Masktools 2

Sep 13, 2013

What's a lut?

Lut is short for **lookup table**, a simple data structure where each predefined key maps to some specific value. It is basically a dictionary implemented as an array, where keys are indexes of elements in said array. For example, imagine a table `9 8 7 6 5 4 3 2 1 0`. The first element of this table is 9, so it will map value 0 (array indexing starts with 0) to 9, 1 to 8 and so on. There's nothing insanely hard about the concept. It is used for optimization purposes, because when you have only a limited range of possible input values, it might be faster to calculate output value for each of them once and then look it up in a table, than doing the same calculations over and over again (it's not always faster though).

There's one important thing about this optimization - it's impossible to optimize with SIMD unless AVX2 instruction set is available (right now - only in Haswells). For simple operations like binarization, it is faster to binarize each pixel individually without a LUT, because with SIMD you can process multiple pixels at once (8-32 depending on the instruction set used). For a real example refer to my tcolormask plugin, that switches to LUT only if SIMD operation is likely to be slower.

mt_lut (part 1)

Now imagine a video frame. In 8-bit world, video frame is a simple array of values in a range [0, 255]. It is possible, and in fact trivial, to build a lut that would map any possible value of any pixel in the frame to some output value. And this is exactly what mt_lut is doing (this operation is called a "point operation" in all image processing books, but the definition is somewhat more formal and abstract). If you look at the source code, you'll see a loop over all rows in a frame, with an inner loop running over each pixel in the row, processing *each pixel individually*. This is important - no other pixel in the same image can affect the current one, they are completely independent.

But how does masktools build this lut?

So far I've been talking about a lut as a given, but how does one actually build this table? There are various ways to specify this data, but basically you always have a loop somewhere that runs over all possible values of pixels and computes some expression for each of them (unless you specify the whole lut by hand, of course). Masktools uses so-called reverse polish notation (RPN for short) to specify this expression. Then for each possible 256 values it computes this expression, storing the result in appropriate position in the lut. RPN might look complex and dumb at the first glance, but believe me - after some time you'll get used to it and you *will* love it.

The expression given to a lut filter is evaluated as a stack. It basically puts all operators (separated by spaces) in the expression into a collection in LIFO order, then takes the top element and evaluates it as expression, recursively extracting any other elements from the stack if needed. Do note that it doesn't evaluate any operators it cannot reach from the top element. For example, expression `mt_lut("Hey look I can put some random stuff in luts x")` will be evaluated to x

and all other garbage will be ignored. You can find the source code in symbol.cpp.

Let's go through an example. Assume that the expression we gave was "x 25 - 2.5 *". For each possible value of source pixel [0; 255] run the following algorithm (example assumes value of x = 10).

1. First, expression is split into tokens [x, 25, -, 2.5, *].
2. Token * gets evaluated. It requires two arguments, so top two elements are extracted from the stack [-, 2.5].
3. The first element is - and it again requires two elements [x, 25];
4. Expression x is evaluated to 10
5. Both arguments for - are ready. Result of expression is -15.
6. Both arguments for * are ready. Result of expression is -37,5.
7. -37,5 is not an allowed pixel value for a 8-bit clip, so it gets clipped to zero.
8. Zero is stored in position 10 of the lut.

Internally it works in double precision, unless integer-only operations are used (shifts, negation, modulo or binary operations), then it converts the value to 64-bit integer. Output result is clipped to range [0,255] because it's impossible to store other values in 8-bit output clip.

Alternative way to build the lut is to make script writers do it themselves in a Real Programming Language. This is exactly what vapoursynth is asking you to do. Whether you like this way more or not is up to you.

mt_lut (part 2)

Now that you know how luts are evaluated (of course you don't, go get at least some experience with RPN first), let's look at some examples.

- `mt_lut("255 x -")` - inversion of the clip. Always use mt_invert over this lut.
- `mt_lut("x 50 > 255 0 ?")` - binarization. All values greater than 50 map to 255, all others - to zero. Always use mt_binarize over this lut.
- `mt_lut("x 126 < 126 x 130 > 130 x ? ?")` - limits the value of a clip. Always use mt_clamp combined with blankclip for this (or limiter filter in avs core).
- `mt_lut("x 128 / 0.7 ^ 128 *")` - increase value of all pixels lower than 128 and decrease value of other pixels. This decreases contrast of the image (look at a histogram to understand what it's doing).

mt_lutxy

This filter is basically identical to mt_lut, except it uses values from two clips rather than one. This is great for temporal stuff, but pixels are still completely independent spatially. The lut size is 256*256 = 65536 bytes, or 64KB per plane. No new theory here, so let's go straight to examples.

- `mt_lutxy(c1, c2, "x y max")` or `mt_lutxy(c1, c2, "x y > x y ?")` - gets maximum value of two corresponding pixels. Always use mt_logic over this lut.
- `mt_lutxy("x y - 128 +")` - difference between two clips plus 128. Always use mt_makediff over this.
- `mt_lutxy("x y + 128 -")` - sum of two clips minus 128. Always use mt_adddiff over this.

- `mt_lutxy("x y - abs 5 *")` - absolute difference between two pixels multiplied by 5. This is extremely useful for comparisons.

