University of Texas at San Antonio, Department of Computer Science
Technical Report CS-TR-2007-002

ERROR ANALYSIS OF VARIOUS FORMS OF FLOATING POINT DOT PRODUCTS *

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

Abstract. This paper discusses both the theoretical and statistical errors obtained by various dot product algorithms. 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 the 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. Therefore, assumptions about limited problem size that used to bound the linear rise in worst-case error due to canonical dot products can no longer be assumed to be true today, and will certainly not be true in the near future. Therefore, this paper discusses several implementations of dot product, their theoretical and achieved error bounds, and their suitability for use as performance-critical building block linear algebra kernels.

Key words. Dot product, inner product, error analysis, BLAS, ATLAS

1. Introduction. This paper discusses both the theoretical and statistical errors obtained by various dot product algorithms. Canonical dot product has a worst-case error bound that rises linearly with vector length. Because vector length has been constrained by the availability of computational and memory resources, and due to the fact that worst case errors are almost never encountered for long vectors, this error has not been deemed intolerable, and many high performance linear algebra operations use algorithms with this worst-case error. With the commoditization of 64-bit memories and multicore CPU dies, the present generation of workstations have already greatly relaxed the computational and memory restraints preventing very large problems from being solved. As continuing software and architectural trends increase both the amount of installed memory in a typical system and the number of CPUs per die, these historical restraints on problem size will weaken further and lead to the possibility of routinely solving linear algebra problems of unprecedented size. With such a historical expansion of problem size, it becomes important to examine whether the assumption that the linear rise in worst-case error is still tolerable, and whether we can moderate it without a noticeable loss in performance. It is for this reason that we have undertaken these dot product studies.

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 [26, 25] project, which uses GEMM as the building block of its high performance BLAS [14, 17, 9, 10, 8] implementation. Therefore, we are keenly interested in both

*This work was supported in part by National Science Foundation CRI grant SNS-0551504
† [castaldo,whaley,atc]@cs.utsa.edu

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

1.1. Background and Related work. Because dot product is so key 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[15], which extended the foundation provided by Stewart in [24]. We will therefore adopt and extend the notation from [15] 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) β 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.

A floating point summation will be represented by $\sum^{n}_{\oplus,i=1}$, and a floating point product by $\prod^{n}_{\odot,i=1}$. We use bold lower case letters to denote vectors. Thus $\mathbf{x}$, $\mathbf{y}$ denote the vectors $[x_1, \ldots, x_n]^T$ and $[y_1, \ldots, y_n]^T$. The dot product of these vectors is $\mathbf{x}^T\mathbf{y} = \sum_{i=1}^{n} x_i y_i$. Therefore we observe that 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 individual elementwise products. Therefore, the most closely related work is on reducing error in summations, as in [20, 12, 19, 23, 22, 11, 2, 6, 21, 13]. To understand floating point summation consider the summation of $x_1, \ldots, x_n$. For example if $n = 5$ and we use $\delta_1 \ldots \delta_4$ to represent the errors, we have

$((((x_1 \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) = x_1(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4) + x_2(1+\delta_1)(1+\delta_2)(1+\delta_3)(1+\delta_4) + x_3(1+\delta_2)(1+\delta_3)(1+\delta_4) + x_4(1+\delta_3)(1+\delta_4) + x_5(1+\delta_4)$

Using the standard notation of [15] we have

$$\prod_{i=1}^{n}(1+\delta_i) = (1+\theta_n), \quad \text{with } |\delta_i| \leq u.$$

We can now bound θ_n. For example, assuming $0 \leq x_i, i = 1, \ldots, n$ we obtain

$$\sum_{\oplus i=1}^{n} x_i \leq x_1\,(1+\theta_{n-1}) + x_2\,(1+\theta_{n-1}) + x_3\,(1+\theta_{n-2}) + \ldots + x_n\,(1+\theta_1)\,.$$

x_1 and x_2 both have $(1+\theta_{n-1})$ as their error factor because they are equally involved in the innermost addition, after which the error factor decreases by a factor of $(1+\delta)$ for each subsequent variable added, indicating how many additions it was involved in. It is important to note that multiple $(1+\theta_x)$ may represent distinct sets of δ_i

and are not necessarily equal to each other. Equal $(1+\theta_x)$ cannot, in general, be factored out for different terms in an equation. As we will show, one cannot even assume some combination of δ_i can be found that would allow this factoring to take place. In short, the $(1+\theta_x)$ are not multiplicatively distributive. However, these terms can be combined and manipulated in the following way (for details and proofs see [15]): A floating point multiply of two numbers with pre-existing error adds the exponents of the errors together, and multiplies the error factor by $(1+\delta)$. So from the definitions we have: $x_1(1+\theta_p) \odot x_2(1+\theta_m) = (x_1 \cdot x_2)(1+\theta_{p+m+1})$. Critically, however, $x_1(1+\theta_p) \oplus x_2(1+\theta_m) = (x_1+x_2)\left(1+\theta_{\max(p,m)+1}\right)$. This is the key observation that leads to algorithmic reduction in error: if we can evenly balance the size of p and m during addition, we can minimize the resulting error. Conversely, if $m=1$ and $p=i$, as in canonical summation (where i is the induction variable), then error growth is maximized.

This paper is concerned only with algorithmic improvements for controlling error. A related and orthogonal approach is using extra and/or mixed precision arithmetic to reduce error. Since this approach is not the focus of this paper, we do not cover this area in detail, but [6, 7, 18] provide some information. The broader effects of floating point error in linear algebra is also too large a pool of literature to survey in detail, but more information can be found in the overview texts [24, 15], and in [5, 16].

1.2. Outline. The remainder of this paper is organized in the following way: §2 discusses bounds for the $(1+\theta_n)$ products that these equations require for simplification, including a tighter bound than we have found in the literature, while §3 introduces a more concise notation that we will use in our later proofs, which can be used to track error more precisely than the standard $(1+\theta_n)$ terms. Section 4 surveys some known dot product implementations, including their error bounds, while §5 introduces a general class of dot product algorithms we call superblock which we believe is new. Section 6 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 §7.

2. Bounding the Floating Point Error Terms. Lemma 2.1 (pg 69 of [15]) is a bound on the term $(1+\theta_n) = \prod_{i=1}^{n}(1+\delta)^{\rho_i}$, with each $\rho_i \in \{-1,+1\}$ and $|\delta| \le u$.

LEMMA 2.1.

$$|\delta_i| \le u, \rho_i = \pm 1, nu < 1, \quad \prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1+\theta_n; \quad |\theta_n| \le \frac{nu}{1-nu} =: \gamma_n. \tag{2.1}$$

Noting that equality holds for $|\theta_1|$.

Since $1+\delta$ can be >1, a useful bound requires an additional limiting factor. If n is the number of flops applied to a variable, we assume $n \cdot u < 1$. We emphasize n is the limit of the number of flops needed to compute a *single result*; so for IEEE single precision this limits us to about 2^{23} flops.

The above error bound is not tight, and in fact we can produce a tighter bound by tracking the specific amount of floating point error in terms of the $(1+\theta_n)$ factors. First, we are given $\delta \in [-u,+u]$ and $\rho \in \{-1,+1\}$, thus $(1+\theta_n) = \prod_{i=1}^{n}(1+\delta_i)^{\rho_i}$ has a minimum value of $(1-u)^n$ and a maximum value of $\left(\frac{1}{1-u}\right)^n$. Thus, observe that (i) $\min_{\delta,\rho} \prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = \prod_{i=1}^{n} \min_{\delta_i,\rho_i} (1+\delta_i)^{\rho_i} = \prod_{i=1}^{n}(1-u) = (1-u)^n$ and

(ii) $\max\limits_{\delta,\rho} \prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = \prod_{i=1}^{n} \max\limits_{\delta_i,\rho_i} (1+\delta_i)^{\rho_i} = \prod_{i=1}^{n} \left(\frac{1}{1-u}\right) = \left(\frac{1}{1-u}\right)^n$. These observations are used in the proof of Lemma 2.2.

