Code Quality Analyzer (CQA)

CQA for IA32, x86-64 and AArch64 architectures

Version 2.2

November 2024

www.maqao.org

1 Introduction

MAQAO-CQA (MAQAO Code Quality Analyzer) is the MAQAO module addressing the code quality issues. Based on a detailed performance model, MAQAO-CQA (i) returns a lower bound on the number of cycles needed to run a binary code fragment, (ii) estimates performance gain if resources were optimally used. It processes the binary code statically, hence the binary code does not have to be run to be analyzed. And it assumes that most of execution time is spent in loops.

MAQAO-CQA compares a binary code against a given machine model and determines the location of the performance bottlenecks. In order to do so, some assumptions are made such as infinite loop trip count and the absence of dynamic hazards such as denormalized numbers and so on. This tutorial deals with the command line version of MAQAO-CQA.

2 Analyzing performance

2.1 Compilation

For a better experience, please compile with -g. Remark: with Intel compilers, -g implies -O0 (no optimization) and requires you to explicit your optimization level (default is O2). To analyze loops in the “my_div” function defined in my_div.c, MAQAO can use either the div.o object file or the whole application executable. Analysis will be faster with the object file. Instead of specifying functions, you can directly analyze binary loops by their MAQAO identifier (displayed by the MAQAO profiler).

2.2 Running MAQAO-CQA on loops

```
maqao cqa <BINARY_FILE> fct-loops=<FUNCTION> uarch=<MICRO-ARCHITECTURE>
```

Name of the binary file to analyze (or path if not present in the current directory)

Most recent supported micro-architectures (non exhaustive list, CF maqao --list-procs):

- Intel: Sierra Forest and Granite Rapids
- AMD: Zen 5
- ARM: Neoverse V2 (AWS Graviton 4 and NVidia Grace)

Name of functions to analyze. You can give a list of regexps: foo,^bar$ will match foo29, my_foo and bar but not my_bar.

Identifier of loops to analyze (comma-separated). Ex: 17,458

```
maqao cqa <BINARY_FILE> loop=<LOOP_ID> uarch=<MICRO-ARCHITECTURE>
```

The module can be invoked either by specifying a function to analyze (all the **innermost** loops) or directly a set of loops (MAQAO loop ids).

The output report is printed on the standard output.

N.B.:If you want to analyze performance of a code for a machine with the same micro-architecture as the machine you are running MAQAO and if this micro-architecture is supported, you can omit to specify the micro-architecture.
To list all options or available micro-architectures:

```
maqao cqa –help or man maqao-cqa
```

2.3 Running MAQAO-CQA on advanced targets

To analyze, in a given function, the code that is **outside loops** (and, by extension, functions not containing loops), use **`fct-body`** instead of `fct-loops`.

```
maqao cqa <BINARY_FILE> fct-body=<FUNCTION> …
```

To analyze a specific code path (ordered sequence of basic blocks), use **`path=<comma-separated list of basic block IDs>`** instead of `fct-loops`.

```
maqao cqa <BINARY_FILE> path=48,49,52 …
```

One way to get basic blocks ID is to run (B30 = block ID):

```
maqao analyze -li <BINARY_FILE> fct=<FUNCTION>
 0x402b40[B30]: PUSH %R12
 0x402b40[B30]: PUSH %R12
```

Remark: to analyze a asm/object file containing a single basic block (typically a micro-benchmark kernel), use `path=0`

2.4 Confidence levels

CQA filters information by "confidence levels":

- Gain: following CQA reports will result into speedup
- Potential: good chance to gain
- Hint: not sure but worth be tried
- Expert: mostly for advanced users: assembly code…

By default, only "gain" and "potential" reports are printed.
To add "hint" reports use:

```
maqao cqa (...) --confidence-levels=gain,potential,hint
```

And for all of them:

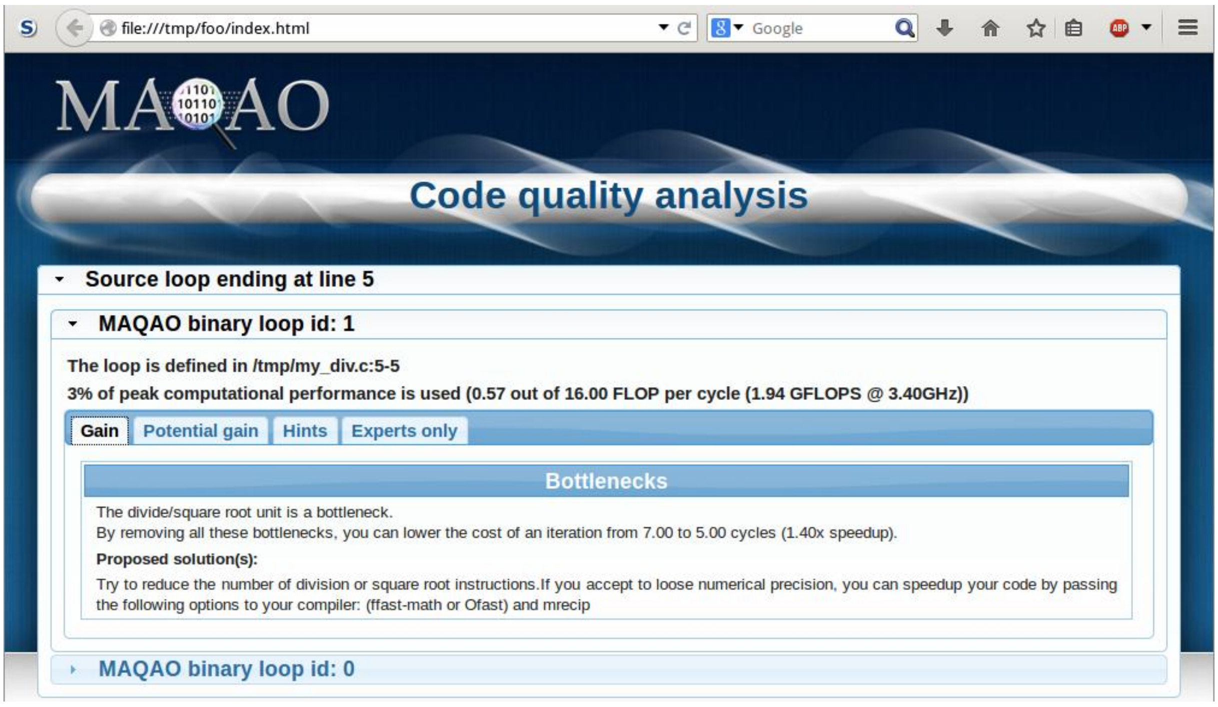
```
maqao cqa (...) --confidence-levels=all
```

2.5 HTML report

HTML output can be generated (and displayed in any web browser) with:

```
maqao cqa (...) --output-format=html --output-path=foo
<my_web_browser> foo/index.html
```

Reports from all confidence levels can be displayed.



Source loop ending at line 5

MAQAO binary loop id: 1

The loop is defined in /tmp/my_div.c:5-5

3% of peak computational performance is used (0.57 out of 16.00 FLOP per cycle (1.94 GFLOPS @ 3.40GHz))

