superblock` algorithm. The second choice is how many temporaries to require. Depending on the types and levels of cache blocking applied by ATLAS's GEMM, each additional temporary beyond the problem's output $N \times N$ matrix and the machine's registers (which handle the post-load dot product in the innermost loop) would require either an $N_b \times N_b$ temporary in the best case or an $N \times N_b$ in the worst. Also, additional storage locations will tend to depress performance due to added cache pollution. Therefore, we choose to add only

one additional workspace beyond the problem's output and the machine's registers, leading to the $t = 3$ algorithm shown in Figure 3.1(b) (for clarity we again assume that all blocksize calculations produce nonzero integral answers). This produces an error bound of $\gamma_{(N_b+2(\sqrt{\frac{N}{N_b}}-1))}$ (note that, as expected, this is the same as (3.2) when $N_b = \sqrt[3]{N}$). We believe this algorithm, requiring only one additional buffer, will provide reasonable error reduction on pretty much all problem sizes that are practical in the near and medium term (section 4 puts some statistics behind this belief), without insupportable workspace requirements or sharp performance reductions.

**4. Statistical studies.** It is widely known that worst-case errors are almost never seen in practice. This is mostly due to the fact that a prior overestimation is often balanced by a later underestimation, so that the worst-case bound is indeed loose. Many practitioners believe that with these extremely loose bounds and the self-cancelling nature of floating point error, all of the algorithms perform fairly indistinguishably for most data. This idea is endorsed in a limited way in [19], which demonstrates that there exist particular data and orderings which will make any of these "better" dot product algorithms produce worse results than the others (e.g., a case can be constructed in which `pairwise` gets worse error than `canonical`). The conclusion of this paper goes further (remember that the "recursive summation" of Higham is our "`canonical`"):

> However, since there appears to be no straightforward way to predict which summation method will be the best for a given linear system, there is little reason to use anything other than the recursive summation in the natural order when evaluating inner products within a general linear equations solver.

This section provides results of statistical studies we have undertaken, which show that there is indeed a benefit on average to using the lower worst-case error algorithms and that this benefit grows with length (though not at anything like the rate suggested by the worst-case analysis).

**4.1. Experimental methodology.** A frequently used approach for experimental exploration of algorithmic improvements is to select vectors with known properties that will behave in known ways. In contrast to this approach, we wish to understand how much error reduction a user can expect on more typical data. To get a feel for this, we chose a more statistical approach, where we contrast the behavior of various algorithms using the same randomly generated vectors. For each vector length $n$ we randomly generate 10,000 different vector pairs, and each algorithm (including `canonical`) is run on each of these vector pairs (so algorithms are always compared using the same data) using IEEE single precision. The amount of error is found by comparison to an "exact" answer for the same pair computed with IEEE double precision and compensated addition (originally described by Kahan [20]; various error analyses reported by Higham (p. 85 of [18])), which is theoretically accurate to $\pm u$ for IEEE single precision.[3] We consider two cases of interest for unstructured data, and so we have separate charts for when the elements are generated continuously in the range [−1,1] and [0,1]. All of these experiments were run on an x86 machine using single precision SSE instructions (so that true 32 bit precision is enforced).

[3]Keep in mind the error bound is for an accumulation in double precision and then translated to single precision for comparison.