LEMMA 2.2. *We can bound θ_n as follows:*

$$|\theta_n| \leq \frac{1-(1-u)^n}{(1-u)^n}$$

Proof.

$(1-u)^n \leq (1+\theta_n) \leq \left(\frac{1}{1-u}\right)^n \Rightarrow (1-u)^n - 1 \leq \theta_n \leq \left(\frac{1}{1-u}\right)^n - 1$

$\Rightarrow |\theta_n| \leq \max\left(|(1-u)^n - 1|, \left|\left(\frac{1}{1-u}\right)^n - 1\right|\right)$

Since $(1-u) < 1$, $|(1-u)^n - 1| = 1-(1-u)^n$, and since $\left(\frac{1}{1-u}\right) > 1$:

$\left|\left(\frac{1}{1-u}\right)^n - 1\right| = \left(\frac{1}{1-u}\right)^n - 1 = \frac{1}{(1-u)^n} - \frac{(1-u)^n}{(1-u)^n} = \frac{1-(1-u)^n}{(1-u)^n} > 1-(1-u)^n$.

The final inequality holds because the divisor $(1-u)^n < 1$. Thus, we obtain

$$\max\left(|(1-u)^n - 1|, \left|\left(\frac{1}{1-u}\right)^n - 1\right|\right) = \frac{1-(1-u)^n}{(1-u)^n}.$$

□

We now prove this bound is tighter than Lemma 2.1 $\forall n > 1$.

PROPOSITION 2.3.

$$\forall n > 1, \frac{1-(1-u)^n}{(1-u)^n} < \frac{nu}{1-nu}$$

Proof. To prove Proposition 2.3 we use an inductive proof by contradiction, where we first show by contradiction that the new bound is not greater than or equal to that of Lemma 2.1, for $n = 2$, and then use induction to show it is false $\forall n > 2$. Obviously, once we have proved that the new bound is never greater than or equal, it must be strictly less than, and Proposition 2.3 is proved. To help with this proof, (2.2) shows the reverse proposition which will be contradicted $\forall n > 1$.

Proposition to be disproved:

$$\frac{1-(1-u)^n}{(1-u)^n} \geq \frac{nu}{1-nu} \Rightarrow \frac{1}{(1-u)^n} - 1 \geq \frac{nu}{1-nu} \Rightarrow \tag{2.2a}$$

$$\frac{1}{(1-u)^n} \geq \frac{(1-nu)+nu}{1-nu} \Rightarrow \frac{1}{(1-u)^n} \geq \frac{1}{1-nu} \Rightarrow (1-u)^n \leq 1-nu \tag{2.2b}$$

We now disprove (2.2) by induction. For $n = 2$, (2.2b) reduces to

$$(1-u)^2 \leq 1-2u \Rightarrow 1-2u+u^2 \leq 1-2u \Rightarrow u^2 \leq 0,$$

a contradiction. So this inequality does not hold for $n = 2$. Now suppose it is true for some first integer $n+1 > 2$. If true, we have

$$(1-u)^{n+1} \leq 1-(n+1)u \Rightarrow (1-u)(1-u)^n \leq 1-nu-u.$$

Since this $(n+1)$ case is the *first* for which (2.2b) holds true, it was false for the prior (n) case, which implies $(1-u)^n > 1-nu$. Therefore, our inequality keeps the same direction and becomes strict if $1-nu$ on the right is replaced by $(1-u)^n$:

$$(1-u)(1-u)^n < (1-u)^n - u \Rightarrow (1-u)^n - u(1-u)^n < (1-u)^n - u \Rightarrow -u(1-u)^n < -u.$$

Dividing both sides by $-u$ reverses the inequality, resulting in

$$u(1-u)^n > u \Rightarrow (1-u)^n > 1.$$

But this is a contradiction since $0 < u < 1$ and $n > 1$, so the proof is complete. □

3. The Error Product Notation. Although the $(1+\theta_n)$ notation is useful for straightforward analysis, it becomes unwieldy when large numbers of terms are involved and we must track the sources of the error factors. To facilitate such analysis we introduce a notation similar to that of Stewart's bracket notation, <*k*>[24]. We denote by $\widehat{\Pi}_j^a$ the product $\prod_{i=1}^{j}(1+\delta_{a_i})^{\rho_{a_i}}$, where the superscript a is used to indicate a particular collection of δ_{a_i} and ρ_{a_i}, with i running from 1 to j. If no superscript is present, then $\widehat{\Pi}_j$ refers to a general collection of δ_i and ρ_i with i running from 1 to j. To understand the motivation for distinguishing between generic and specific sets, consider this hypothetical series of operations: We add x to y in floating point: $z = x \oplus y = (x+y)(1+\delta_1)$, then expand the right hand side: $z = x(1+\delta_1)+y(1+\delta_1)$, and subtract the y term from both sides in exact arithmetic, $z - y(1+\delta_1) = x(1+\delta_1)$.

For some analyses (in particular backward error analyses) we require terms on the right to be represented without error. To accomplish this here, we must divide both sides by $(1+\delta_1)$. But division of error terms presents a unique problem: we know the $(1+\delta_1)$ is identical on both sides because we distributed an original single error value to two terms. So we know it would cancel out, and we could write $\frac{z}{(1+\delta_1)} - y = x$. But during analysis, in order to make computations tractable, we often generalize in ways that lose track of the individual δ. In those cases we can no longer divide: If all we know is that $|\delta| \leq u$, then by definition $\frac{1+\delta}{1+\delta} = \widehat{\Pi}_2$. To overcome this problem, a superscripted $\widehat{\Pi}_j^a$ is used to represent a particular combination of j δ's and ρ's named "a". This allows us to cancel them by division; *i.e.* $\frac{\widehat{\Pi}_j^a}{\widehat{\Pi}_j^a} = 1$.

We use $\widehat{\Pi}_j$ to represent j terms of the form $\prod_{i=1}^{j}(1+\delta_i)^{\rho_i}$ where for the purpose of simplification we may have purposely lost track of the exact values of the δ_i and ρ_i. Given $\widehat{\Pi}_j$, all we know is that all $|\delta_i| \leq u$ and each $\rho_i \in \{-1, +1\}$.

When collections are unnamed, we must presume such collections to be a unique set of δ's and ρ's, and we cannot perform any simplifications. Under these circumstances, division has a different result: $\frac{\widehat{\Pi}_j}{\widehat{\Pi}_j} = \widehat{\Pi}_{2 \cdot j}$. Since we cannot know whether there are any cancellations, we must follow the definition. Expanding the fraction by the definition yields

$$\frac{\prod_{i_1=1}^{j}(1+\delta_{i_1})^{\rho_{i_1}}}{\prod_{i_2=1}^{j}(1+\delta_{i_2})^{\rho_{i_2}}} = \prod_{i_1=1}^{j}(1+\delta_{i_1})^{\rho_{i_1}} \cdot \prod_{i_2=1}^{j}(1+\delta_{i_2})^{-\rho_{i_2}} = \widehat{\Pi}_{2 \cdot j}.$$

Superscripts can be used to track composition of a product in the superscript, just as we track the number of entries in the subscript: $(\widehat{\Pi}_j^a)(\widehat{\Pi}_k^b) = \widehat{\Pi}_{j+k}^{a,b}$. This allows a concise notation with the possibility of regrouping or simplifying parts of the error later in the development of an equation. We next state and prove a bound on the $\widehat{\Pi}_j$ error terms.

