Innovations in H.264/AVC software decoding

Thibault Raffaillac
Postdoctoral researcher, INRAE Montpellier

4 February 2024

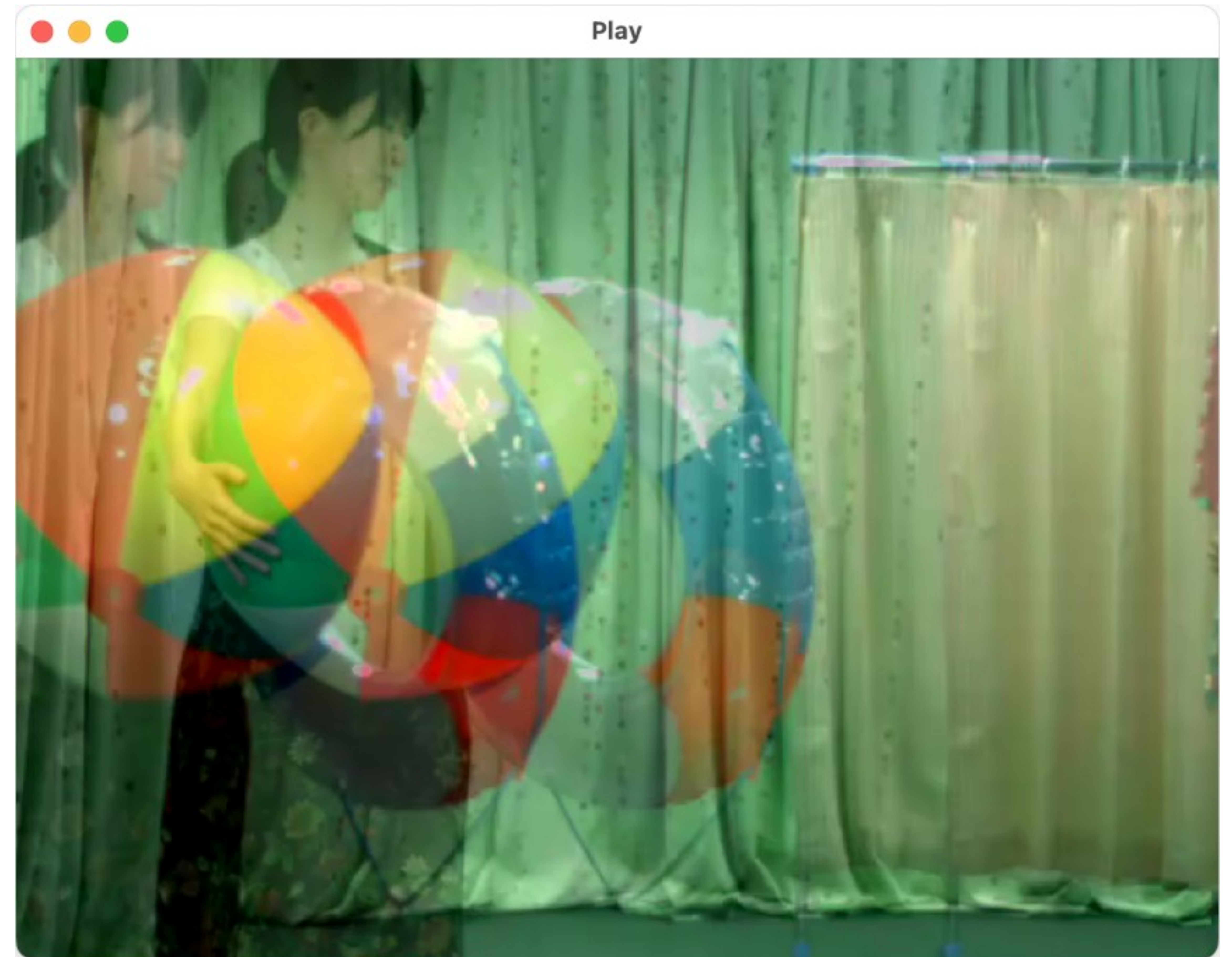
Introduction - edge264

Software decoder for H.264/AVC

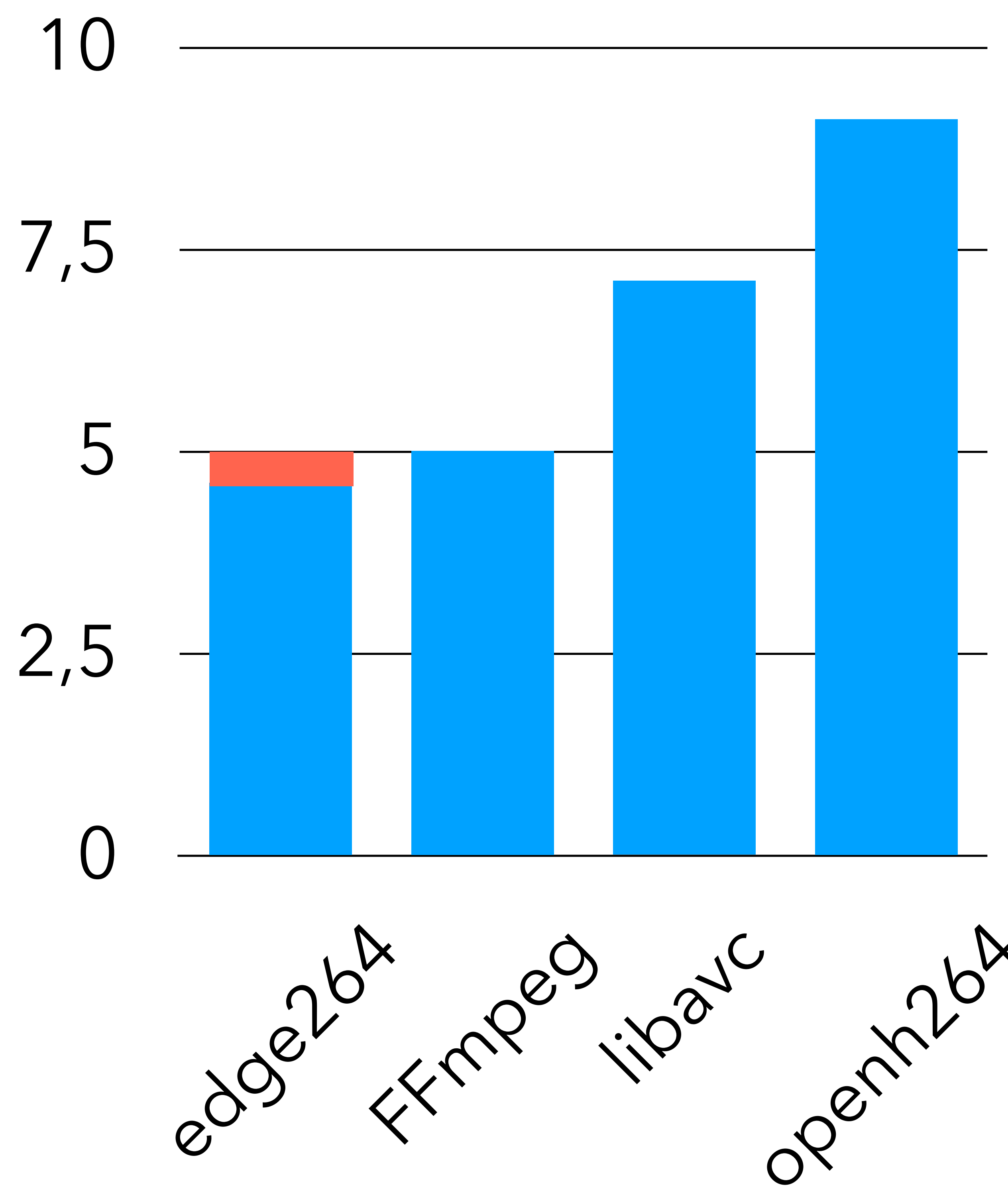
Experimental since 2013, working towards stable release since 2020

