Daala Transforms

Timothy B. Terriberry
Nathan Egge
Christopher “Monty” Montgomery
Transform Design Goals

• Exact integer implementation
  – Lots of iterated prediction with unstable (gain=1.0) filters, no drift acceptable

• Many variations
  – Low bit-depth, high bit-depth, rectangular, DCT, DST, etc.

• High accuracy
  – We don’t need to compromise quality for complexity

• Low software complexity
  – In particular implementation in SIMD

• Reasonable hardware complexity
  – Low latency for small sizes
  – Transform re-use/embedded designs
H.264 4-point DCT

- Very low complexity (8 adds, 2 shifts):

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
2 & 1 & -1 & -2 \\
1 & -1 & -1 & 1 \\
1 & -2 & 2 & -1
\end{bmatrix}
\]

- Drawback: non-uniform scale
  - Saves one multiply/coeff. by combining it with quantization
  - But costs several multiplies/coeff. when rate-distortion optimizing coefficients in an encoder
    - Need uniform scale for distortion to make good trade-offs
    - Encoder costs multiplied by search space
  - Costs a large table of constants (very large for large transform sizes)

- New goal: uniform scaling (4 multiplies)
  - Achievable with much less than 1 multiply/coeff for large sizes
VP9 4-point DCT

- 6 multiplies (full 32-bit products needed), 8 adds (2 at 32 bits), 4 shifts

```
747x40
230x506
6 multiplies (full 32-bit products needed), 8 adds (2 at 32 bits), 4 shifts

+ 11585 (√½ in Q14) 1/16384
+ 11585 1/16384
+ 6270 (cos(π/8) in Q14) 1/16384
+ 15137 (sin (π/8) in Q14) 1/16384
- 6270
```
Avenues for Improvement

• Simplify the multiplies
  – Just scaling the output of the H.264 transform only costs 4 multiplies (but less accurate)

• Scaling
  – Adds a factor of $\sqrt{2}$ relative to a unitary transform
  – VP9 adds an additional $\sqrt{2}$ each time the size doubles
  – When $\log_2(\text{width}) + \log_2(\text{height})$ is even, correct with a shift
  – But it's odd for rectangular transforms (e.g., 8x4)
  – Costs 1 multiply/coeff. to correct for
Extra Scaling

• Where does this scaling come from structurally?
N-point Type II DCT

N/2 Type II DCT

N/2 Type IV DST
N-point Type II DCT

This part is non-unitary

\[ \sqrt{1^2 + 1^2} = \sqrt{2} \]
N-point Type II DCT

This part is non-unitary
\[ \sqrt{1^2 + 1^2} = \sqrt{2} \]
Getting Rid of the Extra Scaling

• Can use multiplies
  – Source of 2 of the multiplies in VP9’s 4-point DCT
  – Kind of expensive

• Another approach:
  – Restrict ourselves to shifts and adds
  – Use *asymmetric scaling*
Asymmetric Scaling (1)

- Asymmetric output scales
- Overall scaling remains unity
- Cancel out the asymmetry in subsequent steps

\[ y_0 = x_0 + x_1 = x_0 + x_1 \]
\[ y_1 = (y_0 >> 1) - x_1 = (x_0 - x_1)/2 \]

OR

\[ y_0 = x_0 - (y_1 >> 1) = (x_0 + x_1)/2 \]
\[ y_1 = x_0 - x_1 = x_0 - x_1 \]
Asymmetric Scaling (2)

- Asymmetric input scales
- Cancels out the asymmetry from previous steps

\[
\begin{align*}
\text{OR} & \quad & \quad \quad \quad & \quad y_0 &= x_0 - y_1 \quad = x_0/2 + x_1 \\
& & \quad y_1 &= (x_0 >> 1) - x_1 \quad = x_0/2 - x_1
\end{align*}
\]

\[
\begin{align*}
y_0 &= x_0 + (x_1 >> 1) \quad = x_0 + x_1/2 \\
y_1 &= y_0 - x_1 \quad = x_0 - x_1/2
\end{align*}
\]
Simplifying the Multiplies

• Multiplies arise from \textit{plane rotations} between two variables

\[
\begin{bmatrix}
y_0 \\
y_1
\end{bmatrix} = \begin{bmatrix}
\cos(\theta) & \sin(\theta) \\
-sin(\theta) & \cos(\theta)
\end{bmatrix} \begin{bmatrix}
x_0 \\
x_1
\end{bmatrix}
\]

• Can trade one multiply for one addition

\[
p_0 = x_0 - \frac{\cos(\theta) - 1}{\sin(\theta)} x_1
\]
\[
y_1 = x_1 - \sin(\theta) p_0
\]
\[
y_0 = p_0 - \frac{\cos(\theta) - 1}{\sin(\theta)} y_1
\]
Asymmetric Scaling Multiplies

