Fun with subnormals

Hadrien Grasland 2025-02-20

Previously…

- μarch course gave you a taste of hardware complexity
- We mentioned **HW performance can be input-dependent**
  - If only because of memory access caching
  - Any single-number metric is for a typical or best case
- Today: More complex example in line with today's theme

IEEE-754 representation

- Recall that IEEE-754 uses base-2 scientific notation
  - Normal case: **$(-1)^{sign}$ x 1.fraction x $2^{exponent}$**
- Notice clever trick to avoid storing 1 bit of mantissa
  - Elegant weapon for a more civilized age
  - Just one problem: **How do we encode zero?**

Enter subnormals/denormals

A recurring issue

- Sometimes, my **code runs much slower on Intel CPUs**
  - I notice when I run on my AMD CPUs at home
- Perf annotate points to **hot float arithmetic instructions**
  - That’s good, I did my homework
  - Obviously, Intel did theirs worse than AMD
  - But why? And can they be saved from themselves?

Let’s ask the internet

- The easy stuff :
  - Floating-point ALUs may struggle with subnormal inputs
  - These are in every computation with diffusion or decay
  - You can disable subnormals with a global CPU flag
- If you investigate a bit more…
  - This is a much bigger problem for Intel CPUs*
  - Disabling subnormals is UB in many languages
  - It also breaks important math (iterative, minimization…)

* Rumor says that they implement subnormal math by trapping to a microcode emulation.

My contribution

- The **quantitative data** I could find is very lacking
  - Only covers very old CPUs
  - Not clear which operations are affected
  - Not clear what kind of input data was used and why
  - Only covers problematic global flag remediation
- So I started writing **a better microbenchmark**
  - Still WIP, but getting there
  - Today's talk is still early research, please be kind

Benchmark workload

- Take each operation implemented in hardware
- Combine with others if needed to get pseudo-operation that…
  - Can form an accumulation chain: acc ⟶ op(acc, input1, …)
  - Enforces that if acc is initially normal, it remains so
- Use biased random input generation to…
  - Enforce a controlled share of subnormal inputs
  - Keep accumulator & normal inputs close to]1/2; 2[range*

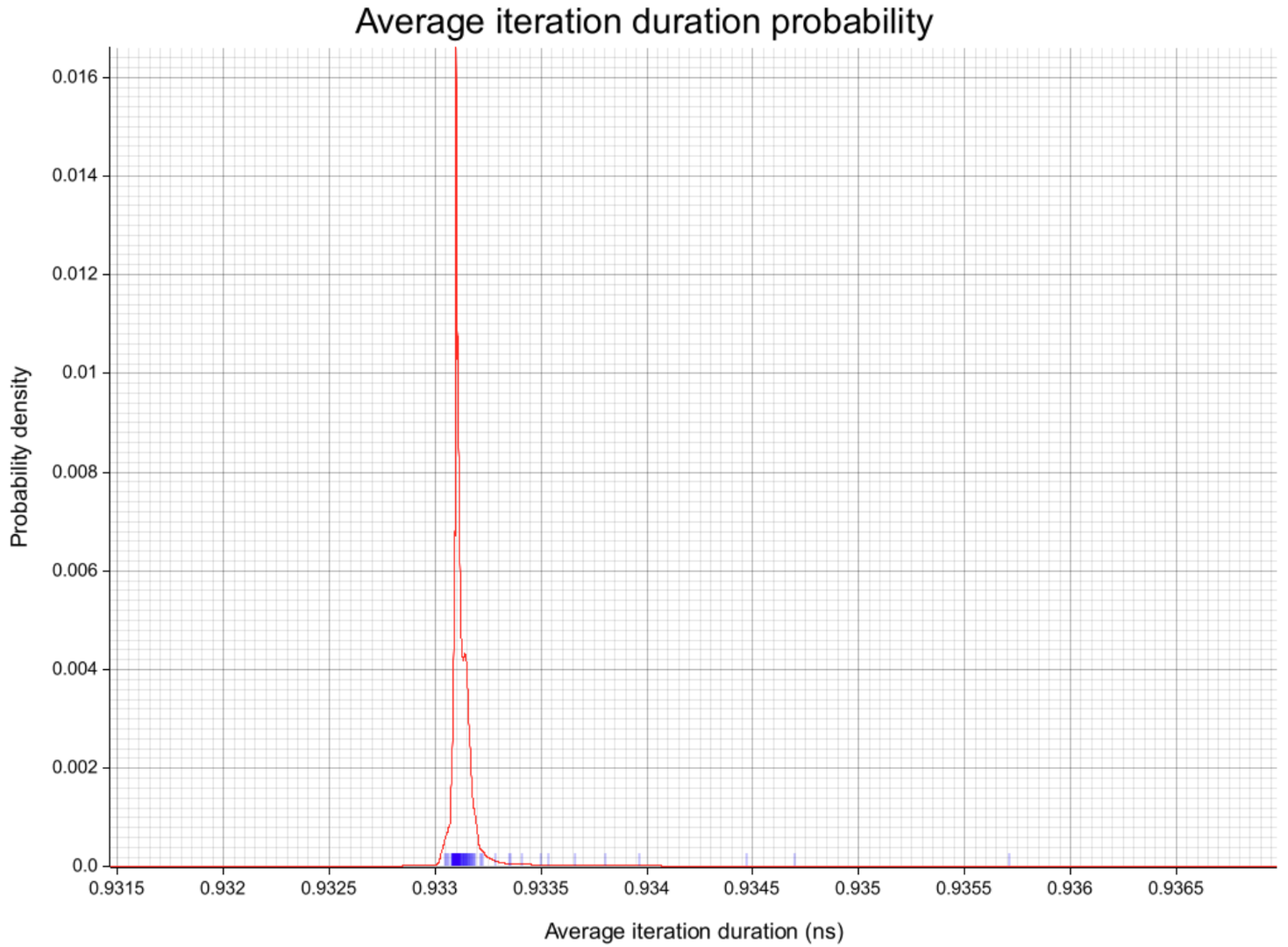
* Reduces rounding issues, increases chance that op(acc, subnormal) output is subnormal.

Combinatorics

- Now run that workload many times…
  - For all **data types** (single/double, scalar/SSE/AVX/512…)
  - For all floating-point **operations** supported in hardware
  - For all degrees of **instruction-level parallelism**
  - For all **input locations** (registers, L1/L2/L3 cache…)
  - For all **subnormal shares** (0-100%)
- Repeat N times with various N to detect instability/nonlinearity

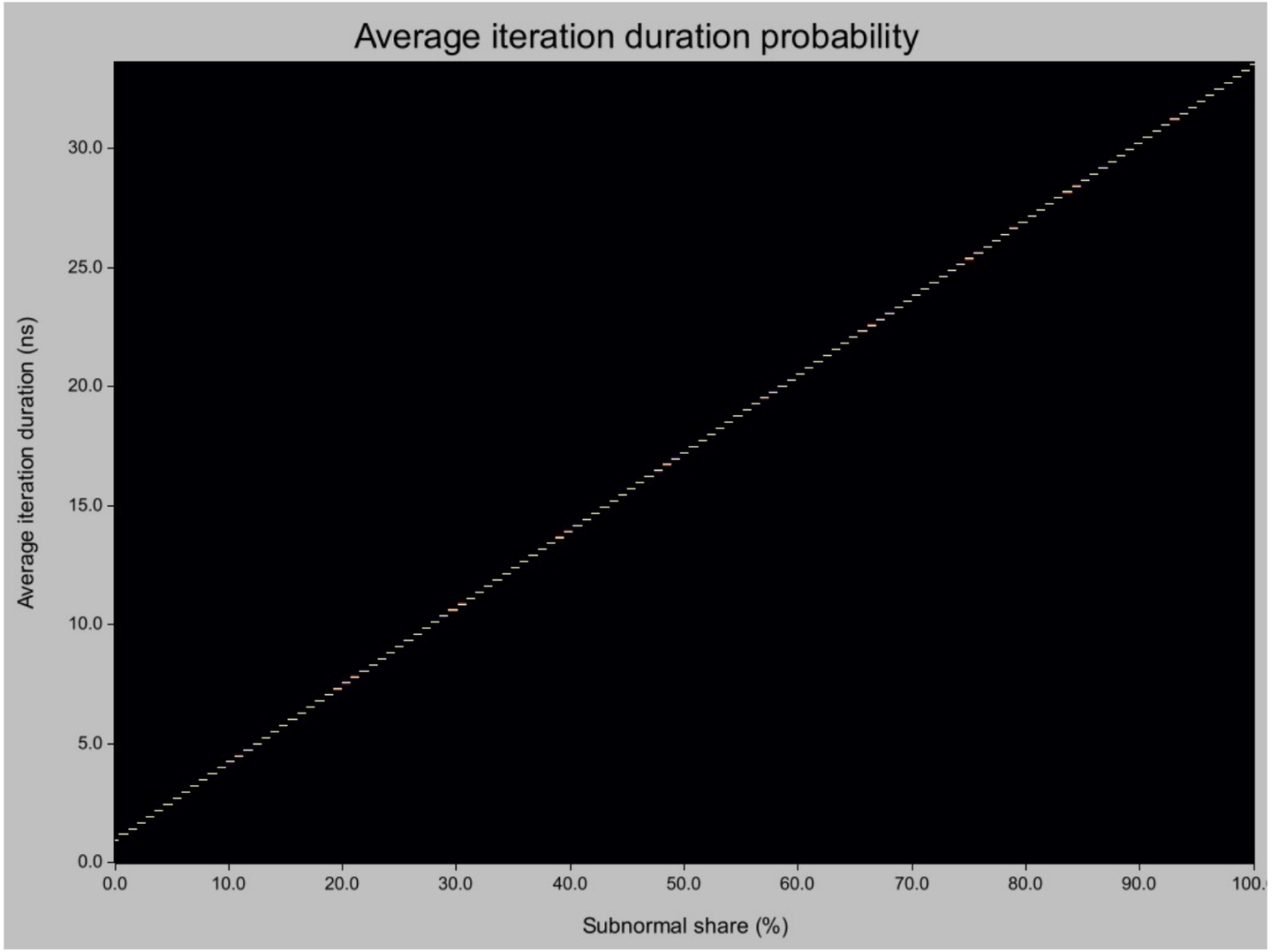
Per-configuration output

- From timing samples, estimate the operation duration PDF...