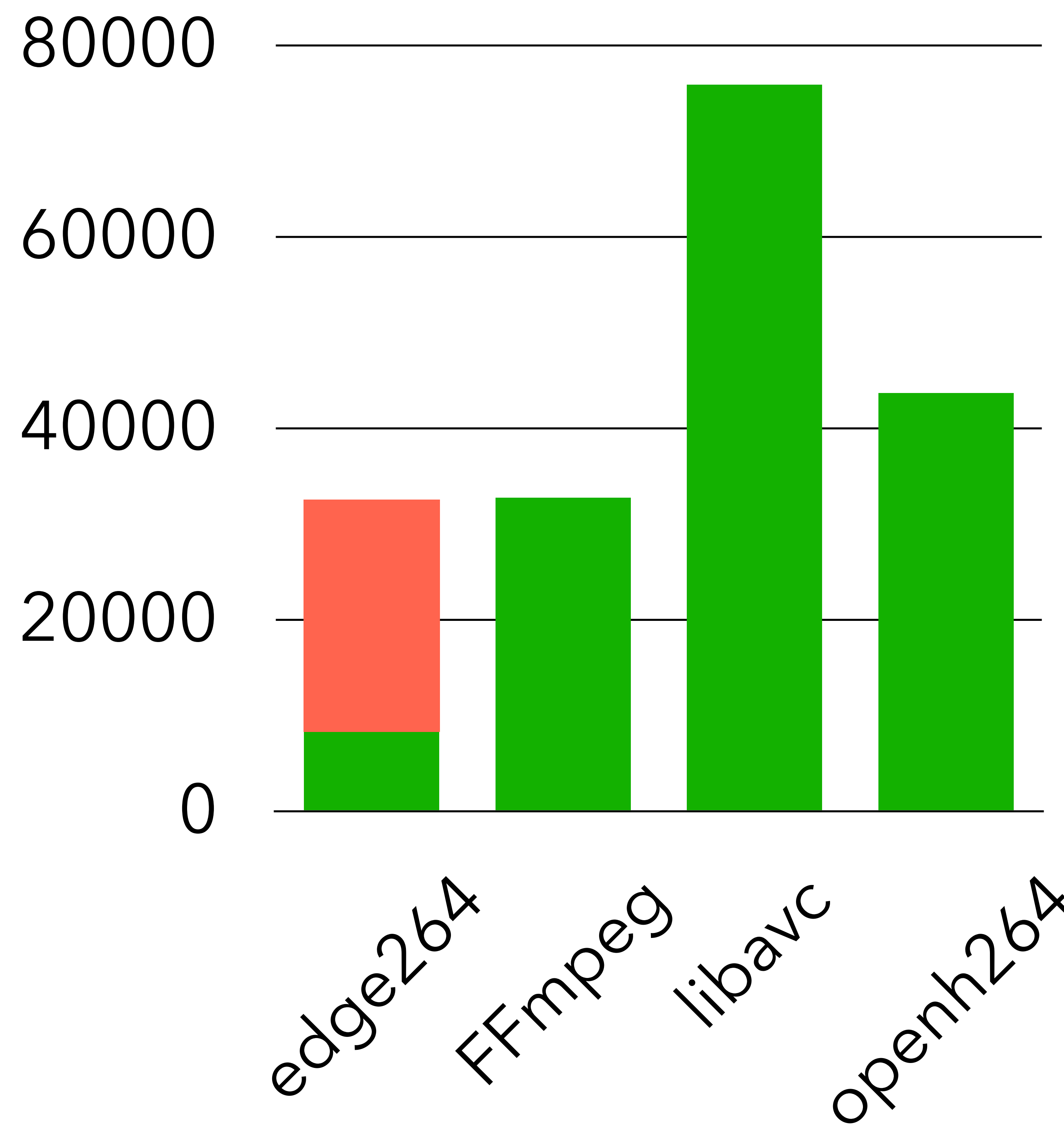
Supports Intel 32/64 with SSSE3

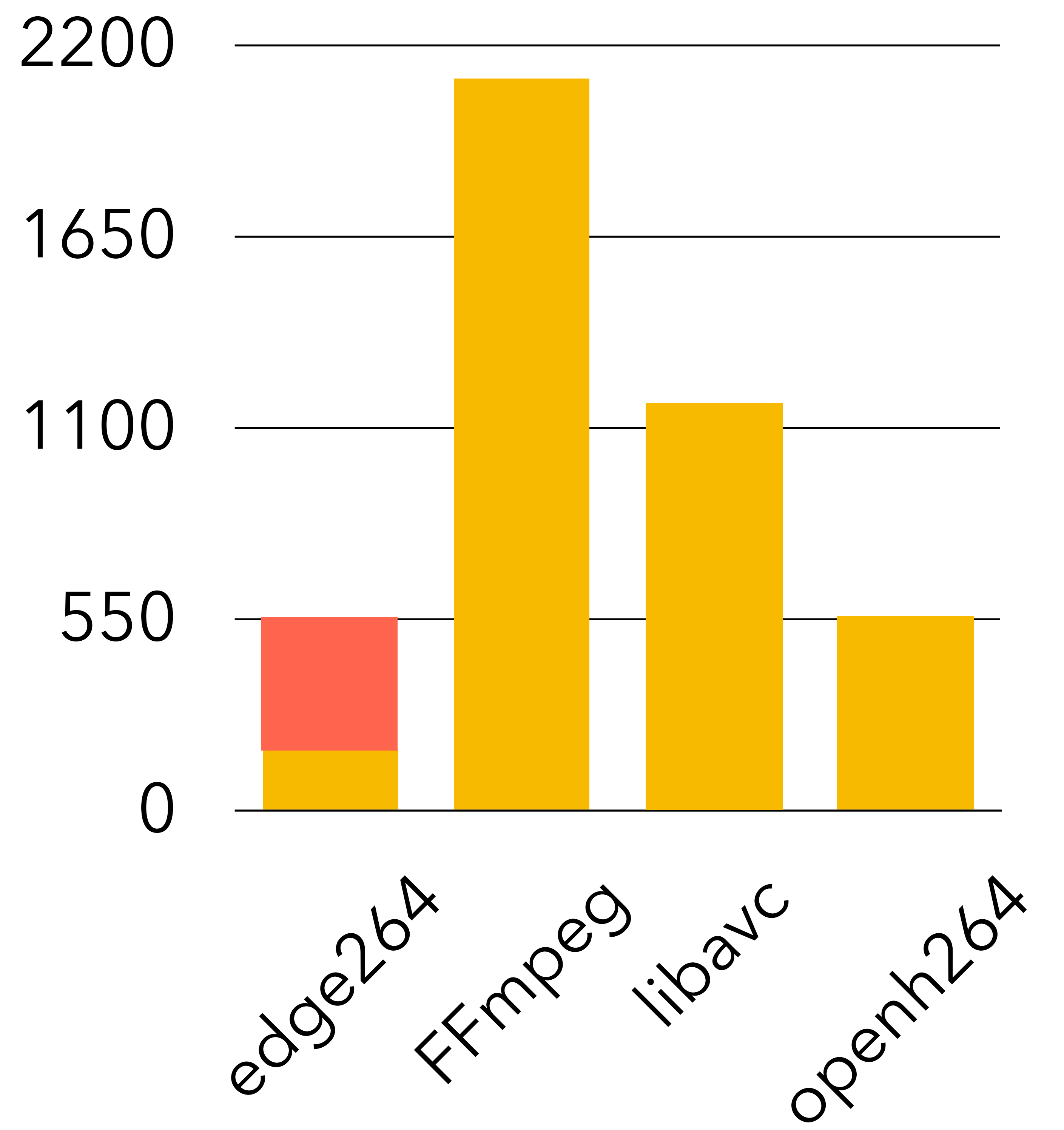
Profiles: **Progressive High** & **MVC** (**Bluray 3D** 🎉)