• Can also arbitrarily scale inputs and outputs

\[
\begin{bmatrix}
  y_0 \\
  y_1
\end{bmatrix} =
\begin{bmatrix}
  u & 0 \\
  0 & \frac{1}{stu}
\end{bmatrix}
\begin{bmatrix}
  \cos(\theta) & \sin(\theta) \\
  -\sin(\theta) & \cos(\theta)
\end{bmatrix}
\begin{bmatrix}
  s & 0 \\
  0 & t
\end{bmatrix}
\begin{bmatrix}
  x_0 \\
  x_1
\end{bmatrix}
\]

• Becomes

\[
p_0 = x_0 - \frac{t}{s} \frac{\cos(\theta) - su}{\sin(\theta)} x_1
\]

\[
y_1 = x_1 - \frac{\sin(\theta)}{tu} p_0
\]

\[
y_0 = p_0 - tu \frac{su \cos(\theta) - 1}{\sin(\theta)} y_1
\]
Advantages

- 25% fewer multiplies
  - Much more expensive than adds
- All have x += a*y structure
- Becomes x += (a*y + 16384) >> 15 in fixed point
  - Only need top part of multiplier output
  - 16-bit SIMD stays in 16 bits
    - Going to 32 bits halves SIMD throughput
- SSSE3 and NEON both have an instruction for this
  - PMULHRSW (parallel multiply high, round, and shift word)
  - VQRDMULH.S16 (vector saturated rounding doubled multiply high)
- Single instruction to multiply, add rounding offset, and shift down
Putting It All Together

- 9 adds, 3 multiplies, 2 shifts

\[ M_0 = \frac{2\cos\left(\frac{3\pi}{8}\right) - \sqrt{2}}{\sin\left(\frac{3\pi}{8}\right)} \]
\[ M_1 = \sqrt{\frac{1}{2}} \sin\left(\frac{3\pi}{8}\right) \]
\[ M_2 = \frac{\cos\left(\frac{3\pi}{8}\right) - \sqrt{2}}{\sin\left(\frac{3\pi}{8}\right)} \]
Putting It All Together

- 9 adds, 3 multiplies, 2 shifts

$$M_0 = \frac{2\cos\left(\frac{3\pi}{8}\right) - \sqrt{2}}{\sin\left(\frac{3\pi}{8}\right)}$$
$$M_1 = \sqrt{\frac{1}{2}} \sin\left(\frac{3\pi}{8}\right)$$
$$M_2 = \frac{\cos\left(\frac{3\pi}{8}\right) - \sqrt{2}}{\sin\left(\frac{3\pi}{8}\right)}$$

Same value
8-Point DCT
16-Point DCT
And more...

- Up to 64-point DCT implemented
  - The margin of this slide is too small to contain...

- Embedded structure
  - Both N-point DCT and N-point DST are embedded in the 4N-point DCT
    - Embedding skips a level because of the asymmetries
Accuracy (1)

- Right shifts and multiplies introduce rounding errors
- Want to keep these as small as possible
- Solution?
  - Shift up input
  - Forward transform, quantize, code, inverse transform
  - Shift down output
- Diminishing returns at 4 bits (for 8-bit input)
  - Enough to make all DCTs match a double-precision floating point implementation after rounding to nearest integer
    - Error $\leq 0.5$
Accuracy (2)

- How does this compare with VP9?
  - Also shifts up inputs (by a smaller amount)
  - And shifts down outputs (by a larger amount)
    - Sometimes between row and column transforms, too
- Scale of VP9 coefficients grows as transform progresses
  - Rounding errors early in process get magnified
- Daala: all stages have the same scale
  - All errors injected at the same level
  - Accumulate, but aren’t magnified
High Bit Depth

• Accuracy less important for higher bit depths (10 or 12 bits)
  - Importance is accuracy *relative* to quantizer, and higher bit depths use larger quantizers

• We shift up less for higher bit depths
  - 10 bits = 2 bit shift
  - 12 bits = no shift

• Result: Can use same transforms for all bit depths
Dynamic Range (1)

- Everything has orthonormal (unitary) scaling
- Dynamic range of the outputs still increases
  - Dynamic range = minimum/maximum output values
  - Unitary transforms are N-dimensional rotations
  - If the input is a box, the length of the diagonal is longer than the length of an edge
    - By a factor of $\sqrt{2}$ every time $N$ doubles
- So how big can the outputs be?
Dynamic Range (2)

- All transforms with 64 pixels or less fit in 16 bits
  - 9-bit residual + 4-bit up shift + 3 bits of dynamic range expansion
  - Includes 4x4, 4x8, 8x4, 8x8, 4x16, 16x4

- All column transforms fit in 16 bits
  - Maximum size needed for hardware transpose buffer

- VP9 has larger intermediaries in the transforms, but shifts final coefficients down to fit in 16 bits
  - Think this is a mis-optimization
  - Just as easy to pack during quantization
  - Avoids double-rounding, simplifies RDO (no special cases)
Reversibility (1)

• Steps of the form
  \[ x_i = x_i + f(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_N) \]
  are called *lifting steps*

• Exactly reversible:
  \[ x_i = x_i - f(x_0, \ldots, x_{i-1}, x_{i+1}, \ldots, x_N) \]