Because we wish to make a statistical argument, it is important that we are measuring something that can be meaningfully averaged over our 10,000 experiments. Relative error is a fairly standard metric of error, so we originally averaged the individual relative errors of each algorithm (i.e., for each input $X$ and $Y$, compute $\frac{|X \cdot Y - \widehat{X \cdot Y}|}{|X \cdot Y|}$). Unfortunately, this approach is fundamentally flawed for mixed-sign data: over the 10,000 trial vector pairs it is not uncommon to find a few outliers for which the relative error denominator, $|X \cdot Y|$, is very small, making the individual ratio so large that these outliers dominate the overall average (in some cases eliminating a single outlier would reduce the average by an order of magnitude). Therefore, we see that averaging ratios where the denominator can be arbitrarily near zero makes the average too sensitive to outliers and is therefore unlikely to represent the "typical" case. We also considered averaging the ratios of the relative errors produced by the two algorithms, but of course on identical input vectors the denominators ($|X \cdot Y|$) would cancel, leaving the ratio of two absolute errors in the form $\frac{|e_1|}{|e_2|}$, and this suffers from the same sensitivity drawback: If on a particular vector pair the second algorithm produces an $|e_2|$ very near zero, a huge individual ratio can overwhelm the overall average ratio.

We chose instead to average the absolute errors for each algorithm *first* and *then* find the ratio of these two averages. It is important to point out that "average absolute error" is a relatively meaningless measure of algorithmic error in isolation; but the *ratio* of these averages, when both algorithms are processing *identical vector pairs*, does have statistical value: For a given $n$ this ratio is proportional to the ratio of the statistical expected value of their forward errors[4] and proportional to the ratio of their expected relative errors (assuming the dot product is never zero).

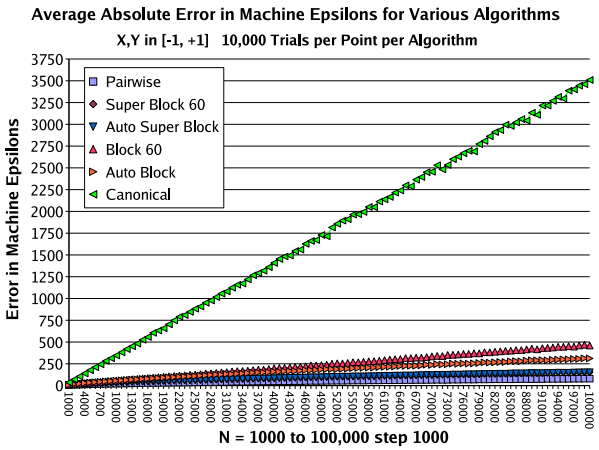
Further, if on a particular dot product one algorithm produces error $|e_{1_i}|$ and a second algorithm $|e_{2_i}| = \alpha \cdot |e_{1_i}|$, with $\alpha > 1$, then in general it requires $\log_2(\alpha)$ more bits to represent $|e_{2_i}|$ than to represent $|e_{1_i}|$, implying $\log_2(\alpha)$ fewer bits of accuracy in $|e_{2_i}|$, regardless of the magnitude of $|e_{2_i}|$. The same formula does not hold for the averages[5] $\overline{|e_1|}$ and $\overline{|e_2|}$, but in our experiments we did calculate the average bits of accuracy and found over all algorithms tested a greater than 99% correlation[6] between our ratio, $\log_2(\overline{|e_1|}/\overline{|e_2|})$, and the average bits of significance lost (or added) when it is positive (or negative, due to $\overline{|e_1|}/\overline{|e_2|}$ being less than 1). In fact for our algorithms $\log_2(\overline{|e_1|}/\overline{|e_2|})$ by itself predicted the average advantage in bits of significance within 0.9 bits for the same-sign vectors and within 0.5 bits for mixed-sign vectors.

**4.2. Results.** For each of the charts shown here, we compare the surveyed algorithms against `canonical`. The algorithms are `pairwise`, `autol3superblock` (`superblock` with $t = 3$ and $N_b = \sqrt[3]{N}$), `l3superblock60` (`superblock` with $t = 3$ and a fixed lowest-level blocking of $N_b = 60$), `autoblock` (`postload blocked` with