There's one thing I want to say about mt*lutxy* - ***stop overusing it***. *Many avisynth script writers tend to forget that there are functions like mt_clamp or mt***diff, making their scripts slower. Why? I don't know.

mt_lutxyz

Same as previous two, except for three clips. The lut size is 256*256*256 = 16777216 bytes or 16MB **per plane**. This results in 48MB per each filter call. **Always**. I don't know what Manao was thinking, but even if you don't process some planes, even if some luts are completely identical - this filters will **always** compute and store three independent luts (this is fixed in my masktools fork, of course). This makes the filter basically unusable. Processing is still fast, but lut computation that happens on startup (after you press f5) takes **time**. No examples, avoid at all costs.

mt_lutf

Now, the fun stuff. On a high level, you can think of it as mt_lutxy call, when one of the clips has constant value, equal to output of `mode` function, applied to all pixels in the clip, for example

```
scriptclip("""mt_lutxy(mt_lut(y=-c1.YPlaneMax()), c2, "x y -")""")
```

is equivalent to

```
mt_lutf(c1, c2, "max", "x y -")
```

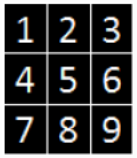
Obviously, mt_lutf version is way more efficient.

I think the most common use-case for this function is optimization of runtime filtering. So if your script uses things like `YPlaneMax`, `AverageLuma` etc., consider rewriting it to `mt_lutf` and other masktools operations (downscaling `mt_lutf` output to 4x4 or 8x8 to keep things fast). It will make your script much faster and more stable. But do note that `mt_lutf` works on clips, so its output values are clipped to 8-bit integers, so e.g. `AverageLuma`-dependent processing will have lower precision when implemented with mt_lutf. Here's a partial example of runtime script rewritten to masktools (this was long ago so they might not be completely identical). You should decide for yourself if this kind of optimization is worth it.

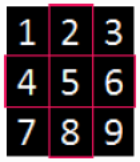
mt_luts

First of all, this is the first "spatial" function (lutf was also spatial but it doesn't count). It means that neighborhood pixels *can* affect output value of the current one. Parameter `pixels` determines what pixels do.

Second - this is one of the hardest masktools function. For every pixel defined by the `pixels` parameter, calculate the expression value using the same old lut. For example, imagine a block from your image (current pixel is in the center).



Your pixels value defines a cross, shown on the next image.



You expression is x y + and your mode is "max". Then it will first calculate 5 values:

1. 5 + 2 = 7
2. 5 + 4 = 9
3. 5 + 5 = 10
4. 5 + 6 = 11
5. 5 + 8 = 13

Then it will apply mode function to calculated value, resulting in `max(7, 9, 10, 11, 13) = 13`. This value will be written to the output frame.

Strictly speaking, x and y values come from different clips (hence two clip parameters), but most of the times you'll be specifying the same clip for both arguments. And actually, there is a bug in masktools, that makes it always use values from clip2 (no longer true for my fork).

This function is so slow you'll most likely never use it. Lut size is only 65KB per plane, but actual calculation is costly. Every time you see this used (in gradfun2dbmod, for example) - try to think of a way to implement the algorithm without it.

mt_lutsx

This one is simple. Just like in mt_luts, you specify range of pixels and modes (two this time). Mt_lutsx calculates mode1 and mode2 for clip1 and clip2 respectively. Only then the final lut is applied.

It's easy to understand by example:

```
mt_lutsx(c1, c2, c2, "med", "avg", mt_square(1), "x y + z + 3 /")
```

is equivalent but much slower than

```
mt_lutxyz(c1, c2.removegrain(4), c2.removegrain(20), "x y + z + 3 /")
```

You might think that it would be more precise because there are no intermediate clips, but since this is a lut function, values of z and y are converted to byte before the actual lookup happens.

Lut size is 16MB per plane and the function is insanely slow, thus not really usable even for SD content.

mt_lutspa

The most unusual lut function. In fact, this isn't even a lut function - for whatever reason Manao just implemented it with a lookup operation while it actually requires just a single memcpy.

Unlike other lut functions, this one doesn't care about input values at all - its lut contains values for all possible combination of x and y coordinates of a frame. Lut size is equal to frame size at a given resolution. I'm not gonna describe how this function works because it doesn't introduce anything new (check the source code if you're wondering what some parameters are doing).

One interesting thing about this function - since it doesn't depend on the actual input (other than resolution and colorspace), the result value can effectively be cached and reused without any additional processing. This is done by placing e.g. `trim(0,-1).loop(c.framecount())` after the lutpsa call. This is important if you're using reference implementation which is lolslow.

16-bit

One interesting thing about luts is 16-bit support. Basically, the only function that can be trivially implemented in 16-bit is `mt_lut`, which will have 65KB lut size because of the extended range of possible input values. `Mt_lutxy` will require 65536*65536 bytes (4GB) lut, which is not a reasonable thing to do. There are some optimization techniques, but most implementations just switch to runtime value calculation and abandon any luts whatsoever.

One more thing - reference masktools does not include 16-bit lut and Firesledge implemented it in terms of `mt_lutxy` in his dither package. I strongly advice you to check the source out to at least understand what kind of hackery avisynth devs need to implement to support 16-bit processing. The function is called Dither_lut16.

Back to main page