• Inverse transform: just reverse all of the steps

• Why is this good?
Reversibility in Daala

- Daala used lapping instead of a deblocking filter
- Deblocking filters are low pass
  - Tend to blur out details over consecutive frames
- Forward and inverse lapping are matched
  - No low passing
- If that match is not exact, errors will build up over multiple frames
  - Costs bits to correct
Reversibility (2)

• Do we need perfect reversibility?
  – It seems to help (small coding gain improvements)
  – Probably not required, but it’s basically free
  – Don’t actually have it in Daala anymore
    • 4 bit down shift after inverse breaks it
    • Using 12-bit references (even for 8-bit data) restores it [1]
    • But using CLPF/deringing also solves the problem
      – Adds the low pass filter we were missing from deblocking

Reversibility and Dynamic Range

- Transform coefficients values are larger than pixel values
  - Forward transform expands dynamic range
- Inverse transform is also an N-dim. rotation
  - How do we know it doesn't expand dynamic range?
- E.g., if $x_0$ and $x_1$ just barely fit in 16 bits, how do we know $x_0 + x_1$ won't overflow?
- Answer: Reversibility
  - Values computed in inverse same as forward transform
    - $\pm$ quantization error
  - Only guaranteed if coefficients result of transforming pixels
Type IV vs. Type VII DST

• For intra prediction residuals, prediction error is asymmetric
  – Less error closer to edges we’re predicting from

• Want an asymmetric transform to code them

• Optimal transform is a Type VII DST
  – Compute correlation matrix, solve eigensystem problem in the limit as the correlation approaches 1

• Type VII DST factorizations are much nastier than Type IVs
Type VII vs. Type IV DST

- **Type IV**

\[ y_k = \sum_{n=0}^{N-1} x_n \sin \left( \frac{\pi}{N} (n + \frac{1}{2}) (k + \frac{1}{2}) \right) \]

- **Type VII**

\[ y_k = \sqrt{\frac{2}{N + \frac{1}{2}}} \sum_{n=0}^{N-1} x_n \sin \left( \frac{\pi}{N + \frac{1}{2}} (n + 1) (k + \frac{1}{2}) \right) \]
Type VII vs. Type IV

- Type IV transforms *almost* as good, and already embedded inside our DCTs
- Current approach
  - Use Type VII for small DSTs (4-point and 8-point)
  - Use embedded Type IV for larger DSTs
### Overall Complexity

<table>
<thead>
<tr>
<th></th>
<th>Daala TX</th>
<th>TXMG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>muls/coeff</td>
<td>adds/coeff</td>
</tr>
<tr>
<td>DCT 4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DST 4</td>
<td>1.25</td>
<td>2.75</td>
</tr>
<tr>
<td>DCT 8 [1]</td>
<td>1.875</td>
<td>3.875</td>
</tr>
<tr>
<td>DST 8 [2]</td>
<td>2.625</td>
<td>9.375</td>
</tr>
<tr>
<td>DCT 16</td>
<td>2.0625</td>
<td>5.1875</td>
</tr>
<tr>
<td>DST 16</td>
<td>3.1875</td>
<td>6.1875</td>
</tr>
<tr>
<td>DCT 32</td>
<td>2.7188</td>
<td>6.2188</td>
</tr>
<tr>
<td>DST 32</td>
<td>3.6562</td>
<td>7.6562</td>
</tr>
</tbody>
</table>

- [1] SIMD benchmarked at 26.2% faster
- [2] Daala TX uses a Type VII DST, while TXMG uses a Type IV
Hardware Considerations (1)

• Intra prediction requires reconstructed pixels from neighboring blocks

• This serializes reconstruction of these blocks
  – Including the inverse transform
  – Particularly a problem for encoders

• Our 3-multiply rotations chain them all consecutively

• This is a bottleneck for small transform sizes
Low-Latency Small Transforms

- 4-point DCT: replace 3-multiply block with 4-multiply version
  - All multiplies can proceed in parallel
  - Still only use top part of multiply
    - Full SIMD throughput
- 4-point Type VII DST:
  - Use custom factorization with 5 parallel multiplies
- These are not exactly reversible
Hardware Considerations (2)

- Most hardware already “multi-standard”
  - Including VP9
- Dedicates a lot of gates to parallel multipliers
- Can replace serial multiplies in rotations with parallel multiplies

\begin{align*}
  u_0 &= x_0 + a x_1 \\
  y_1 &= u_0 + b u_0 \\
  y_0 &= y_1 + a x_1
\end{align*}

becomes

\begin{align*}
  u_0 &= x_0 + x_1 \\
  u_1 &= (1 + ab) u_0 \\
  u_2 &= (b(a - 1) + 1) x_0 \\
  u_3 &= (a + (a - 1)(1 + ab)) x_1 \\
  y_0 &= u_1 + u_3 \\
  y_1 &= u_1 - u_2
\end{align*}

- Still experimenting to see impact on accuracy, potential for overflows
Questions?