Introduction - Benchmarks





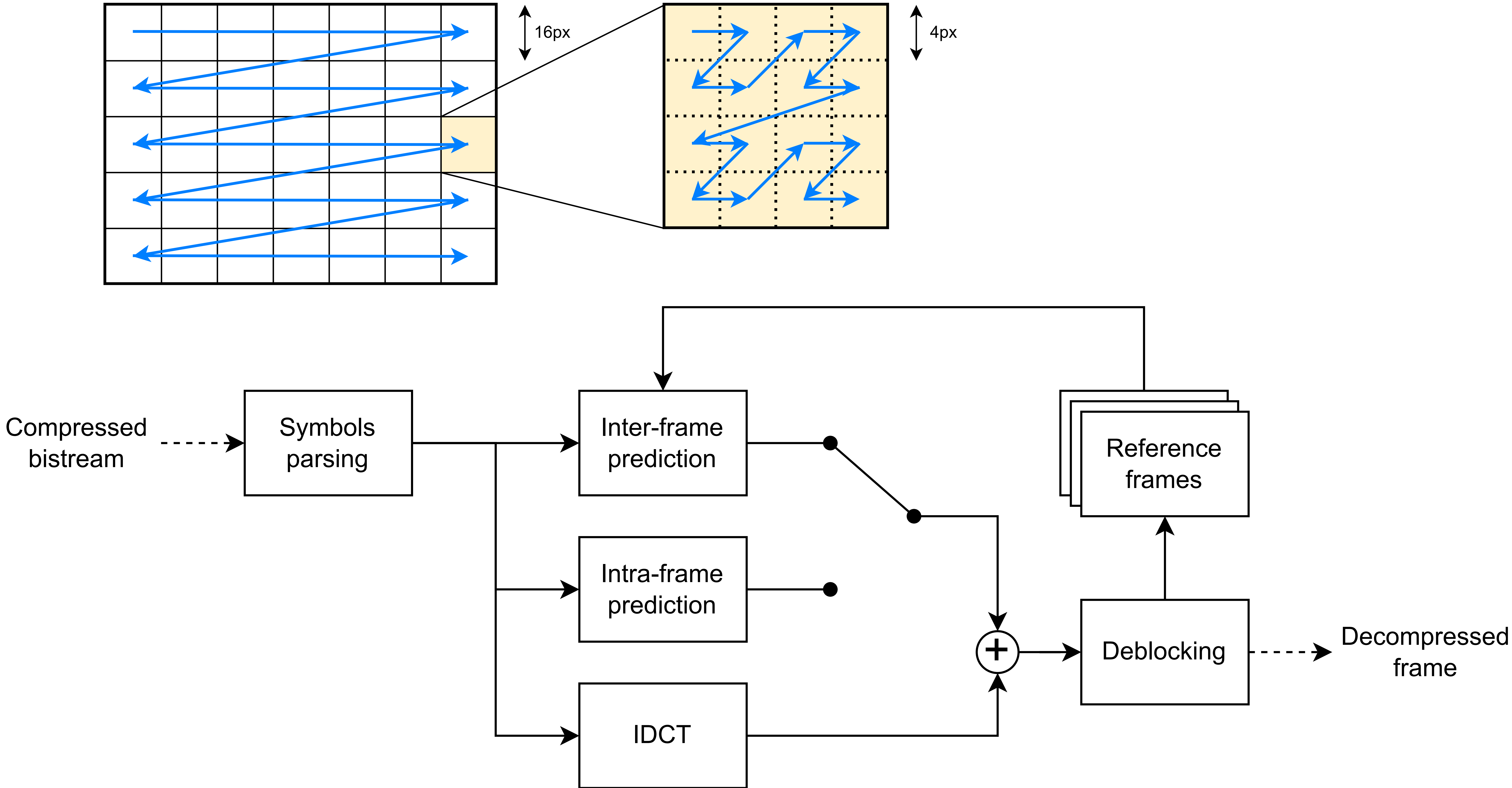


10% faster

3x lighter

3x lighter

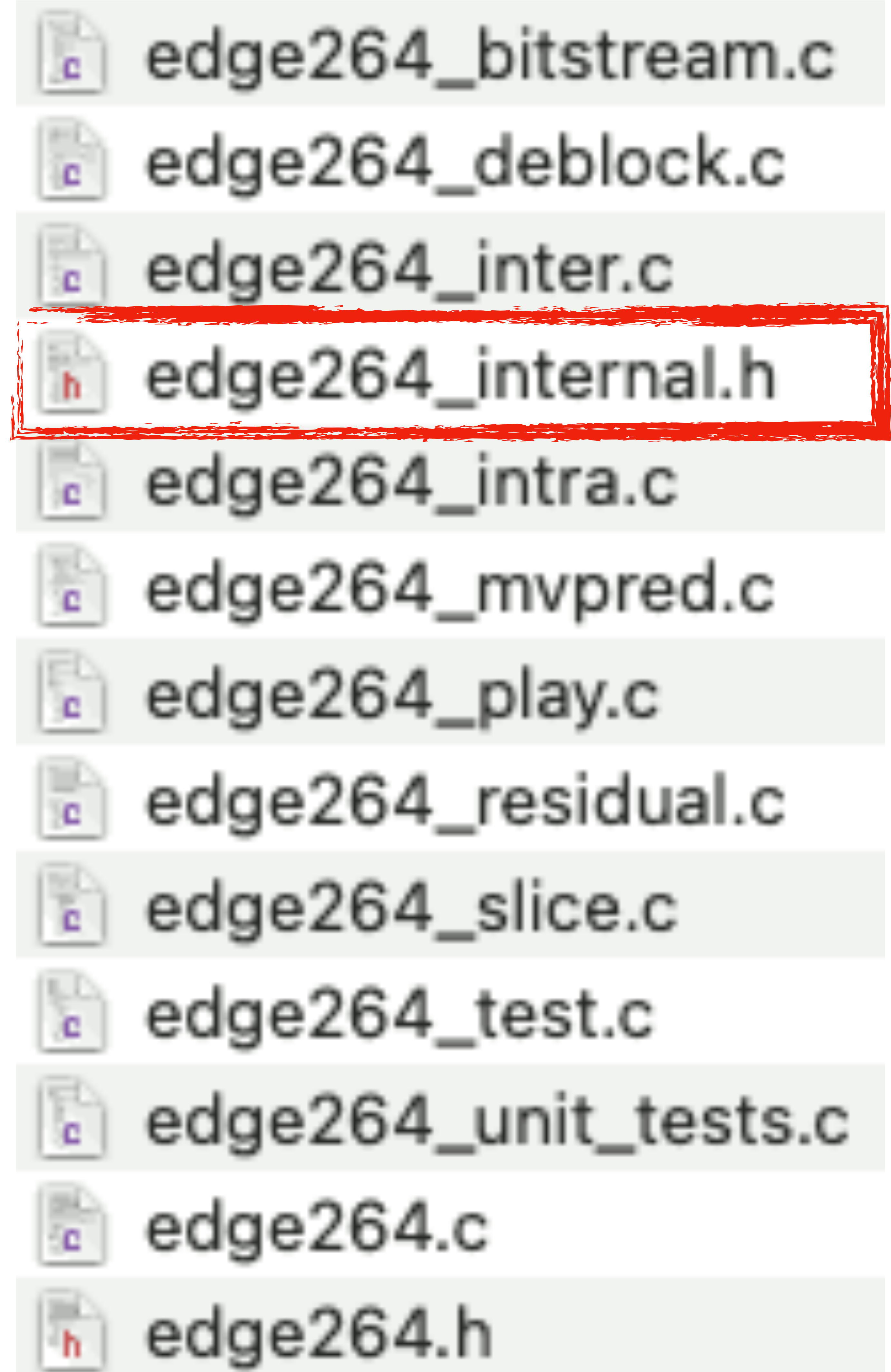
Introduction - H.264/AVC decoding



Technique 1 - Single header file

Merging all internal definitions into a **single file**:

- struct typedefs
- inline functions and macros (e.g. min/max)
- exported functions for each .c file
- SIMD typedefs
- SIMD inline functions and macros

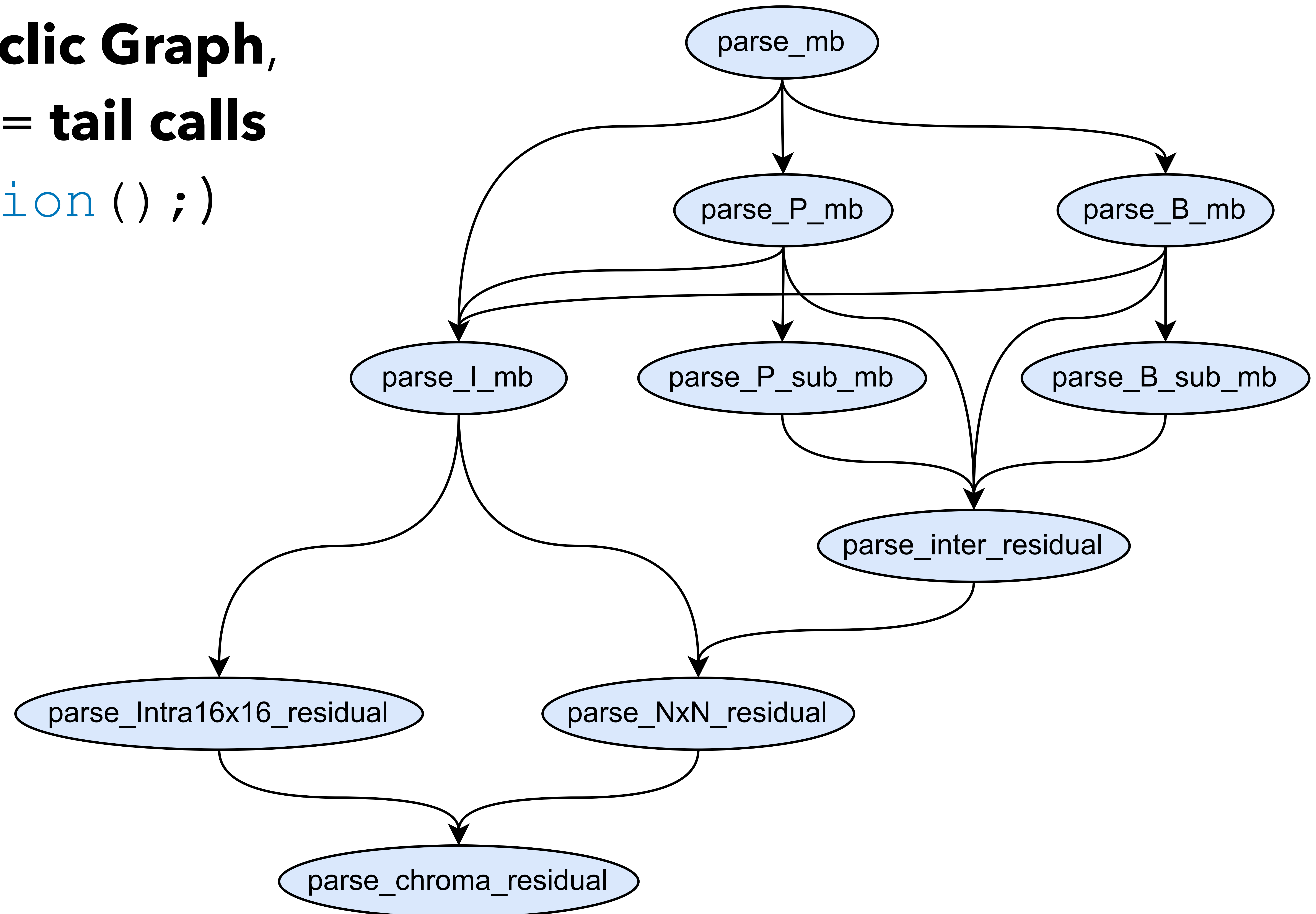


Technique 2 - Code blocks instead of functions

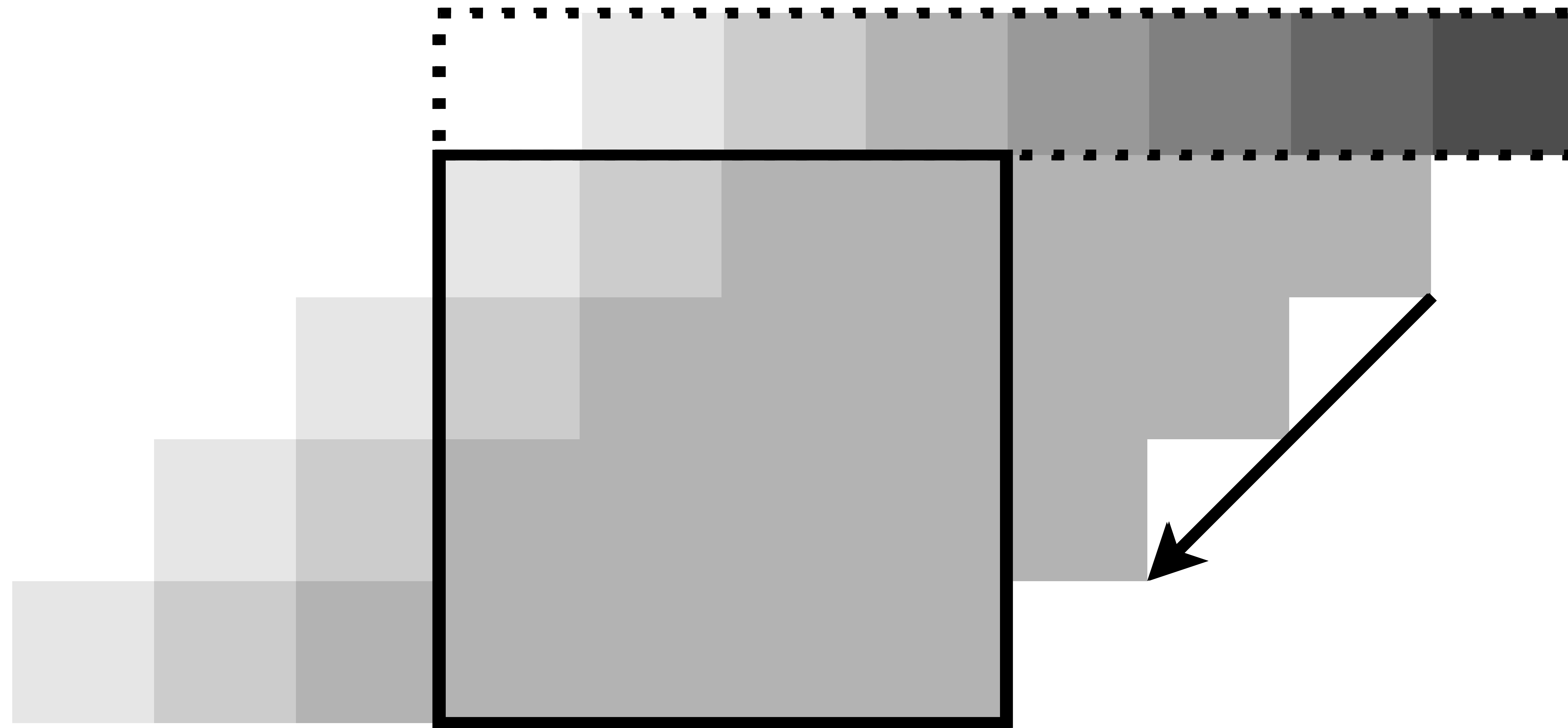
Code flow = **Directed Acyclic Graph**,
nodes = **functions**, edges = **tail calls**
(i.e. `return next_function();`)