PROPOSITION 3.1. *Using the terminology introduced above, we obtain the following bound on the error product terms:* $\left|\widehat{\Pi}_j - 1\right| \leq \frac{1-(1-u)^j}{(1-u)^j} \leq \frac{ju}{1-ju} =: \gamma_j$

Proof. Using the definition and our previous results, we can bound $\widehat{\Pi}_j$

$$(1-u)^j \leq \widehat{\Pi}_j \leq \left(\frac{1}{1-u}\right)^j \Rightarrow$$

$$(1-u)^j - 1 \leq \widehat{\Pi}_j - 1 \leq \left(\frac{1}{1-u}\right)^j - 1 \Rightarrow \left|\widehat{\Pi}_j - 1\right| \leq \frac{1-(1-u)^j}{(1-u)^j} \leq \frac{ju}{1-ju}$$

Note that Proposition 2.3 makes the inequality with γ_j strict $\forall j > 1$. □

$\widehat{\Pi}_j$ is a direct replacement for Higham's $(1+\theta_j)$. Thus Higham's bounds, theories and Lemmas that apply to $(1+\theta_j)$ apply equally to $\widehat{\Pi}_j$ because of (2.1), and our $\widehat{\Pi}$ notation can be easily translated to and from θ notation via (3.1):

$$\widehat{\Pi}_n = 1 + \theta_n \qquad \Longleftrightarrow \qquad \theta_n = \widehat{\Pi}_n - 1 \tag{3.1}$$

REMARK 3.2. *This notation implies the following relations in exact arithmetic, where z_1, z_2 are scalars:*

(i) $z_1\widehat{\Pi}_j \cdot z_2\widehat{\Pi}_k = (z_1 \cdot z_2)\widehat{\Pi}_{j+k}$

(ii) $\frac{z_1\widehat{\Pi}_j}{z_2\widehat{\Pi}_k} = \frac{z_1}{z_2}\widehat{\Pi}_{j+k}$

These multiplication and division results follow directly from the definition.

CLAIM 3.3. *In general, $\widehat{\Pi}_j$ is <u>not</u> distributive over addition i.e.*

$$z_1\widehat{\Pi}_j \pm z_2\widehat{\Pi}_j \neq (z_1 \pm z_2)\widehat{\Pi}_j. \tag{3.2}$$

Proof. If this relation were true, it would require that there exist a <u>single</u> error factor $\widehat{\Pi}_j$ that can be applied to the sum to produce the same amount of error as the two independent $\widehat{\Pi}_j$. This is not the case. Suppose $0 < z_2 < z_1$ and the operation is subtraction. Then 3.2 implies $\exists$ a $\widehat{\Pi}_j^c$ such that

$$z_1\widehat{\Pi}_j^a - z_2\widehat{\Pi}_j^b = (z_1 - z_2)\widehat{\Pi}_j^c \tag{3.3}$$

Where we have named the various $\widehat{\Pi}_j$ to help track them and indicate independence. The left side of 3.3 varies with the error factors; its minimum occurs when $\widehat{\Pi}_j^a$ is at its minimum and $\widehat{\Pi}_j^b$ is at its maximum. Using our previous observations this yields

$$\min\left(z_1\widehat{\Pi}_j^a - z_2\widehat{\Pi}_j^b\right) = z_1(1-u)^j - z_2\left(\frac{1}{(1-u)^j}\right).$$

Our assumption implies $\exists$ a $\widehat{\Pi}_j^c$ that satisfies 3.3. The minimum value of $\widehat{\Pi}_j^c$ is $(1-u)^j$, implying

$$z_1(1-u)^j - z_2\left(\tfrac{1}{(1-u)^j}\right) \geq (z_1 - z_2)\cdot(1-u)^j = z_1(1-u)^j - z_2(1-u)^j \Rightarrow$$
$$-z_2\left(\tfrac{1}{(1-u)^j}\right) \geq -z_2(1-u)^j \Rightarrow \tfrac{1}{(1-u)} \leq (1-u) \Rightarrow 1 \leq (1-u)^2$$

a contradiction since $0 < u < 1$. Thus $\widehat{\Pi}_j$ is not distributive over addition. □

The problem is the mixing of signs which creates a mix of min and max values for $\widehat{\Pi}_j$, and the non-specificity of $\widehat{\Pi}_j$. If we have more information about the operands or error factors we can be more precise. For example, $z_1\widehat{\Pi}_j^a \pm z_2\widehat{\Pi}_j^a = (z_1 \pm z_2)\widehat{\Pi}_j^a$ as expected; because in this case $\widehat{\Pi}_j^a$ represents the same error factor. If scalars $z_1, z_2, \ldots z_k$ are all the same sign we can prove the following:

PROPOSITION 3.4. *Given same-signed values* $\{z_i\}_{i=1}^k$ *then* $\exists$ *a* $\widehat{\Pi}_j$ *such that* $\sum_{i=1}^k z_1\widehat{\Pi}_j^{a_i} = \widehat{\Pi}_j \sum_{i=1}^k z_i$.

Proof. The proof is by induction. Suppose $k = 2$ and we have $z = z_1\widehat{\Pi}_j^{a_1} + z_2\widehat{\Pi}_j^{a_2}$, with $\widehat{\Pi}_j^{a_1}$ and $\widehat{\Pi}_j^{a_2}$ independent, and $z_1, z_2 \geq 0$. Then

$$\min_{\delta,\ \rho} z = \min_{\delta,\ \rho}(z_1\widehat{\Pi}_j^{a_1} + z_2\widehat{\Pi}_j^{a_2}) = \min_{\delta,\ \rho}(z_1\widehat{\Pi}_j^{a_1}) + \min_{\delta,\ \rho}(z_2\widehat{\Pi}_j^{a_2}).$$
$$\text{Now } z_1 \geq 0 \Longrightarrow \min_{\delta,\ \rho} z_1\widehat{\Pi}_j^{a_1} = z_1 \cdot \min_{\delta,\ \rho} \widehat{\Pi}_j^{a_1},$$

and a similar result holds for z_2. Thus,

$$\min_{\delta,\ \rho} z = z_1 \cdot \min_{\delta,\ \rho} \widehat{\Pi}_j^{a_1} + z_2 \cdot \min_{\delta,\ \rho} \widehat{\Pi}_j^{a_2}, \text{ and since}$$
$$\min_{\delta,\ \rho} \widehat{\Pi}_j^{a_1} = \min_{\delta,\ \rho} \widehat{\Pi}_j^{a_2} = (1-u)^j, \text{ we have}$$
$$\min_{\delta,\ \rho} z = z_1(1-u)^j + z_2(1-u)^j = (z_1 + z_2)(1-u)^j.$$

The maximums work similarly, letting us conclude

$$z \in (z_1 + z_2)\cdot\left[(1-u)^j,\ \frac{1}{(1-u)^j}\right].$$

This is precisely the range of $\widehat{\Pi}_j$, and since $\widehat{\Pi}_j$ is continuous on its range, $\exists$ a $\widehat{\Pi}_j^c$ such that $z = \widehat{\Pi}_j^c(z_1 + z_2)$. Then the induction step is trivial; assuming this is true for n terms, for $n+1$ terms we combine the first n terms to create this $k = 2$ case. The proof for all $\{z_i\}_{i=1}^k < 0$ proceeds similarly. □

4. Known Dot Products. In this section we overview 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 $O(N^3)$ flops and $O(N^2)$ 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 additional flops (eg., compensated summation). For performance reasons, we also avoid sorting the vectors of each individual dot product. 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 4.1 discusses canonical dot product, §4.2 surveys two versions of the blocked dot product, and §4.3 presents pairwise dot product.