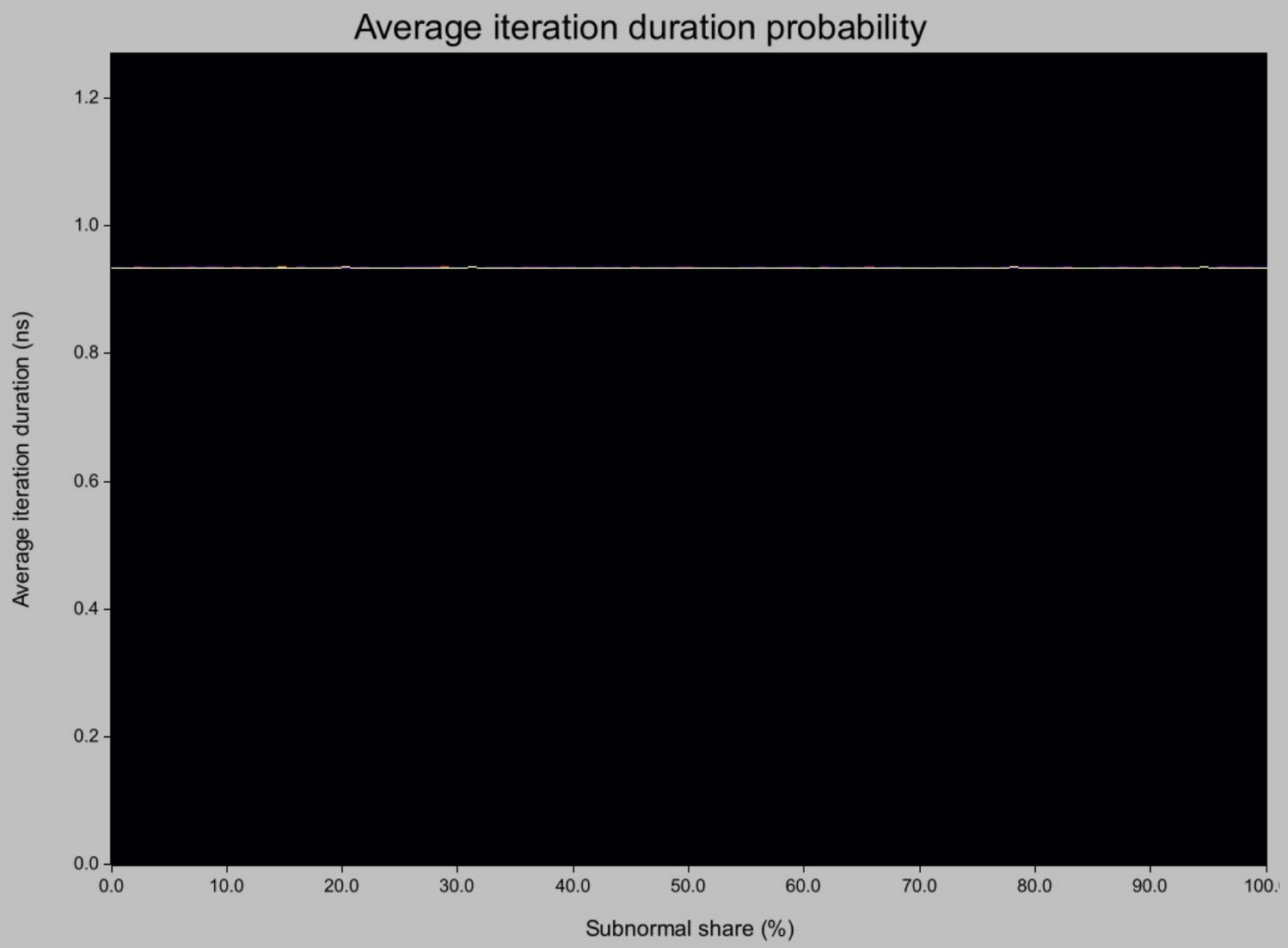
Subnormal impact

- …then we plot over subnormal share to see how the PDF changes



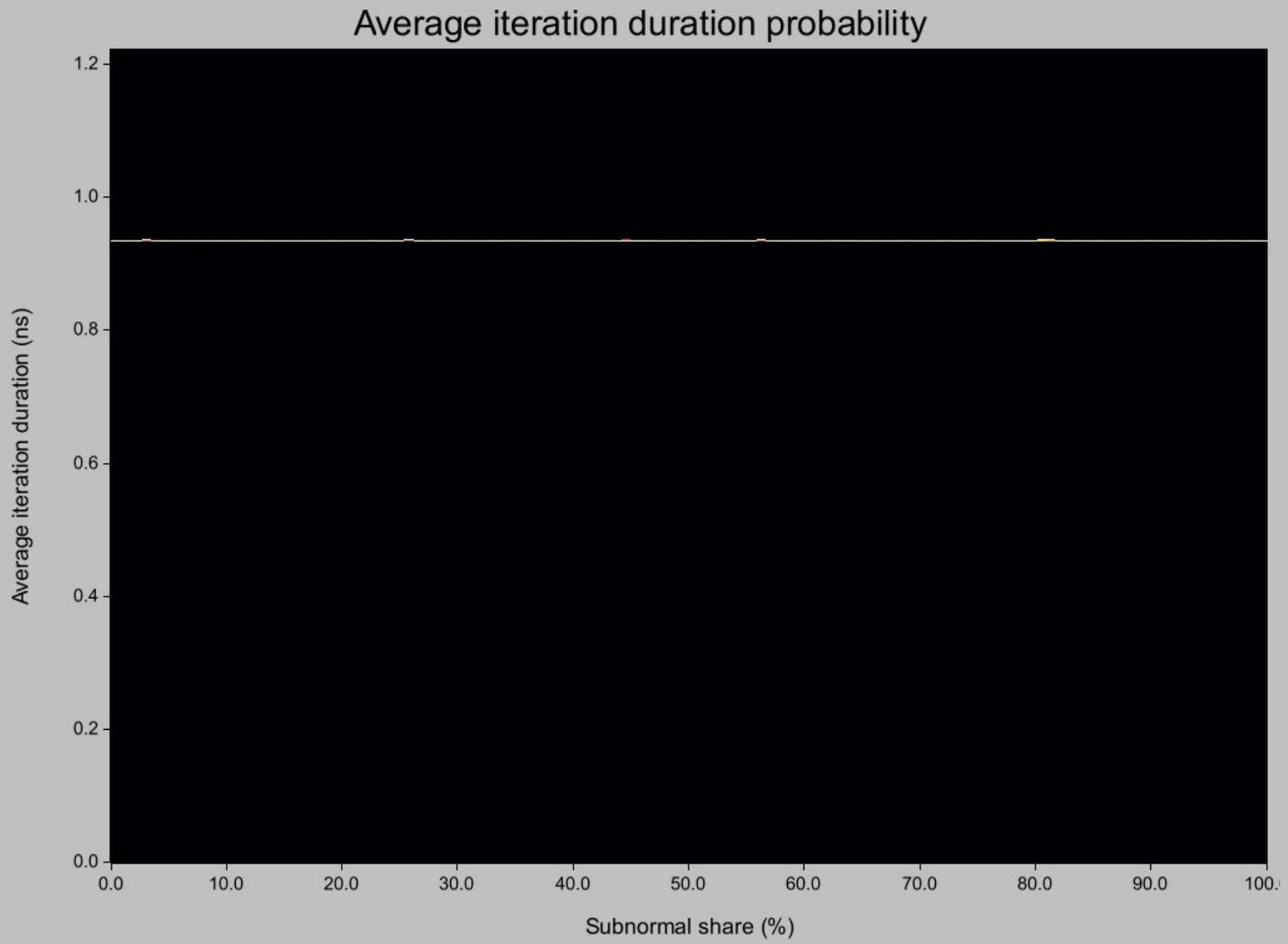
Intel i9-10900 (Comet Lake, 2020)

acc ⟶ acc + input

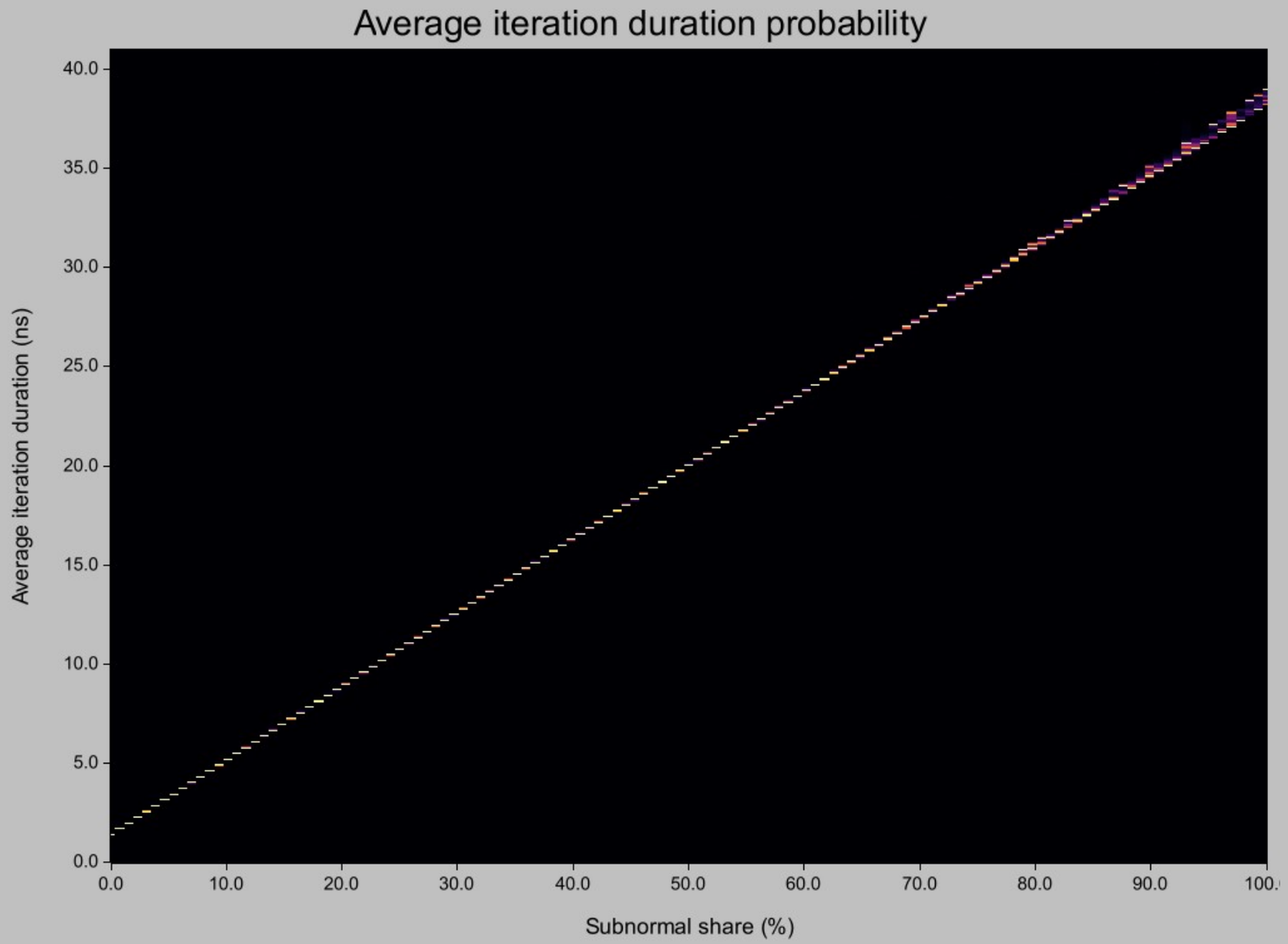


* All results presented today will be on f32 data, AVX vectors, in L1 cache, with no ILP