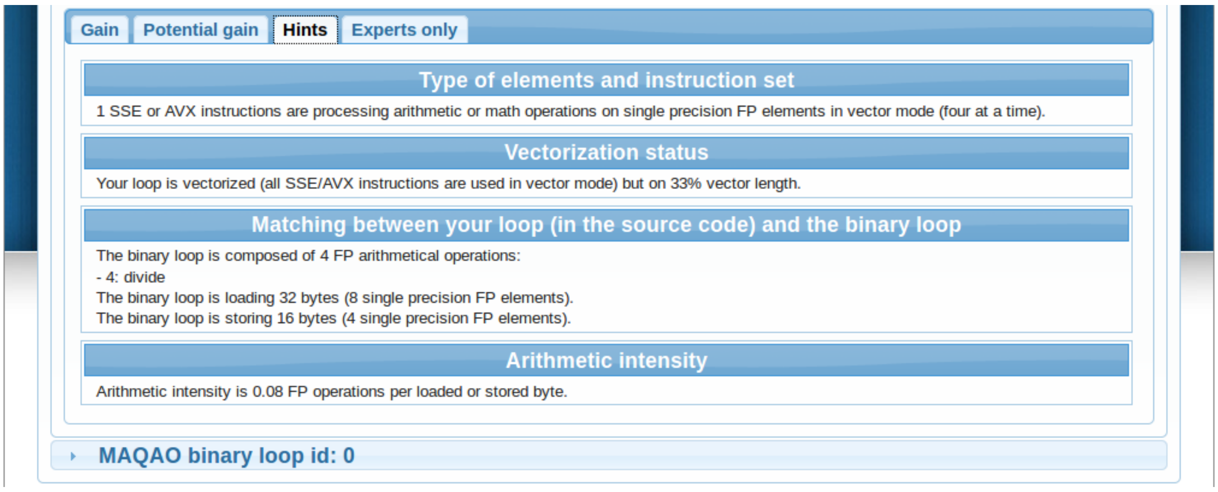
Bottlenecks

The divide/square root unit is a bottleneck.
By removing all these bottlenecks, you can lower the cost of an iteration from 7.00 to 5.00 cycles (1.40x speedup).

Proposed solution(s):

Try to reduce the number of division or square root instructions.If you accept to loose numerical precision, you can speedup your code by passing the following options to your compiler: (ffast-math or Ofast) and mrecip

MAQAO binary loop id: 0



Type of elements and instruction set

1 SSE or AVX instructions are processing arithmetic or math operations on single precision FP elements in vector mode (four at a time).

Vectorization status

Your loop is vectorized (all SSE/AVX instructions are used in vector mode) but on 33% vector length.

Matching between your loop (in the source code) and the binary loop

The binary loop is composed of 4 FP arithmetical operations:
- 4: divide
The binary loop is loading 32 bytes (8 single precision FP elements).
The binary loop is storing 16 bytes (4 single precision FP elements).

Arithmetic intensity

Arithmetic intensity is 0.08 FP operations per loaded or stored byte.

MAQAO binary loop id: 0

2.6 Understanding the output report

2.6.1 Example

Figure 1 shows a simple code example performing a division.

```
/tmp/my_div.c:

  int i;

  for (i=0; i<n; i++)
    c[i] = a[i] / b[i];
}

int main (int argc, char *argv[]) {
...
```

Figure 1

The code is then compiled as follows:

```
gcc -g -O3 my_div.c -o my_div
```

We perform the analysis targeting the **my_div** function and store the output report in the out.txt file.

```
maqao cqa my_div fct-loops=my_div uarch=NEHALEM > out.txt
```

2.6.2 Interpreting the output

Figure 2 present the output report's header which provides a summary of an analyzed (innermost) loop. In our example there is only one innermost loop (which performs the division).

The report is presented hierarchically:

- Function (contains source or binary loops)
- Source loop (contains binary loops)
- Binary loop (contains paths)
- Path (if at least two execution paths)

```
out.txt:
 Section 1: Function: my_div
 ===========================

Code for this function has been compiled to run on any x86_64
processor (SSE2, 2004). It is not optimized for later processors
(AVX etc.).
These loops are supposed to be defined in: /tmp/my_div.c

 Section 1.1: Source loop ending at line 5
 =========================================

 Composition and unrolling
 -------------------------
 It is composed of the following loops [ID (first-last source
line)]:
  - 0 (1-5)
  - 1 (5-5)
 and is unrolled by 4 (including vectorization).

 The following loops are considered as:
  - unrolled and/or vectorized: 1
  - peel or tail: 0
 The analysis will be displayed for the unrolled and/or vectorized
loops: 1

 (report for the loop 1)
```

Figure 2

You can check that your code is vectorized by reviewing the corresponding section of the report:

```
Vectorization
-------------
Your loop is fully vectorized, using full register length.
```

