CV Academy

Pragmatic optimization approach

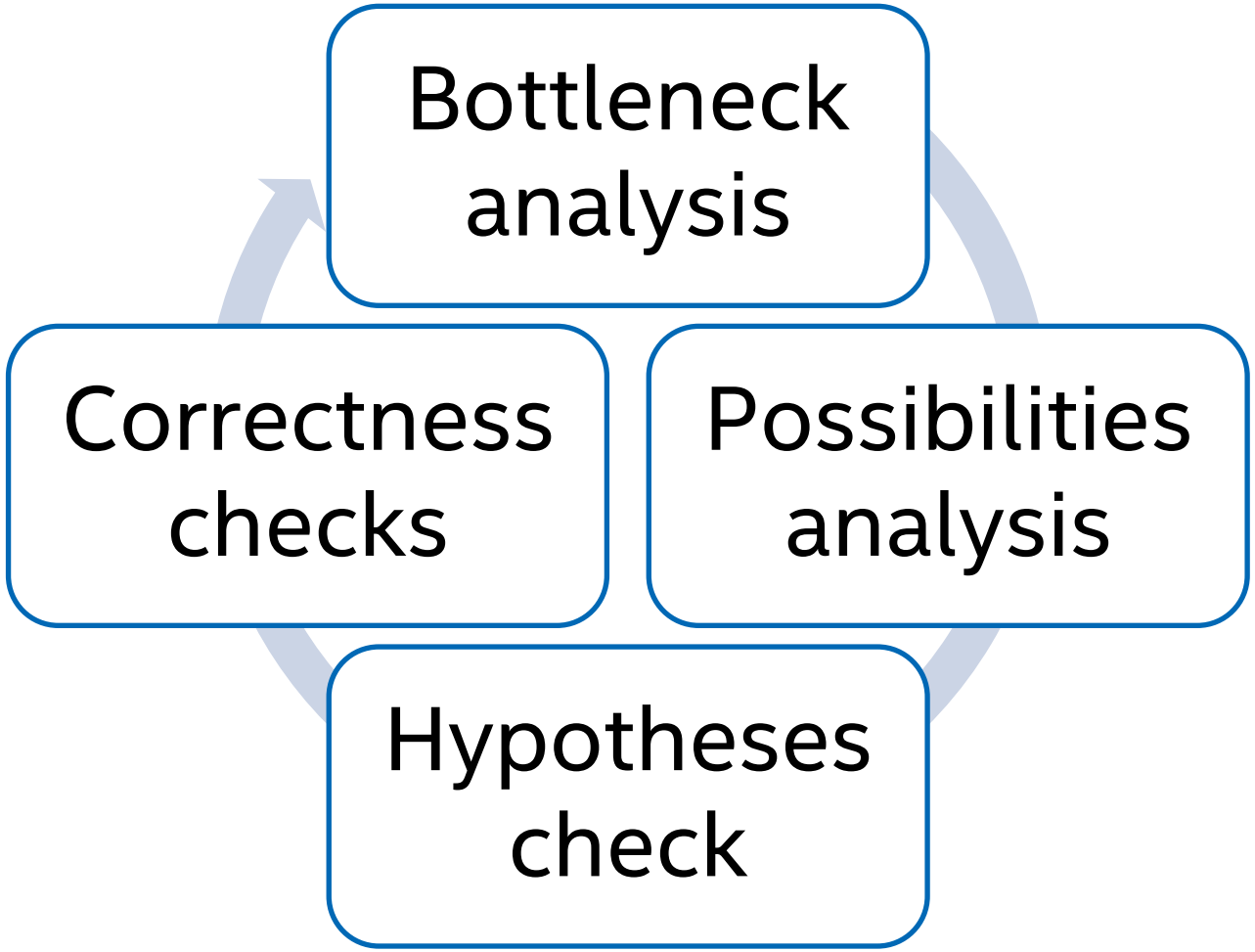
Alexander Shokin

Optimization targets

Optimization is a process of transforming a piece of code to make it more efficient without changing its output.