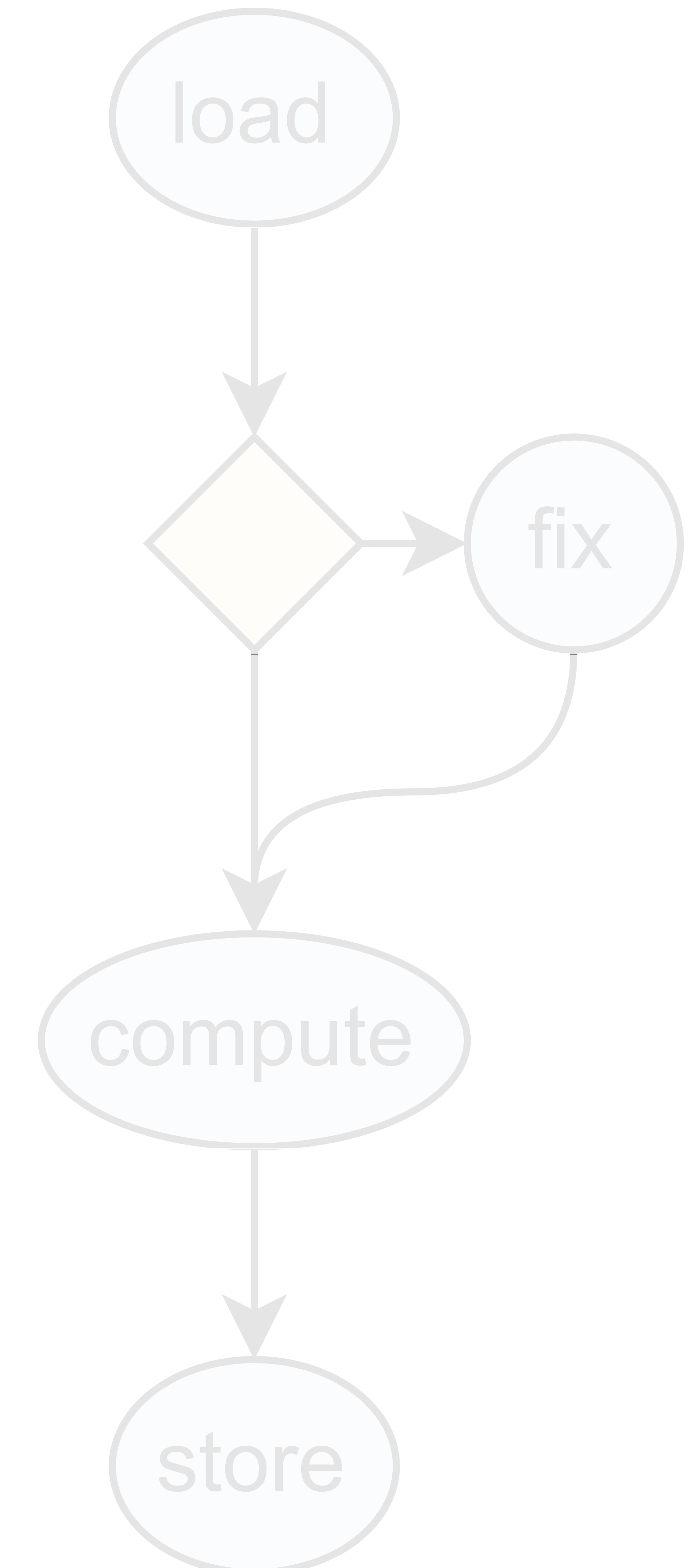
No inlining

Less use of parameters

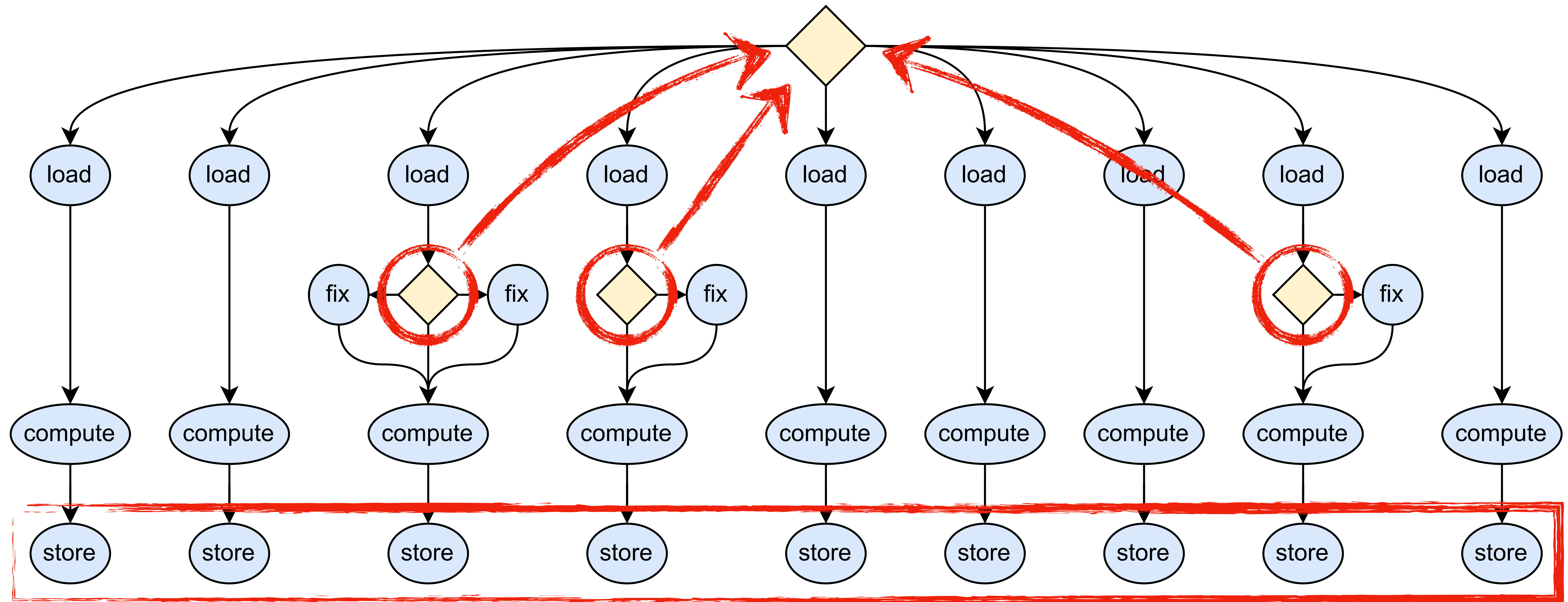


Technique 3 - Tree branching



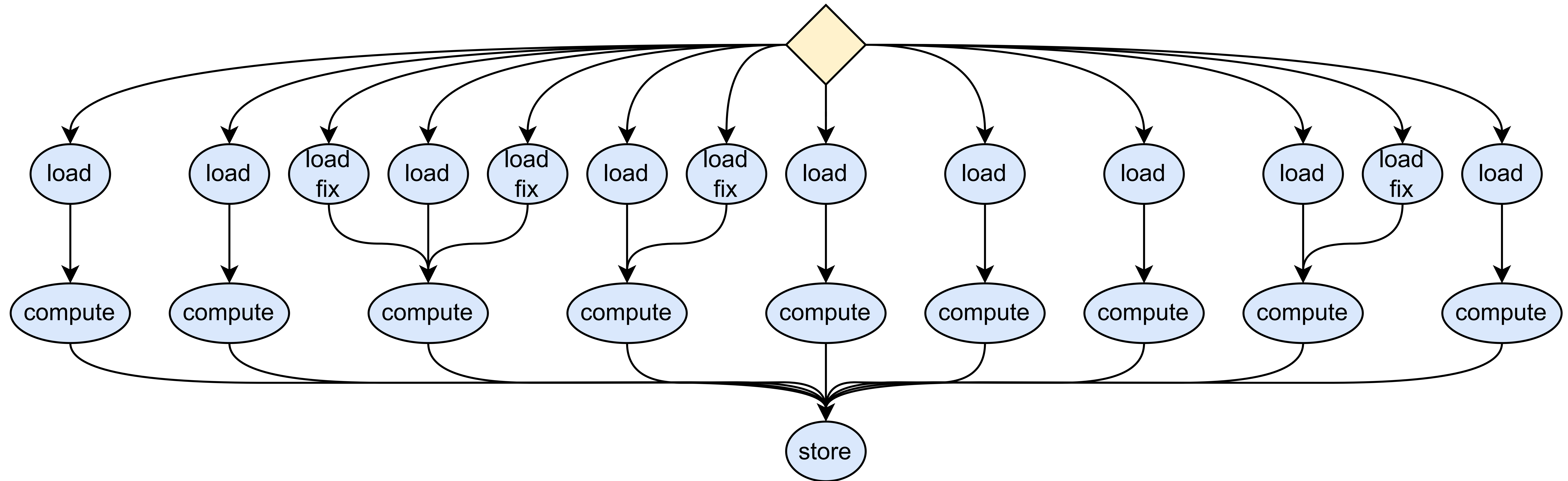


Technique 3 - Tree branching



Branching with array of **function pointers**

Technique 3 - Tree branching



Branching on leaves with **switch**, on branches with **goto**, on trunk with **break**

