A Survey of Multiplier Rounding Techniques on Algorithms with Successive Multiplication

Trey Boehm, Jordan Pamatmat, and Matthew Barondeau

***Abstract*—Multiplication is a fundamental operation of modern computer arithmetic and is especially prominent in digital signal processing and machine learning. In this paper we evaluate different multiplication rounding techniques and their impact on algorithms that have many successive multiplications. We begin with a survey of existing rounding techniques and their logical complements. Then we examine three algorithms consisting primarily of multiplications: general matrix multiplication, the fast Fourier transform, and the discrete cosine transform. We run these algorithms using software implementations of rounding techniques and compare the accumulated errors to a baseline. Finally, we implement RTL versions of 16x16 multipliers with various rounding techniques and compare the implementation results.**

I. Background

Multiplication of two N-bit numbers can result in a product that is $2N$ bits wide. Since most computers work off of fixed sized words, it is often necessary to reduce this product back to a single N-bit value through a rounding technique. Existing rounding techniques offer a variety of error distributions and cases, which we examine in depth in this paper.

A. Rounding types

In our study we primarily focus on evaluating five types of rounding: true rounding, truncation, jamming, alternating between truncation and jamming, and constant correction. We also implemented what we call "clearing" and "stretching," as well as two variants of alternating rounding that use these. We use both true rounding and exact multiplication as baselines as different points in our analysis. In this section, we describe each of the types of rounding.

1) Exact: The first multiplier we consider is one that keeps the full-precision product intact. We call this "exact" since no rounding takes place. At the end of a sequence of successive multiplications, we perform a truncation, removing the lower bits to arrive at a precise result. For the remaining rounding methods, after each multiplication, we perform the specified rounding method. The implementation is simply a standard array multiplier as shown in Figure 1.

2) True Rounding: True rounding works by adding a 1 at the N^{th} least significant bit of the product in a $N \times N$ multiplication. For example, the N^{th} least significant bit in Figure 2 is P7. If the bottom half of the product is greater than the halfway value, adding a 1 will create a carry into bit P8. If this value is below half however, the addition of a 1 will not generate a carry and P8 will remain unmodified. This rounding technique requires substituting a full adder for a half adder in the bottom right corner of the array multiplier.

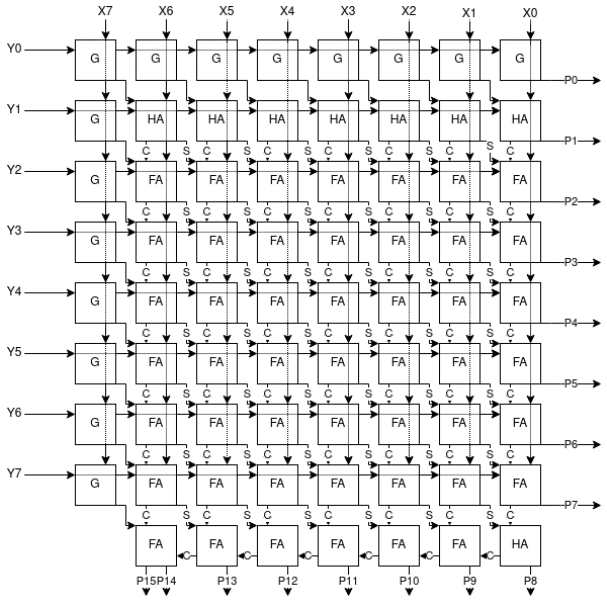

Fig. 1. Array multiplier with no rounding.

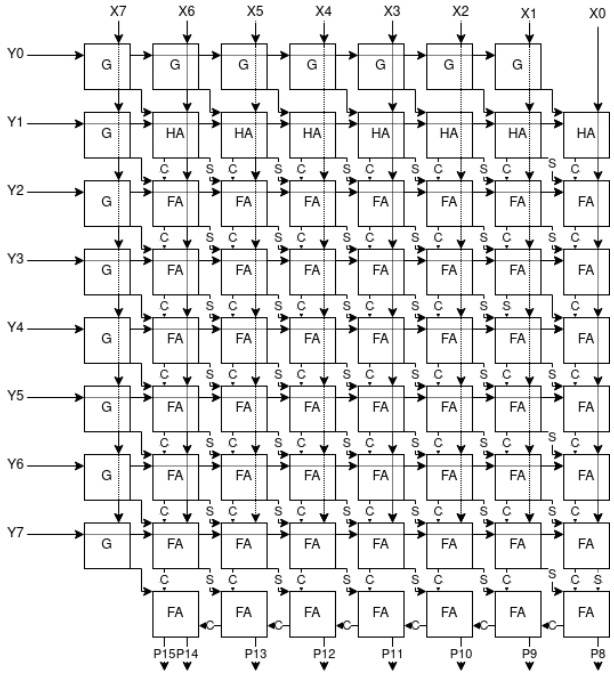

Fig. 2. Array multiplier with true rounding.