4.1. 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

$$((\ldots(((x_1 \odot y_1) \oplus x_2 \odot y_2) \oplus x_3 \odot y_3) \oplus \ldots) \oplus x_n \odot y_n)$$

In [15] and much of the summation literature, this method is called the recursive algorithm. Since the pairwise algorithm (surveyed in §4.3) 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*. We repeat the bound for the forward error presented on page 69 of [15] as Proposition 4.1.

PROPOSITION 4.1. *Let* $s = \mathbf{y}^\mathsf{T}\mathbf{x}$ *and* $\hat{s} = \mathbf{y}^\mathsf{T} \odot \mathbf{x}$. *Then* $\exists$ *a* $\widehat{\Pi}_n$ *such that*

$$|s - \hat{s}| \leq (|\widehat{\Pi}_n - 1|)|\mathbf{x}|^\mathsf{T}|\mathbf{y}| \leq \gamma_n|\mathbf{x}|^\mathsf{T}|\mathbf{y}|. \tag{4.1}$$

Proof. We present this algorithm in more than usual detail to familiarize the reader with a standard analysis using the new notation. We first account for the error introduced by floating point multiplication, replacing $x_i \odot y_i$ with $x_i \cdot y_i\widehat{\Pi}_1$, yielding:

$$((\ldots(((x_1 \cdot y_1\widehat{\Pi}_1) \oplus x_2 \cdot y_2\widehat{\Pi}_1) \oplus x_3 \cdot y_3\widehat{\Pi}_1) \oplus \ldots) \oplus x_n \cdot y_n\widehat{\Pi}_1)$$

We replace $\oplus$ with $+$, adding one to each bracket on each side at each step. We start with the *final* $\oplus$ and work our way inward:

$$\begin{gathered}
(\ldots(x_1 \cdot y_1\widehat{\Pi}_2) \oplus x_2 \cdot y_2\widehat{\Pi}_2) \oplus x_3 \cdot y_3\widehat{\Pi}_2) \oplus \ldots) \oplus x_{n-1} \cdot y_{n-1}\widehat{\Pi}_2) + x_n \cdot y_n\widehat{\Pi}_2) \Leftrightarrow \\
(\ldots(x_1 \cdot y_1\widehat{\Pi}_3) \oplus x_2 \cdot y_2\widehat{\Pi}_3) \oplus x_3 \cdot y_3\widehat{\Pi}_3) \oplus \ldots) + x_{n-1} \cdot y_{n-1}\widehat{\Pi}_3 + x_n \cdot y_n\widehat{\Pi}_2) \Leftrightarrow \\
(\ldots(x_1 \cdot y_1\widehat{\Pi}_{n-2}) \oplus x_2 \cdot y_2\widehat{\Pi}_{n-2}) \oplus x_3 \cdot y_3\widehat{\Pi}_{n-2}) + \ldots) + \\
x_{n-1} \cdot y_{n-1}\widehat{\Pi}_3 + x_n \cdot y_n\widehat{\Pi}_2) \Leftrightarrow \\
\vdots \quad \text{and so forth until we obtain} \quad \vdots
\end{gathered}$$

$$(x_1 \cdot y_1\widehat{\Pi}_n + x_2 \cdot y_2\widehat{\Pi}_n + x_3 \cdot y_3\widehat{\Pi}_{n-1} + \ldots + x_n \cdot y_n\widehat{\Pi}_2) \tag{4.2}$$

The worst potential relative error in this equation is on the terms involving $\widehat{\Pi}_n$. We cannot know which terms actually have the worst error, but note if $a \leq b$, then $\forall\ \widehat{\Pi}_a\ \exists$ a $\widehat{\Pi}_b = \widehat{\Pi}_a$. So, remembering our rule that all unnamed collections must be presumed to be unique, we can replace all of the $\widehat{\Pi}_i$ in the above equation with $\widehat{\Pi}_n$:

$$\mathbf{y}^\mathsf{T} \odot \mathbf{x} = \left(x_1 \cdot y_1\widehat{\Pi}_n + x_2 \cdot y_2\widehat{\Pi}_n + x_3 \cdot y_3\widehat{\Pi}_n + \ldots + x_n \cdot y_n\widehat{\Pi}_n\right)$$

To get the standard forward error we subtract the exact answer as shown in (4.3)

(4.3) $\mathbf{y}^\mathsf{T} \odot \mathbf{x} - \mathbf{y}^\mathsf{T}\mathbf{x} = x_1 \cdot y_1(\widehat{\Pi}_n - 1) + x_2 \cdot y_2(\widehat{\Pi}_n^{'} - 1) + \ldots + x_n \cdot y_n(\widehat{\Pi}_n - 1))$

But $(\widehat{\Pi}_n - 1)$ is not distributive over addition for differently signed elements, so to simplify further we must force identical signs on all the $x_i \cdot y_i$ terms, which we accomplish with the absolute value function.

$$|\mathbf{y}^\mathsf{T} \odot \mathbf{x} - \mathbf{y}^\mathsf{T}\mathbf{x}| = |x_1 \cdot y_1(\widehat{\Pi}_n - 1) + x_2 \cdot y_2(\widehat{\Pi}_n^{'} - 1) + \ldots + x_n \cdot y_n(\widehat{\Pi}_n - 1)|$$

Applying the Triangle Inequality yields

$$|\mathbf{y}^\mathsf{T} \odot \mathbf{x} - \mathbf{y}^\mathsf{T}\mathbf{x}| \leq |x_1 \cdot y_1||(\widehat{\Pi}_n - 1)| + |x_2 \cdot y_2||(\widehat{\Pi}_n^{'} - 1)| + \ldots + |x_n \cdot y_n||(\widehat{\Pi}_n - 1)|$$

Now we apply Proposition 3.4 to find $\exists$ a $\widehat{\Pi}_n$ such that:

$$|\mathbf{y}^\mathsf{T} \odot \mathbf{x} - \mathbf{y}^\mathsf{T}\mathbf{x}| \leq |(\widehat{\Pi}_n - 1)|\mathbf{y}^\mathsf{T} \odot \mathbf{x} \Rightarrow |s - \hat{s}| \leq |(\widehat{\Pi}_n - 1)|\mathbf{y}^\mathsf{T} \odot \mathbf{x}$$

Since by proposition 3.1 $|\widehat{\Pi}_n - 1| \leq \gamma_n$, the proof is complete.

□

Without knowledge of the signs or magnitudes of the vector elements, we see that the worst-case error will therefore depend on the greatest number of flops to which any given input is exposed. 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.

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.); some of these optimizations (eg. scalar expansion on the accumulator) can change the error behavior slightly, but not enough to worry about here. However, canonical dot product results in vector performance (which is often an order of magnitude slower than cache-blocked matrix performance), when building a matrix multiply and so canonical dot product is almost never used to directly build a high performance GEMM implementation. Instead a blocked version of dot product is used, and so we survey this dot product variant next.