Intra 4x4 → 14 leaves, Intra 8x8 → 32 leaves

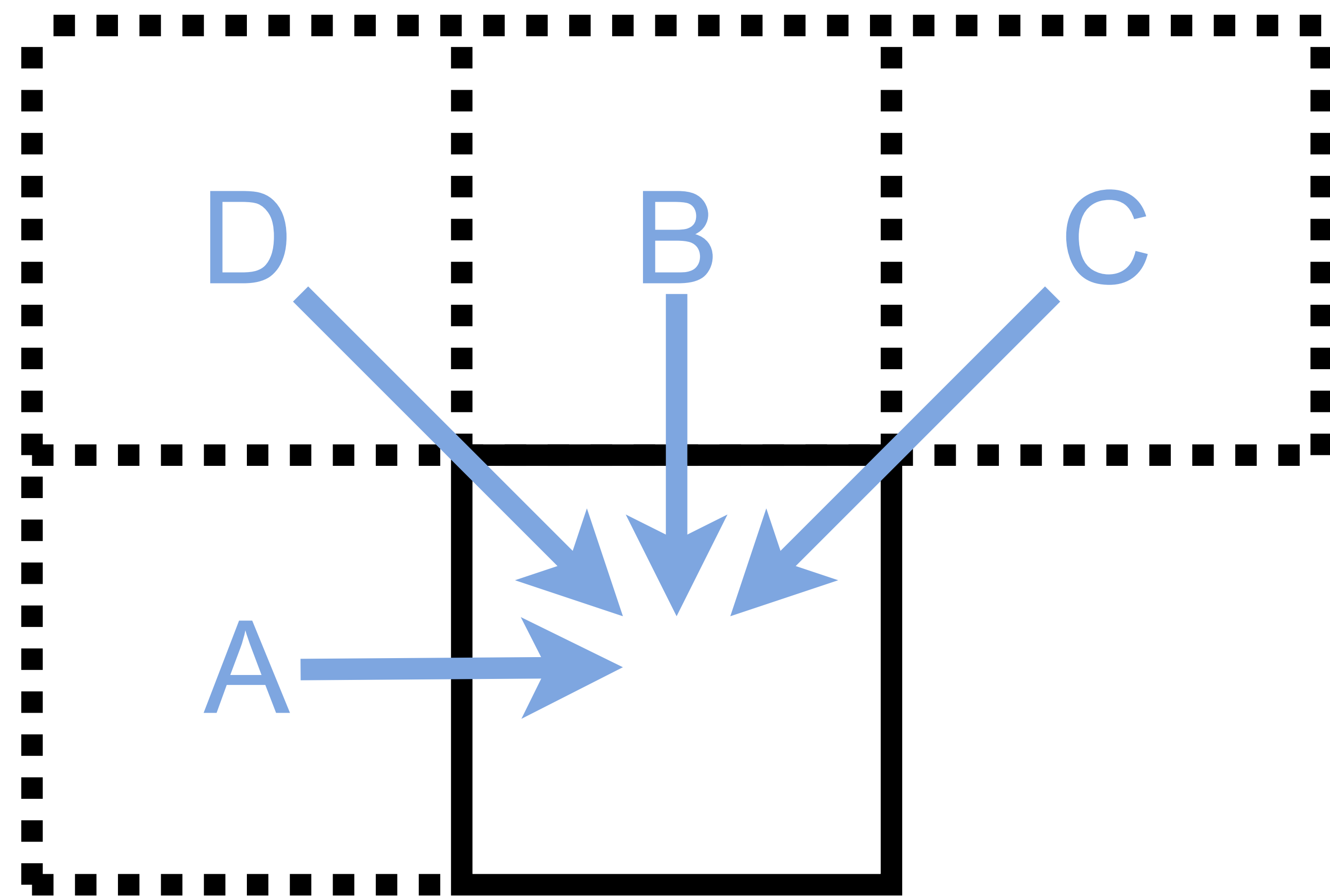
Technique 4 - Global context register

All context data resides in a `Edge264_ctx` structure passed to *every function*

Its pointer is stored in a **Global Register Variable** if possible (GCC)

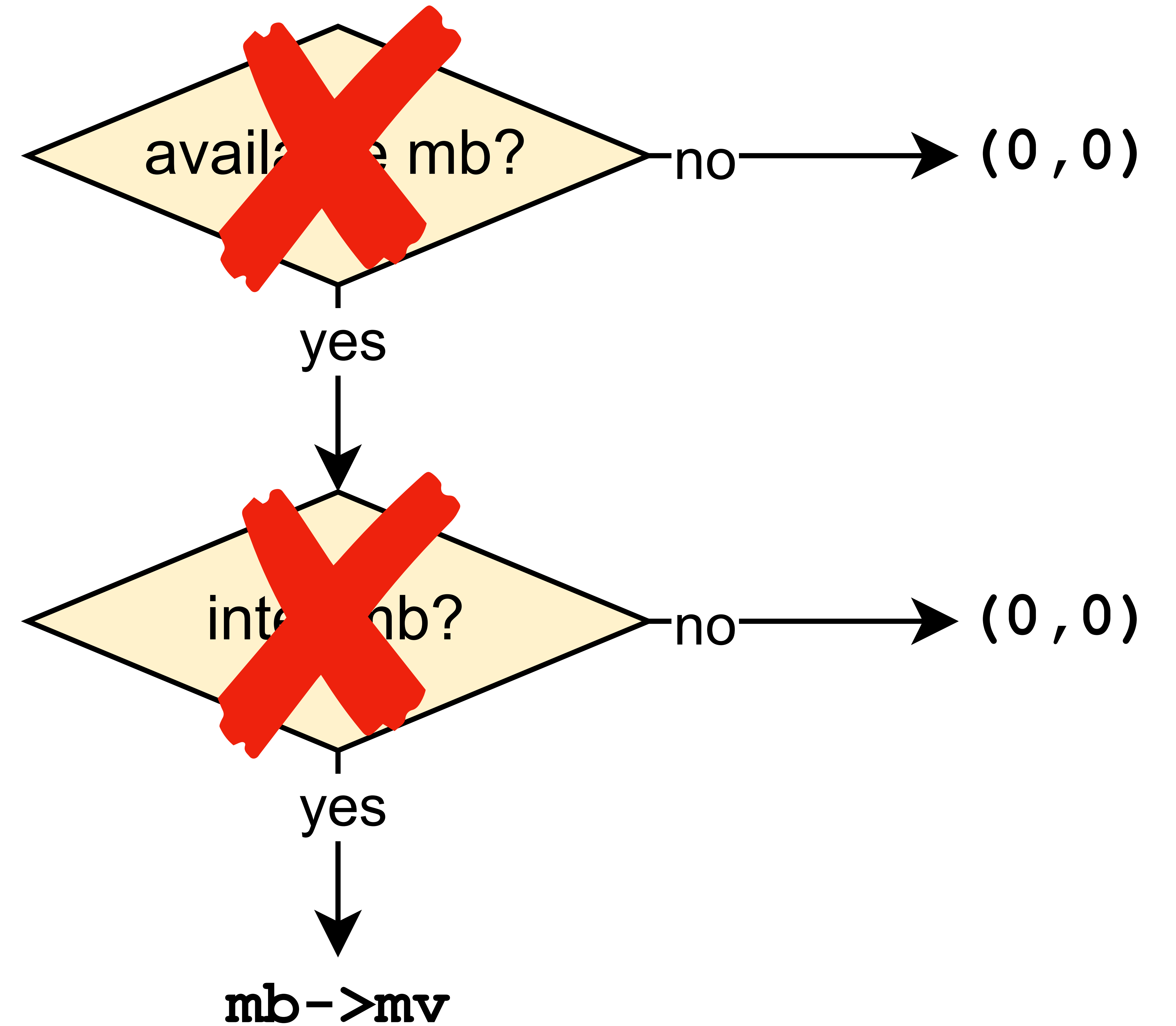
```
#if GCC
    register Edge264_ctx * restrict ctx asm("ebx");
    #define CALL(f, ...) f(__VA_ARGS__)
#else
    #define CALL(f, ...) f(ctx, ## __VA_ARGS__)
#endif
...
CALL(parse_I_mb, 0);
```