3) Truncation and Stretching: Truncation is performed by discarding the least significant half of the product and only using the upper half. To implement truncation, we remove the gate that generates the least significant product bit and discard the outputs of adders for the bottom half of the product. In terms of complexity, the result is slightly simpler than the baseline exact algorithm as we remove one gate. An example array multiplier with truncation rounding is in Figure 3.

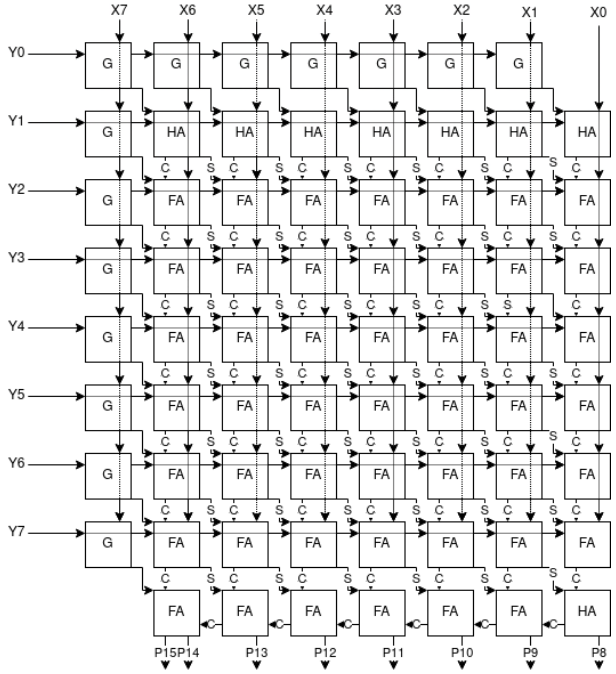

Fig. 3. Array multiplier with truncation rounding.

The opposite of truncated multiplication is one which produces an exactly complementary error. We call this "stretching" since it might stretch the rounded product to a larger value (unlike truncation, which always makes the rounded product smaller). We implement this by ignoring the least significant half of the product and adding a 1. This is effectively true rounding where the round bit is always a 1.

4) Jamming and Clearing: Jamming is performed by dropping the least significant half of the product and then setting the least significant bit of the upper half of the product to one. Similar to truncation, we remove the gate that generates the least significant product bit and discards the outputs of adders for the bottom half of the product. It is exactly the same in terms of complexity as truncation rounding. An example array multiplier with truncation rounding is in Figure 4.

Clearing is a complement to jamming that forces the LSB to be 0 instead of 1. This mode of rounding performs quite poorly, but we include it for the sake of completeness in our comparison of alternating multiplications.

5) Alternating: Alternating rounding is a combination of jamming and truncation in which the rounding mode that is selected is determined by a flip flop. After each multiplication the rounding mode switches between alternating and truncation depending on the value in the flip-flop. As truncation, on average, is slightly lower than the final result

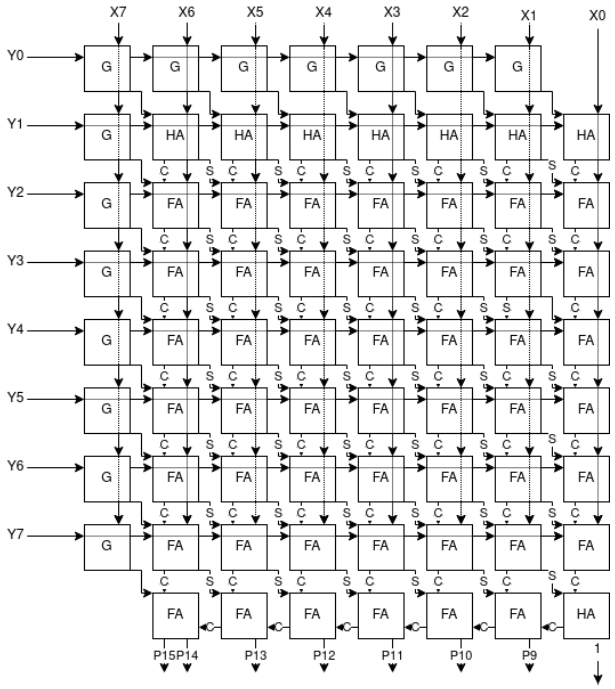

Fig. 4. Array multiplier with jamming rounding.

and jamming is slightly larger than the exact result, we expect this to cancel out some of the error introduced by the two rounding methods. We also study two variants of alternating rounding: one that alternates between truncation and stretching and another that alternates between jamming and clearing. An example array multiplier with alternating rounding between clearing and jamming is shown in Figure 5.