4.2. 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, then added together. There are two obvious algorithms for blocked dot product, which we call *pre-load* and *post-load*; we show that post-load is strongly preferable to the pre-load due to error growth. Figure 4.1 gives pseudo-code for both versions of the algorithm (we assume N is a multiple of N_b for expositional simplicity throughout this section).

The pre-load algorithm of Figure 4.1(a) is probably the most obvious implementation. However, it is not optimal error wise. 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 doesn't cause error. So there are $N - 1$ adds to s, along with the $\widehat{\Pi}_1$ error of the multiply, making the forward error exactly the same as the canonical algorithm error bound.

```
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) Pre-load 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) Post-load blocked dot product

FIG. 4.1. *Pseudo-code for blocked dot products*

Now consider a slight alteration to this algorithm, as shown in Figure 4.1(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. The forward error bound of the post-load algorithm is stated in (4.4), and we prove it in Proposition 4.2:

PROPOSITION 4.2. *If* $s = \mathbf{y}^\mathsf{T}\mathbf{x}$ *and* $\hat{s} = \mathbf{y}^\mathsf{T} \odot \mathbf{x}$ *then* $\exists$ *a* $\widehat{\Pi}_{N_b+\frac{N}{N_b}-1}$ *such that the forward error of post-load dot product is*

$$|s - \widehat{s}| \leq (|\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} - 1|)|\mathbf{x}|^\mathsf{T}|\mathbf{y}| \leq \gamma_{N_b+\frac{N}{N_b}-1}|\mathbf{x}|^\mathsf{T}|\mathbf{y}| \tag{4.4}$$

Proof. Let $\mathbf{x_b}$ and $\mathbf{y_b}$ be the subsections of $\mathbf{x}$ and $\mathbf{y}$ operated on in the inner loop of Figure 4.1(b). These are clearly canonical dot products of size N_b, which we have shown produce an error of $\widehat{\Pi}_{N_b}$. Assigning each such subsection the value $s_1, s_2, \ldots s_{\frac{N}{N_b}}$, we have

$$\hat{s} = s_1\widehat{\Pi}_{N_b} \oplus s_2\widehat{\Pi}_{N_b} \oplus \ldots \oplus s_{\frac{N}{N_b}}\widehat{\Pi}_{N_b},$$

Which by straightforward analysis similar to canonical dot product yields

$$\hat{s} = s_1\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} + s_2\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} + \ldots + s_{\frac{N}{N_b}}\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} \Rightarrow$$
$$|s - \hat{s}| = |s_1(\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} - 1) + \ldots + s_{\frac{N}{N_b}}(\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} - 1)$$

Note $|s_1|+|s_2|+\ldots+|s_{\frac{N}{N_b}}| = |s| = |\mathbf{x}|^\mathsf{T}|\mathbf{y}|$. So the above implies by Proposition 3.4 and by applying the Triangle Inequality, $\exists$ a $\widehat{\Pi}_{N_b+\frac{N}{N_b}-1}$ such that

$$|s - \hat{s}| \leq |s_1||\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} - 1| + \ldots + |s_{\frac{N}{N_b}}||\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} - 1| \Rightarrow$$
$$|s - \hat{s}| \leq |\widehat{\Pi}_{N_b+\frac{N}{N_b}-1} - 1||s| \leq \gamma_{N_b+\frac{N}{N_b}-1}|\mathbf{x}|^\mathsf{T}|\mathbf{y}|.$$

□

Therefore, assuming a fixed N_b, post-load reduces error by a constant factor which depends on N_b. In examining (4.4), it is clear that as N grows with a fixed N_b, the number of blocks eventually dominates the error term, which leads to the idea of finding a way to minimize the error from adding the block results together. This is the essential idea behind superblocking, which is discussed in detail in §5. If we look at the error subscript as a function of N_b, say $E(N_b) = N_b + \frac{N}{N_b} - 1$, minimizing

it reduces the order of the error. In particular if $N_b = \sqrt{N}$ then the error becomes $\widehat{\Pi}_{2\sqrt{N}-1}$, as mentioned in [15](pg 70).

Implementation notes. Most high performance GEMM implementations use one of these blocked algorithms, where the N_b is chosen to fit the operands into the cache (which cache level this GEMM kernel fills depends on both the implementation and the architecture). It is perhaps not obvious, but the post-load algorithm requires no extra storage when extended to GEMM. In GEMM, dot's scalar accumulators 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 pre-load, the summation-so-far is loaded before beginning the loop indexing the common dimension of the input matrices, but in post-load the summation is not loaded until that loop is complete. Therefore, whether the pre-load or post-load algorithm is used varies by library (ATLAS mostly uses the pre-load algorithm at present); indirect experience with vendor-supplied BLAS seems to indicate that many use the pre-load 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 post-load, and have at least historically used it.

Since N_b is selected to fill a level of cache, it cannot typically be varied exactly as called for to get the $\widehat{\Pi}_{2\sqrt{n}-1}$ bound; In our own ATLAS (and we suspect in other libraries as well), N_b is allowed to vary to a limited degree, but does so only due to performance considerations. 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 in §4.2.1.

4.2.1. Optimal and Near-Optimal Blocking for Post-Load Dot Product. Post-load blocked dot product has the error given in (4.4). To derive the optimal N_b to use, we treat the error subscript as a function of N_b; $E(N_b) = N_b + \frac{N}{N_b} - 1$. Notice $E(1) = N$ and $E(N) = N$, but $E(2) = \frac{N}{2} + 1 < N$ for any $N > 2$. Thus $E()$ starts out decreasing but eventually increases, when the number of blocks to be added outweighs the block length. To minimize this function we set its first derivative to 0, and solve for N_b:

$$E'(N_b) = 0 \Longrightarrow -N \cdot N_b^{-2} + 1 = 0 \Longrightarrow N \cdot N_b^{-2} = 1 \Longrightarrow N_b = \sqrt{N} \tag{4.5}$$

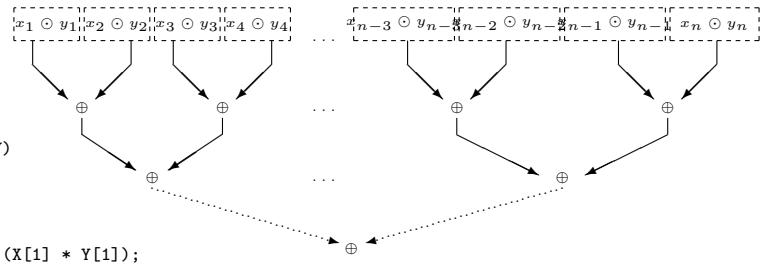
Applying $N_b = \sqrt{N}$ to (4.4) shows the error will now be

$$\widehat{\Pi}_{\left(\sqrt{N}+\frac{N}{\sqrt{N}}-1\right)} = \widehat{\Pi}_{2\sqrt{N}-1}.$$

However, assuming that as discussed in the implementation notes we have only a few fixed block sizes to choose from, can we still get the lower-order error bound for a reasonable range of problem sizes? The answer turns out to be yes: using a slightly non-optimal block factor merely results in a larger coefficient on the square root term. More formally, suppose for some small constant c, $N_b = c \cdot \sqrt{N}$. Then the error factor on each term will be

$$\widehat{\Pi}_{c\cdot\sqrt{N}+\frac{N}{c\cdot\sqrt{N}}-1} = \widehat{\Pi}_{c\cdot\sqrt{N}+\frac{1}{c}\cdot\sqrt{N}-1} = \widehat{\Pi}_{(c+\frac{1}{c})\sqrt{N}-1}.$$

Implementation note. For an intuition of how this could help, consider $c = 2$, and a dot product (or GEMM) with $N_b = 60$. Such an algorithm would achieve a worst-case error of $\widehat{\Pi}_{2.5\sqrt{N}-1}, \forall\{N : 900 \leq N \leq 14,400\}$. Thus, a few carefully selected block sizes could achieve square root worst-case error across the range of reasonable problem sizes (note that very small N do not have substantial errors, and so their N_b could either be omitted or handled by other optimizations with similar error results as small blockings, such as accumulator expansion). $N_b = 60$ is a midsized L1 cache blocking factor; using moderately smaller blocking factors will typically not have a sharp affect on performance. One could therefore imagine a GEMM implementation that has several smaller block sizes available, and also larger sizes that block for the L2 cache, and thus can produce square root worst case error across all typical problem sizes with minimal loss in performance.




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

(a) Pairwise pseudo-code **(b) Pairwise summation diagram**

FIG. 4.2. *Pseudo-code and flop exposure of pairwise dot product*

4.3. Stability of the Pairwise Dot Product. Finally, we consider the pairwise algorithm, which can be naturally implemented using recursion, as shown in Figure 4.2(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 4.2(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 [15] (p70), where Higham produces an error bound for the forward error for this algorithm which we cite as Lemma 4.3:

LEMMA 4.3. *Let us denote by* $|\mathbf{x}|$, $|\mathbf{y}|$ *the vectors of absolute values of the elements of* $\mathbf{x}$ *and* $\mathbf{y}$. *The following bound hold true on the forward error of the floating point pairwise dot product:*

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

We prove the equivalent error bound in our analysis.

PROPOSITION 4.4. *The following bound holds true on the forward error for the recursive dot product computation:* $\exists$ *a* $\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil}$ *s.t.*

$$|s_n - \widehat{s_n}| \leq |\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil} - 1| \cdot |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}| \leq \gamma_{\lceil 1+\log_2 n \rceil} |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}|. \tag{4.6}$$

Proof. We note that in the canonical algorithm the various pairs of vector element-products are subject to different numbers of additions. However, in this algorithm all

products are subject to the same number of additions, namely, $\lceil \log_2(n) \rceil$ of them. In our first (lowest) level, we form the products

$$x_1 \cdot y_1 \widehat{\Pi}_1 \oplus x_2 \cdot y_2 \widehat{\Pi}_1 \oplus x_3 \cdot y_3 \widehat{\Pi}_1 \oplus x_4 \cdot y_4 \widehat{\Pi}_1 \oplus \ldots \oplus x_{n-1} \cdot y_{n-1} \widehat{\Pi}_1 \oplus x_n \cdot y_n \widehat{\Pi}_1$$

The next level we add neighboring pairs, adding $\widehat{\Pi}_1$ error,

$$(x_1 \cdot y_1 \widehat{\Pi}_2 + x_2 \cdot y_2 \widehat{\Pi}_2) \oplus (x_3 \cdot y_3 \widehat{\Pi}_2 + x_4 \cdot y_4 \widehat{\Pi}_2) \oplus \ldots \oplus (x_{n-1} \cdot y_{n-1} \widehat{\Pi}_2 + x_n \cdot y_n \widehat{\Pi}_2)$$

At the next level we add neighboring pairs of parentheticals, adding $\widehat{\Pi}_1$ error,

$$((x_1 \cdot y_1 \widehat{\Pi}_3 + x_2 \cdot y_2 \widehat{\Pi}_3) + (x_3 \cdot y_3 \widehat{\Pi}_3 + x_4 \cdot y_4 \widehat{\Pi}_3)) \oplus \ldots$$
$$\oplus ((x_{n-3} \cdot y_{n-3} \widehat{\Pi}_3 + x_{n-2} \cdot y_{n-2} \widehat{\Pi}_3 + x_{n-1} \cdot y_{n-1} \widehat{\Pi}_3 + x_n \cdot y_n \widehat{\Pi}_3))$$

and so on, until we reach our final sum after $\lceil \log_2(n) \rceil$ steps. Since we also have to account for the $\odot$ which adds $\widehat{\Pi}_1$ error, our final summation is

$$x_1 \cdot y_1 (\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil}) + x_2 \cdot y_2 (\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil}) + \ldots + x_n \cdot y_n (\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil})$$

As we have done on earlier analyses, we can take absolute values, apply the Triangle Inequality and Proposition 3.4 to find $\exists$ a $\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil}$ such that:

$$|s - \hat{s}| \leq |\widehat{\Pi}_{\lceil 1+\log_2(n) \rceil} - 1| |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}| \leq \gamma_{1+\lceil \log_2(n) \rceil} |\mathbf{x}|^{\mathsf{T}} |\mathbf{y}|. \; \square$$

Pairwise demonstrates an instance where recursion, by distributing usage of prior results uniformly, inherently improves the accuracy 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 §6 that these lower worst-case error algorithms do indeed produce lower actual errors on average versus 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. Further, 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 limits or completely removes the freedom to perform these optimizations, and so it generally is much less optimizable than a loop implementation, even when the recursive overhead is 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. With smarter accumulator management, we may reduce the worskpace requirements to $(1 + log_2(n))$, as mentioned in [15] (which cites earlier work [4]). We will derive a similar result using our superblock algorithm, in §5. 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.

5. Superblocked Dot Product. The idea behind superblock is that given t temporaries, perform the necessary accumulations of dot in t levels, where each level gets the same amount of adds, and the lowest level consists of a dot product with that same number of adds. This generalized superblock is shown in Figure 5.1(a), where for clarity it is assumed that $\sqrt[t]{N}$ is an integer. Note that it is possible with superblock to have each temporary experience a different number of adds, but the best error case