Technique 5: Default neighboring values

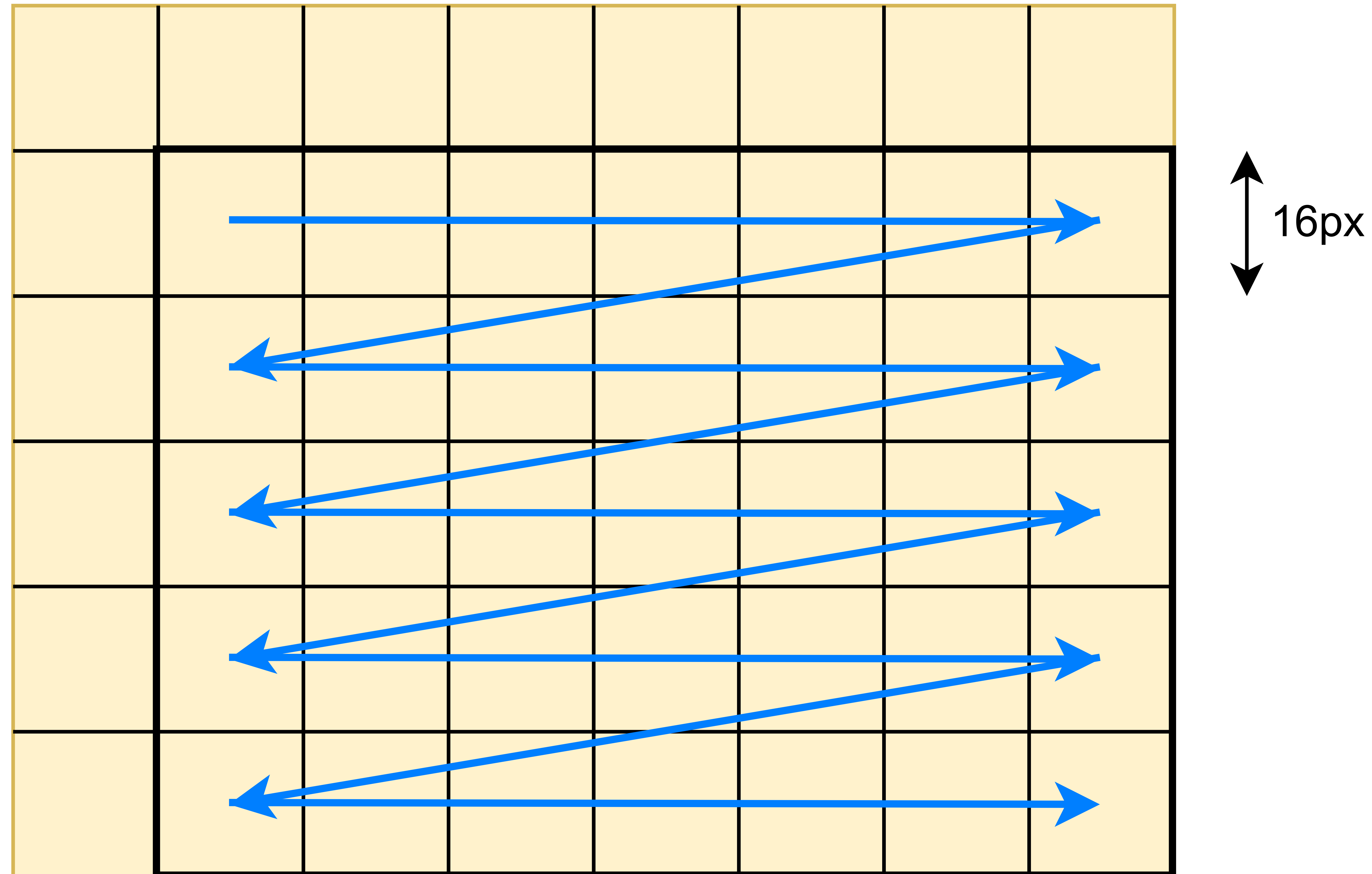


allocate fake neighboring mbs

all mbs have mv

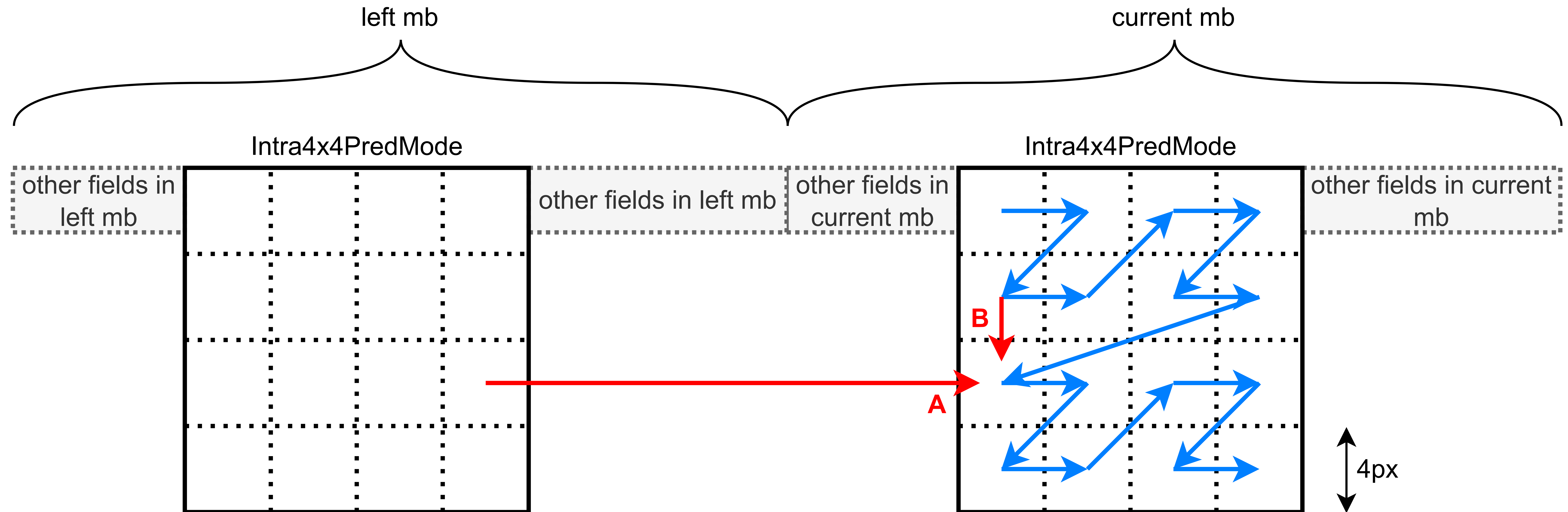


Technique 6 - Relative neighboring offsets



Each picture stores an array of `Edge264_macroblock` structures

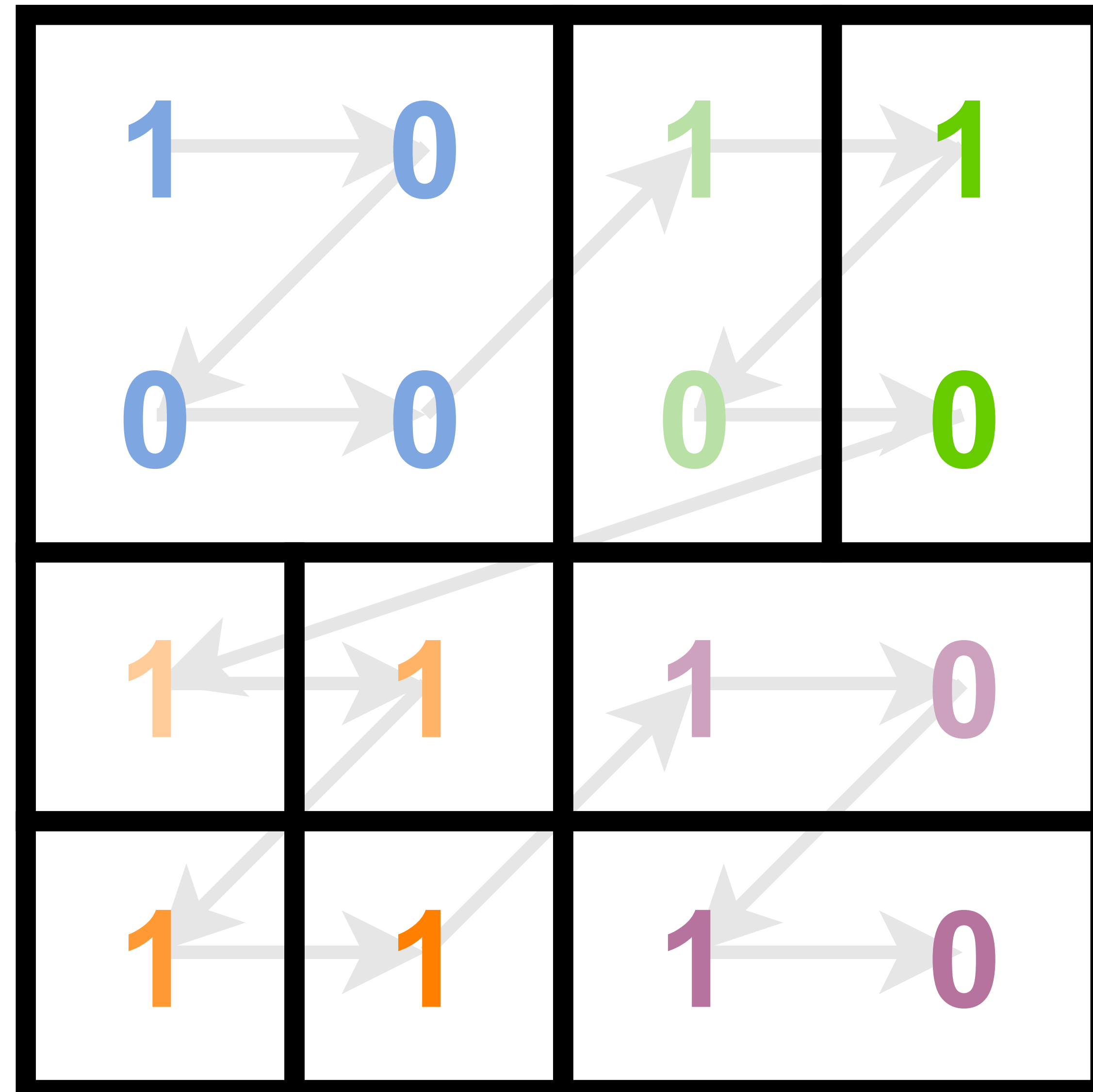
Technique 6 - Relative neighboring offsets