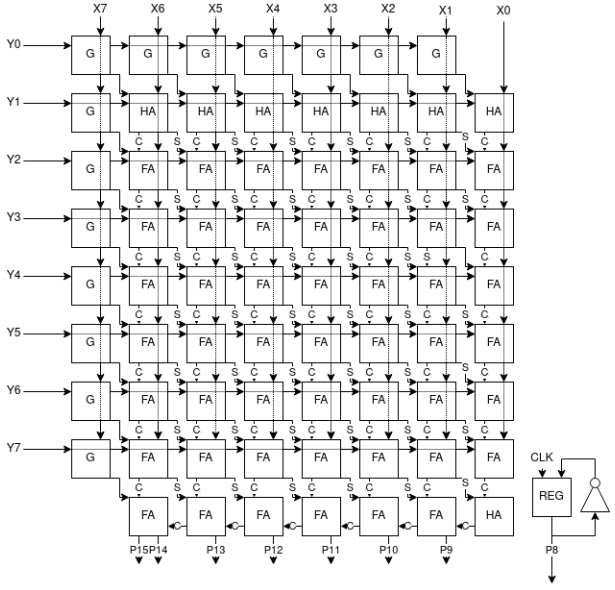

Fig. 5. Array multiplier with alternating rounding.

6) Constant Correction: The final mode of rounding that we evaluate is constant correction, which does not form the least significant n columns of the bit product matrix. Instead, this method adds a constant to the remaining columns. This

constant is derived by the approximation that, on average, bits are equally likely to be a 0 or a 1, and thus their approximate value is 1/4. Thus instead of computing these values, we add this correction factor to the product and then apply true rounding. Compared to the other methods, constant correction has a significantly lower area as we do not compute the least significant bits in the bit product matrix. An example based on an array multiplier is shown in Figure 6.

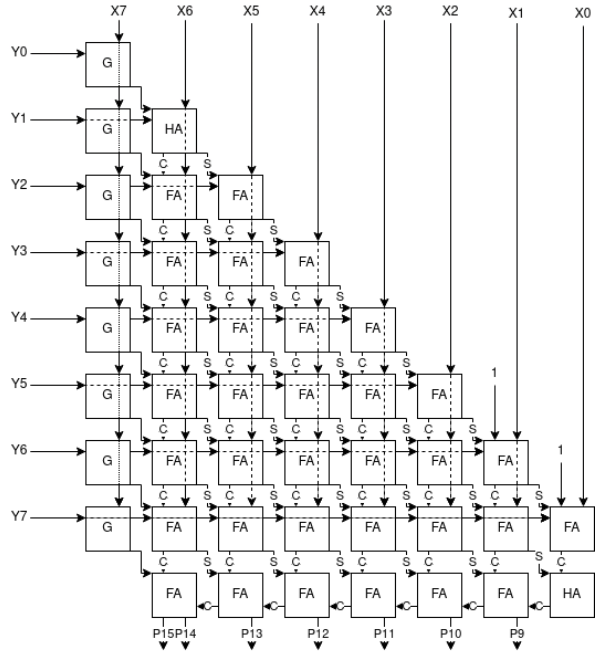

Fig. 6. Array multiplier with constant correction rounding.

B. Algorithm Overviews

We chose three algorithms to study the effects of rounding on cumulative error: general matrix multiply (GEMM), the discrete cosine transform (DCT), and the fast Fourier transform (FFT). Each of the three contributes something unique to the study of rounding techniques.

GEMM is the prototypical example of accumulating the results of multiplication since it is just a collection of dot products. This gives us a rather pure sense of how consecutive rounding might affect cumulative error. We implemented GEMM for matrices of arbitrary shape, but limit our experiments to square matrices (without loss of generality). The GEMM algorithm is shown in Procedure 1, although we modified the loop ordering to improve run times during testing.