```
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. 5.1. *Pseudo-code for Superblock Algorithms*

is the one shown, where all levels are exposed to roughly the same number of adds. This algorithm therefore has the advantage of distributing the additions as optimally as possible for a given amount of workspace. Additionally, notice that at $t = 1$, this algorithm becomes canonical, for $t = 2$, it becomes post-load blocked, and for $t = log_2(N)$ it becomes space-optimal pairwise (this will be shown in Proposition 5.3). Therefore, all the algorithms surveyed in §4 may be viewed as special cases of the superblock class of dot products.

PROPOSITION 5.1. *For a t temporary superblock summation, the optimal blocking factor is $N^{\frac{1}{t}}$, which will produce an error of $\widehat{\Pi}_{t(N^{\frac{1}{t}}-1)}$.*

Proof. The proof is inductive on t. The proposition is trivially true for $t = 1$. For $t = 2$, suppose we block the lowest level by N_b, then the second level will consist of $\frac{N}{N_b}$ results (for simplicity, we assume N evenly divisible by N_b) each of which has a $\widehat{\Pi}_{N_b-1}$ preexisting error from the level-1 summation. Adding these level-1 blocks requires $\frac{N}{N_b} - 1$ nonzero adds.

We therefore see that the error of a two-level superblock of size N will be a function of N_b, as shown in (5.1a), which we minimize by setting its derivative to zero and solving, resulting in an optimal $N_b = \sqrt{N}$ as shown in (5.1b), which leads to the required error bound, and the basis case is complete.

$$E(N_b) = N_b + \frac{N}{N_b} - 2 \tag{5.1a}$$

$$E'(N_b) = 1 - NN_b^{-2} = 0 \Rightarrow N_b^2 = N \Rightarrow N_b = \sqrt{N} \tag{5.1b}$$

Therefore, assume the proposition is true for t. For $t + 1$ level blocking, we must decide on the lowest level blocking factor, which we will call N_b. This will produce an

error of $\widehat{\Pi}_{N_b-1}$ on each block summation, with $\frac{N}{N_b}$ blocks remaining 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 additional error of $\widehat{\Pi}_{t\left(\left(\frac{N}{N_b}\right)^{\frac{1}{t}}-1\right)}$.

