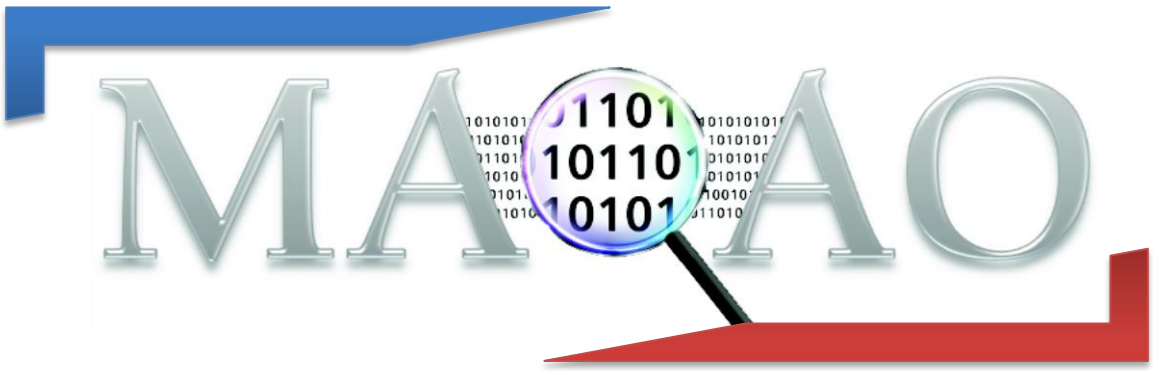


Code Quality Analyzer (CQA)

CQA for Xeon Phi

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.

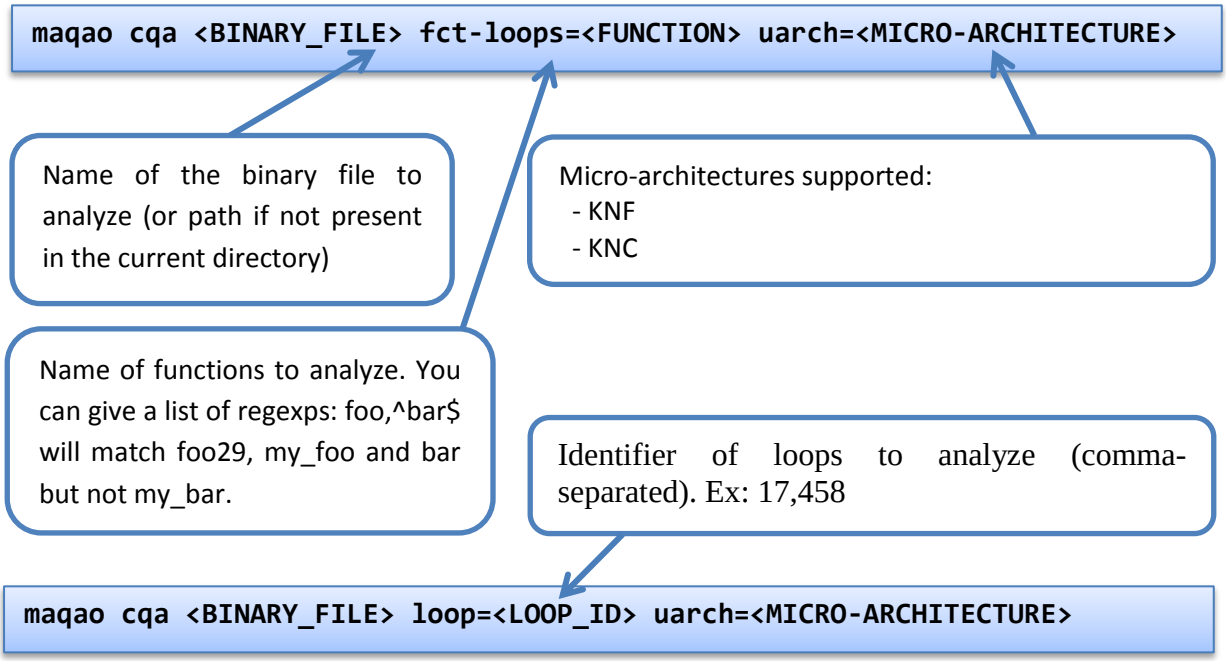
MAQAO-CQA compares a binary code against a given machine model and determines the location of the performance bottlenecks. In order to do so, some assumptions are made such as infinite loop trip count and the absence of dynamic hazards such as denormalized numbers and so on. This manual deals with the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