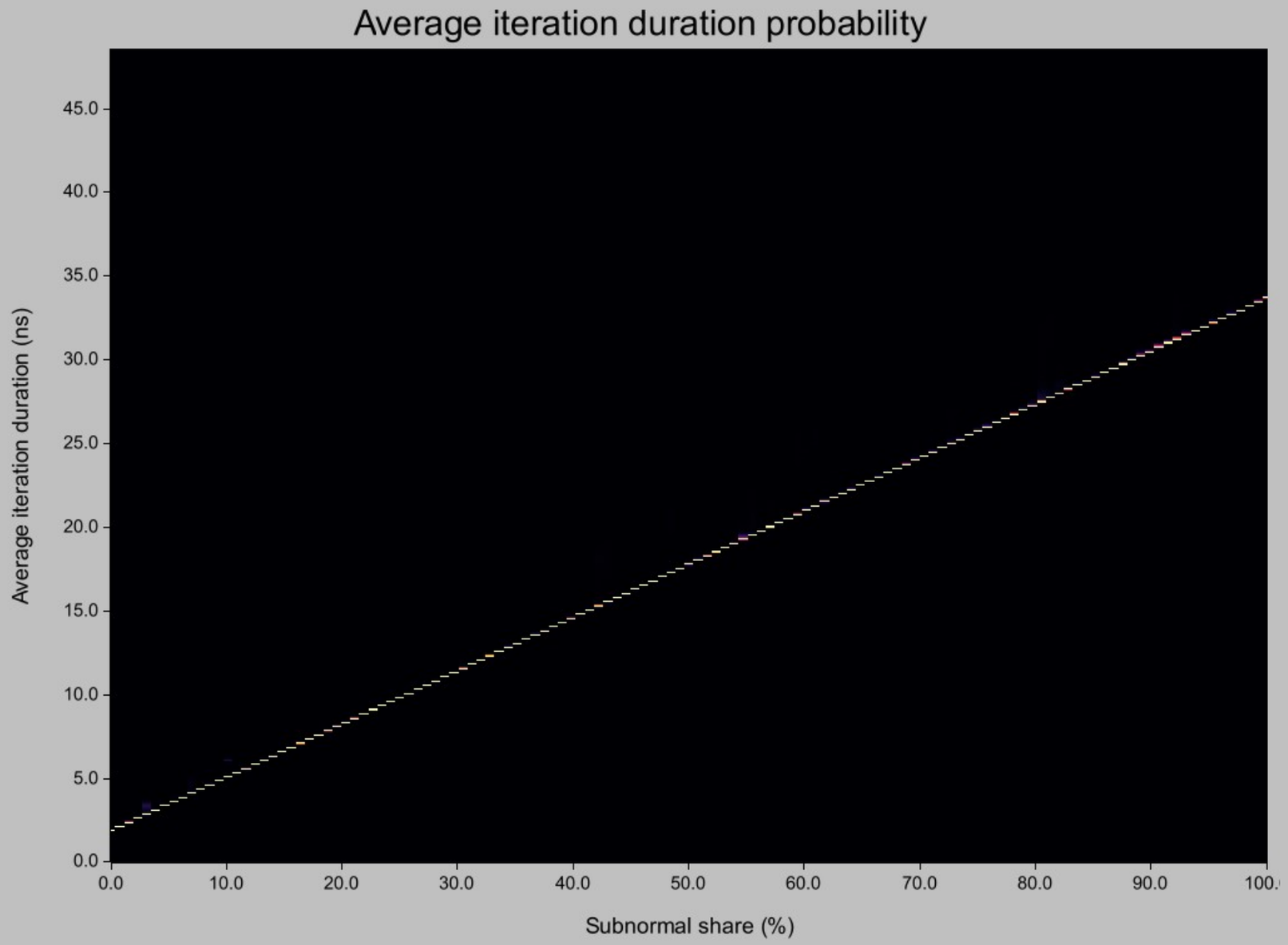
acc ⟶ max(acc, input)



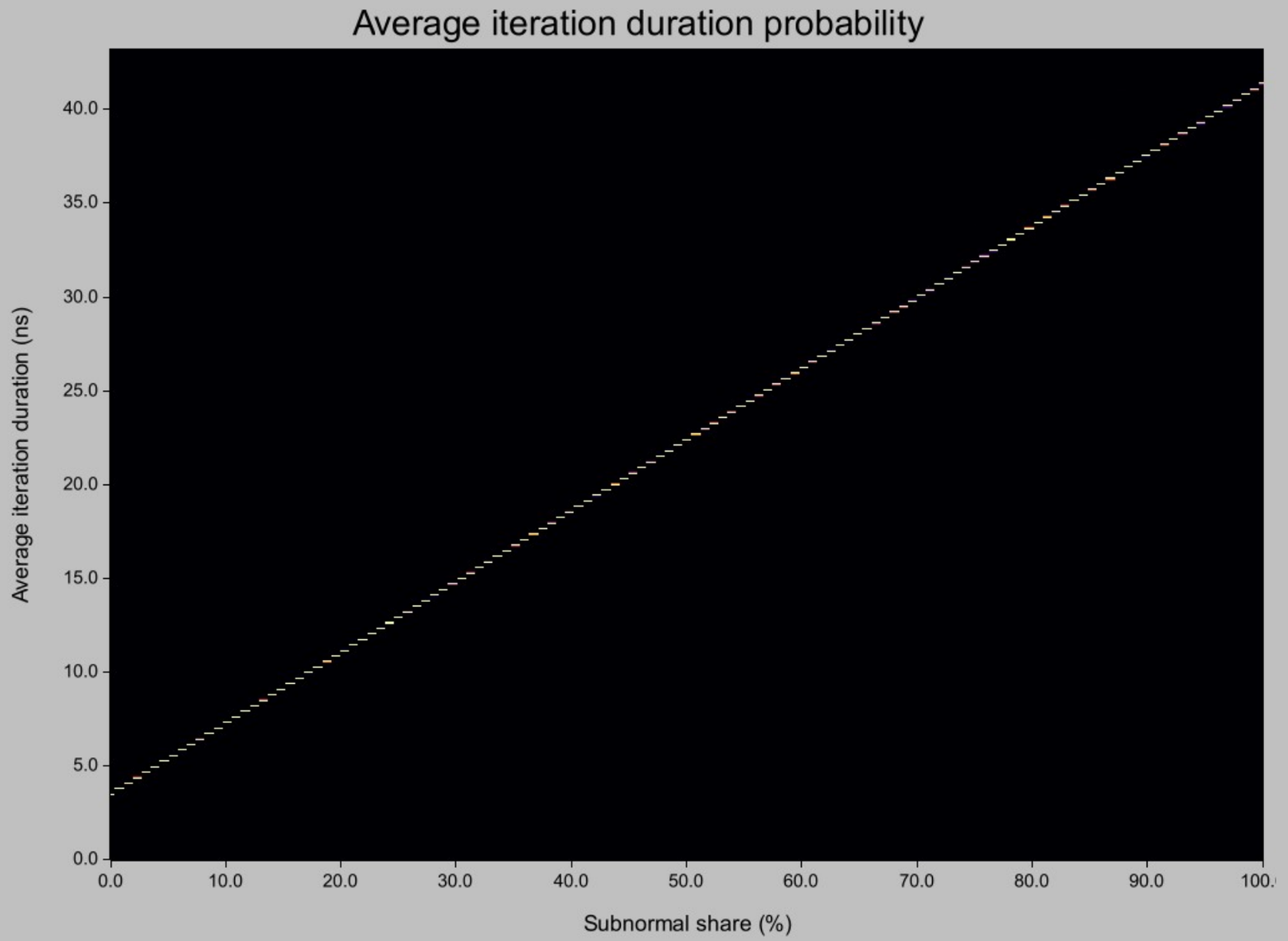
acc ⟶ max(acc, sqrt(input))



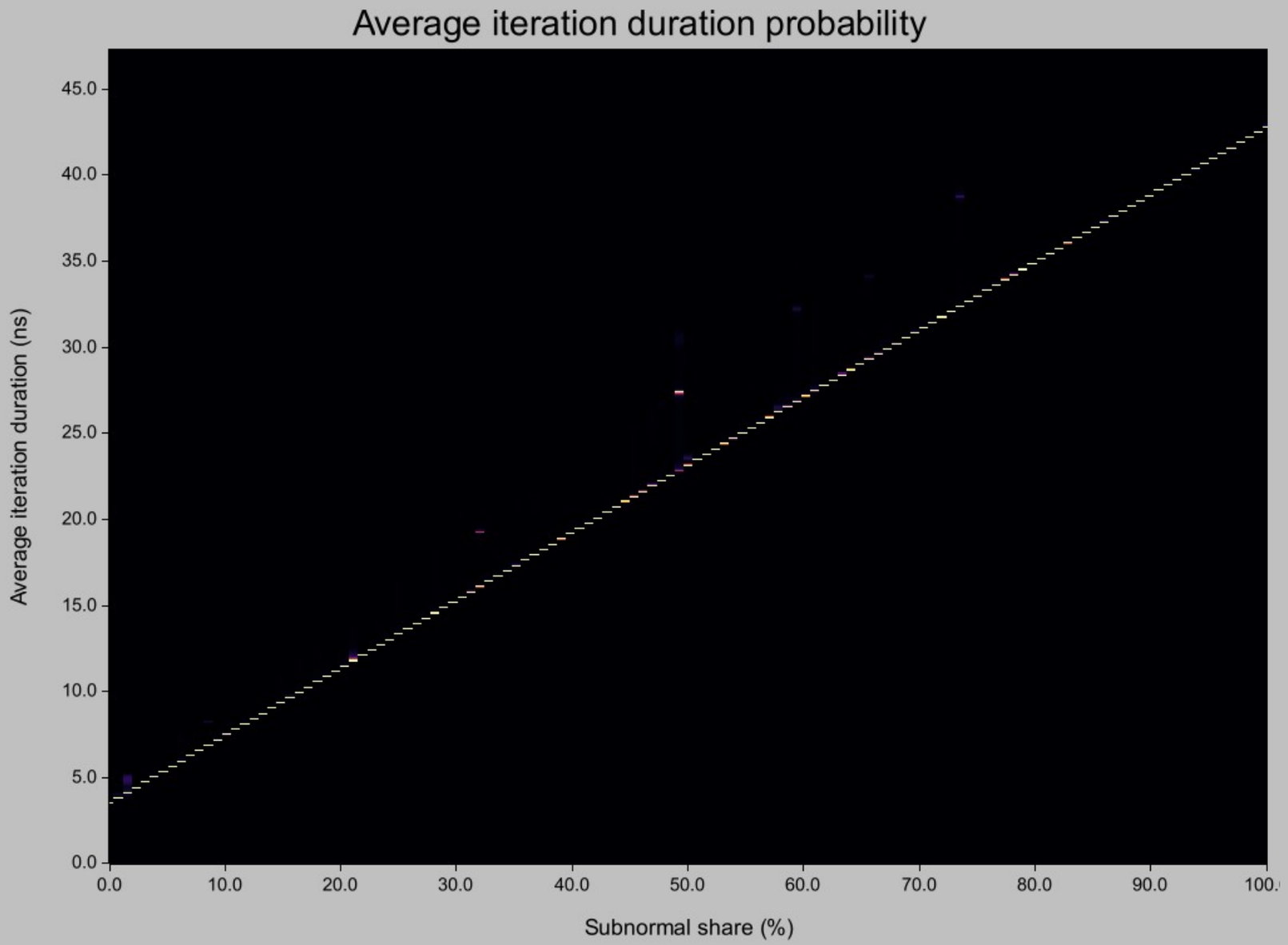
acc ⟶ max(acc * input, 0.25)



