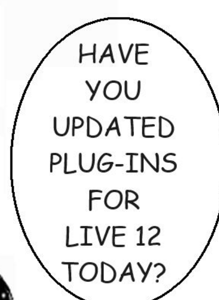
What's new in this Dplug "Winter"?

- Not much this month. Updated roadmap 2024 =>
  https://github.com/AuburnSounds/Dplug/wiki/Roadmap (be patient)

- Two tools exist that can build plug-ins MUCH faster: **reggae** and **redub**
  https://code.dlang.org/packages/redub

- #dlang activity in UI library department: libsoba and Fluid.



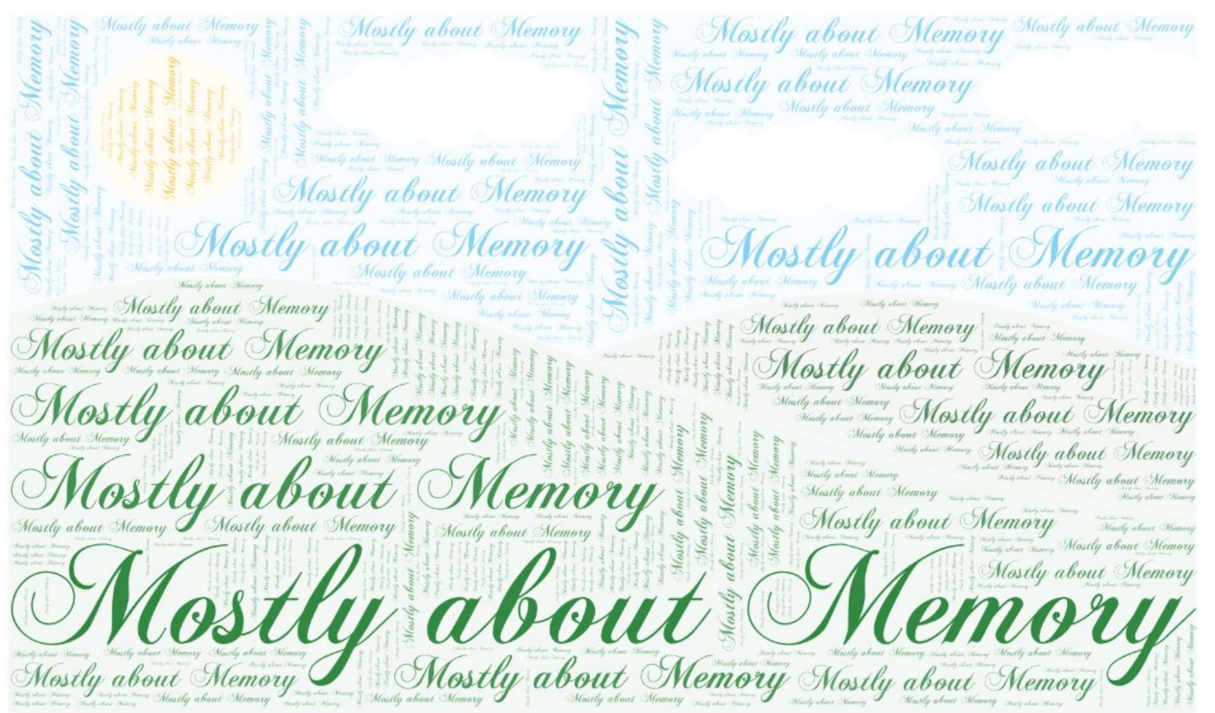
FL Studio is adopting **CLAP** just after we adopt **FLP** format

=> *my 400IQ strategy falls apart*😭

Audio Optimization Idioms you might find useful

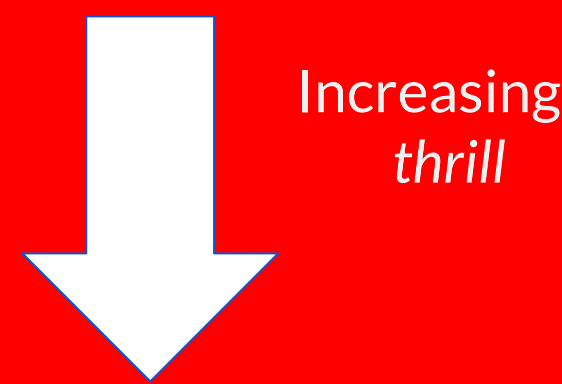
Meeting
Apr 2nd 2024

Nowadays performance is



We will see 4 dangerous optimizations:

1. The “Remainder Loop”
2. The “Padded Buffer”
3. The “Merged Allocation”
4. The “Fixed Allocation”



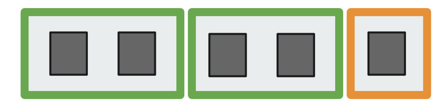
The "Remainder Loop"

say we have this unoptimized loop:

```
// Energy of FFT data
void computeSquaredMagnitudes(Complex!float* fftCoeffs,
                              float squaredMagnitudes,
                              int fftSize)
{
    foreach(bin; 0..fftSize/2+1)
    {
        cfloat c = fftCoeffs[bin];
        squaredMagnitudes[bin] = c.re * c.re + c.im * c.im + 1e-10f;
    }
}
```

We want to go SIMD, but nothing is a multiple of 4 (4 float = 16 byte = NEON and SSE alignment)

The "Remainder Loop"



Possibly, this one run faster

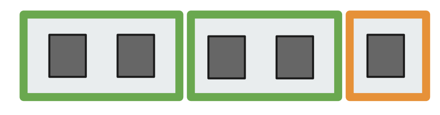
After optimization, 2 loops instead of 1

Reminder loop typically for **1 to 3 elements max**

```
// Energy of FFT data
void computeSquaredMagnitudes(Complex!float* fftCoeffs,
                              float squaredMagnitudes,
                              int fftSize)
{
    __m128 offset = _mm_set1_ps(1e-10f);
    int bin = 0;
    for(; bin + 1 < fftSize/2+1; bin += 2) // this loops compute 2 squares at once
    {
        // read two bins at once and square them
        __m128 bins = _mm_load_ps(cast(float*)(&fftData[bin]));
        bins = _mm_mul_ps(bins, bins);
        bins = _mm_add_ps(bins, _mm_srli_ps!4(bins));
        __m128 squaredMag = _mm_shuffle_ps!0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }

    for(; bin < fftSize/2+1; bin += 1) // reminder loop
    {
        cfloat c = fftCoeffs[bin];
        squaredMagnitudes[bin] = c.re * c.re + c.im * c.im + 1e-10f;
    }
}
```

The “Remainder Loop”



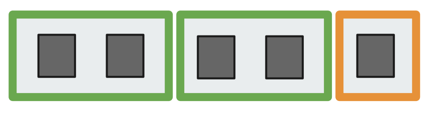
```
// Energy of FFT data
void computeSquaredMagnitudes(Complex!float* fftCoeffs,
                              float squaredMagnitudes,
                              int fftSize)
{
    __m128 offset = _mm_set1_ps(1e-10f);
    int bin = 0;
    for(; bin + 1 < fftSize/2+1; bin += 2) // this loops compute 2 squares at once
    {
        // read two bins at once and square them
        __m128 bins = _mm_load_ps(cast(float*)(&fftData[bin]));
        bins = _mm_mul_ps(bins, bins);
        bins = _mm_add_ps(bins, _mm_srli_ps!4(bins));
        __m128 squaredMag = _mm_shuffle_ps!0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }

    for(; bin < fftSize/2+1; bin += 1) // reminder loop
    {
        cfloat c = fftCoeffs[bin];
        squaredMagnitudes[bin] = c.re * c.re + c.im * c.im + 1e-10f;
    }
}
```

tricky body

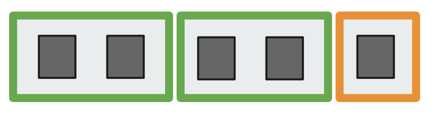
same body as before SIMD

The “Remainder Loop”



- **9 times out of 10** you can upgrade regular code to SIMD code using this simple transformation
  - remember to benchmark
  - bench against naive code, which is often best
  - bench against Dlang Array Operations

The “Remainder Loop”



- **9 times out of 10** you can upgrade regular code to SIMD code using this simple transformation
  - remember to benchmark
  - bench against naive code, which is often best
  - bench against Dlang Array Operations

- More importantly: if you’re unsure about your SIMD translation, just comment the fast loop to check for diffs. Remainder body acts as documentation.

The “Padded Buffer”

Can we do this instead?

```
// Energy of FFT data
void computeSquaredMagnitudes(Complex!float* fftCoeffs,
                              float*squaredMagnitudes,
                              int fftSize)
{
    __m128 offset = _mm_set1_ps(1e-10f);
    int bin = 0;
    for(; bin + 1 < fftSize/2+1; bin += 2) // this loops compute 2 squares at once
    {
        // read two bins at once and square them
        __m128 bins = _mm_load_ps(cast(float*)(&fftData[bin]));
        bins = _mm_mul_ps(bins, bins);
        bins = _mm_add_ps(bins, _mm_srli_ps!4(bins));
        __m128 squaredMag = _mm_shuffle_ps!0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }
}
```

^No Reminder Loop needed!