- **Wall(-clock) time** is a human perception of the span of time from the start to the completion of a task
- **Power consumption** is the electrical energy which is consumed to complete a task
- **Processor (CPU) time** is the total execution time during which a processor was dedicated to a task (i.e., executes instructions of that task)
- **Pipeline latency** is amount of time between when the task is issued and when it completes

Typical optimization pipeline



Bottleneck analysis

1. Benchmark whole application
2. Sort results by stages (can be functions or loops - depends on code size, for example) summary time. Our points of interests - top 5 or top 10 hardest stages. Count depends on time distribution between stages. In most cases top 5 stages has in sum more than 80% application running time.
3. Estimate optimization potential and time on its implementation
4. Choose stage that seems most important to optimize

Memory or compute bound differentiation

In most CV kernels there are streaming memory access pattern

1. Benchmark current kernel time – baseline time
2. Benchmark memcpy-like kernel time in the same memory environment – best kernel time from memory bound point of view
3. Calculate minimum clocks count using information about CPU operations latency, throughput and simultaneous running possibilities from documentation. From clocks and CPU frequency calculate minimum possible kernel time from compute bound point of view.
4. Difference between baseline time and maximum time from possible minimums – is out potential for optimization on this kernel level. Is it enough to get started?

Checklist before starting

- There are enough tests on this part of code?
- There are tests with cover “strange” cases (picture sizes 1x1, 1x11, 11x1, 11x11, 10Kx10K, for example)
- There are fast and simple way to disable optimized code and run reference one?
- There are list of typical parameters for this kernel?

Possibilities analysis

Memory bound:

- Can we use different algorithm?
- What memory level has most problems? (L1, L2, L3, DDR, etc.)
- Throughput limitation
- Cache misses
- Access pattern

Compute bound:

- Can we use different algorithm?
- Vectorization potential
- Multithreading potential
- Fixed point arithmetic
- CPU pipeline stalling:
  - Loop unrolling
  - If-conversion

Memory bound: L1/L2 cache misses

- There are unclear memory access pattern for prefetch unit? Try to update algorithm to use simpler memory access pattern.
- There are inefficient cache usage because of cache thrashing? Try to update algorithm to use streaming memory access pattern.
- All data stream load and store through cache once? Try to merge several stages into one.
- What percent of cache line is using in calculation? Try to update data storage format, for example use SoA istead of AoS.

Memory bound: other memory hierarchy levels problems

There are multithreading pipeline in this application? If yes – it's paralleled inside stages or different stages runned simultaneously in different threads? Independent on answers there are only one way to optimize this part of code – increase data locality. Often, the best realization – is divide data on several tile and merge calculation of several stages together. The fastest way to load some data – don't load data but using already loaded data.

Compute bound: vectorization

- Find the most suitable data type – as short as possible but meet accuracy requirements.
- Open and analyze assembler code, generated by compiler and check auto-vectorizer work. Can you do better? If yes – try to understand why compiler not did that? For example, for compiler point of view, input and output data can looks like overlapped or for floating point type it tried to keep calculation sequence. Also, correctness is always emphasized over performance. Try to help compiler generate better code.
- If previous step not help – use intrinsics. They are extremely powerful, but after this step you too close to CPU architecture and at least should check if this intrinsics supported by current CPU.

Compute bound: multithreading

- Simplest way to calculate this optimization potential – set process affinity to one CPU core and benchmark result.
- Important decisions: task granularity size,
- Should be memory bound reserve

Compute bound: fixed point arithmetic

Pros

- Wide types to choose – from s8 to u64
- Faster calculation, in common
- Stability from the order of calculations

Cons

- Overflow possibility
- Worse precision, in common
- Often generate slightly different result compared to reference floating point version
- Code can be harder to understand

Compute bound: loop unrolling

- Modern CPU has very long computation pipeline
- Small size loop generate a lot of small pieces of code, divided by brunches from loop. Compiler haven’t information about typical picture width or height, for example. We should provide it.
- Several concatenated loop pieces give much more space to compiler optimizations. In addition, total brunches count will be smaller – that reduce overhead from loop.
- Don’t forget about tile calculation.

Compute bound: if-conversion

- Branches inside loops can dramatically drop performance. Moreover, loops with them is really hard to vectorize.
- If branch can be moved outside loop – compiler already did that in 99% cases.
- Most common way to solve this problem – is using masks, combined with binary operations. All vector comparison generate mask as output. Simplest way to concatenate vector results from different branches – using *blend* intrinsics.

Write pure code

Pure function is a function for which the following statements are true:

- always evaluates the same result on the same argument value(s)
- result must not depend on any hidden information or state that may change while program execution proceeds or between different executions of the program, as well as on any external input from I/O devices
- Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices

Q/A