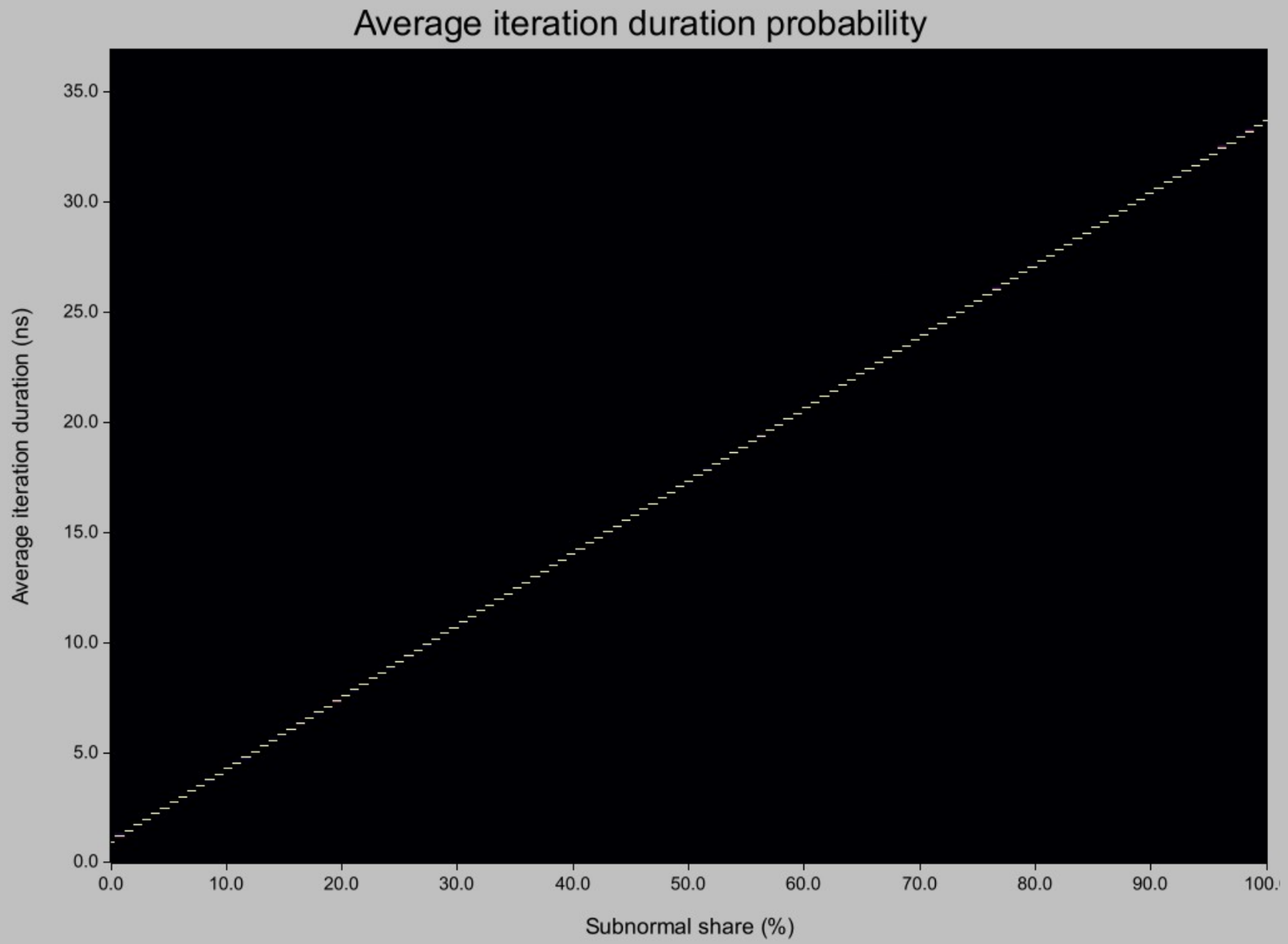
acc ⟶ max(input / acc, 0.25)



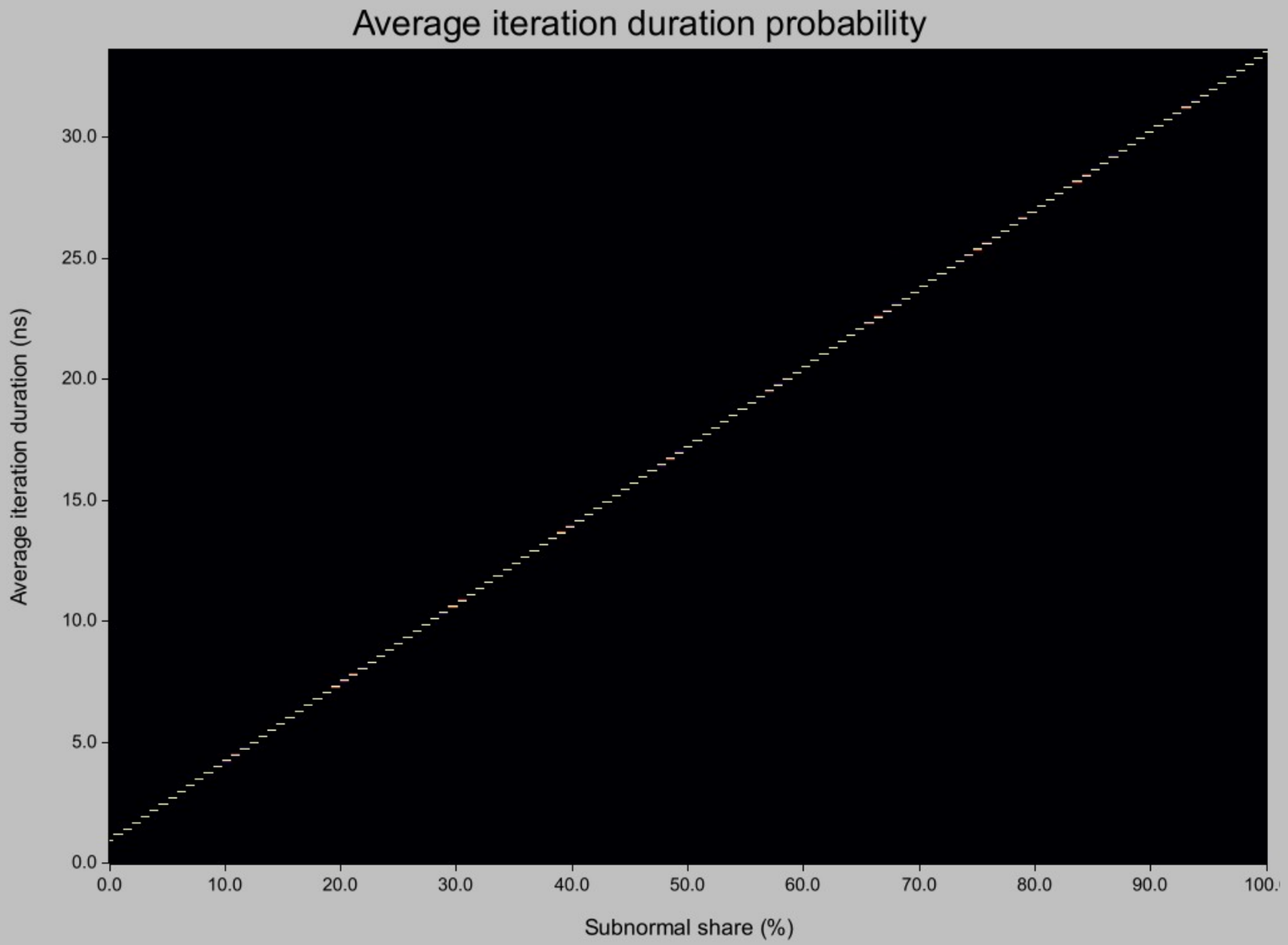
acc ⟶ min(acc / input, 4.0)