Figure 3

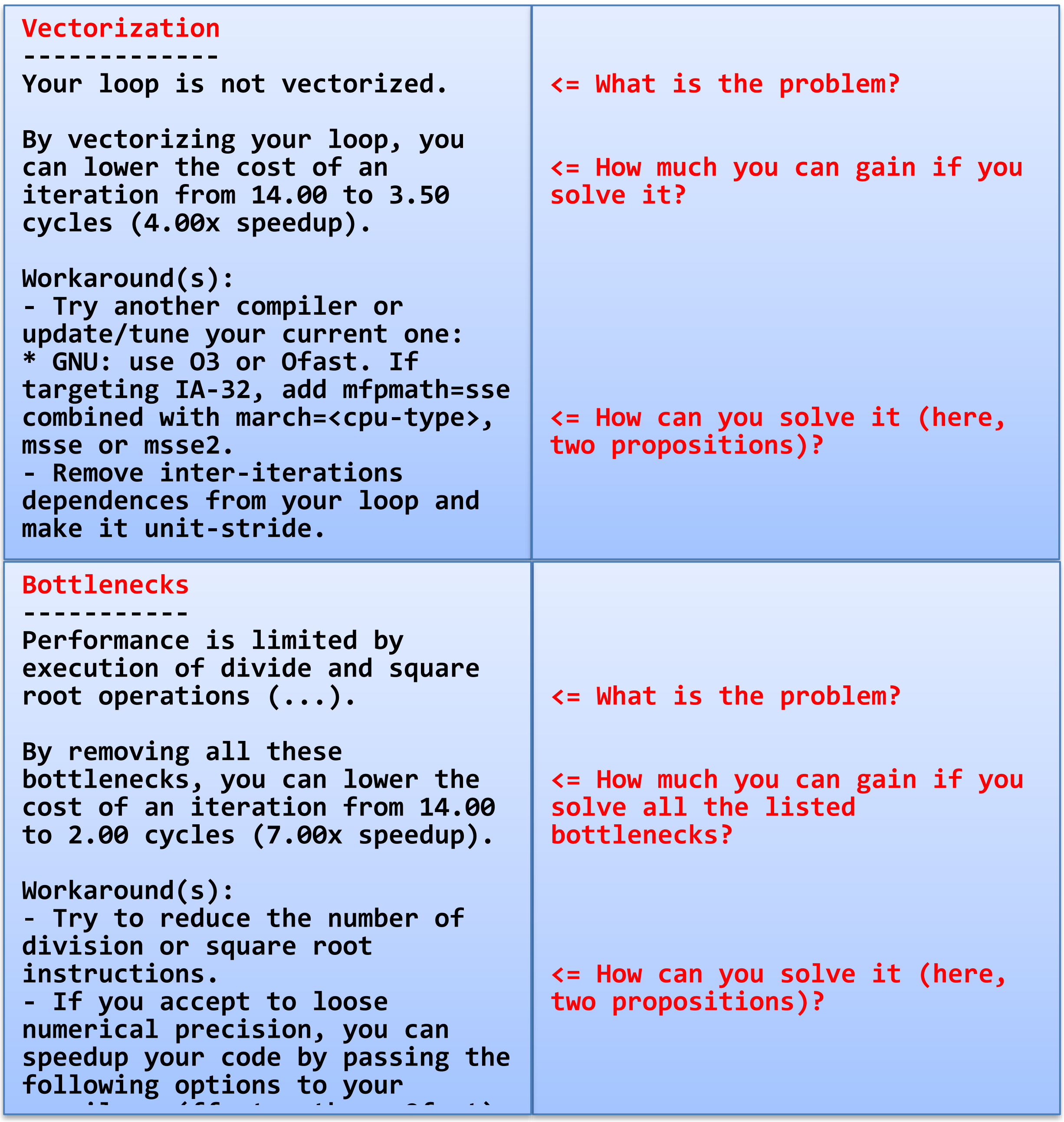
And review cycles and resources usage:

```
Cycles and resources usage
--------------------------
Assuming all data fit into the L1 cache, each iteration of the
binary loop takes 14.00 cycles. At this rate:
- 0% of peak computational performance is reached (0.07 out of
8.00 FLOP per cycle (GFLOPS @ 1GHz))
- 3% of peak load performance is reached (0.57 out of 16.00 bytes
loaded per cycle (GB/s @ 1GHz))
- 1% of peak store performance is reached (0.29 out of 16.00 bytes
stored per cycle (GB/s @ 1GHz))
```