The DCT behaves similarly to a dot product, but each product is scaled by a deterministic factor that is less than one. Since these factors follow a predictable pattern, the DCT might favor different rounding techniques than GEMM. We chose to implement the venerable 2-dimensional DCT-II. This is the variant used in JPEG encoding and is already familiar to two of the authors. Procedure 2 shows how we compute it.

Finally, the FFT consists of many *levels* of multiplication along with deterministic twiddle factors. Furthermore, the basic functional unit is a "butterfly" consisting of a complex multiplication and two complex additions. The cumulative error we observe for the FFT is thus a sum of magnitudes of complex numbers rather than just the absolute value of differences. We chose the Cooley-Tukey formulation of the FFT shown in Procedure 3 because it is straightforward to implement and, as we will see in subsubsection II-B2, lends itself to error analysis. We based our Python implementation on an excerpt from the Python Numerical Methods book [2].

Procedure 1 General matrix multiplication.

Input: M, N, and K, the matrix dimensions.
Input: A, an $M \times K$ matrix.
Input: B, a $K \times N$ matrix.
Output: C, an $M \times N$ matrix.

for $i \in \{1 \dots M\}$ **do**
  for $j \in \{1 \dots N\}$ **do**
    $c \leftarrow 0$
    for $k \in \{1 \dots K\}$ **do**
      $a \leftarrow A[i,k]$
      $b \leftarrow B[k,j]$
      $c \leftarrow c + \text{round}(a \times b)$
    end for
    $C[i,j] \leftarrow c$
  end for
end for
return C

Procedure 2 The Discrete Cosine Transform II.

Input: w, the width of the input and output matrices.
Input: X, a $w \times w$ matrix.
Output: Y, a $w \times w$ matrix.

for $u \in \{1 \dots w\}$ **do**
  for $v \in \{1 \dots w\}$ **do**
    $s \leftarrow 0$
    for $x \in \{1 \dots w\}$ **do**
      for $y \in \{1 \dots w\}$ **do**
        $g \leftarrow X[x,y]$
        $a \leftarrow \cos[((2x+1)u\pi)/(2w)]$
        $b \leftarrow \cos[((2y+1)v\pi)/(2w)]$
        $s \leftarrow s + g \times (a \times b)$
      end for
    end for
    $\alpha_u \leftarrow \begin{cases} 1/\sqrt{2}, & \text{if } u = 1 \\ 1, & \text{otherwise} \end{cases}$
    $\alpha_v \leftarrow \begin{cases} 1/\sqrt{2}, & \text{if } v = 1 \\ 1, & \text{otherwise} \end{cases}$
    $Y[u,v] \leftarrow (s \times (\alpha_u \times \alpha_v)) >> 2$
  end for
end for
return Y

II. Experiment Design

A. Single Error

Although the main focus of this project is cumulative error, we also looked at the error for single rounded multiplications to replicate the results in Dr. Swartzlander's paper on the

Procedure 3 The Cooley-Tukey Fast Fourier Transform.

Input: n, the length of the input vector. Must be an integral power of 2.
Input: X, a vector of length n.
Output: Y, a vector of length n.

if $n = 1$ **then**
  return X
else
  $E \leftarrow$ FFT $\left[\frac{n}{2},\ \text{even-indexed elements of } X\right]$
  $O \leftarrow$ FFT $\left[\frac{n}{2},\ \text{odd-indexed elements of } X\right]$
  for $k \in \{1 \dots \frac{n}{2}\}$ **do**
    $f \leftarrow \exp\left[-\frac{2k\pi j}{n}\right]$
    $x_0 \leftarrow E[k]$
    $x_1 \leftarrow O[k]$
    $q \leftarrow x_1 \times f$
    $Y[k] \leftarrow x_0 + q$
    $Y\left[k + \frac{n}{2}\right] \leftarrow x_0 - q$
  end for
  return Y
end if

topic [1]. We ran 100,000 trials in which we randomized the multiplicand and multiplier and computed the error as follows for each type of rounding:

$$E = \begin{cases} (a \times b) - \text{rounded}\,(a \times b)\,, & \text{if } a \times b \geq 0 \\ \text{rounded}\,(a \times b) - (a \times b)\,, & \text{if } a \times b < 0 \end{cases}$$

B. Cumulative Error

To study the effect of rounding on cumulative error for each algorithm, we swept several parameters: the size of the input, the word size, and the number of bits we rounded off. Furthermore, each of the three algorithms required a slightly different setup to examine cumulative error. The two main differences among the algorithms are how we handle fixed-point arithmetic and how we specify the "size" of the algorithm.