We therefore see that the error of a $t+1$-level superblock of size N will be a function of N_b, as shown in (5.2a), which we minimize by setting its derivative to zero and solving for N_b. as shown in (5.2b).

$$E(N_b) = N_b - 1 + t\left(\left(\frac{N}{N_b}\right)^{\frac{1}{t}} - 1\right) \tag{5.2a}$$

$$E'(N_b) = 1 + t\left(N^{\frac{1}{t}} \times \frac{-1}{t} \times N_b^{\frac{-1}{t}-1}\right) = 0 \Rightarrow 1 = N^{\frac{1}{t}} \times N_b^{\frac{-(t+1)}{t}} \Rightarrow N_b = N^{\frac{1}{t}} \tag{5.2b}$$

Therefore, the remaining blocks to be added by all subsequent levels is $\frac{N}{N_b} = \frac{N}{N^{\frac{1}{t}}} = N^{\frac{t}{t+1}}$, which by our assumption means that all subsequent levels will also be using $N_b = N^{\frac{1}{t}}$, and so this is our optimal N_b $\forall t$. Substituting this N_b in (5.2a) gives the error

$$\widehat{\Pi}_{N^{\frac{1}{t}}-1+t\left(\left(N^{\frac{t}{t+1}}\right)^{\frac{1}{t}}-1\right)} \iff \widehat{\Pi}_{(t+1)(N^{\frac{1}{t+1}}-1)}$$

which is the rest of the required result.

□

PROPOSITION 5.2. *The following bound holds true on the forward error for the t temporary superblock dot product computation:*

$$|s_n - \widehat{s_n}| \leq \left(|\widehat{\Pi}_{t\left(\sqrt[t]{N}-1\right)+1} - 1|\right)|\mathbf{x}|^{\mathsf{T}}|\mathbf{y}| \leq \left(\gamma_{t\left(\sqrt[t]{N}-1\right)}\right)|\mathbf{x}|^{\mathsf{T}}|\mathbf{y}| \tag{5.3}$$

Proof. Superblock dot product differs from summation only in that it has 1 additional error from the multiply, and we note that adding 1 to the result proven in Proposition 5.1 yields (5.3). □

PROPOSITION 5.3. *An $N = 2^t$, t-level superblock dot product is equivalent to the space-efficient pairwise dot product.*

Proof. Applying Proposition 5.2 tells us the error must be

$$\widehat{\Pi}_{t(N^{\frac{1}{t}}-1)+1} = \widehat{\Pi}_{t(2^{\frac{t}{t}}-1)+1} = \widehat{\Pi}_{t+1} = \widehat{\Pi}_{\log_2(N)+1}$$

If we apply this result to vectors $\mathbf{x}$ and $\mathbf{y}$ of length N we will find $\exists$ a $\widehat{\Pi}_{\log_2(N)+1}$ such that:

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

identical to the pairwise result (4.4), accomplished in $t = \log_2(N)$ workspaces. □

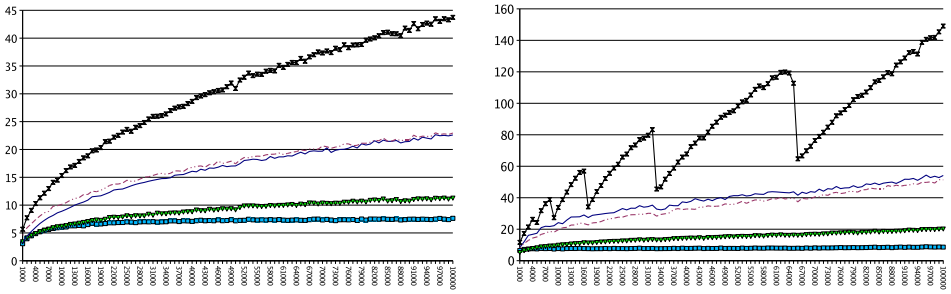
Higham[15] notes that Caprani[4] showed pairwise takes $\lfloor log_2(N) \rfloor + 1$ storage locations, which disagrees with our count of $t = log_2(N)$ (where we assume $\lfloor log_2(N) \rfloor = \lceil log_2(N) \rceil$) as just shown in (5.3). We have been unable to obtain a copy of [4] directly to be sure, but from our own analysis it appears likely the extra storage location for powers of two comes from insisting on an extra storage location to hold the result of an individual `x[i] * y[i]`, which we do not count as storage in our algorithm (given a machine with a multiply and accumulate instruction, no such location is necessary, though registers for loading x[i] and y[i] would be, which neither algorithm accounts for). Therefore, we do not claim to require less storage, despite the differing counts.

5.1. 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 post-load GEMM alone to 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 machines 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 a $N \times N_b$ 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 5.1(b) (for clarity we again assume that all block calculations produce non-zero integral answers). This produces an error of $\widehat{\Pi}_{N_b+2(\sqrt{\frac{N}{N_b}}-1)}$ (note that, as expected, this is the same as (5.3) 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 (§6 puts some statistics behind this belief), without unsupportable workspace requirements or sharp performance reductions.

6. Numerical Experiments. 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 under-estimation, so that the worst-case bound is loose indeed. Many practitioners believe that with these extremely loose bounds and the self-cancelling nature of floating point error, all the algorithms perform fairly indistinguishably for most data. This idea is endorsed in a limited way in [22], 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 (eg., 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). 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 (post-load blocked with $N_b = \sqrt{N}$), and block60 (post-load blocked with $N_b = 60$). These parameter values were chosen due to practical reasons: $N_b = 60$ is a typical midrange blocking



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

FIG. 6.1. *Ratio of average absolute errors of canonical over various algorithms*

factor, and $t = 3$ requires only one additional workspace in GEMM. Therefore, the symbol key to each graph is: pairwise (hourglasses, black) autol3superblock (dashed line, red) l3superblock60 (solid line, dark blue), autoblock (point-down triangle, light green) and block60 (squares, light blue).

For each size, we randomly generate 10,000 different vectors, and each algorithm (including canonical) is run on each of these vectors (so algorithms are always compared using the same data). We believe that there are two main cases of interest in unstructured data, and so we have separate charts for when the elements are generated continuously in the range [-1,1] and [0,1]. In general, the mixed sign vectors have large condition numbers, while the all-positive vectors of course always have a condition number of 1. All of these experiments were run on an x86 machine using the single precision SSE instructions (so that true 32 bit precision is enforced).

Our error results are tracked as absolute errors in terms of u, but if these are charted directly, the fast-rising canonical error makes all other algorithms indistinguishable due to scale. Therefore, all of the charts given here track the ratio of canonical's average absolute error divided by the average absolute error achieved by the improved method. 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. The average is over the 10,000 trial vectors (each algorithm uses the same unique 10,000 vectors for each vector size).

Figure 6.1 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 $N = [1000, 100, 000]$ in steps of 1000, with the problem sizes along X-axis and error ratio along the Y. Figure 6.1(a) shows this chart for mixed-sign vectors, and Figure 6.1(b) shows the same for all-positive vectors. 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 counter-intuitive, 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 grow at each step as it does with same-sign data.