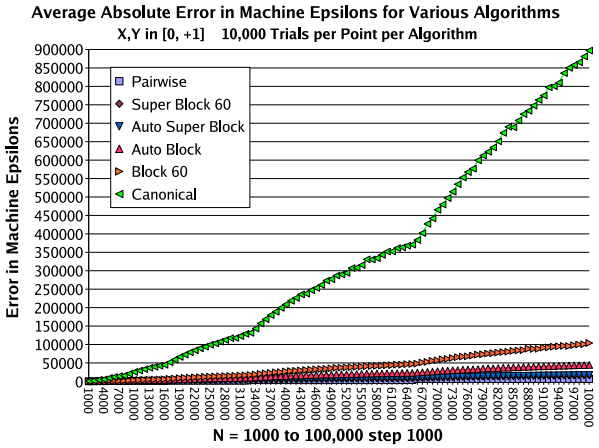
[4]Consider the form of the upper bound on error for an individual dot product of $n$ length vectors $X_i$ and $Y_i$, $|e_i| \leq c \cdot |X_i|^{\mathsf{T}}|Y_i|$, where $c$ is a constant (usually computed as $\gamma_{f(n)}$, e.g., $\gamma_n$ or $\gamma_{\log_2(n)}$, but with $n$ given this reduces to a constant). The constant $c$ is statistically independent of the values of $|X_i|^{\mathsf{T}}|Y_i|$. We adopt this form as the model for the *expected* amount of error; so for algorithm 1 we assume a linear model for the error on each vector pair of $|e_{1_i}| = c_{1_i} \cdot |X_i|^{\mathsf{T}}|Y_i|$, and for algorithm 2 (on the same vector pair) $|e_{2_i}| = c_{2_i} \cdot |X_i|^{\mathsf{T}}|Y_i|$. Using appropriate random variables, taking expected values on both sides, using the independence argument, and assuming the errors are normally distributed, we find $E(c_1)/E(c_2) = \overline{|e_1|}/\overline{|e_2|}$; i.e., the ratio of the average absolute errors is proportional to the ratio of forward errors. A similar argument holds for relative errors; thus we believe this ratio is a reasonable general measure for comparing algorithmic error.

[5]The average of the logs of a set of values does not equal the log of the set's average value.

[6]This result is with the exception of `block60` on same-sign data, which had only a 95% correlation.




**(a) For random data in range [−1,1]**




**(b) For random data in range [0,1]**

**Legend:** *Black hourglasses: Pairwise. Red dashed line: autol3superblock. Dark blue solid line: l3superblock60. Light green point-down triangle: autoblock. Light blue squares: block60.*

FIG. 4.1. *Average absolute error of tested algorithms. Color is available only in the online version.*

$N_b = \sqrt{N}$), and `block60` (`postload blocked` with $N_b = 60$). These parameter values were chosen due to practical reasons: $N_b = 60$ is a typical midrange blocking factor (when $N_b$ is selected for performance), and $t = 3$ requires only one additional workspace in GEMM.

Average absolute errors for the various algorithms are charted in Figure 4.1(a) and (b), scaled by the constant $\epsilon_M = 2^{-23}$ ($\epsilon_M$ is the smallest power of 2 which can be added to 1 to get a different number) for display purposes. In both charts, for the majority of the span the ranking of the curves is (from top (worst) to bottom (best)): `canonical`, `block60`, `autoblock`, `autol3superblock`, `l3superblock60`, and `pairwise` (although `autol3superblock` and `l3superblock60` are nearly indis-

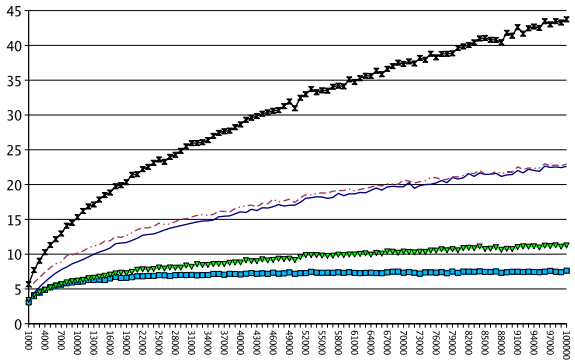
tinguishable). We see that the improved algorithms perform notably better than `canonical`. However, if we take the mixed-sign data as the more typical case, we can perhaps better understand the above quote from Higham: despite the obvious win experienced on average by the improved methods, the total error is quite low, and it seems unlikely many problems would be so sensitive to error that an improved algorithm would be critical.