1) Algorithm Simulation: To gather data, we implemented each multiplier type and algorithm in Python and ran trials at various test points. Implementing the multipliers was straightforward in most cases. However, since constant correction does not form the low bits of the product, we could not simply use the $\star$ operator to compute the multiplication and apply shifting, masking, and addition like we did in the other cases.

Each algorithm accepted the same set of arguments:

Width (w) The size of the input: matrix width for GEMM and DCT; vector length for FFT.

Maximum value (m) The highest value (and negative lowest value) that inputs can take.

Round bits (r) The number of bits that will be rounded off after each multiplication.

Fractional bits (f) The number of bits corresponding to the fractional part of the fixed-point number.

In the end, we always set $f = r$. We also added the restriction that r be less than the maximum possible multiplicand/multiplier. Our dataset includes 30 trials for each algorithm (3) for each rounding type (9) for each input width (either 3 or 4) for each of the 6 following combinations of m and r:

1) $m = 2^{16}$, $r = f = 8$
2) $m = 2^{32}$, $r = f = 8$
3) $m = 2^{32}$, $r = f = 16$
4) $m = 2^{64}$, $r = f = 8$
5) $m = 2^{64}$, $r = f = 16$
6) $m = 2^{64}$, $r = f = 32$

We did, however, omit the width-32 DCT and GEMM is omitted for constant correction. This is because our structural constant correction function took tens of thousands of times longer to simulate than the other more straightforward rounding techniques. Thus, our dataset contained 6480 points for FFT, 4680 for DCT, and 4680 for GEMM.

2) Fixed-Point Arithmetic: Both the DCT and FFT use weights or factors that are less than one. In DCT, these are weights resulting from cosines, and in FFT, these are the twiddle factors that come from complex exponentiation. Thus, we needed to specify a fixed point for these two algorithms. GEMM, on the other hand, can operate entirely on integer values. For consistency, we decided to use fixed-point arithmetic there as well. Since the placement of the fixed point can have some effect on cumulative error, we initially swept this parameter in addition to the number of bits that are rounded off. We limited our experiments to cases in which the number of bits we round off after multiplication is greater than the number of fraction bits we have. Without this restriction, our results would show no difference among the different methods of rounding. However, as mentioned above, the choice of the number of fraction bits appeared to have no affect on error, so we held it constant with respect to the number of round bits in order to simplify the analysis.

Typical fixed-point multiplication with f fractional bits looks like $C = (A \times B) >> f$. When we add in rounding, we get $C = (\text{round}\,[A \times B]) >> f$. Clearly, even with unrounded multiplication, we still have some loss of precision when we sum consecutive products computed in this manner. So, when comparing against exact multiplication, we save the right-shift until the end of the algorithm. For GEMM, we loop over all of the values and shift each one to the right f times. For DCT, we shift right $3f$ times since each element is the result of three levels of multiplication: one to get the weights, another to scale each $X_{x,y}$ element, and a third to scale the $G_{u,v}$ entry by α. Finally, FFT requires $\log_2(\text{width})f$ shifts since there is one multiply per element for each recursive invocation of the FFT function.

3) Cumulative Error: To compute cumulative error, we do a sum of the absolute value of the errors on an element-by-element basis. For GEMM and DCT, this meant taking an element-wise difference of the reference matrix (computed with exact multiplication) and the rounded matrix:

$$E = \sum_{i=1}^{w} \sum_{j=1}^{w} |C_{ref}[i,j] - C[i,j]|$$

Since the vector resulting from an FFT could, in general, have complex elements, we computed the magnitude of the

errors as follows:

$$E = \sum_{i=1}^{w} \sqrt{\mathrm{Re}\left(C_{ref}\left[i\right] - C\left[i\right]\right)^2 + \mathrm{Im}\left(C_{ref}\left[i\right] - C\left[i\right]\right)^2}$$

C. Area and Performance

We developed RTL models to synthesize and compare area and latency for each rounding mode. We based our modules on the designs in [1]. The baseline design is the array multiplier designated by the "exact" rounding type. Each rounding mode is a variation of the initial array multiplier design. We designed our modules in structural Verilog to accurately replicate the intended architecture of each multiplier. We parameterized the designs and used "generate" blocks to allow for easy creation of multipliers with different word sizes — the only difference between an 8×8 multiplier and 32×32 multiplier is a simple change of a macro value. During the synthesis of our modules, we used a 16×16 configuration.