Values for sub-blocks are stored in arrays inside each mb

Neighboring values are fetched using **relative memory offsets**

Technique 7 - Parsing uneven block shapes



1 0 0 0 1 1 0 0 1 1 1 1 1 0 1 0

```
while (bitmask) {
    int i = __builtin_clz(bitmask);
    ...
    bitmask &= bitmask - 1;
}
```

General pattern:

1. convert mb type to bitmask
2. iterate on set bits

Used for reference indices, motion vectors, residual coefficients

Technique 8 - Using vector extensions

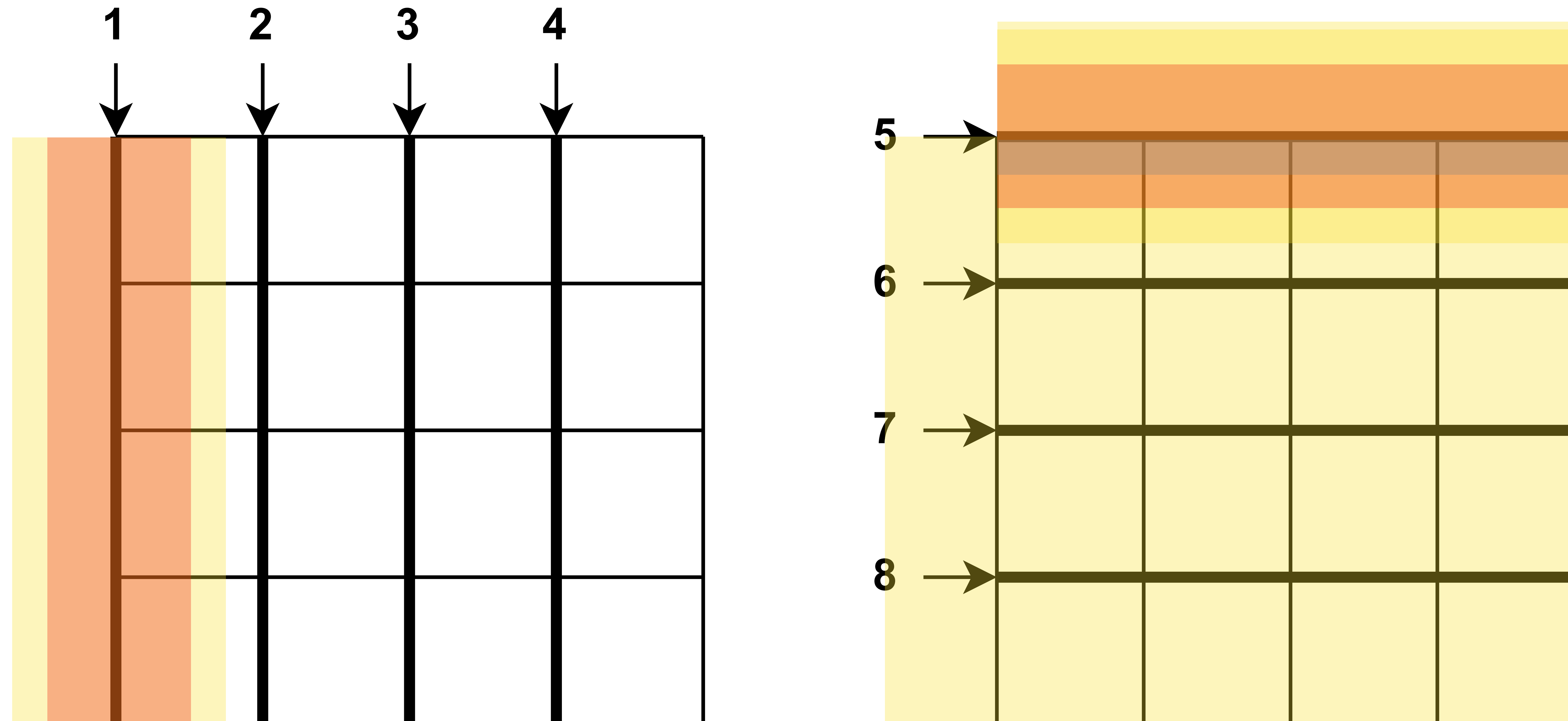
SIMD are used *everywhere* in edge264 thanks to **GCC vector extensions**

```
typedef int8_t i8x16 __attribute__((vector_size(16)));
union { int8_t array[64]; i8x16 array_v[4]; };
#define shuffle8(a, m) (i8x16)_mm_shuffle_epi8(a, m)
```

Caveats:

- Don't use __builtin_shuffle & __builtin_convertvector (may output big code)
- Don't use vector sizes other than native
- Don't index with variable (e.g. `pic[i]`)
- Don't use automatic vectorization!

Technique 9 - Register-saturating SIMD

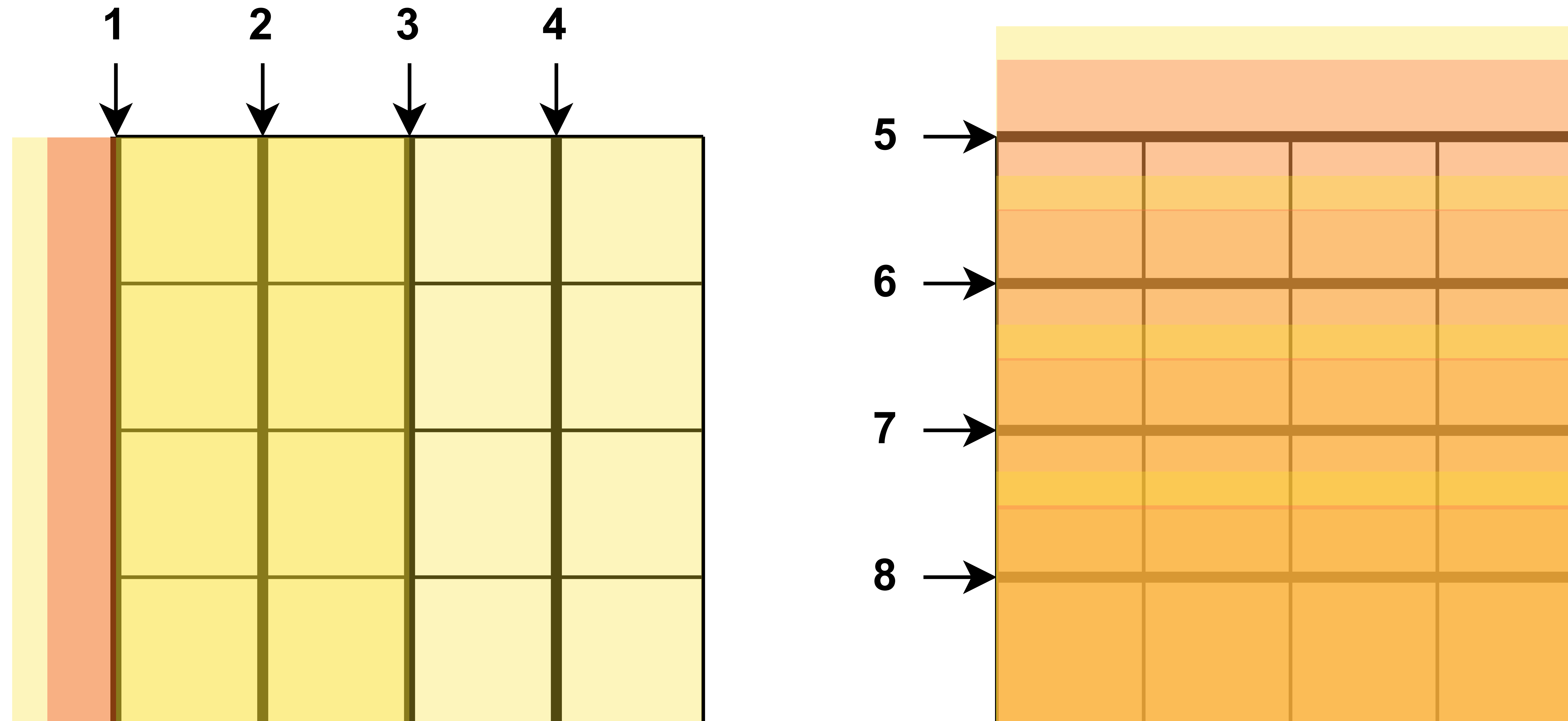


Vertical edge: 16 reads, 16 writes, 50 shuffles

Horizontal edge: 6 reads, 4 writes

All 8 edges: 88 reads, 80 writes, 200 shuffles

Technique 9 - Register-saturating SIMD



Ideal: 35 reads (-53), 34 writes (-46), 163 shuffles (-37), 22 spills

Actual: 40 spills (*less on later architectures*)

Technique 9 - Register-saturating SIMD

Easy to design and implement in C (esp. *reordering instructions*)

Used for deblocking, inter prediction (6-tap filters)

Using **C instead of ASM** also improved filtering code (20% shorter):

- eliminating redundant ops from macros
- reasoning with more compact code

Thank you for your attention!

