Optimizing rav1e

Luca Barbato

Who am I?

- Luca Barbato
  - **rav1e** and **dav1d** contributor among many other open source software.
  - Contacts
    - lu_zero@gentoo.org / lu_zero@videolan.org
    - https://twitter.com/lu_zero_
    - https://github.com/lu-zero

We will talk about rav1e and optimization

- rav1e is an AV1 encoder
  - rav1e is written in Rust
  - With a fair amount of arch-specific SIMD
    - Some written using stdarch intrinsics
    - Lots shared with dav1d and written in plain assembly
  - A good deal of multi-threaded code
    - Most leveraging rayon
- We will see what tools helped in speeding up rav1e and how we proceeded about it.

Why optimizing?

- To enable some use-case
  - Optimizing for size so your application fits within some storage constraint
  - Optimizing for minimal latency so your application can be used in real-time scenarios
  - Optimizing for the least amount of cpu usage, so your application will not drain your mobile battery or burn your device to a crisp.
- To make some use-case cheaper
  - Optimizing for overall throughput so your application can process the largest amount of data for the amount of resources that your budget let you afford.
- To prove how smart you are
  - Ok, this is not a good reason…

- Every encoder may target different use-case
  - Best quality (according to some quasi-objective metric)
    - No matter the amount of time, memory and cpu used.
  - Single encoding speed
    - No matter the amount of resources, you want that the overall process takes the least amount of time.
  - Lowest possible latency
    - The time between the video frame entering the encoder and the packet containing it must be the least possible.
  - Maximum throughput
    - Largest amount of frames processed per amount of resources (memory and cpu) used.

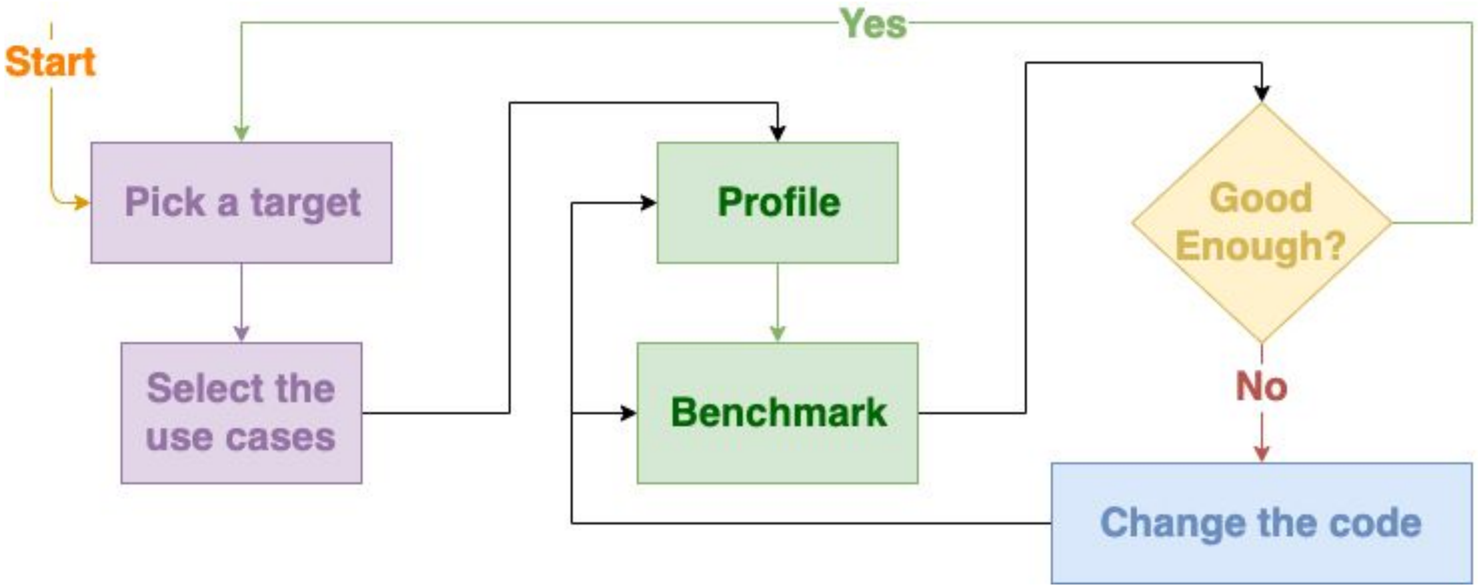
With rav1e we want to provide a sweet spot among the 4, often conflicting, targets above.

