MAQAO
Performance Analysis and Optimization Framework

Cédric VALENSI, Emmanuel Oseret
cedric.valensi@uvsq.fr, emmanuel.oseret@uvsq.fr
Performance Evaluation Team, University of Versailles Paris-Saclay
http://www.maqao.org
VI-HPS 47th TW Essen, Germany
02–05 June 2025

Motivating example

Code of a loop representing ~10% walltime

```
do j = ni + nvalue1, nato

    nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
    u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
    rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
    rij = demi*(rvwi + rvwalc1(j))
    drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
    Eq = qq1*qq(j)*drtest
    ntj = nti + ntype(j)
    Ed = ceps(ntj)*drtest2*drtest2*drtest2
    Eqc = Eqc + Eq ; Ephob = Ephob + Ed
    gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
    u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
    g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
    gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
    gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
    gr(nj3, thread_num) = gr(nj3, thread_num) + u3g

end do
```

Where are the bottlenecks?

Motivating example

Code of a loop representing ~10% walltime

6) Variable number of iterations

2) Non-unit stride accesses

1) High number of statements

```
do j = ni + nvalue1, nato
  nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
  u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
  rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
  rij = demi*(rvwi + rvwalc1(j))
  drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
  Eq = qq1*qq(j)*drtest
  ntj = nti + ntype(j)
  Ed = ceps(ntj)*drtest2*drtest2*drtest2
  Eqc = Eqc + Eq ; Ephob = Ephob + Ed
  gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
  u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
  g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
  gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
  gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
  gr(nj3, thread_num) = gr(nj3, thread_num) + u3g
end do
```

4) DIV/SQRT

3) Indirect accesses

5) Reductions

2) Non-unit stride accesses

1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Variable number of iterations

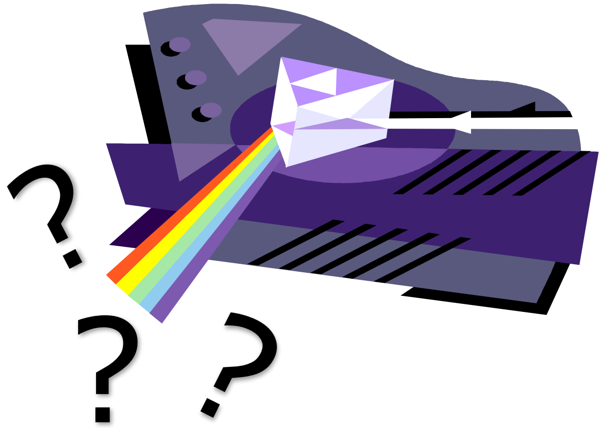
Which is the dominant one?

➔ Need analysis tools to identify performance issues

A multifaceted problem

What type of problems are we facing?

- Identifying the dominant issues: Algorithm, implementation, parallelisation, compilation, …
  - CPU or data access problems
- Making the **best use** of the machine features



What levers do we have to address them?

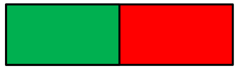
- Compiler switches, Partial/full vectorization
- Loop blocking/array restructuring, If removal, Full unroll
- Binary transforms (prefetch)
- …

➔ **Need for dedicated and complementary tools**

Which issues will be the most rewarding to fix?

- **40%** total time, expected **10%** speedup
  - ➔ TOTAL IMPACT: **4%** speedup

- **20%** total time, expected **50%** speedup
  - ➔ TOTAL IMPACT: **10%** speedup



Our Approach

Nobody wants problems everybody wants solutions ☺

- Focusing on the knobs that code developers can operate:
  - Compiler flags and runtime settings
  - Code restructuring
  - Data restructuring
- Assisting the user in using these knobs

➔ In addition to pinpointing problems, guiding the user towards a way to address them.