The error buildup is much more appreciable on same-sign data, but of course here we might expect the fact that the answer is also large to help hide the increased error. In this case, however, if the vectors are sufficiently long, new elements cease to change the result due to alignment error, and thus, for long enough vectors, this should eventually prove intolerable, even in relative error. Therefore, areas in which an improved algorithm might be critical include extremely long same-sign vectors, vectors which have long-running patterns of same-sign results, but whose answer is nonetheless small (e.g., imagine multiplying two vectors which produce positive results for the first $\frac{n}{2}$ elements and negative for the remainder), highly iterative algorithms which accumulate error, or applications processing inherently low resolution data (such as 8-bit sensor readings). However, even absent these conditions, our results show that these improved methods substantially reduce the average error, and therefore it makes sense to employ at least those methods which do not negatively impact performance whenever possible.

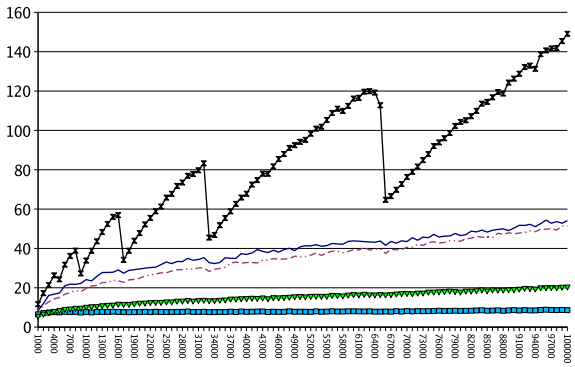
These graphs are revealing, but the fast-rising `canonical` error makes all other algorithms almost indistinguishable due to scale, as well as making all errors appear somewhat linear. Therefore, in our remaining error charts, we track the ratio of `canonical`'s average absolute error divided by the average absolute error achieved by the improved method (as discussed in section 4.1). Thus, an algorithm with a plotted ratio of 10 achieved an average absolute error 10 times smaller than `canonical` on the vectors of that size. Note this inverts the order of the charts relative to Figure 4.1, so `canonical` will always be "1" and `pairwise`, with the least error, will always be the curve on top. The average is over the 10,000 trial vector pairs; each algorithm uses the same unique 10,000 vectors for each vector length $n$.

Figure 4.2 shows the average (over the 10,000 trials) absolute error of the `canonical` algorithm divided by the average error of the surveyed algorithms on the range $N = [1000,\ 100000]$ in steps of 1000, with the problem sizes along the X-axis and the error ratio along the Y-axis. Figure 4.2(a) shows this chart for mixed-sign vectors, and Figure 4.2(b) shows the same for all-positive vectors. In Figure 4.2(a) the order of the curves from top (best) to bottom (worst) is `pairwise`, `autol3superblock`, `l3superblock60`, `autoblock` and `block60`. The `canonical` algorithm is not shown; it would be the horizontal line of "1" (and thus the most error prone algorithm in the chart). In Figure 4.2(b) the algorithm `l3superblock60` is above `autol3superblock`, although they are both still quite close.

The first thing to note is that the error ratios for the all-positive data is much larger than for the mixed sign. This may at first be counterintuitive, as all-positive vectors have a condition number of 1. However, one of the main ways these algorithms reduce error is by minimizing alignment error (i.e., bits lost when mantissas with differing exponents must be aligned prior to adding) by tending to add elements that are likely to be in the same basic range (since they have been exposed to a more balanced number of flops). Alignment error is less of an issue for mixed-sign data, as the dot product accumulator does not necessarily grow at every step as it can with same-sign data.