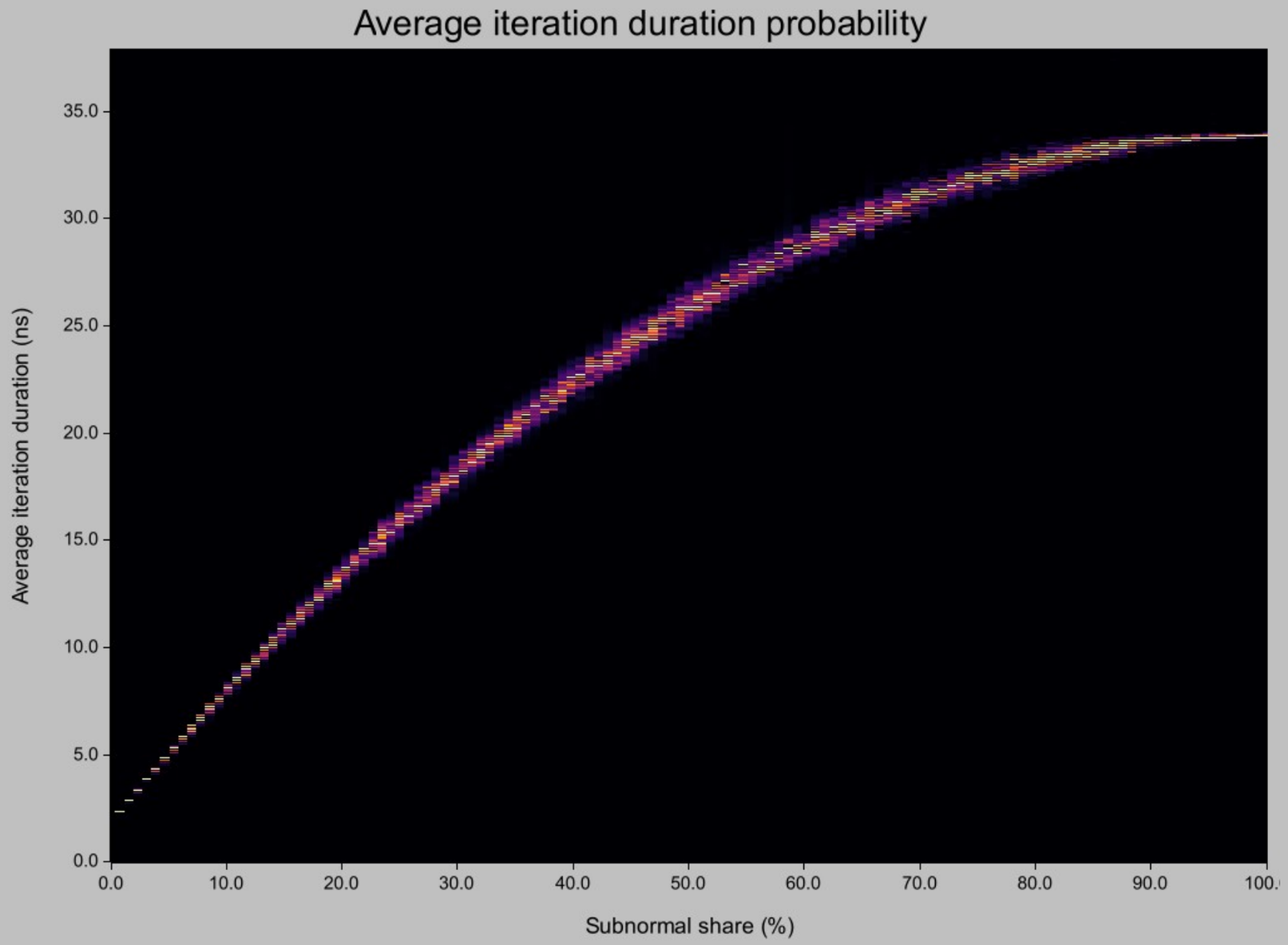
acc ⟶ fma(acc, cst, input)



acc ⟶ fma(input, cst, acc)



acc ⟶ max(fma(acc, in1, in2), 0.25)



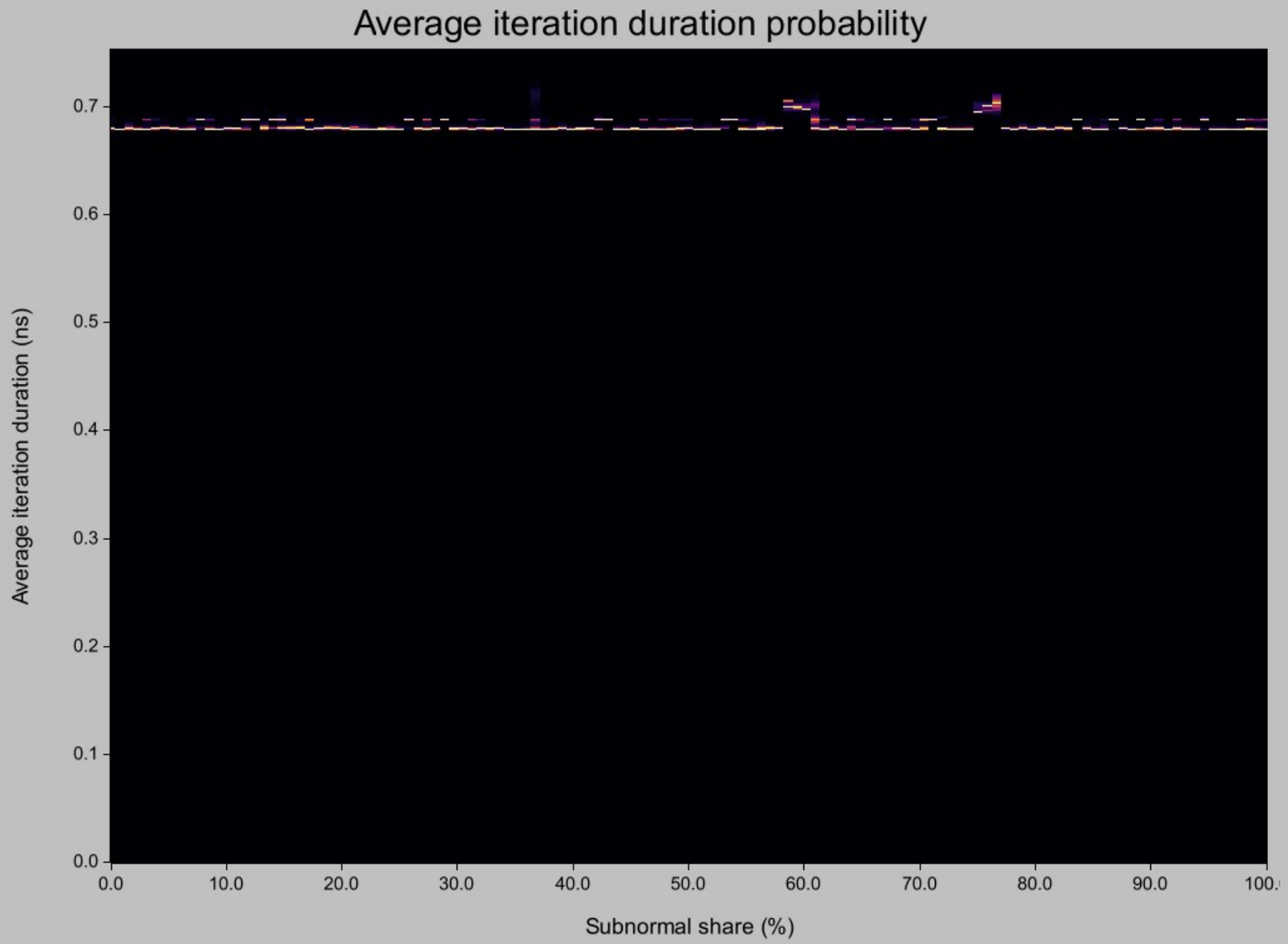
Intel recap

- ADD/SUB and MIN/MAX not affected
- Anything else pays a big penalty **(>30x slowdown)**
  - Happens if any input is a nonzero subnormal number
- What I don’t have time to show you :
  - Similar behavior across all data types
  - Relative penalty gets worse as amount of ILP increase*

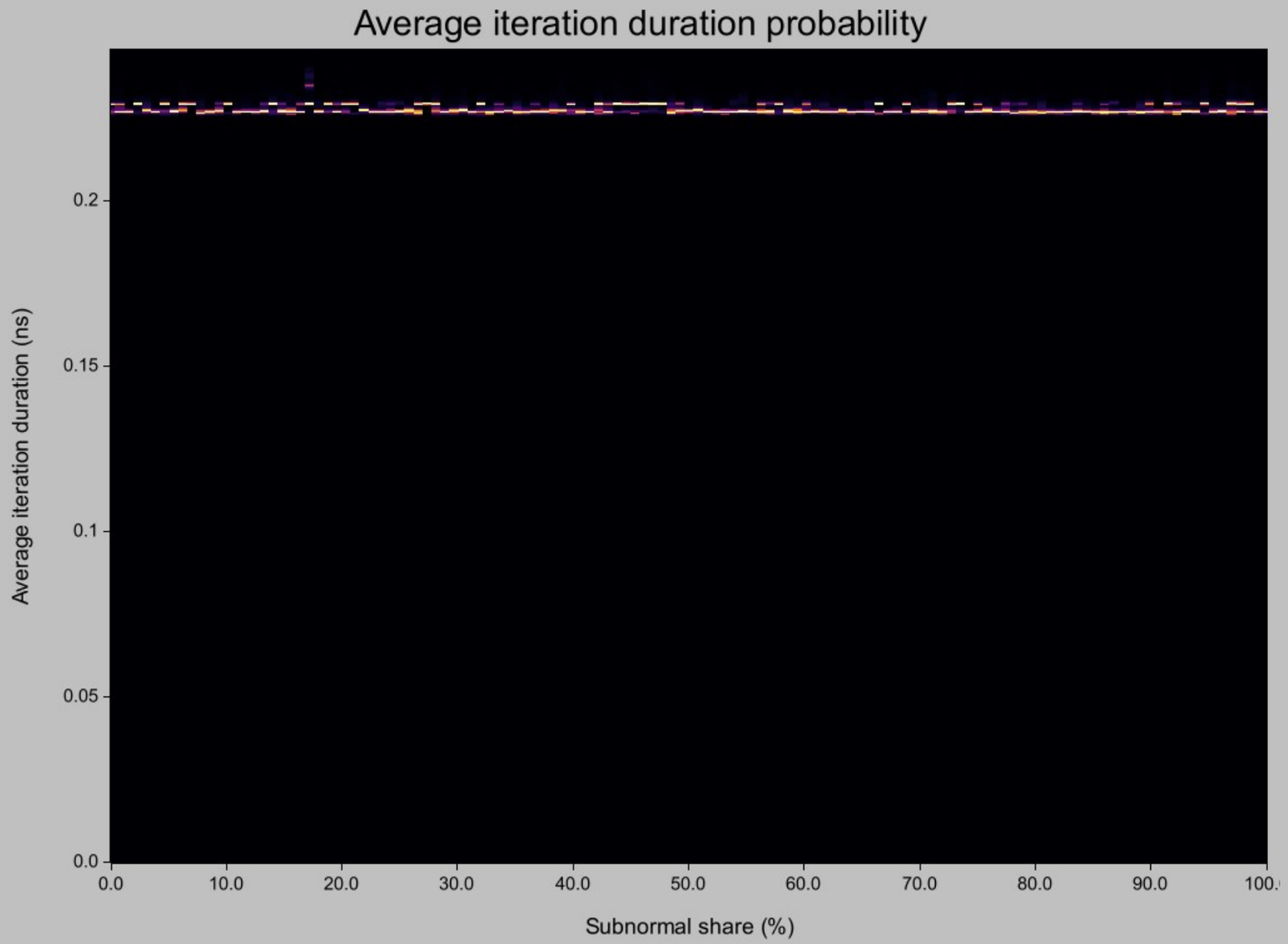
* Time at 100% subnormal similar to no ILP even if time at 0% shrunk → Becomes latency bound.