Optimizing is an iterative process

1. Prepare the use-cases you want to optimize for
2. Measure their behavior
   - Change the code and go back to *2.*
   - If the results are good enough you go back to *1.*
     - and change your optimization target

Let's unpack it a little.

Optimizing is an iterative process



You may try to optimize for a number of **metrics**

- Speed
  - Single execution time
  - Latency
- Memory usage
  - Maximum resident set
  - Allocation count
- Throughput
  - Number of results per unit of time
  - Number of results per resource spent
- Quality
  - Application-dependent

For rav1e our main trade-off point is between **Quality** and **Speed**

- We try to alternate the main focus every release
  - 0.2.0 was mainly about **speed**
  - 0.3.0 was mainly about **quality**
  - 0.4.0 will be about **throughput** and latency
- Yet we try to keep a **balanced** approach
  - We try to keep the amount of memory used within reason
  - We try to not require too many cores
  - The quality/speed trade-offs are often re-evaluated

Notwithstanding the metric, you have to come up with **good use-cases**

- It should represent well the common usage of your application
- It can be **non-exhaustive**
  - Coverage 99% is unnecessary
  - Coverage 50%+ is nice to have
- It should the **right amount** of time and resources to execute, but not more than that.
  - Encoding hours of video vs encoding the right amount of frames to trigger the scene-change detection logic enough times.
  - Encoding 8k videos vs encoding 4k videos or even 1080p videos.

- For video encoding there are collections of short and not so short raw samples that are used to do quality and performance comparisons among encoders
  - We just have to select a subset that is well representative
    - The easiest way to do that is to run some encodes and measure the code coverage
- For rust there are a number of tools available
  - rustc has an not-yet stable *-Zprofile* flag that produces information that can be parsed and formatted by grcov, gcovr and similar tools.
  - kcov and cargo-kcov provide similar information without the need to have instrumented binaries. (It is 2x-3x faster than -Zprofile, but less precise)
  - tarpaulin is a pure-rust solution, but currently supports only linux on x86_64 and pure-rust binaries. (Sadly does not work for my use-case)

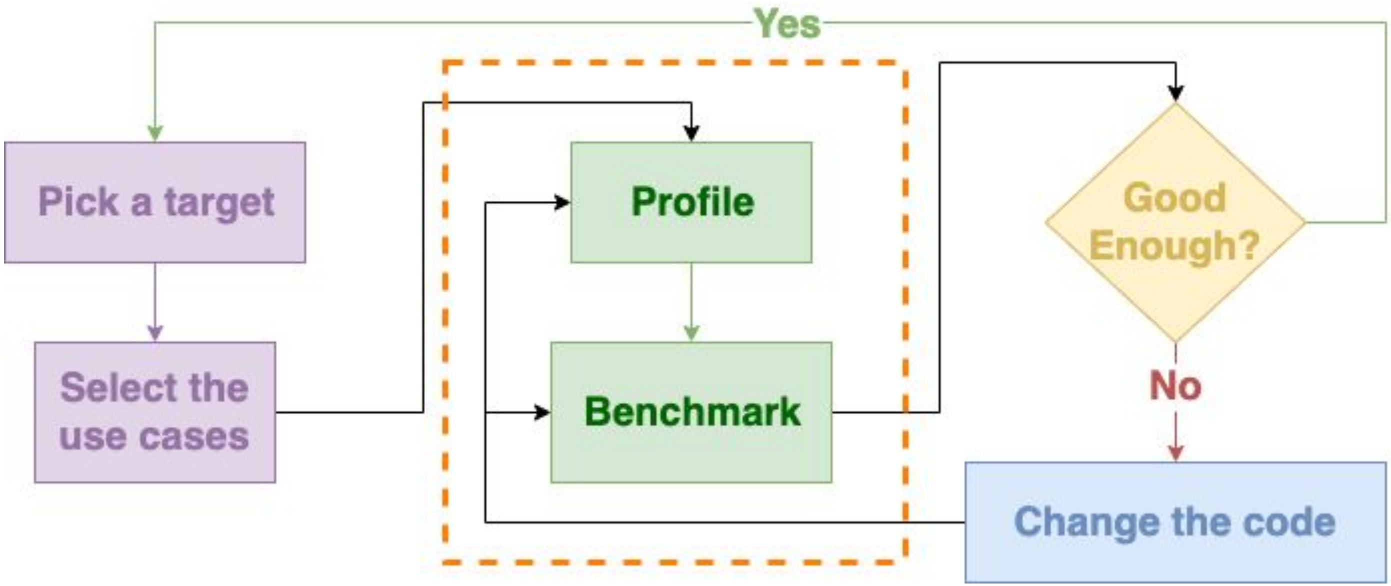
Optimization target selection

```
$ export PROJECT=rav1e
$ export RUSTFLAGS="-C target-cpu=native \
-Zprofile -Ccodegen-units=1 -Cinline-threshold=0 \
-Clink-dead-code -Coverflow-checks=off -Zno-landing-pads"
$ export CARGO_INCREMENTAL=0
$ cargo +nightly run --release -- $SAMPLE -s $SPEED --tiles $TILES -o /dev/null
$ gcovr -r . --gcov-executable "llvm-cov gcov" --filter src/
```

```
$ cargo build --release
$ kcov --include-path=src/ /tmp/kcov target/release/rav1e $SAMPLE -s $SPEED --tiles $TILES -o /dev/null
```