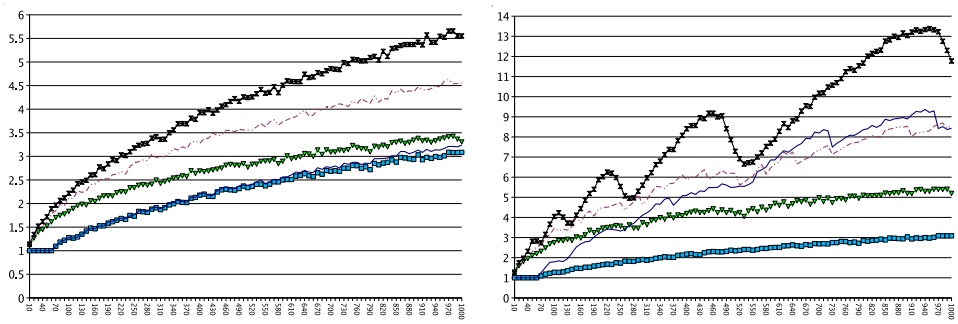
The worst performer of the improved algorithms is always block60, which nonetheless produces 7 (8) times less error on average than canonical for long vectors (mixed and same sign, respectively). It may at first seem surprising how competitive with

autoblock block60 is, since block60 essentially has $O(N)$ error where autoblock has $O(\sqrt{N})$. However, our analysis in 4.2.1 shows that block60 will give $O(\sqrt{N})$ error across much of this range *in the worst case*. Since errors actually build 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, $N < 10,000$, is well within the lower-order range proven in Section 4.2.1). Since this form of GEMM requires no extra storage, this is a strong suggestion that even in the absence of other measures, it makes sense to utilize post-load in modern GEMM algorithms, and that it effectively produces lower order worst-case errors on most problem sizes in use today.

Another surprising result is how much difference separates the l3superblock algorithms from the autoblocked algorithm, since they both have $O(\sqrt{N})$ error. The different 3-level superblock algorithms, as expected, behave almost the same in practice, which indicates that a fixed-N_b (much friendlier to HPC implementation) superblock will be adequate for error control. In mixed sign data, the 3-level superblock algorithms behave as expected: autol3superblock is general 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 worst-case error, which doesn't happen in practice, so using larger N_b should not cause a problem. It may be that $N_b = 60$ results in less alignment error on average when adding the higher level blocks (eg., 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 error-wise (with an error almost 23 (55) times better than canonical, respectively) than post-load 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 across the range. The amount of area between pairwise and l3superblock60 indicates that it may be interesting to study higher level superblocks, and see how the performance/error win tradeoff plays out in practice. We note that the pairwise algorithm displays a sawtooth pattern for the same-sign data, with the least 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, and thus more error is incurred. It seems likely that changing the algorithm to manually rebalance the addition tree could moderate the drops, but since pairwise is not our main focus, we did not investigate this further.

Space considerations rule out detailed presentation of our full experimental results, but we can summarize some interesting points. First, since we presented ratios, we give some context by mentioning that canonical's absolute error does indeed start small, and then rise: for $N = 1000$, its average error was $70.5u$ ($1403u$), while for $N = 100,000$ the average error was $7018u$ ($1,794,144u$) for mixed (same) sign data. We mentioned that worst-case bounds are rarely seen in practice: in fact, for $N = 1000$ canonical, the worst error found in 10,000 trials was merely 0.33% (2.84%) of the Proposition 4.1 bound. In general, the longer the vector, the less % of the bound is achieved (for instance, over the $10,000$ trials, the worst error seen for the $N = 100,000$ case was 0.004% (0.40%) of the Proposition 4.1 bound). These algo-



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

FIG. 6.2. *Ratio of average absolute errors for small problems*

rithms' advantage over canonical is also not due to a one-time win in an uncommon case: in the above experiments for $N = 1000$ to $N = 100,000$, l3superblock60 won or tied vs canonical for 98.75% of all cases, and in those cases where it lost to canonical, l3superblock60 error was still, on average, less than 12.5% (4.1%) of canonical's average error, and never exceeded 82% (23%) of canonical's average error (i.e., the cases where l3superblock60 lost were cases where canonical did abnormally well).

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 6.2 gives the same information for $N = [10, 1000]$ in steps of 10. Here we see less algorithmic difference, as one would expect. For instance, $\forall N \leq 60$, canonical, l3superblock60 and block60 are all 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 build up of error, this is not a drawback in practice.

7. Summary and Conclusions. In summary, we have presented a tighter bound for the forward errors of dot products and summation (§2), a concise notation that retains more information for tracking details in these kinds of error analyses (§3), and a survey of several of the most important known dot products along with their error properties (§4), which includes some discussion on getting lower-order error with a range of block sizes (§4.2.1). 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 (§5). Finally, we have presented a statistical argument that despite the extreme looseness of these worst-case bounds, and the dependence on order inherent in these types of calculations, there is still a significant statistical difference between these algorithms in practice (§6).

Our main conclusion is two-fold. 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 HPC 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, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide, Second Edition*, SIAM, Philadelphia, PA, 1995.

[2] I. J. Anderson, *A distillation algorithm for floating-point summation*, SIAM Journal on Scientific Computing, 20 (1997), 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, PA, 1995.

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

[5] J. Demmel, *Underflow and the reliability of numerical software*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 887–919.

[6] James Demmel and Yozo Hida, *Accurate floating point summation*, Tech. Report UCB/CSD-02-1180, University of California, Berkeley, 2002.

[7] James Demmel, Yozo Hida, William Kahan, Xiaoye S. Li, Sonil Mukherjee, and E. Jason Riedy, *Error bounds from extra-precise iterative refinement*, ACM Trans. Math. Softw., 32 (2006), pp. 325–351.

[8] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 16 (1990), pp. 1–17.

[9] 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. Softw., 14 (1988), pp. 18–32.

[10] ———, *An Extended Set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Softw., 14 (1988), pp. 1–17.

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

[12] James Gregory, *A comparison of floating point summation methods*, Communications of the ACM, 15 (1972), p. 838.

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

[14] R. Hanson, F. Krogh, and C. Lawson, *A Proposal for Standard Linear Algebra Subprograms*, ACM SIGNUM Newsl., 8 (1973).

[15] Nicholas J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, 1996.

[16] Ph. Langlois and N. Louvet, *Solving triangular systems more accurately and efficiently*, Tech. Report N RR2005-02, Laboratoire LP2A, University of Perpignan, 2005.

[17] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308–323.

[18] 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 Transactions on Mathematical Software, 28 (2002), pp. 152–205.

[19] Seppo Linnainmaa, *Analysis of some known methods of improving the accuracy of floating-point sums*, BIT Numerical Mathematics, 14 (1974), pp. 167–202.

[20] Peter Linz, *Accurate floating-point summation*, Communications of the ACM, 13 (1970), pp. 361–362.

[21] John Michael McNamee, *A comparison of methods for accurate summation*, SIGSAM Bull., 38 (2004), pp. 1–7.

[22] Nicholas J. Higham, *The Accuracy of Floating Point Summation*, SIAM Journal on Scientific Computing, 14 (1993).

[23] T. G. Robertazzi and S. C. Schwartz, *Best "ordering" for floating-point addition*, ACM Trans. Math. Softw., 14 (1988), pp. 101–110.

[24] G.W. Stewart, *Introduction To Matrix Computations*, Academic Press, 1973, ch. 2, 3 & Appendix 3.

[25] R. Clint Whaley and Antoine Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, 35 (2005), pp. 101–121. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[26] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35.