AMD Ryzen 5 5600G (Zen 3, 2021)

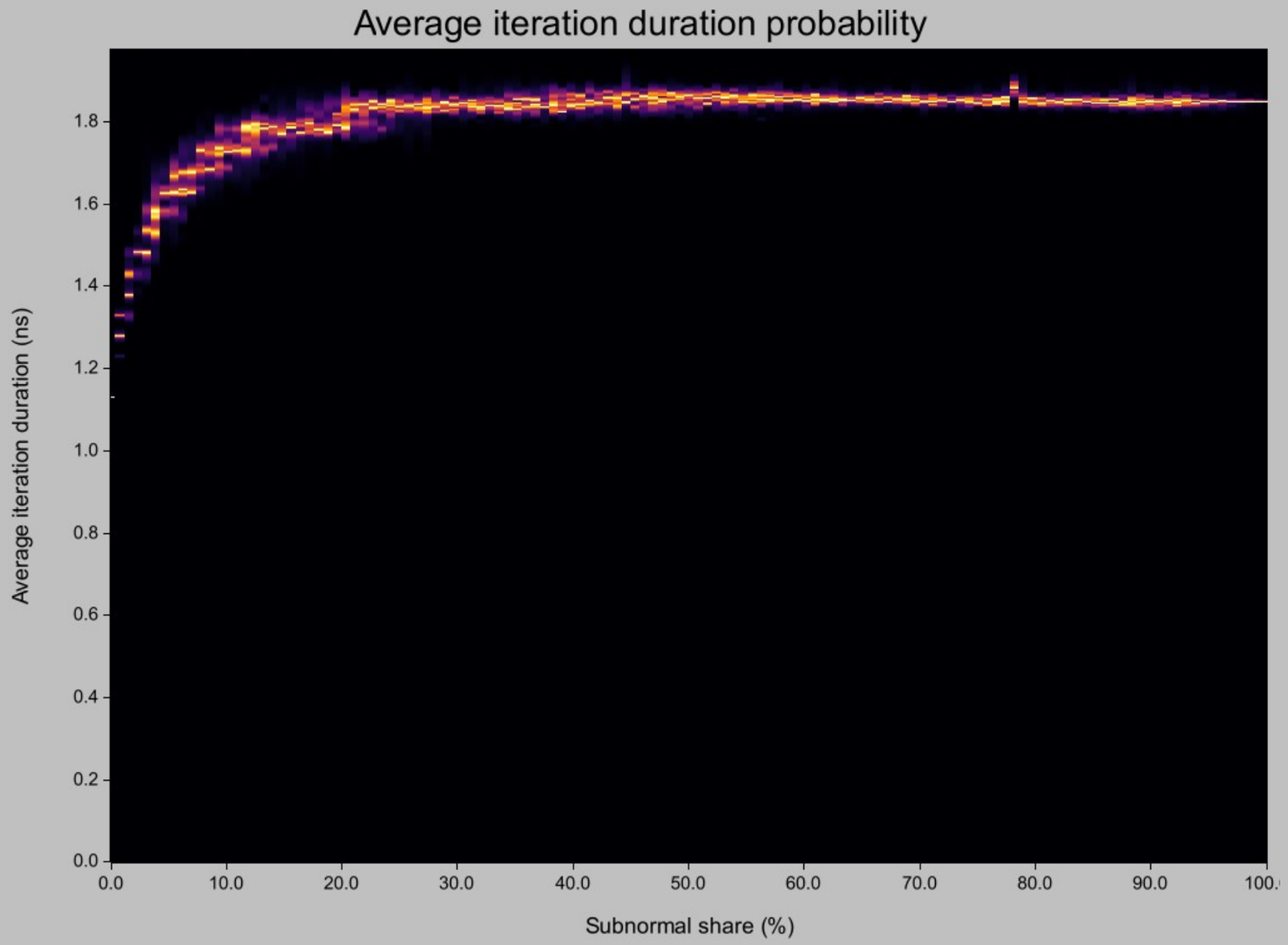
acc ⟶ acc + input



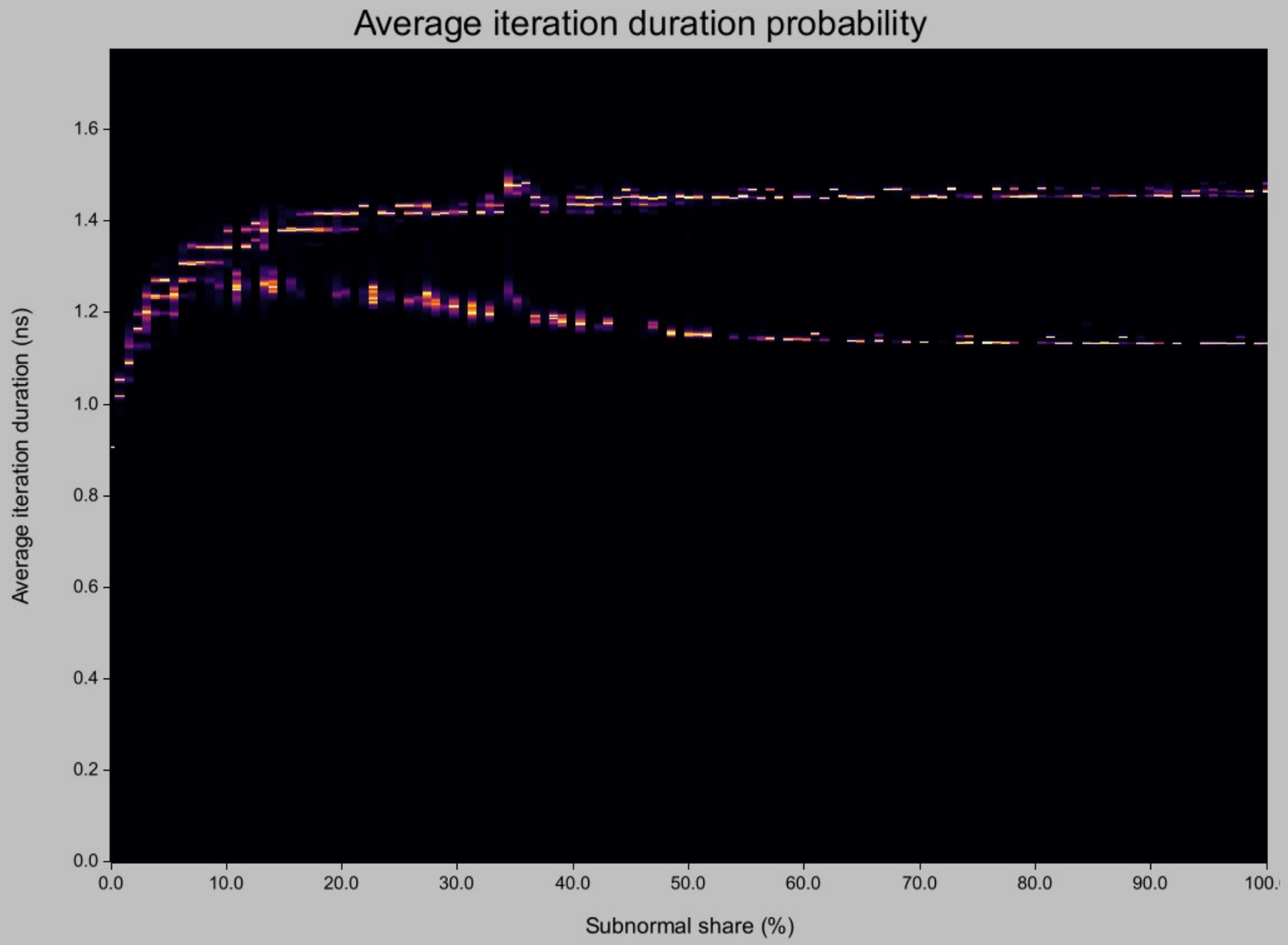
acc ⟶ max(acc, input)



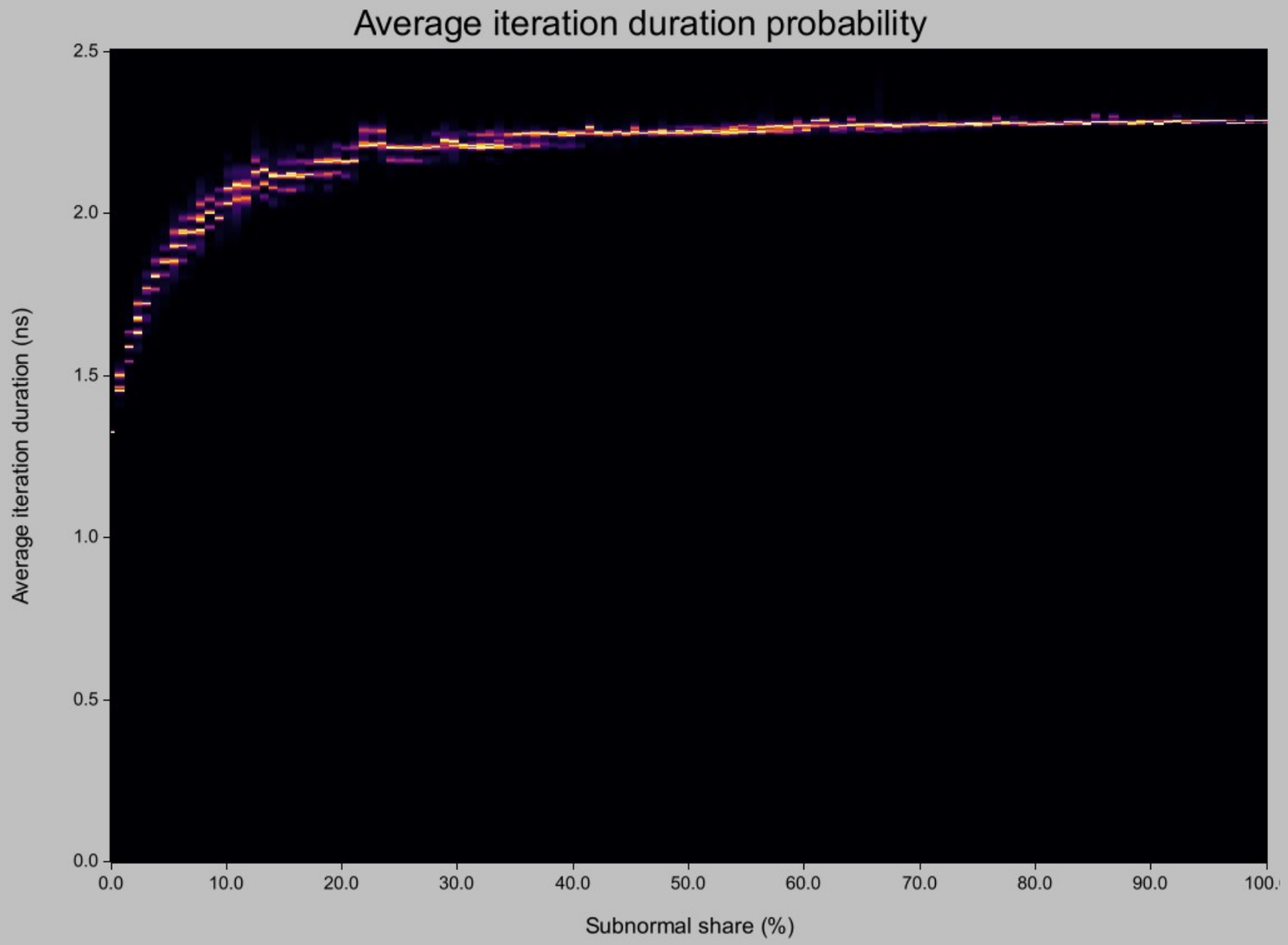
acc ⟶ max(acc, sqrt(input))