Once we have our set of use-cases we have to profile it

- And possibly produce benchmarks out of it



I split the process of measuring in two

- **Profiling** the full use-case execution instrumenting the application
  - Figuring out what are the slow paths
  - Getting a list of potential places to optimize first
- Writing and executing more precise **benchmarks** to measure how the selected code-paths behave
  - The profiling instrumentation **slows down** the execution potentially many-folds
  - Executing the **benchmarks** should take much less time, by few orders of magnitude

NOTE: Doing well in microbenchmarks may not translate in doing as well in the actual use case

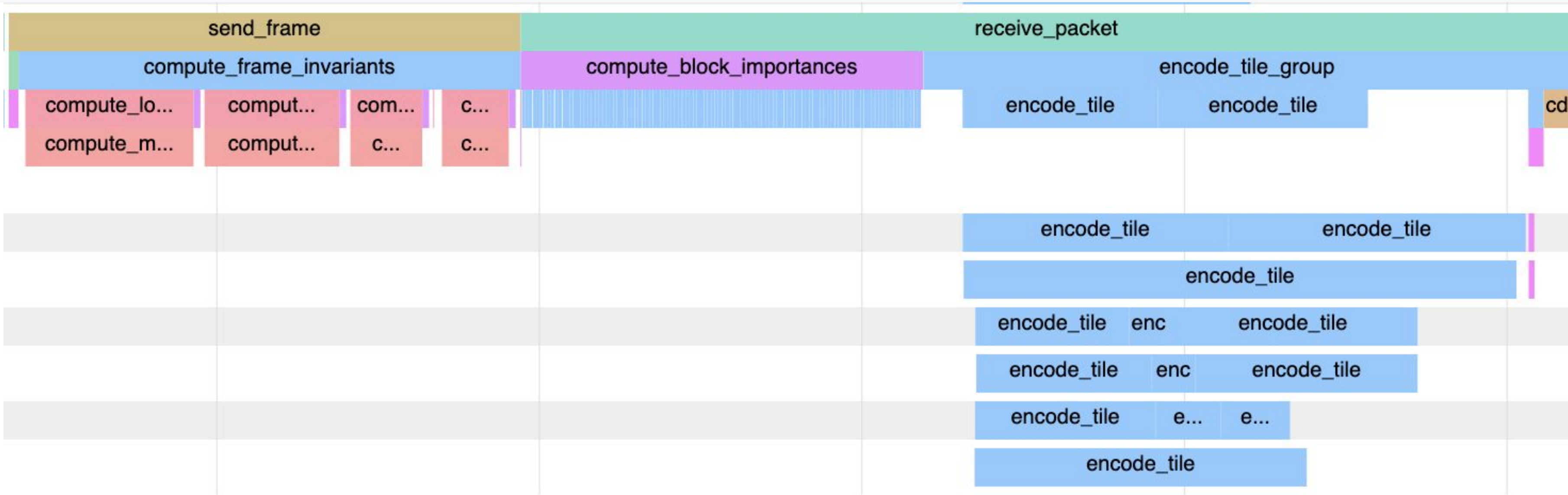
We have a number of tools we can use to extract useful information

- hyperfine is quite useful to get you an overall measurement and its noise.
  - If the variance is low you can do without having too many runs.
- (cargo-)flamegraph produces nice interactive flamegraphs
  - It uses under the hood perf or dtrace, so it supports a good variety of systems.
- not-perf is a pure-rust alternative to perf
  - It produces similar flamegraphs and it can be more viable than perf sometimes.
- uftrace is a faster function-tracer that works coupled with -Zinstrument-mcount
  - Supports only Linux on x86(_64) and ARM/AArch64, and produces all sort of useful data presentation including flamegraphs and chrome-tracing json
- cargo-instruments makes even easier to use Xcode Instruments.