We verified each implementation using a test bench that generated random inputs, fed them to each module, and compared the results with the expected output from behavioral versions of the designs. The waveform from 7 shows how the result for our array multiplier aligns with the expected result. We used the same technique to verify all of the RTL models.

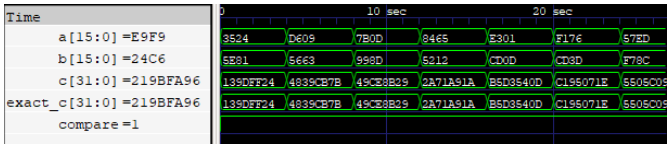

Fig. 7. Verification of array multiplier.

Rounding type	Mean error	Variance	Minimum error	Maximum error
True rounding	0	0.08	-0.5	0.5
Truncation	-0.5	0.08	-1	0
Stretching	0.5	0.08	0	1
Jamming	0	0.33	-1	1
Clearing	-1	0.33	-2	0
Alt. (trunc/jam)	-0.25	0.25	-1	1
Alt. (clear/jam)	-0.5	0.59	-2	1
Alt. (trunc/stretch)	0	0.33	-1	1
Constant correction	0	0.27	-2.5	2.5

TABLE I
SINGLE ERROR STATISTICS FOR EACH TYPE OF ROUNDING.

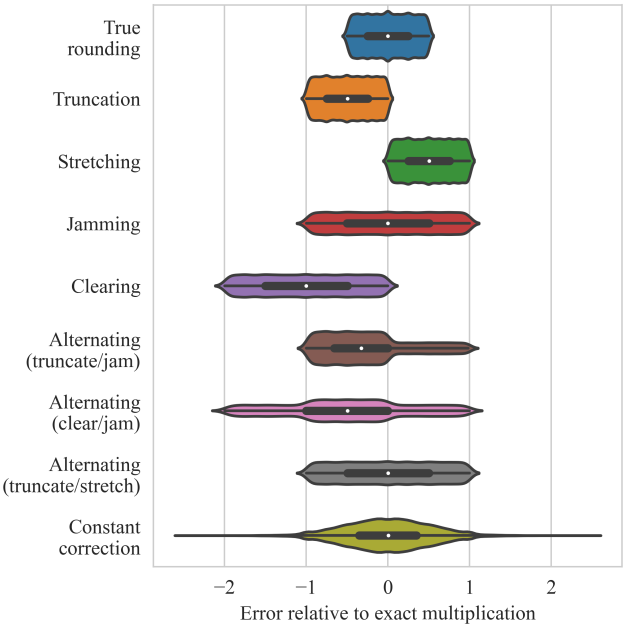

Fig. 8. Distribution of errors over 100,000 randomized multiplications.

III. RESULTS

A. Single error

The single error statistics are in Table I. Note that these come from empirical findings from performing 100,000 randomized trials, not from exhaustive experimentation or analytical methods. We noticed that our statistics for jamming were different than in the original paper. We suspect that our method of error calculation may have differed from that used in the original paper. We also used a different constant correction scheme and hence have different statistics there, as well. Interestingly, alternating between truncation and stretching has the same error statistics as jamming. However, the variance, minimum error, and maximum error are all higher than true rounding. True rounding is slightly more expensive (as discussed in subsection III-C), but it seems that this tradeoff would typically be a good one.

We also include a plot showing the distribution of errors in Figure 8. Most have either a uniform distribution or, in the case of alternating, a sum of two uniform distributions. Constant correction exhibits a distribution that resembles a Gaussian.

B. Cumulative error

There is no single plot, table, or ranking that can tell us how they different rounding types compare in terms of cumulative error. However, there are some clear takeaways. First, true rounding is invariably better than any of the other rounding techniques in terms of cumulative error. We also noticed that word size and the number of bits we rounded off had no significant effect on cumulative error. Thus, each bar in the plots in this section are the average of 180 data points, which typically makes the error bars relatively small. However, constant correction consistently had a fairly large variance. Additionally, the various alternating techniques typically performed worse than either truncation or jamming alone, making the value proposition of the extra flip-flop and inverter suspicious at best. Finally, we saw differences between the rounding techniques that each algorithm favored and how cumulative error was affected by word size.

1) DCT: The DCT has the largest variance in cumulative errors of the three algorithms we studied. The "clearing"

technique, alternated clearing and jamming, and constant correction have relatively high amounts of accumulated error, while the other techniques all have about twice the error of true rounding. Figure 9 shows the relative error for each rounding type on each size of DCT we tried. Recall that we skipped the 32×32 DCT due to the remarkably long simulation times.