Figure 4

To optimize your code (or check if already “statically optimal”), review the “pathological cases” section (and then, “bottlenecks”). For some of reported items, you can found answers to three critical questions:

- what is the problem?
- how much you can gain if you solve it?
- how you can solve it?



```
Vectorization
-------------
Your loop is not vectorized.

By vectorizing your loop, you
can lower the cost of an
iteration from 14.00 to 3.50
cycles (4.00x speedup).

Workaround(s):
- Try another compiler or
update/tune your current one:
* GNU: use O3 or Ofast. If
targeting IA-32, add mfpmath=sse
combined with march=<cpu-type>,
msse or msse2.
- Remove inter-iterations
dependences from your loop and
make it unit-stride.
```

```
<= What is the problem?

<= How much you can gain if you
solve it?

<= How can you solve it (here,
two propositions)?
```

```
Bottlenecks
-----------
Performance is limited by
execution of divide and square
root operations (...).

By removing all these
bottlenecks, you can lower the
cost of an iteration from 14.00
to 2.00 cycles (7.00x speedup).

Workaround(s):
- Try to reduce the number of
division or square root
instructions.
- If you accept to loose
numerical precision, you can
speedup your code by passing the
following options to your
```

```
<= What is the problem?

<= How much you can gain if you
solve all the listed
bottlenecks?

<= How can you solve it (here,
two propositions)?
```

Figure 5

2.6.3 Other possible outputs

The previous example introduces a subset of the available issues. The following table extends it with other available hints (non exhaustive list).

Composition and unrolling ------------------------- It is composed of the loop 0 and is not unrolled or unrolled with no peel/tail code (including vectorization). The analysis will be displayed for the first found loop: 0	The "Composition and unrolling" paragraph explains how the source loop was break down to binary loops by your compiler. If the (source) loop is unrolled and/or vectorized, it will contain in most cases at least two binary loops: the main loop and a tail loop to process leftover iterations when the loop trip count is not a multiple of the unroll factor. If the loop is vectorized, two extra loops are often generated to increase the proportion of vector aligned loads/stores: another main loop with a different memory offset and a peel loop to set the first iteration of a main loop on a vector-aligned address.
Type of elements and instruction set ---------------------------------- 1 SSE or AVX instructions are processing arithmetic or math operations on single precision FP elements in scalar mode (one at a time).	This paragraph explains how your source will be mapped in assembly instructions to process your data. You will know which type of instructions was generated (arithmetic, math...), on which type of elements it will operated (single or double precision FP element, integers...) and how many elements at a time (one=scalar instructions or more=vector instruction).
Vectorization ------------- Your loop is not vectorized.	This paragraph tells you if your loop were vectorized or not.
Matching between your loop (...) -------------------------------- The binary loop is composed of 1 FP arithmetical operations: - 1: divide The binary loop is loading 8 bytes (2 single precision FP elements). The binary loop is storing 4 bytes (1 single precision FP elements). Arithmetic intensity is 0.08 FP operations per loaded or stored byte.	This paragraph gives the matching between your loop (in the source code) and the binary loop which is useful to: • check the unroll factor and vectorization • see how the work exposed at source level is spread in the different binary loops Arithmetic intensity displays the ratio between computation load and memory load, that is number of FP arithmetic operations divided by number of loaded/stored bytes