- We want to the amount of time spent per-function, for all the functions.
- We want to profile our corpus at least once
  - If the top 10 functions are always the same we can select a reduced use-case
- If possible we should prepare a unit-test-like benchmark
  - If it is too much effort we can just use the reduced testcase
    - We can use lightweight probes instead of fully profile
- Once we start using threads we should try to be aware of the critical path
  - Every improvement in functions running in parallel has less global impact
    - The focus should move to the functions that are in the least parallelized paths first
  - Running in parallel sub-tasks from a tasks that is already parallelized requires additional care
  - Lightweight probes such as hawktracer come handy to visualize what is going on.

Profiling - Speed

```
$ cargo install hawktracer-converter
$ cargo run --release --features=tracing $SAMPLE --tiles $TILES -s $SPEED -o /dev/null
# produce a chrome-tracing compatible json
$ hawktracer-converter-rs -s trace.bin -o rav1e-$SAMPLE-t$TILES-s$SPEED.json
```



- Memory
    - gnu time and getrusage provide a quick way to get the overall maximum resident set for a single run.
    - malt provides a large amount of information regarding memory usage
        - Its web-ui is among the nicest available
        - It has multiple means to trace the memory allocation, allowing a large degree of platform support
    - memory-profiler is a linux-only memory tracer
        - It provides a rich web-ui and supports visualizing multiple traces
        - It supports only x86(_64), ARM/AArch64 and mips64.
        - Faster than the default malt, but not as straightforward to use.
    - cargo-instruments can be used to trace the memory usage on macOS.
    - heaptrack provides a really nice GUI that works great if you have KDE.
        - malt and memory-profiler both provides compatible outputs.

- We want to keep the maximum resident set to the minimum
    - The smaller it is the higher the number of concurrent instances
- We want to minimize the number of allocations as well
    - The higher the number, the higher the chance to fragment the memory
    - Allocating and deallocating in an hot path is highly disruptive
        - A syscall might be involved
        - You are almost certain to fragment the memory
        - Your cache access pattern might be ruined
- We want to make sure we do not leak memory
    - Leaking memory is safe and possible in rust, but unlikely.

memory-profiler does not come with a run-script like malt, so I you can come up with one like:

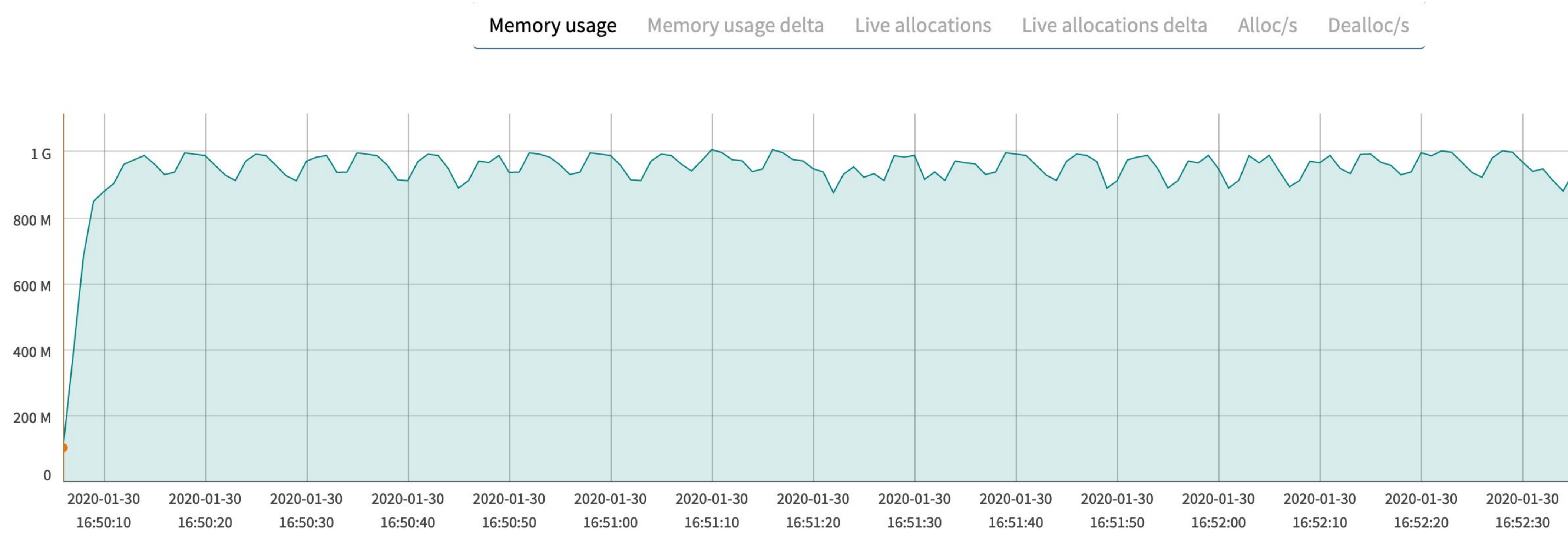
```
#!/bin/sh

MEM_PROF_LIB=/opt/memory_profiler/libmemory_profiler.so

LD_PRELOAD="${MEM_PROF_LIB}:${LD_PRELOAD}" "$@"
```