**(a) For random data in range [−1,1]**



**(b) For random data in range [0,1]**

**Legend:** *Black hourglasses: Pairwise. Red dashed line: autol3superblock. Dark blue solid line: l3superblock60. Light green point-down triangle: autoblock. Light blue squares: block60.*

FIG. 4.2. *Ratio showing reduction in average absolute error. Color is available only in the online version.*

The worst performer of the improved algorithms is always `block60`, which nonetheless produces 7 (8) times less error on average than `canonical` for long vectors (mixed and same sign, respectively). It may seem surprising that `block60` is competitive with `autoblock`, since `block60` has $\mathcal{O}(N)$ error bound where autoblock has $\mathcal{O}(\sqrt{N})$. However, our analysis in section 2.2.1 shows `block60` has an $\mathcal{O}(\sqrt{N})$ error bound for much of this range. Since actual error builds up much slower in practice, `block60` maintains this lower-order behavior longer as well (note that the range where `block60` is almost the same as `autoblock`, $N < 10000$, is well within the lower-order range proven in section 2.2.1). Since this form of GEMM requires no extra storage, this is a strong suggestion that, even absent other measures, it makes sense to utilize `postload blocking` in modern GEMM algorithms and that it produces lower average errors on most problem sizes in use today.

Another surprising result is how much difference separates the `l3superblock` algorithms from the `autoblock` algorithm, since they both have an $\mathcal{O}(\sqrt{N})$ error bound. The different 3-level `superblock` algorithms, as expected, behave almost the same in practice, which indicates that a fixed-$N_b$ superblock, which allows for much greater tuning freedom than autoblocked, will be adequate for error control. In mixed-sign

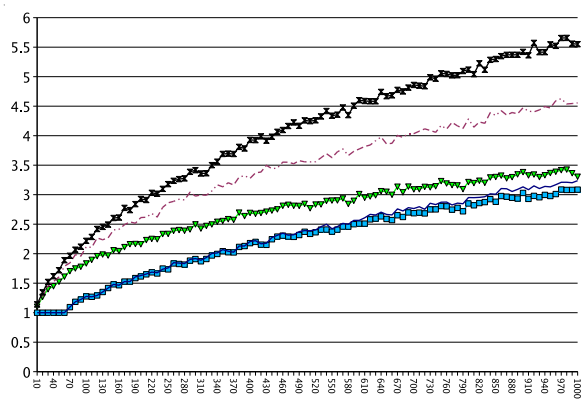
data, the 3-level `superblock` algorithms behave as expected: `autol3superblock` is generally slightly better, but since the lines are so close, `superblock60` occasionally wins. For same-sign data, `superblock60` actually wins across the entire range, though again the difference is minor. It is difficult to say why this might be the case, but we note that the optimal block factor is based on a worst-case analysis which does not happen in practice, so using larger $N_b$ should not cause a problem. It may be that $N_b = 60$ results in fewer alignment error on average when adding the higher level blocks (e.g., the summation totals for $N_b = 60$ are more uniform than for $N_b = \sqrt[3]{N}$), but this is pure speculation on our part. We note that `l3superblock` is substantially better errorwise (with an error almost 23 (55) times better than `canonical`, respectively) than `postload blocked` for all but the very beginning of this range, which suggests that error-sensitive algorithms may want to employ `superblock` even for reasonably sized vectors if the performance effect can be made negligible.

Finally, we notice that `pairwise` is decidedly better on average across the range. The clear superiority of `pairwise` over the next best tested algorithm, `l3superblock60`, indicates it may be interesting to study higher level `superblocks` to see how the performance/error win trade-off plays out in practice. We note that the `pairwise` algorithm displays a sawtooth pattern for the same-sign data, with the least average error (maximum ratio) found at powers of two. Again, the reason is probably due to alignment error: when `pairwise`'s binary tree becomes unbalanced because $N$ is slightly above a given power of 2, each branch of the tree will yield markedly different-sized values, thus increasing the probability of alignment error. It seems likely that changing the algorithm to better balance the addition tree could moderate the drops, but, since `pairwise` is not our main focus, we did not investigate this further.

