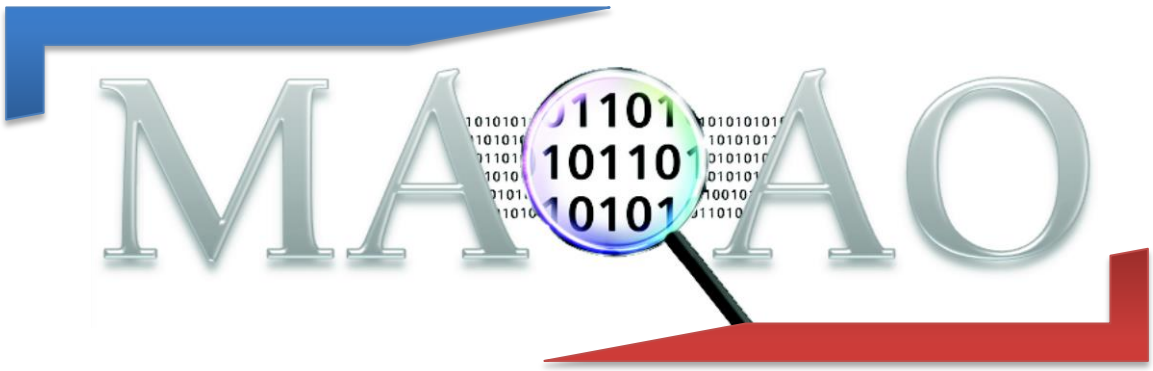


Code Quality Analyzer (CQA)

CQA for Intel 64 architectures

Version 1.1

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.

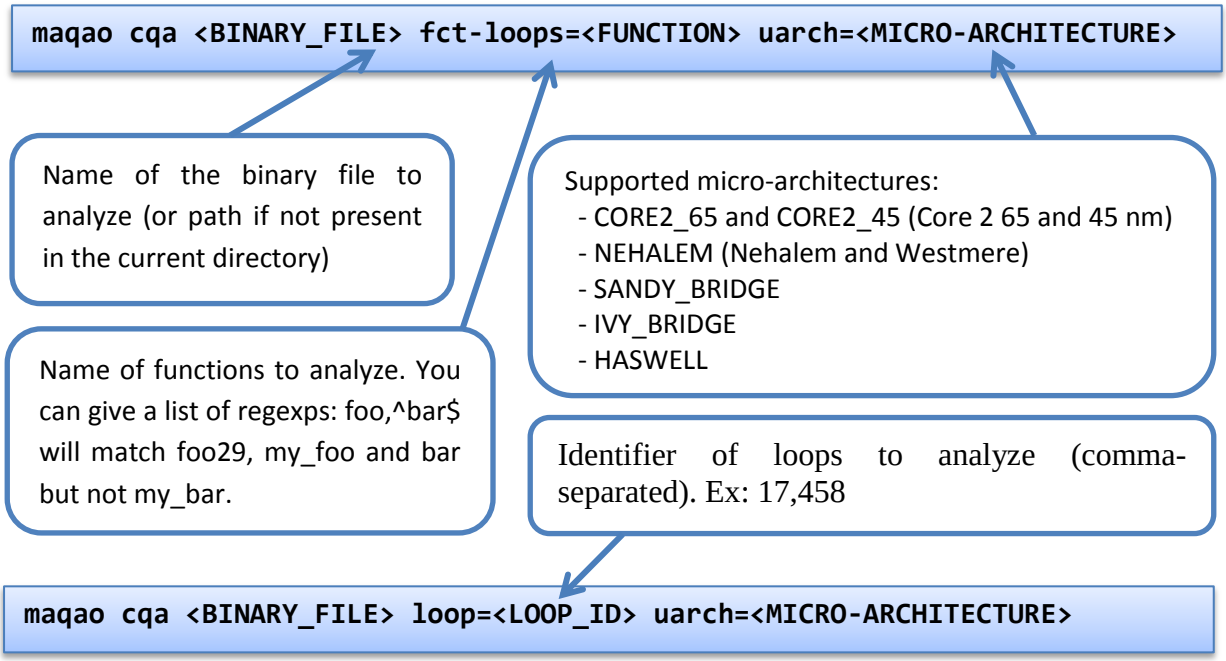
MAQAO-CQA compares a binary code against a given machine model and determines the location of the performance bottlenecks. In order to do so, some assumptions are made such as infinite loop trip count and the absence of dynamic hazards such as denormalized numbers and so on. This manual deals with the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



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 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:

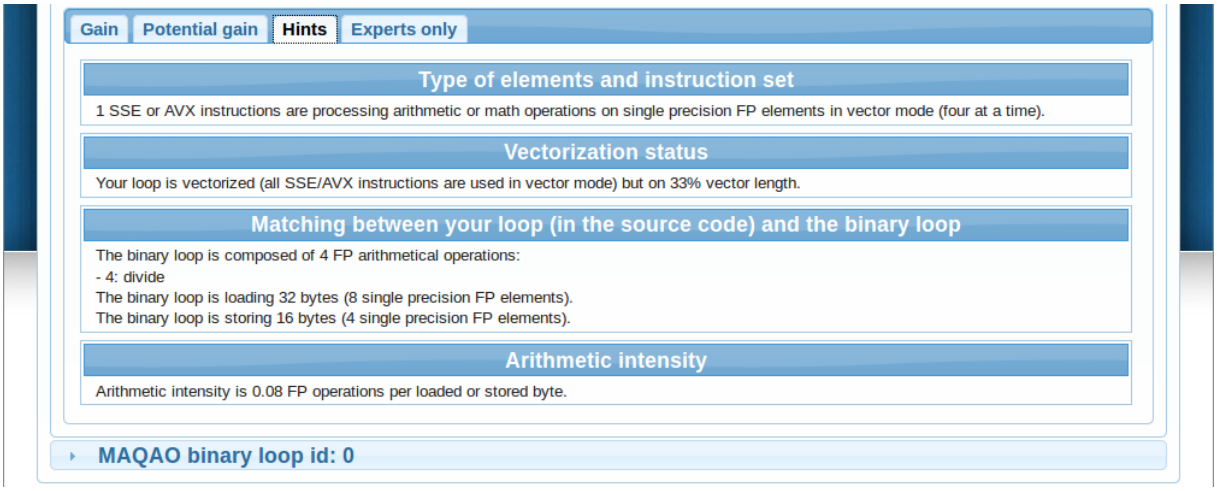
```
maqao cqa (...) --confidence-levels=all
```

2.4 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.



2.5 Understanding the output report

2.5.1 Example

Figure 1 shows a simple code example performing a division.

```
/tmp/my_div.c:
1
2   int i;
3
4   for (i=0; i<n; i++)
5     c[i] = a[i] / b[i];
6 }
7
8 int main (int argc, char *argv[]) {
9 ...
```

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
```

2.5.2 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
 ===========================

 Section 1.1: Source loops in the function named my_div
 ======================================================

 These loops are supposed to be defined in: /tmp/my_div.c

 Section 1.1.1: 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 (all SSE/AVX instructions are used
in vector mode).
```

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 ?

Pathological cases ---------- Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED.	<= What is the problem ?
Since your execution units are vector units, only a fully vectorized loop can use their full power. By fully vectorizing your loop, you can lower the cost of an iteration from 14.00 to 3.50 cycles (4.00x speedup).	<= How much you can gain if you solve it ?
Two propositions: - 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.	<= How can you solve it (here, two propositions) ?
Bottlenecks ---------- The divide/square root unit is a bottleneck. Try to reduce the number of division or square root instructions.	<= What is the problem ?
If you accept to loose numerical precision, you can speedup your code by passing the following options to your compiler: GNU: (ffast-math or Ofast) and mrecip	<= How can you solve it (here, two propositions) ?
By removing all these bottlenecks, you can lower the cost of an iteration from 14.00 to 9.00 cycles (1.56x speedup).	<= How much you can gain if you solve all the listed bottlenecks ?

Figure 5

2.5.3 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 (all SSE/AVX instructions are used in scalar mode).**	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.
Pathological cases **----------** **Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED.** **Since your execution units are vector units, only a fully vectorized loop can use their full power.** **By fully vectorizing your loop, you can lower the cost of an iteration from 14.00 to 3.50 cycles (4.00x speedup).** **Two propositions:** **- 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.** **WARNING: Fix as many pathological cases as you can before reading the following sections.**	This paragraph does not report bottlenecks but bad assembly patterns that can generate bottlenecks. If the loop is not fully vectorized, the speedup factor that can be gained by full vectorization is given. For each pathological case, MAQAO tells you what you can do with your compiler (update, optimization flags) and on your code (loop transformation, pragmas) to help your compiler generate a better code.
Bottlenecks **----------** **The divide/square root unit is a bottleneck.** **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:** **GNU: (ffast-math or Ofast) and**	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.

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

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 mode).**	**Vectorization** **-------------** **Your loop is fully vectorized (all SSE/AVX instructions are used in vector mode).**

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).
Pathological cases --- Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED.	Pathological cases --- (This message is no more displayed)