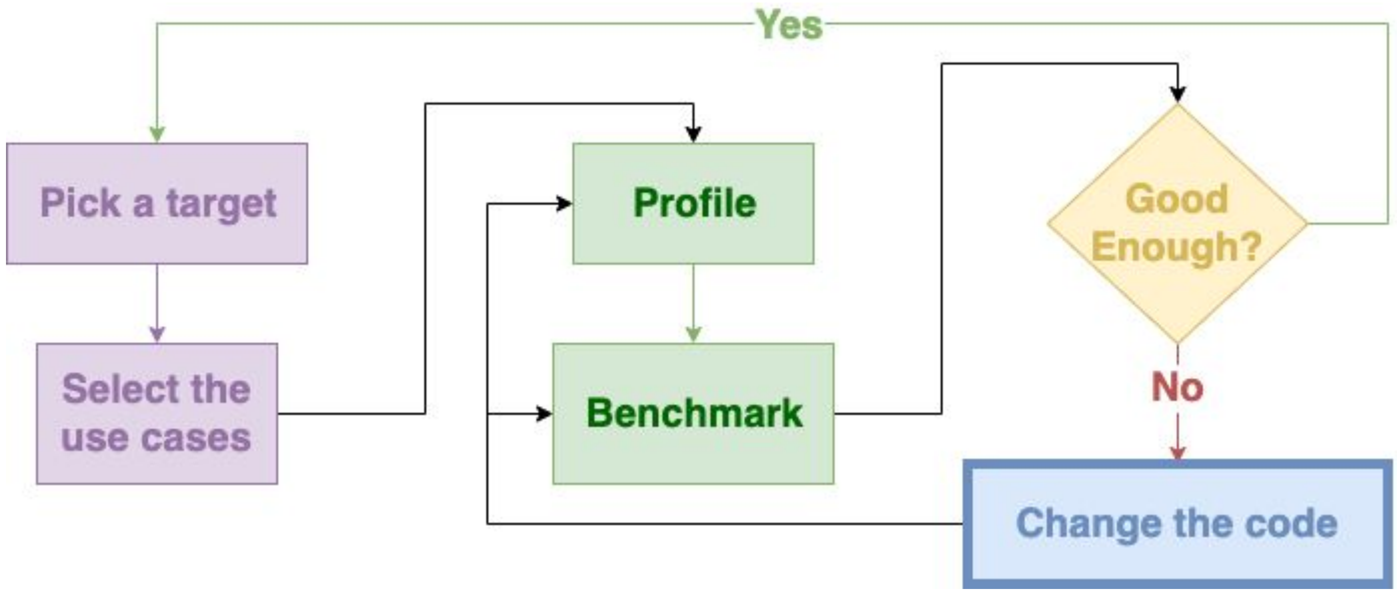
```
$ cargo build --release
$ memprof target/release/rav1e $SAMPLE --tiles $TILES -s $SPEED -o /dev/null
$ memory-profiler-cli server -p 8084 -i 0.0.0.0 memory-profiling_*.dat
```

Profiling - memory-profiler



The rust built-in benchmarking support is still currently in flux, within the pending testing framework overhaul

- As measuring speed or throughput goes criterion does provide a fairly rich API to build good micro-benchmarks and paired with critcmp gives fairly good results
  - Just make sure you disable the built-in bench support.
- There isn't anything as good to precisely measure the memory usage, to my knowledge, so using the memory profilers over the standard **tests** is the most viable solution.