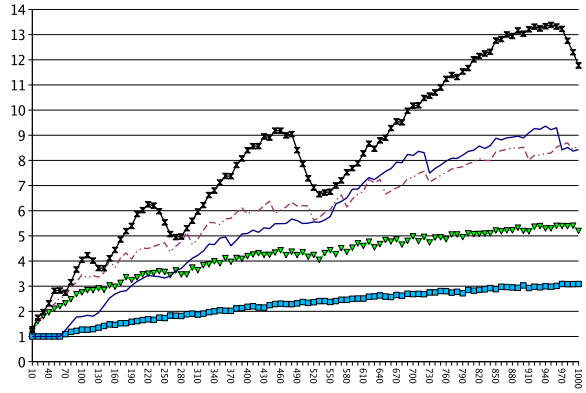
Since standard methods work fine in practice for smaller problems, we have concentrated primarily on these large problem sizes. However, as many people are interested in small-case behavior, Figure 4.3 gives the same information for $N = [10,\ 1000]$ in steps of 10. Here we see less algorithmic difference, as one would expect. For instance, for all $N \leq 60$, `canonical`, `l3superblock60`, and `block60` are the same algorithm. These fixed-size block algorithms do not strongly distinguish themselves from `canonical` until any $N$ mod $N_b$ is overwhelmed by problem size, as we see. Therefore, the fixed-$N_b$ algorithms are mainly interesting for decreasing error for large problems; since these are precisely the cases in which we most need to ameliorate the buildup of error, this is not a drawback in practice.

Having considered average performance, we next consider exceptions to the average. For each algorithm we tallied its "Win/Lose/Tie" record against `canonical` on the same vectors. Here we discuss `l3superblock60`, but very similar conclusions apply to all the algorithms in keeping with the ranking of their error bounds. "Win" means the alternative algorithm had less error, "Loss" means it had more error, and "Tie" means the error was identical within the roundoff error. For mixed-sign data, `l3superblock60` wins or ties against `canonical` 95% of the time, and for same-sign data 99% of the time, becoming increasingly superior as the vector length increases.

For the small percentage of cases in which `l3superblock60` loses to `canonical`, it is never because `l3superblock60` produced an excessively large error. In fact, the largest error it produces in such circumstances is still only 52% of `canonical's` *average* error: The reason `l3superblock60` loses is because `canonical` produced an unusually small error on that particular vector: On average, when `canonical` wins against `l3superblock60`, the canonical error is just 10% of the overall average of canonical error and the `l3superblock60` error is 20% of that overall average.



**(a) For random data in range [−1,1]**



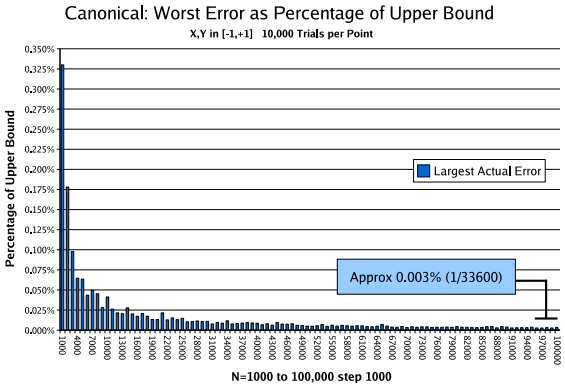
**(b) For random data in range [0,1]**

**Legend for Figure 4.3:** *Black hourglasses: Pairwise. Red dashed line: autol3superblock. Dark blue solid line: l3superblock60. Light green point-down triangle: autoblock. Light blue squares: block60.*