The large minimum and maximum error that come with constant correction mean repeated multiplications can really suffer. Each element in the output matrix of the DCT is ultimately the product of four terms (in this implementation): two cosines, an element from the input matrix, and an α factor (which can be either 1, $1/\sqrt{2}$, or $\sqrt{2}$). The multiplication by α comes at the end, so unlike the accumulation of products while computing the DCT, there is no way for these errors to cancel each other out. Thus, constant correction's ability to occasionally get an error of up to ± 2.5 makes it a poor choice for DCT compared to true rounding, truncation, or jamming.

The clearing technique also has a large possible error — from 0 to -2. Consequently, error in the DCT accumulates in the same way as in constant correction.

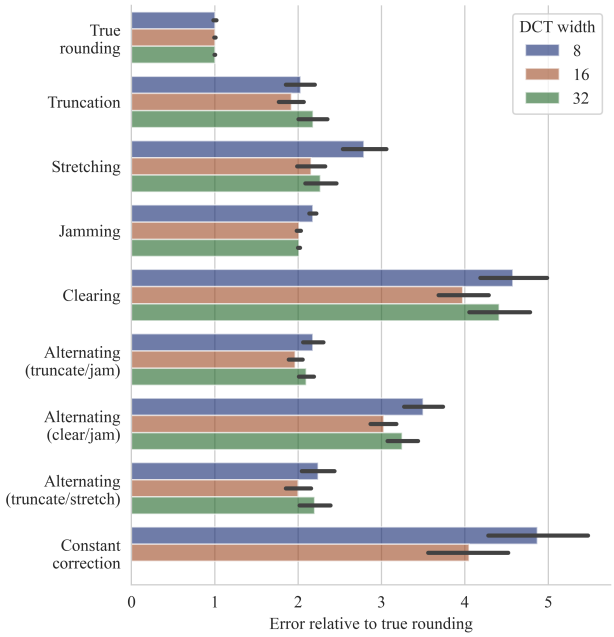

Fig. 9. Effect of rounding type and DCT size on cumulative error.

2) FFT: The FFT displays similar error patterns to the DCT. Some immediately obvious differences visible in Figure 10 are that truncation is the clear winner after true rounding, alternating truncation/jamming beats jamming alone, and clearing performs somewhat better than it did in the DCT. Another striking difference is the size of the error bars. These are much tighter statistics than we saw on the DCT results. Even constant correction is more consistent in terms of total accumulated error, although it still has substantially more variance than the other types.

Like DCT, the elements in the FFT are accumulated products. On the other hand, a doubling in the width of the FFT adds an additional layer of multiplication in the form of a butterfly. One operand to the multiplication is a twiddle factor that has a magnitude of 1, so the products and accumulated sums will not grow much beyond their initial order of magnitude as they would in, for example, an arbitrary GEMM.

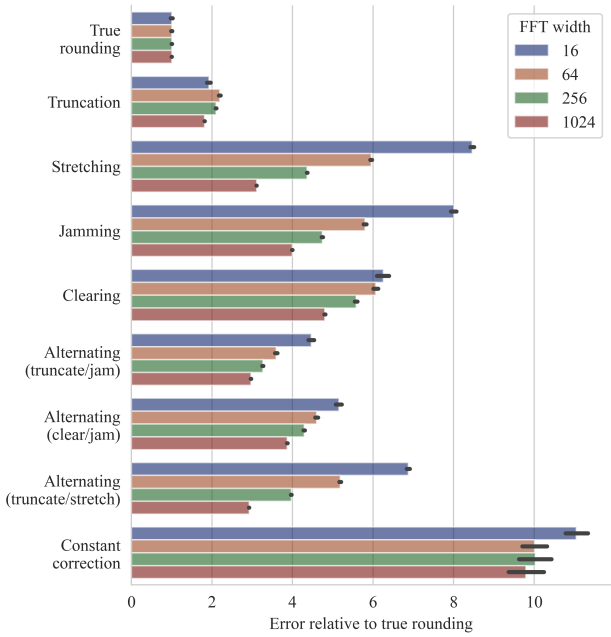

Fig. 10. Effect of rounding type and FFT size on cumulative error.