`Cycles and resources usage` `--------------------------` `Assuming all data fit into the L1 cache, each iteration of the binary loop takes 14.00 cycles. At this rate:` `- 0% of peak computational performance is reached (0.07 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))` `- 3% of peak load performance is reached (0.57 out of 16.00 bytes loaded per cycle (GB/s @ 1GHz))` `- 1% of peak store performance is reached (0.29 out of 16.00 bytes stored per cycle (GB/s @ 1GHz))`	This paragraph explains how well the assembly code can use computational and memory units in the specified processor. On optimal conditions (infinite trip count, all data in L1, no branch mispredictions...), it will give the minimal cost in cycles for one (binary) loop iteration. To translate to source loop iterations, use previous paragraphs.
`Bottlenecks` `-----------` `Performance is limited by execution of divide and square root operations (the divide/square root unit is a bottleneck).` `By removing all these bottlenecks, you can lower the cost of an iteration from 14.00 to 2.00 cycles (7.00x speedup).` `Workaround(s):` `- Reduce the number of division or square root instructions:` `* If denominator is constant over iterations, use reciprocal (...)` `- If you accept to loose numerical precision up to 2 ulp, you can speedup your code by passing the following options to your compiler: (ffast-math or Ofast) and mrecip`	This paragraphs lists performance bottlenecks. Fixing pathological cases will fix most critical ones. This is why the user is invited to fix as many of them as he can before reading this section.

A very important point to check is vectorization. A loop is said “vectorized” if the compiler generated vector instructions to process iterations, that is instructions processing in parallel multiple data (using vector registers). In general, a loop is vectorized if it processes consecutive elements (in that case, elements in vector registers are consecutive in memory). On the report, check the following paragraphs (on the following examples, 32 bits FP elements can be processed four at a time for the same cost when vectorized).

Not vectorized	Vectorized

`Composition and unrolling`
`-------------------------`
`It is composed of the loop 0`
`and is not unrolled or unrolled`
`with no peel/tail code (including`
`vectorization).`
	`Composition and unrolling`
	`-------------------------`
	`It is composed of the following`
	`loops [ID (first-last source`
	`line)]:`
	`- 0 (1-5)`
	`- 1 (5-5)`
	`and is unrolled by 4 (including`
	`vectorization).`
	
	`The following loops are`
	`considered as:`
	`- unrolled and/or vectorized: 1`
	`- peel or tail: 0`
	`The analysis will be displayed`
	`for the unrolled and/or`
	`vectorized loops: 1`

`Type of elements and instruction set`
`------------------------------------`
`1 SSE or AVX instructions are`
`processing arithmetic or math`
`operations on single precision FP`
`elements in scalar mode (one at a`
`time).`
	`Type of elements and instruction set`
	`------------------------------------`
	`1 SSE or AVX instructions are`
	`processing arithmetic or math`
	`operations on single precision FP`
	`elements in vector mode (four at`
	`a time).`

`Vectorization`
`-------------`
`Your loop is not vectorized.`
`All SSE/AVX instructions are used`
`in scalar version (...).`
	`Vectorization`
	`-------------`
	`Your loop is fully vectorized,`
	`using full register length.`
	`All SSE/AVX instructions are used`
	`in vector version (...).`

`Matching between your loop (...)`
`--------------------------------`
`The binary loop is composed of 1`
`FP arithmetical operations:`
`- 1: divide`
`The binary loop is loading 8 bytes`
`(2 single precision FP elements).`
`The binary loop is storing 4 bytes`
`(1 single precision FP elements).`
	`Matching between your loop (...)`
	`--------------------------------`
	`The binary loop is composed of 4`
	`FP arithmetical operations:`
	`- 4: divide`
	`The binary loop is loading 32`
	`bytes (8 single precision FP`
	`elements).`
	`The binary loop is storing 16`
	`bytes (4 single precision FP`
	`elements).`

3 Expert

Some items (in *Italic*) are architecture-specific. Presented below: x86-64.