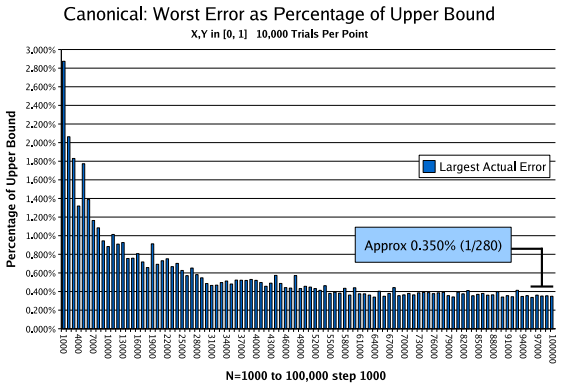
FIG. 4.3. *Ratio of average absolute errors and small problems. Color is available only in the online version.*

**4.2.1. Results as a percentage of worst-case error bound.** It is well known that the theoretical worst-case error bound is extremely loose and almost never achieved for long vectors. Our statistical studies agree strongly with this common wisdom, as shown in Figure 4.4, where we plot the worst error achieved over 10,000 randomly generated vector pairs for each problem size. As expected, mixed-sign data shows lower actual error, and the percentage of the worst-case error goes down strongly with vector length. What may be less well known is just how small the actual error is, at least on this type of data: we see that, even for short vectors of same-sign data (the worst case for error), the worst error ever achieved over 10,000 trials was less than 3% of the theoretical worst-case error bound and that for long vectors this drops off to a paltry 0.35% (which is nonetheless more than two orders of magnitude greater than the percentage encountered for long-vector mixed-sign data).

Essentially this is because, at a high enough condition number for a given $N$, the interim sum of the dot product *of our randomly generated vectors* tends to remain




**(a) For random data in range [−1,1]**




**(b) For random data in range [0,1]**

FIG. 4.4. *Canonical: worst actual error vs. upper bound.*

relatively close to zero and thus does not get large enough to create significant amounts of alignment error. We emphasize that this is a feature of our vector generation, since it is easy to create pairs of vectors whose dot product will have an arbitrarily high condition number and absolute error on par with a dot product with a condition number of 1 (e.g., imagine a product producing all positive numbers for the first half of the product and all negative for the rest). However, our randomized method is extremely unlikely to produce such cases.

One seemingly counterintuitive result that we can observe from these figures is that the lower the condition number, the *larger* the absolute error. This is because low condition numbers are associated with low *relative* error, and here we are charting *absolute* error. It is not a contradiction to have both. Upon reflection it will make sense that absolute error will be higher for low condition numbers, because a low condition number implies $|\mathbf{X}|^{\mathsf{T}} \cdot |\mathbf{Y}| \approx |\mathbf{X}^{\mathsf{T}} \cdot \mathbf{Y}|$, and, for randomly generated data, this implies a growing interim sum and more opportunity for alignment error. On the other hand, for our randomly generated mixed-sign data the interim sum is equally likely to increase or decrease with each add, reducing the opportunity for the buildup of alignment error.

**5. Summary and conclusions.** In summary, we have presented a survey of several of the most important known dot products along with their error properties (section 2), and we showed that the lower-order error achieved by `postload blocking` is not overly sensitive to block size (section 2.2.1), so a relatively small selection of hand optimized block sizes can accommodate an extremely wide range of vector sizes. Further, we have presented a new class of dot product which subsumes these known algorithms, including a modification which is suitable for high performance GEMM (section 3). Finally, in section 4 we demonstrated that despite the very large difference between the theoretical worst-case bounds and the errors observed in practice, which make the worst-case bounds of the algorithms extremely poor estimates of actual error, the worst-case bounds do provide a fairly reliable guide to sorting out which algorithms will provide less error in practice.

Our main conclusion is twofold. The first is that, contrary to some thought, algorithms with lower worst-case error bounds behave noticeably better in practice. The second is that, with the strategies we have outlined, using such algorithms should be possible with little to no performance loss in high performance libraries such as ATLAS. As solvable problem size continues to rise, we believe it will become increasingly important that such libraries do so.