Yes
Pick a target
Select the use cases
Profile
Benchmark
Good Enough?
No
Change the code

There are many strategies one could use, here is mine:

- Maximize the impact
  - Pick the easiest code-path among the top 5
    - Optimize and get some instant gratification
  - Iterate until all the functions at the top are similar metrics-wise
- Try to be conservative with the tradeoffs
  - Try first to get improvements w/out impairing other metrics.
  - Try to set some kind of budget, thinking of your ideal users.
- Always be ready to undo some early work
  - And to accept your work could be undone
    - *It is not disrespectful to delete code*

In order to be fast you have the following choices

- Use less resources
  - By improving the algorithm in use
  - By avoiding unneeded computation
- Use the same resources but in better ways
  - Leverage the SIMD extensions available
  - Cache locality optimization
- Use more resources
  - Multithread processing

A good deal of code is inherently parallel.

- The **rav1e** works together with the **dav1d** in sharing the SIMD assembly optimized routines that are common between encoders and decoders, nasm-rs and cc-rs make the integration fairly easy.
- Encoder-specific codepaths are usually optimized using the rust **arch-specific** intrinsics.
- Since the Rust language provides more chances for the compiler to unroll and auto-vectorize a good part of the codebase it is compiled to **SSE2** instructions on x86_64 and **NEON** instructions on AArch64.
  - Using -C target-features=+avx2,+fma produce an even **faster** binary, with the shortcoming of working only on recent CPUs.

- Writing multithreaded code is usually cumbersome and error prone.
  - In rust most of the common pitfalls are just **impossible**.
  - The standard library offers already good primitives, including easy to use channels.
- There are external crates that make even easier to make high performance multi-threading implementations.
  - parking_lot replacing the standard library primitives with better ones.
  - crossbeam sporting better channels and additional primitives.
  - rayon provides an easy to use threadpool and let you convert normal Iterators in parallel iterators in literally **one line of code**.

This is our main encoding loop

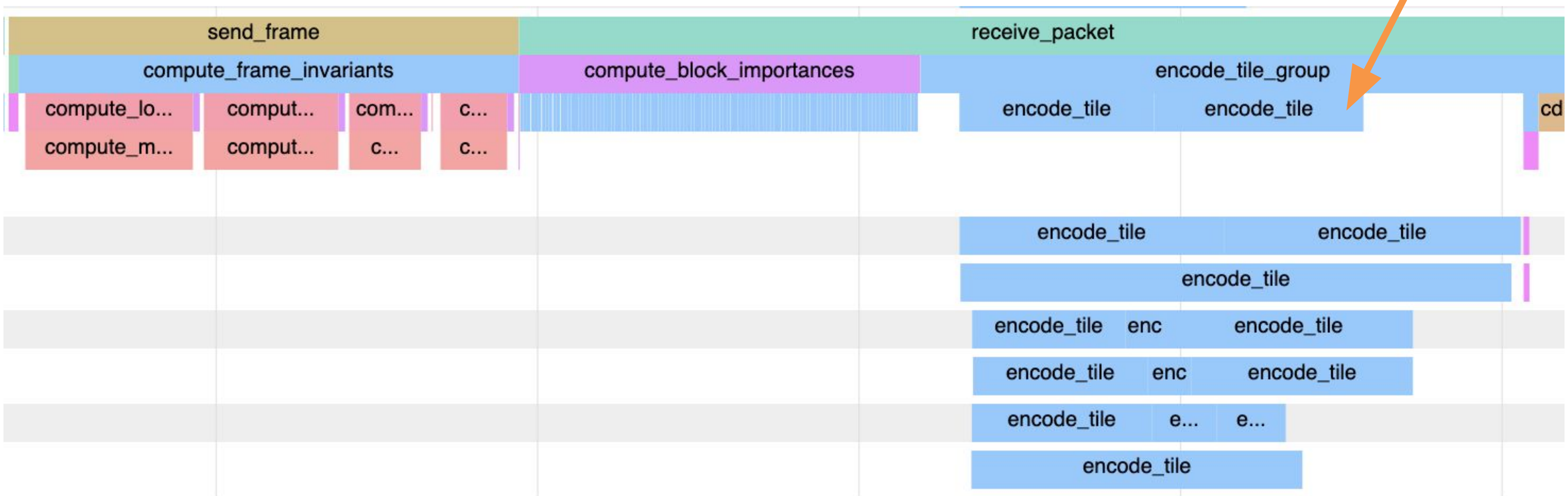
```
let (raw_tiles, tile_states): (Vec<_>, Vec<_>) = ti
  .tile_iter_mut(fs, &mut blocks)
  .zip(cdfs.iter_mut())
  .collect::<Vec<_>>()
  .map(|(mut ctx, cdf)| {
    let raw = encode_tile(fi, &mut ctx.ts, cdf, &mut ctx.tb, inter_cfg);
    (raw, ctx.ts)
  })
  .unzip();
```