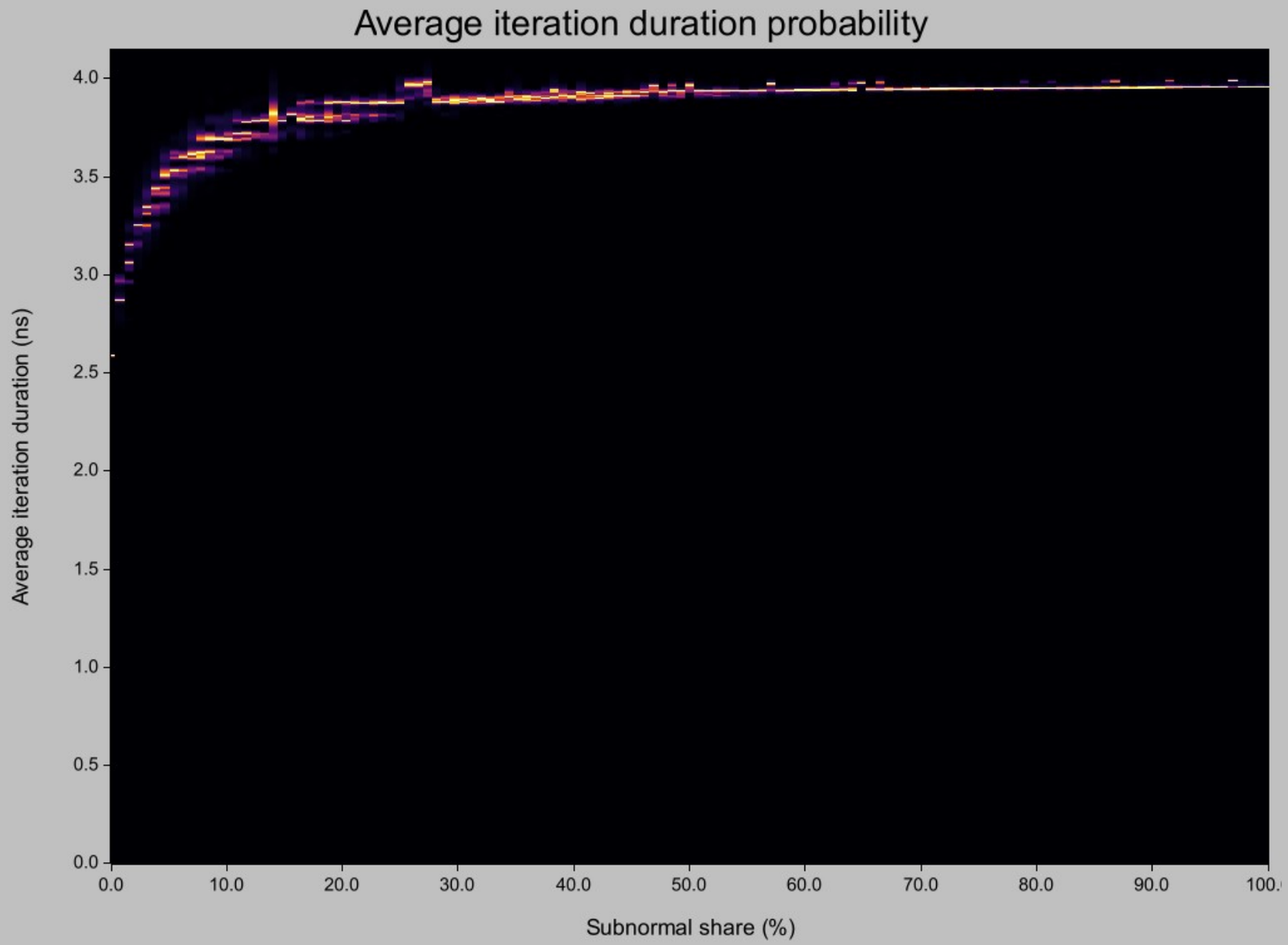
acc ⟶ max(acc * input, 0.25)



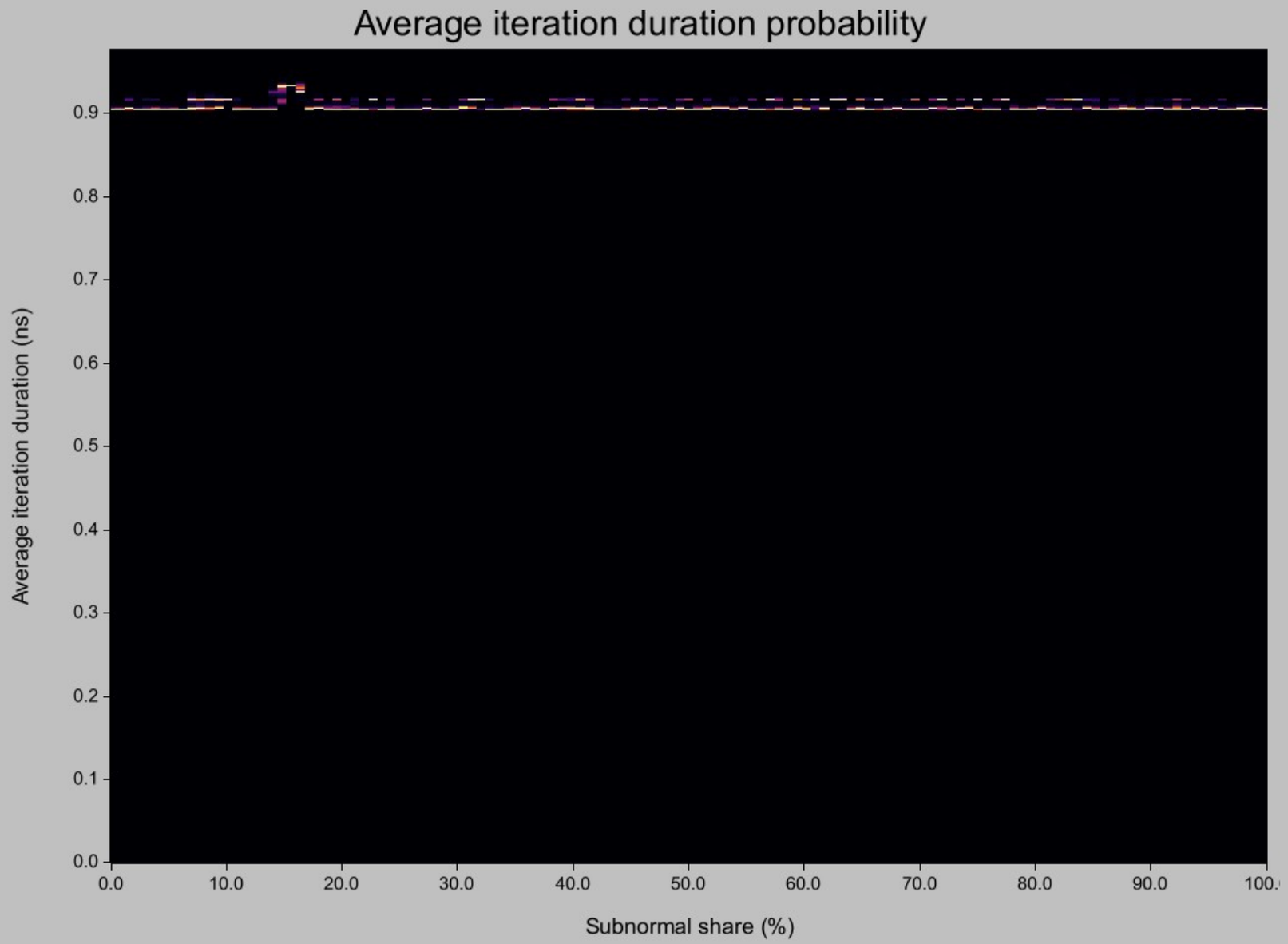
acc ⟶ max(input / acc, 0.25)