3.1 General Properties

- nb instructions: number of instructions
- nb uops: number of front-end (decoded) micro-ops, typically 1 per instruction (minus 1 if fused compare and branch)
- loop length: code size in bytes
- used registers: number of distinct registers used
    - *x86 registers: general purpose (scalar integer)*
    - *used mmx registers: MMX (legacy SIMD)*
    - *used xmm registers: XMM (SIMD 128 bits)*
    - *used ymm registers: YMM (SIMD 256 bits)*
    - *used zmm registers: ZMM (SIMD 512 bits)*
    - nb stack references: number of stack-pointer relative operands, typically corresponds to spill/fill (lower is better)

3.2 Front-end

- ASSUMED MACRO FUSION: if displayed, macro fusion (compare and branch) is applied. Allows to spare one front-end micro-operation
- FIT IN UOP CACHE: if displayed, loop length and nb uops are below micro-op cache capacity. Streaming from uop cache is faster (that refetching/decoding instructions at each loop iteration)
- micro-operation queue: number of cycles needed to stream micro-ops from the uop queue that is between the front- and the back-end
- front end: number of cycles to stream micro-ops from the overall front-end (fetching, decoding, micro-operation queue)

3.3 Back-end

- uops PX: number of (back-end) micro-ops that will flow through the PX execution port/unit
- cycles PX: minimum number of cycles required to stream uops through that unit/port
- Cycles executing div or sqrt instructions: number of cycles spent in the divide/square-root unit

- Longest recurrence chain latency (RecMII): minimum number of cycles caused by inter-iteration dependencies. For a loop summing elements inside an array, that metric equals to the latency of the related ADD instruction

3.4 Front-end and detailed OoO resources (UFS) (x86 only)

UFS (Uops Flow Simulator), contrary to CQA, is a pseudo cycle accurate simulator that takes into account (limited) buffers size and simulate what happens every cycle for instructions and related micro-operation. It is presently supported only for the x86 architecture.

- FE+BE cycles: min and max number of cycles for overall core (Front-end and Back-end)
- Stall cycles: number of stall cycles during which core cannot dispatch/retire any instruction. Lower is better
- X full (events): number of times the X resource is full (and then not stalled and no more accepting new micro-operations)

3.5 Cycles Summary

- Front-end: idem "front end" in the previous "Front-end" section, number of cycles spent in the front-end
- Dispatch: number of cycles spent in the most contended execution port/unit, in other words is MAX (PX)
- Data deps.: idem "Longest recurrence chain latency (RecMII)" in the previous "Back-end" section
- Overall L1: MAX across the 3 previous items

3.6 Vectorization ratios

Proportion of vectorizable instructions that are actually vectorized. Higher is better.

That section provides a breakdown per element type (integer vs FP) and then per operation type (add/sub, mul etc.).

3.7 Vector efficiency ratios

Will/must be renamed to vector length use/efficiency. Effective/average relative length/width used in vector/SIMD registers. Higher is better.

That section provides a breakdown similar to section 3.6.

Remark: Vectorization ratio = 0 does not imply vector efficiency = 0 since some scalar (not vectorized) instructions actually use some bits in vector registers (for instance 32 bits for a scalar instruction dealing with FP32 numbers).

3.8 Cycles and memory resources usage

Compare the CQA-computed load/store/compute metrics with the capacity of the core. For instance if CQA computes 24 bytes in the loop in 2 cycles (that is 12 cycles per cycle) and if the core can process up to 16 bytes per cycle, that metric is 75%.

3.9 Front-end bottlenecks

List of bottlenecks detected by CQA. A bottleneck is an issue which reduces front-end throughput below the core capacity.

3.10 ASM code

For each instruction, details:

- Nb FU: number of Fused (Front-end) uops, typically 1 (fast path). 2 or more corresponds to complex and/or micro-coded instructions (slow path)
- PX: number of uops going through PX. If an instruction can go to X ports, value is 1/X. For instance if an instruction dispatch a micro-operation that can go to P2 or P3: 0.50 in P2 and 0.50 in P3
- Latency: instruction latency (number of cycles needed for execution)
- Recip. throughput: reciprocal throughput, that is average number of cycles to retire that instruction if repeated (assuming independent instructions, with no read-after-write dependency)