Changing the code - rayon

This is our main encoding loop, multithreaded

```
let (raw_tiles, tile_states): (Vec<_>, Vec<_>) = ti
  .tile_iter_mut(fs, &mut blocks)
  .zip(cdfs.iter_mut())
  .collect::<Vec<_>>()
  .into_par_iter()
  .map(|(mut ctx, cdf)| {
    let raw = encode_tile(fi, &mut ctx.ts, cdf, &mut ctx.tb, inter_cfg);
    (raw, ctx.ts)
  })
  .unzip();
```

This is our main encoding loop, multithreaded



Changing the code - rayon

Adding **par_iter()** requires that the Iterator obeys certain constraints

- It is working on **Send** data types
- It is not mutating variables captured by the closure

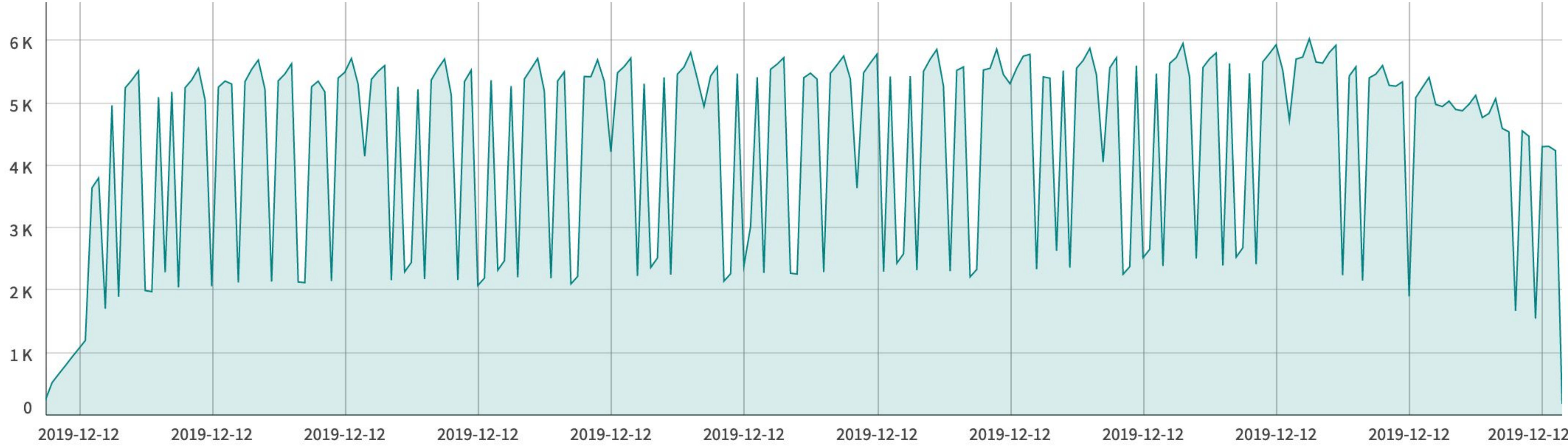
That may require some initial refactor but it usually pays off well.

Currently we are using crossbeam channels to experiment with additional levels of parallelism and provide the users an alternative channel-based API.