(end of postdoc in March, will work full time on it then)

1. Single header file
2. Code blocks instead of functions
3. Tree branching
4. Global context register
5. Default neighboring values
6. Relative neighboring offsets
7. Parsing uneven block shapes
8. Using vector extensions
9. Register-saturating SIMD

Bonus - First matching condition

37	201	134	0	8	37	3	72	0	0	3	1	64	75	182	0	rest of bytestream

`== 0`

`== 3`

0	0	0	-1	0	0	0	0	-1	-1	0	0	0	0	0	-1

0	0	0	0	0	0	-1	0	0	0	-1	0	0	0	0	0

`m0 & m0<<1 & m3<<2`

0	0	0	0	0	0	0	0	-1	0	0	0	0	0	0	0

`clz(pmovmskb(..))`

8

Bonus - First matching condition

General pattern:

1. Compute all conditions in parallel with SIMD
2. Convert vector of results to unsigned int (PMOVMSKB)
3. Get indices based on set bits (e.g. with CLZ)

Used for escape sequences (003), start codes (001), spatial inter prediction

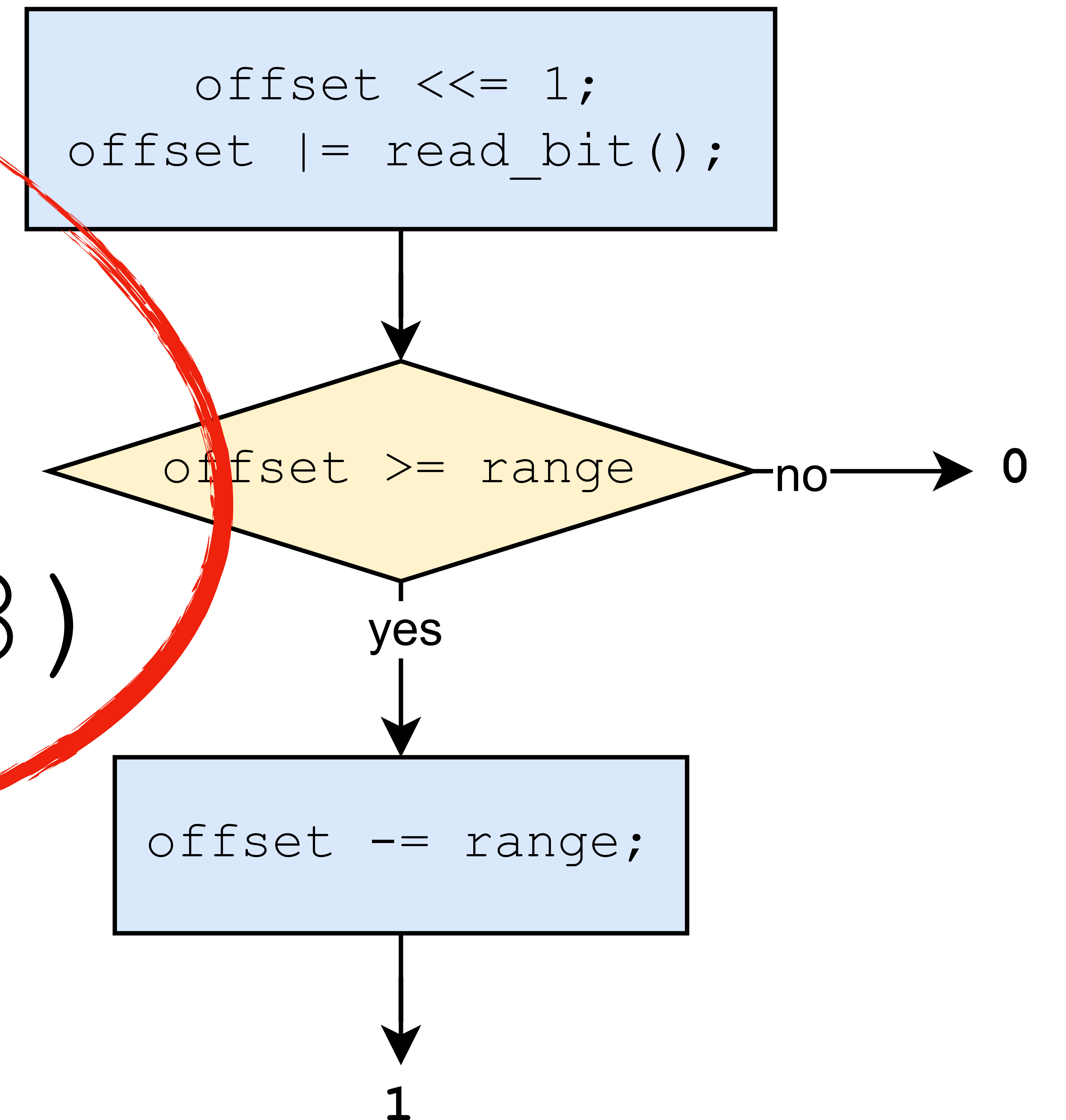
Bonus - Extraction of bypass bits

bypass bits

1010

```
range           offset
101011001 | 0111001010  (458)
    (345) |  0011100010  (226)
               0111000101  (453)
                0011011001  (108)
                 011011001
```

binary division



Bonus - Extraction of bypass bits

Requires offset on 32/64 bits

General pattern:

1. get a batch of bypass bits with offset/range
2. extract and remove Exp-Golomb code from it
3. compute new offset by multiplying remainder with range

Used for motion vectors, residual coefficients

Impact: speed?

Bonus - General rules

Allow all C versions, GCC/Clang extensions and processor intrinsics

Avoid storing redundant variables e.g. width_px & width_mbs => less bugs

Mutualize code paths => less potential bugs with rare code paths

Avoid small functions => makes code flow easier to follow

Inline & unroll manually => big chunks of code are SIMD opportunities

Write compact code => easier to spot SIMD opportunities

1 conditional test per bit of input => optimize away well-predicted tests

Avoid build-time options => easier maintenance

Minimize number of mallocs => spares hidden pointer arithmetic

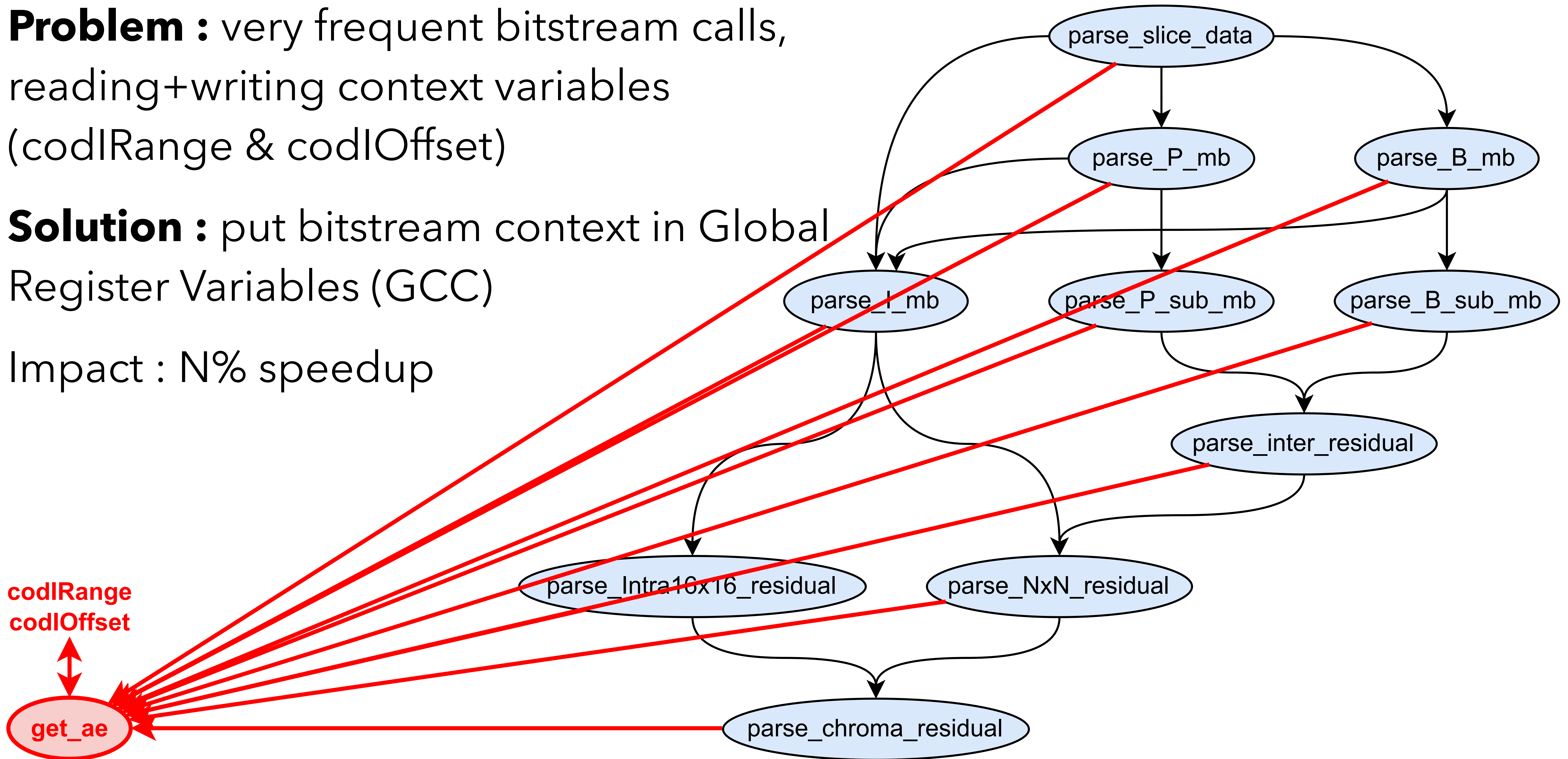
Avoid relying on compiler e.g. for math => spares the need to double check

Bonus - Global bitstream registers

Problem : very frequent bitstream calls, reading+writing context variables (codIRange & codIOffset)

Solution : put bitstream context in Global Register Variables (GCC)

Impact : N% speedup



Bonus - Coding for a CPU rather than C

Any C version => mixed declarations & code, restrict qualifier, _Generic, …

Any GCC extension => __builtin_{ctz,clz,popcount,expect,unreachable}, vector extensions, Global Register Variables, …

Target x86/SSSE3 exclusively => SIMD everywhere, no fallback C code

Intrinsics for unobtainable instructions => SHLD, PEXT, …

Impact: binary size

Bonus - Code and memory layout

Goal: be more compact than 1 byte per bit, speedup inits/reads/writes

AVC always requests 4x4/8x8 neighbors as left+2*top

Bit pattern for 4x4 and 8x8

Init = shift+mask, read = shift+mask, write = shift+or

Bonus all 8x8 inits can be vectorized

Actually used only for 8x8, not sure about performance but happy with size