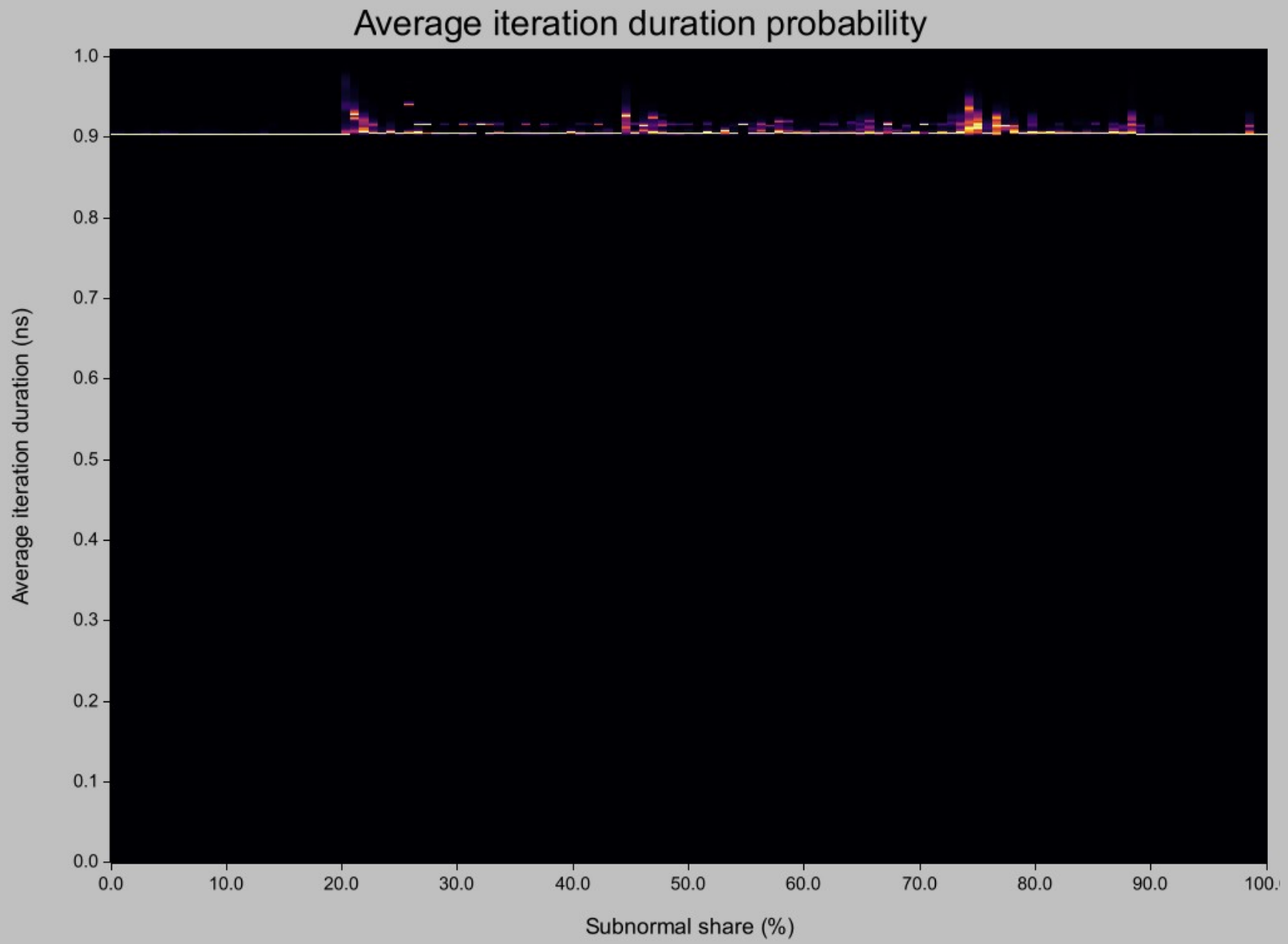
acc ⟶ min(acc / input, 4.0)



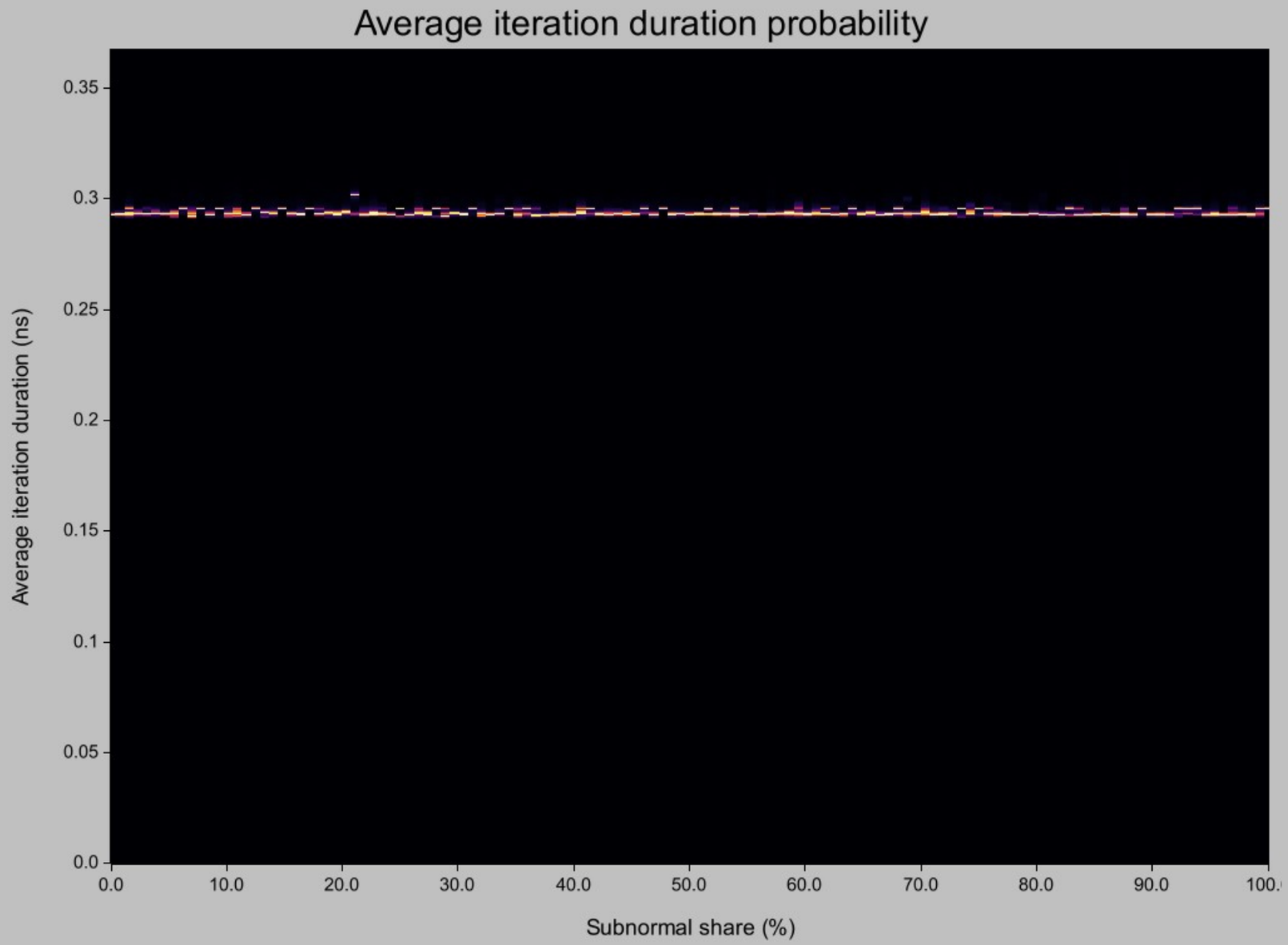
acc ⟶ fma(acc, cst, input)



acc ⟶ fma(input, cst, acc)



acc ⟶ max(fma(acc, in1, in2), 0.25)

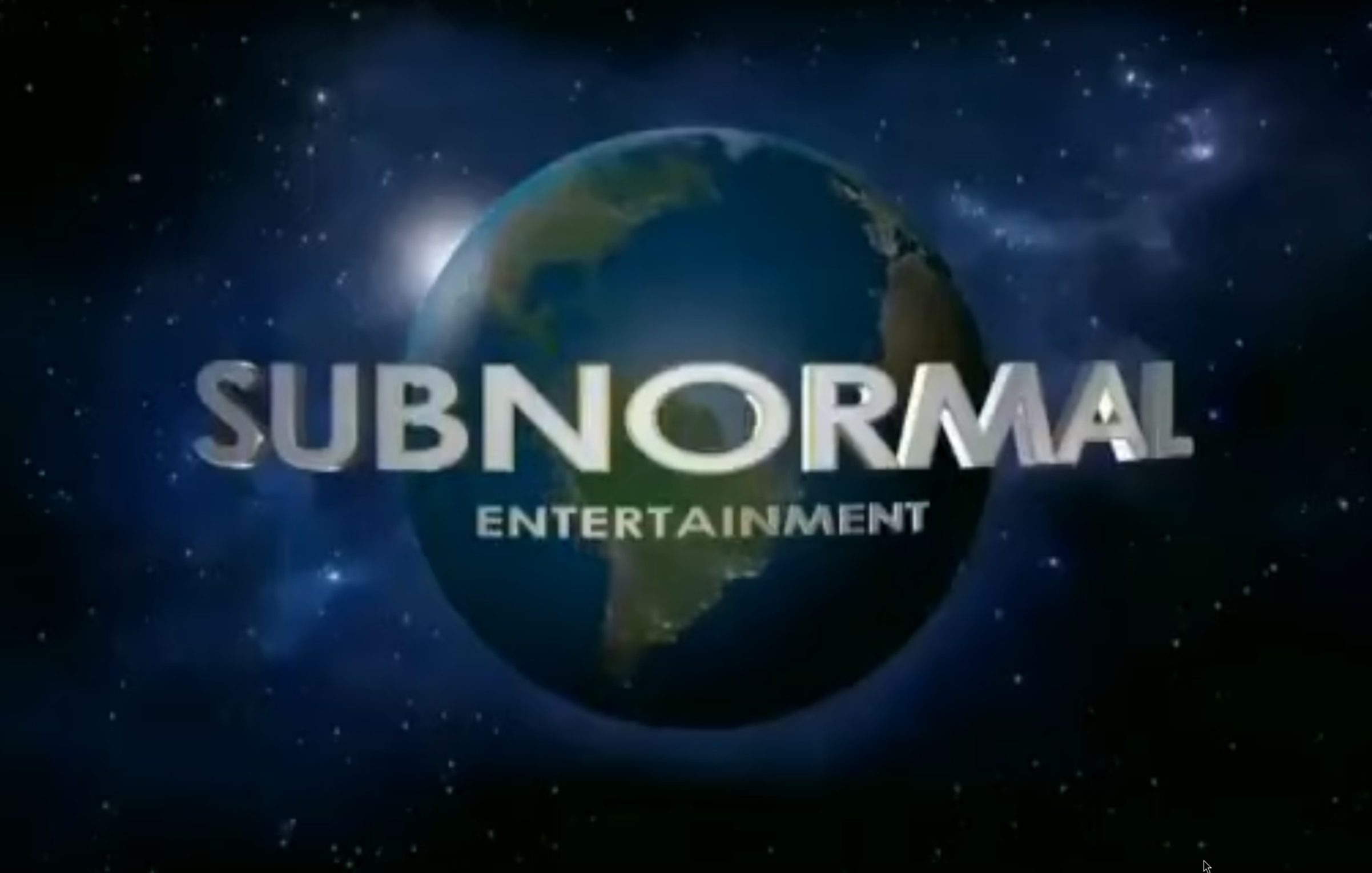


AMD recap

- ADD/SUB/MIN/MAX still unaffected, **FMA too for AMD**
  - Weird when we know MUL is affected
- Performance impact is much smaller (typically <2x range)
  - Saturating curve suggests a mode switch implementation?
- Bimodal MUL timing is very weird
  - Suggests variable behavior across runs, to be investigated

Conclusion

- Some float ops struggle with subnormal inputs
  - MUL/DIV/SQRT for Intel & AMD + FMA too for Intel
- Associated performance penalty varies
  - Can be >30x for Intel, more like <2x for AMD
- Much more work to be done
  - Understand bimodal AMD MUL run timing
  - Investigate remediation beyond FTZ/DAZ flag
  - Check other CPU manufacturers, GPUs…


SUBNORMAL
ENTERTAINMENT