To list all options or available micro-architectures:

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

2.3 Understanding the output report

2.3.1 Example

Figure 1 shows a simple code example performing a division.

```
/tmp/my_div.c:
1 void my_div (int n, float a[n], float b[n], float c[n]) {
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:

```
icc –mmic -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=KNC > out.txt
```

2.3.2 Controlling confidence of reports in the output

Four levels of confidence are available:

- Gain. Following workarounds will surely provide gain
- Potential. Following workarounds will probably provide speedup
- Hint. Some hints are given but no estimated speedup. Try them
- Expert. Low level metrics will be displayed, they can help only experts

By default, gain and potential reports are reported but you can provide levels you need with the *conf* stub.

```
maqao cqa (…) conf=hint,expert …
```

2.3.3 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 (4-5)
  - 1 (4-5)
  - 4 (4-5)
  - 5 (4-5)
  - 6 (4-5)

 and is unrolled by 32 (including vectorization).

 The following loops are considered as:
  - unrolled and/or vectorized: 0, 4, 5
  - peel or tail: 1, 6
 The analysis will be displayed for the unrolled and/or vectorized
loops: 0, 4, 5
```

Figure 2

Report begins with binary loop location (in source and binary code) and computational resources usage:

```
The loop is defined in /tmp/my_div.c:5-5
In the binary file, the address of the loop is: 68
3% of peak load performance is reached (0.57 out of 16.00 bytes
loaded per cycle (GB/s @ 1GHz))
```

Figure 3

To optimize your code (or check if already “statically optimal”), just review the following reports. For some of reported items, you can found answers to three critical questions:

- what is the problem ?
- how you can solve it ?

Pathological cases **----------** **Detected X87 INSTRUCTIONS.** **x87 is the legacy x86 extension to process FP values. This instruction set is much less efficient that MIC. In particular, it does not support vectorization (x87 units not being vector units).** **If you don’t need 80 bits precision, use only single or double precision elements in your code and then, if necessary, tune your compiler to make it generate only MIC instructions.** **A recent compiler should never generate x87 code as soon as you don’t use any 80 bits FP or complex divides in your code.** **Intel: if complex divides are used, use complex-limited-range that is included in fp-model fast=2 (see manual for safe usage).**	**<= What is the problem ?** **<= How can you solve it ?**

Figure 4

2.3.4 Complete output reference

The previous example introduces a subset of the available issues. The following table provides a complete description of the available hints.

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 **----------** **12 x87 instructions are processing arithmetic or math operations on FP elements in scalar mode (one at**	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

a time).	(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 VPU 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 **----------** **Detected X87 INSTRUCTIONS.** **x87 is the legacy x86 (…)** **Intel: if complex divides are used, use complex-limited-range that is included in fp-model fast=2 (see manual for safe**	This paragraph does not report bottlenecks but bad assembly patterns that can generate bottlenecks. 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

usage). **WARNING: Fix as many pathological cases as you can before reading the following sections.**	compiler generate a better code.

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). On MIC, all loops should be vectorized as soon as only standard data types are used (16/32/64 bits integers, single and double precision FP elements). On the report, check the following paragraphs (on the following examples, 32 bits FP elements can be processed sixteen at a time for the same cost when vectorized).

Not vectorized	Vectorized
Type of elements and instruction set **----------** **2 x87 instructions are processing arithmetic or math operations on FP elements in scalar mode (one at a time).**	**Type of elements and instruction set** **----------** **12 VPU instructions are processing arithmetic or math operations on single precision FP elements in vector mode (up to sixteen at a time).**
Vectorization **----------** **No VPU instruction.**	**Vectorization** **----------** **Your loop is fully vectorized (all VPU instructions are used in vector mode).**
Matching between your loop (...) **----------** **The binary loop is composed of 2 FP arithmetical operations:** **- 2: divide**	**Matching between your loop (...)** **----------** **The binary loop is composed of 256 FP arithmetical operations:** **- (…)** **- 32: fast reciprocal**
Pathological cases **----------** **Detected X87 INSTRUCTIONS.** **(…) does not support vectorization (x87 units not being vector units).**	**Pathological cases** **----------** **(This message is no more displayed)**