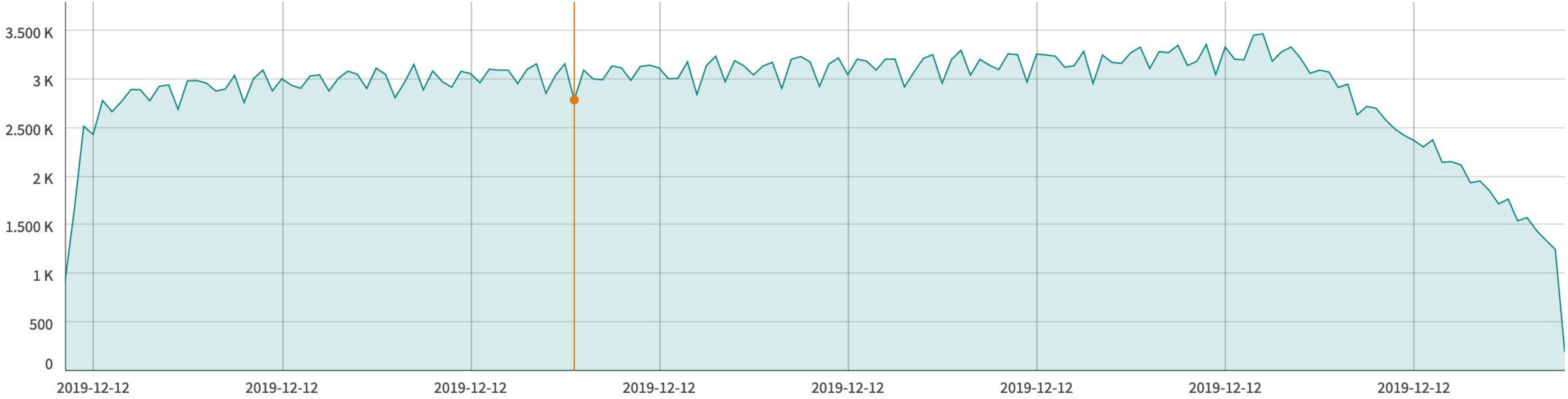
Optimizing the memory usage is usually less interesting

- Most of the dynamic allocation come from Vec-overuse
  - ArrayVec/SmallVec/TinyVec let you use the same Vec API but using a stack-allocated fixed size array as backing storage.
    - This makes the memory access cheaper
    - Gets you less allocations
    - Depending on your workload does not increase a lot the resident set.
  - arraydeque and similar richer stack-based data structures might come handy
    - But they might be less used and tested, so use additional care.
- You might have unneeded intermediate buffers
  - In this case you might use creatively the standard Iterator trait
  - itertools may come handy as well.

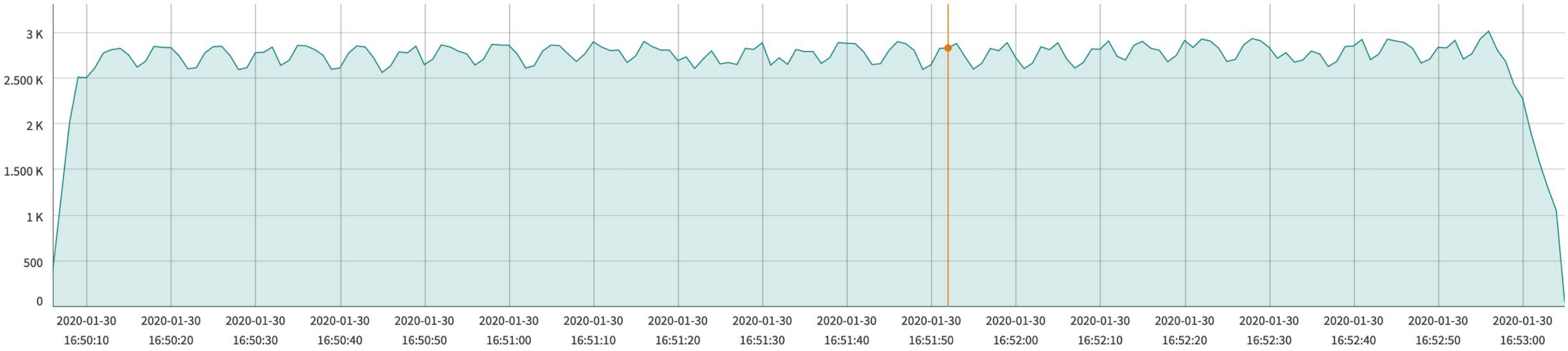
Live allocations for rav1e 0.1.0: **6039 peak**



Live allocations for rav1e 0.2.0-p20191201: **3500 peak**



Live allocations for rav1e current (da62d7a46): **3000 peak**



Thank You