The "Padded Buffer"

YES, IF 1. BUFFERS ARE PADDED

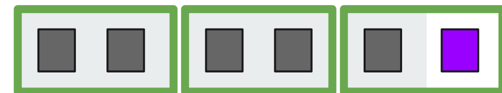
```
void computeSquaredMagnitudes(Complex!float* fftCoeffs,
                              float* squaredMagnitudes,
                              int fftSize)
{
    // blah blah
}
```

has extra space at the end

has extra space at the end

AND 2. WE CAN PROCESS MEANINGLESS SAMPLES (STATELESS)

The “Padded Buffer”



Wrapping it up:

1. Allocate one extra sample(s) if non-multiple

```
// Round up: A + (B - 1) / B
int SSE_ALIGN = 16;
fftCoeffs.reallocBuffer( (len + 1) /2, SSE_ALIGN );
squaredMagnitudes.reallocBuffer( (len + 1) /2, SSE_ALIGN );
```

```
// Energy of FFT data
void computeSquaredMagnitudes(Complex!float* fftCoeffs,
                              float* squaredMagnitudes,
                              int fftSize)
{
    __m128 offset = _mm_set1_ps(1e-10f);
    int bin = 0;
    for(; bin + 1 < fftSize/2+1; bin += 2) // this loops compute 2 squares at once
    {
        // read two bins at once and square them
        __m128 bins = _mm_load_ps(cast(float*)(&fftData[bin]));
        bins = _mm_mul_ps(bins, bins);
        bins = _mm_add_ps(bins, _mm_srli_ps!4(bins));
        __m128 squaredMag = _mm_shuffle_ps!0x88(bins, bins);
        squaredMag = _mm_add_ps(squaredMag, offset);
        _mm_storel_epi64(cast(__m128i*)(&squaredMagnitudes[bin]), cast(__m128i) squaredMag);
    }
}
```

^No Reminder Loop => smaller code size, no speed loss usually

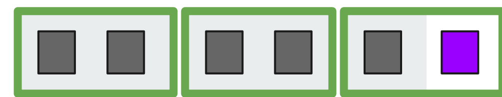
2. Process it and discard

Helpful because SIMD cos/sin/tan/exp/pow/log are usually same cost when parallel like this, and will inline only once.

The “Padded Buffer”

- Simplify some SIMD loops by being multiple of 2, 4...
- Same speed as Reminder Loop and smaller code size

The “Padded Buffer”



- Simplify some SIMD loops by being multiple of 2, 4...
- Same speed as Reminder Loop and smaller code size
- **doesn’t work for recursive DSP tasks**
- **error-prone**
- **padded area might be NaN, out of bounds etc...**

Is this familiar?

```
struct MyDopeProcess
{
    void initialize(int maxFrames)
    {
        _alpha.reallocBuffer(maxFrames);
        _peak.reallocBuffer(maxFrames);
        _rms.reallocBuffer(maxFrames);
        _crest.reallocBuffer(maxFrames);
        // [...many buffers...]
    }
    // [...]
}
```

...also don't forget to reclaim L🤣L

```
~this()
{
    _alpha.reallocBuffer(0);
    _peak.reallocBuffer(0);
    _rms.reallocBuffer(0);
    _crest.reallocBuffer(0);
    // [...many buffers...]
}
```

_alpha[0..maxFrames]

megabytes of memory

_rms[0..maxFrames]

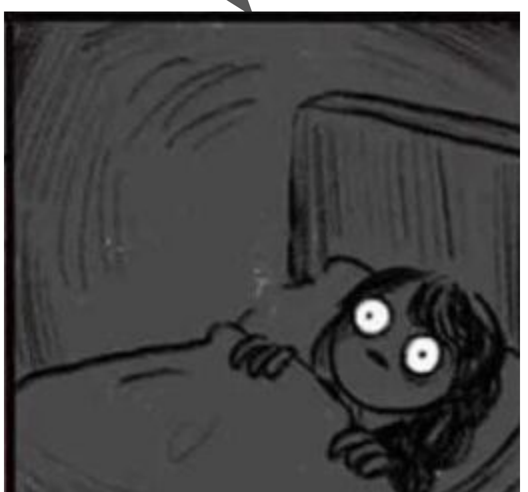
megabytes of memory

_peak[0..maxFrames]

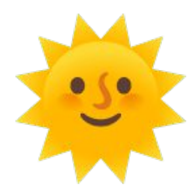
megabytes of memory

_crest[0..maxFrames]

HOW DO I KNOW THE BUFFERS ARE NOT IN A WORST-CASE POSITION? FAR FROM EACH OTHER AND IN ANY ORDER?