3) GEMM: GEMM is the only algorithm we studied where jamming has the lowest cumulative error after true rounding. In fact, constant correction and alternating truncation/jamming also perform better than truncation, as shown in Figure 11. This plot also shows that as we increase the width of the matrices we are multiplying, the relative cumulative error increases. This is because GEMM has no mechanism to multiply by factors that are less than or equal to 1 (like DCT's α and cosines or FFT's twiddle factors). Thus, the products become very large, and the accumulation in each dot product grows large as well. Any small single error in rounding propagates in the cumulative error.

C. Area and Performance

We synthesized our RTL models with Synopsys's Design Vision tool using the FreePDK45 45 nanometer library [3]. Table II summarizes the area and performance results from synthesis. Truncation and jamming have slightly smaller area compared to the baseline array multiplier because they do not compute the bit product of the LSB of each input and therefore omit one gate. The alternating rounding type is slightly larger since the design has a flip-flop and inverter to alternate between the two rounding modes. Constant correction has significantly less area compared to the other implementations. This scheme cuts out about of half the gates from the original design and uses constant values to approximate the computation omitted from the lower half of the bit product

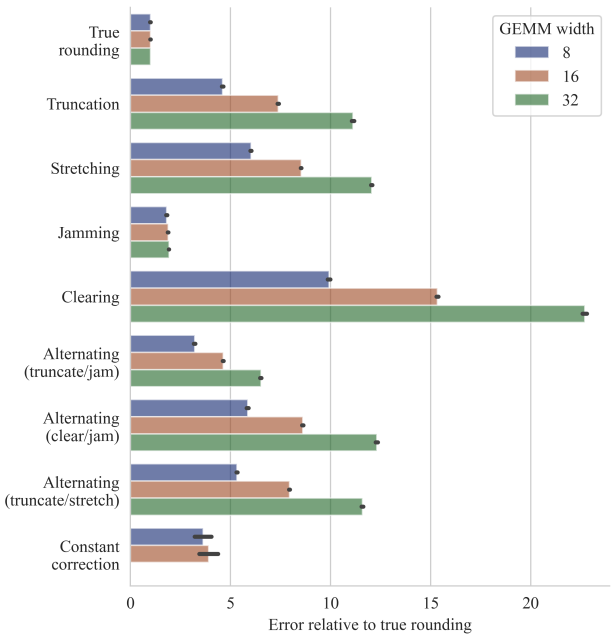

Fig. 11. Effect of rounding type and GEMM size on cumulative error.

matrix. The area increase for true rounding compared to the baseline comes from using a full adder instead of a half adder in the last row of computation. The effect of the full adder also manifests in the performance. The latency metric we present is the static timing analysis measuring the critical path delay of each circuit. True rounding takes longer than the other designs because the half adder that is changed to a full adder is in the critical path. All other designs exhibited the same latency.

Rounding type	Area (μm^2)	Latency (ns)
Exact	4392.64785	4.41
True rounding	4399.68735	4.53
Truncation	4390.30135	4.41
Jamming	4390.30135	4.41
Alt. (trunc/jam)	4402.03385	4.41
Constant correction	2151.74043	4.41

TABLE II
AREA AND PERFORMANCE SYNTHESIS RESULTS OF 16X16 MULTIPLICATION FOR EACH TYPE OF ROUNDING.

IV. Conclusion

Multiplier designers have many options for rounding methods, and their selection will impact the overall accuracy and performance of the result. While true rounding consistently presented the lowest error out of the existing rounding mechanisms, it also presented the slowest implementation by a small margin. Across the algorithms, different rounding methods, with the exception of true rounding, had widely varying performance. This suggests that the rounding technique a designer chooses should be tailored depending on the algorithm at hand and the design requirements. In general, if the chief constraint is area, constant correction is a good candidate, otherwise true rounding is almost certainly the best choice to minimize cumulative error.

Acknowledgment

The authors would like to thank Dr. Earl Swartzlander for an interesting class and a great semester. We learned a lot and hope the Purdue football team is doing well.

References

[1] E. E. Swartzlander, *Truncated multiplication with approximate rounding*, Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems, and Computers (Cat. No.CH37020), 1999, pp. 1480-1483 vol.2. doi:10.1109/ACSSC.1999.831996.

[2] Q. Kong, T. Siauw, and A. Bayen, *Fast Fourier Transform (FFT) — Python Numerical Methods*, Elsevier Academic Press, 2020. https://pythonnumericalmethods.berkeley.edu/notebooks/chapter24.03-Fast-Fourier-Transform.html.

[3] NC State EDA, *FreePDK45*, https://eda.ncsu.edu/freepdk/freepdk45/.