## REFERENCES


[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, 1999.

[2] I. J. Anderson, *A distillation algorithm for floating-point summation*, SIAM J. Sci. Comput., 20 (1999), pp. 1797–1806.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.

[4] O. Caprani, *Implementation of a low round-off summation method*, BIT, 11 (1971), pp. 271–275.

[5] J. Demmel, *Underflow and the reliability of numerical software*, SIAM J. Sci. Statist. Comput., 5 (1984), pp. 887–919.

[6] J. Demmel and Y. Hida, *Fast and accurate floating point summation with application to computational geometry*, Numer. Algorithms, 37 (2004), pp. 101–112.

[7] J. Demmel and Y. Hida, *Accurate and efficient floating point summation*, SIAM J. Sci. Comput., 25 (2003), pp. 1214–1248.

[8] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, *Error bounds from extra-precise iterative refinement*, ACM Trans. Math. Software, 32 (2006), pp. 325–351.

[9] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[10] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *Algorithm* 656: *An extended set of basic linear algebra subprograms: Model implementation and test programs*, ACM Trans. Math. Software, 14 (1988), pp. 18–32.

[11] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An extended set of FORTRAN basic linear algebra subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.

[12] T. O. Espelid, *On floating-point summation*, SIAM Rev., 37 (1995), pp. 603–607.

[13] J. Gregory, *A comparison of floating point summation methods*, Comm. ACM, 15 (1972), p. 838.

[14] J. F. Groote, F. Monin, and J. Springintveld, *A computer checked algebraic verification of a distributed summation algorithm*, Formal Aspects of Computing, 17 (2005), pp. 19–37.

[15] R. Hanson, F. Krogh, and C. Lawson, *A proposal for standard linear algebra subprograms*, ACM SIGNUM Newsl., 8 (1973).

[16] N. J. Higham, *Exploiting fast matrix multiplication within the level* 3 *BLAS*, ACM Trans. Math. Software, 16 (1990), pp. 352–368.

[17] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.

[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, Philadelphia, 2002.
[19] N. J. Higham, *The accuracy of floating point summation*, SIAM J. Sci. Comput., 14 (1993), pp. 783–799.
[20] W. Kahan, *Pracniques: Further remarks on reducing truncation errors*, Comm. ACM, 8 (1965), p. 40.
[21] Ph. Langlois and N. Louvet, *Solving triangular systems more accurately and efficiently*, in Proceedings of the 17th IMACS World Congress, Paris, France, 2005.
[22] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Software, 5 (1979), pp. 308–323.
[23] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. J. Yoo, *Design, implementation and testing of extended and mixed precision BLAS*, ACM Trans. Math. Software, 28 (2002), pp. 152–205.
[24] S. Linnainmaa, *Analysis of some known methods of improving the accuracy of floating-point sums*, BIT, 14 (1974), pp. 167–202.
[25] P. Linz, *Accurate floating-point summation*, Comm. ACM, 13 (1970), pp. 361–362.
[26] J. M. McNamee, *A comparison of methods for accurate summation*, SIGSAM Bull., 38 (2004), pp. 1–7.
[27] T. Ogita, S. M. Rump, and S. Oishi *Accurate sum and dot product*, SIAM J. Sci. Comput., 26 (2005), pp. 1955–1988.
[28] T. G. Robertazzi and S. C. Schwartz, *Best "ordering" for floating-point addition*, ACM Trans. Math. Software, 14 (1988), pp. 101–110.
[29] G. W. Stewart, *Introduction to Matrix Computations*, Academic Press, New York, London, 1973.
[30] R. C. Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, 35 (2005), pp. 101–121.
[31] R. C. Whaley, A. Petitet, and J. J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Comput., 27 (2001), pp. 3–35.