IN AN IDEAL WORLD



How to speed-up processing AND allocation?

```
struct MyDopeProcess
{
    void initialize(int maxFrames)
    {
        _alpha.reallocBuffer(maxFrames);
        _peak.reallocBuffer(maxFrames);
        _rms.reallocBuffer(maxFrames);
        _crest.reallocBuffer(maxFrames);
        // [...many buffers...]
    }
    // [...]
}
```

Three methods:

1. Allocators

Three methods:

1. **Allocators** (but Dplug doesn’t have those)

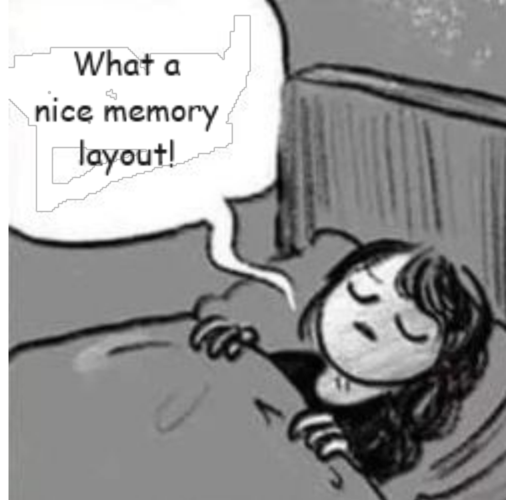
Three methods:

1. Allocators (but Dplug doesn't have those)
2. "Merged Allocation"

The “Merged Allocation”

One single merged allocation

_alpha[0..maxFrames]
_peak[0..maxFrames]
_rms[0..maxFrames]
_crest[0..maxFrames]



```
struct MyDopeProcess
{
    void initialize(int maxFrames)
    {
        _mergedAlloc.start();
        layout(_mergedAlloc, maxFrames);
        _mergedAlloc.allocate();
        layout(_mergedAlloc, maxFrames);
    }

    void layout(ref MergedAllocation ma, int maxFrames)
    {
        ma.allocArray(_alpha, maxFrames);
        ma.allocArray(_peak, maxFrames);
        ma.allocArray(_rms, maxFrames);
        ma.allocArray(_crest, maxFrames);
    }

private:
    MergedAllocation _ma;
}
```

Place buffers after NULL

Place buffers after alloc

MUST NOT initialize buffers here, just says how much memory you want

It's a struct, so automatic reclaim on ~this

The “Merged Allocation”

- **Usually faster to process and allocate nearby buffers**
- Alignment control with padding bytes

- **BUT Not always faster** than malloc for processing, often malloc has excellent locality

See also: https://p0nce.github.io/d-idioms/#The-merged-allocation-optimization

Three methods:

1. **Allocators** (but Dplug doesn't have those)
2. "Merged Allocation"
3. "Fixed Allocation"

- **Why not use buffer-splitting?**

- 1. Use the maxFramesInProcess() callback to limit the maximum number
of frames you receive.

THEN

2. Use regular static arrays, as stack variables or fields.

```
class MyDopeProcess
{
    enum int MAX_POSSIBLE_MAXFRAMES = 128;

    void initialize(int maxFrames)
    {
        assert(maxFrames <= MAX_POSSIBLE_MAXFRAMES);
    }

    override int maxFramesInProcess() pure const
    {
        return MAX_POSSIBLE_MAXFRAMES;
    }

private:
    MergedAllocation _ma;
    float[MAX_POSSIBLE_MAXFRAMES] _alpha;
    float[MAX_POSSIBLE_MAXFRAMES] _peak;
    float[MAX_POSSIBLE_MAXFRAMES] _rms;
    float[MAX_POSSIBLE_MAXFRAMES] _crest;
}
```

The “Fixed Allocation”

Same allocation/place than owning objects, or the stack.

_alpha[0..MAX_POSSIBLE_MAXFRAMES]

_peak[0..MAX_POSSIBLE_MAXFRAMES]

_rms[0..MAX_POSSIBLE_MAXFRAMES]

_crest[0..MAX_POSSIBLE_MAXFRAMES]

The “Fixed Allocation”

- Zero allocation potentially, controllable layout.

- **BUT Dangerous for space,** because space might be limited in a (unknown capacity) thread stack.
- **Dangerous for locality**: maxFrames might be even smaller than specified, leading to wasted space.
- **Even worse for locality:** Large objects and T.init, pessimized distance.
- Smaller buffer size usually lower performance below 128/256

Three methods:

1. **Allocators** (but Dplug doesn't have those)
2. "Merged Allocation"
3. "Fixed Allocation"

That said, malloc is a pretty good allocator.

Any nice idioms you want to share?