Philosophy: Analysis at Binary Level

- Compiler optimizations increase the distance between the executed code and the source code
- Source code instrumentation may prevent the compiler from applying certain transformations
- Allows to be agnostic with regard to compiled source code language

➔ What You Analyse Is What You Run

MAQAO: Modular Assembly Quality Analyzer and Optimizer

Objectives:

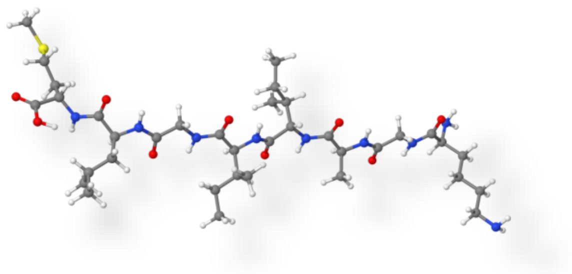
- Characterizing performance of HPC applications
- Focusing on performance at the **core/node level**
- **Guiding users** through the optimization process
- Estimating return on investment (**R.O.I.**)

Characteristics:

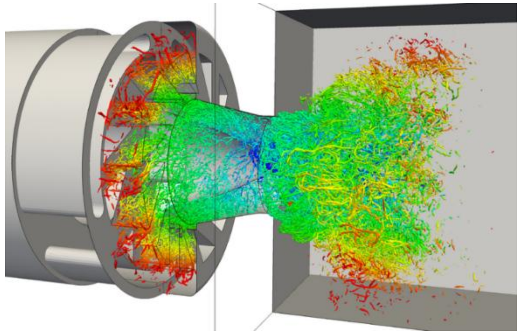
- **Modular tool** offering complementary views
- Support for **x86-64** and **aarch64** (beta version)
  - Work in progress on GPU support
- LGPL3 Open Source software
- Developed at UVSQ since 2004
- Binary release available as a **static executable**

Success stories

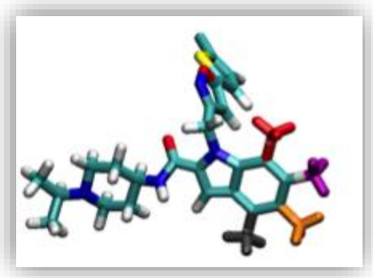
Optimizing industrial and academic HPC applications:



- QMC=CHEM (IRSAMC)
  - Quantum chemistry
  - Speedup: **> 3x**
    - Optimization: moved invocations of functions with identical parameters out of the loop body
- Yales2 (CORIA)
  - Computational fluid dynamics
  - Speedup: **up to 2.8x**
    - Optimization: removing double structure indirections



- Polaris (CEA)
  - Molecular dynamics
  - Speedup: **1.5x – 1.7x**
    - Optimization: enforcing loop vectorization through compiler directives
- AVBP (CERFACS)
  - Computational fluid dynamics
  - Speedup: **1.08x – 1.17x**
    - Replaced divisions by reciprocal multiplications
    - Complete unrolling of loops with a small number of iterations



Partnerships

MAQAO is part of the POP Centre of Excellence

- Provides performance optimisation and productivity services for academic and industrial codes
- https://pop-coe.eu/

MAQAO has been funded by UVSQ, Intel and CEA (French department of energy) through Exascale Computing Research (ECR) and through various European projects (FUI/ITEA: H4H, COLOC, PerfCloud, ELCI, POP2 CoE, TREX CoE, etc...)

Provided core binary analysis and instrumentation capabilities and features for other tools:

- TAU performance tools with MADRAS patcher through MIL (MAQAO Instrumentation Language)
  - X86_64 only, aarch64 under development
- Intel Advisor

MAQAO team and collaborators

MAQAO Team
- William Jalby, Prof.
- Cédric Valensi, Ph.D.
- Emmanuel Oseret, Ph.D.
- Mathieu Tribalat, M.Sc.Eng.
- Hugo Bolloré, M.Sc.Eng
- Kévin Camus, Eng.
- Lucas Neto, Eng.

Collaborators
- David J. Kuck, Prof. (Intel US)
- Pablo de Oliveira, Prof. (UVSQ)
- Eric Petit, Ph.D. (Intel US)
- David C. Wong, Ph.D. (ARM US)
- Othman Bouizi, Ph.D. (Eviden)
- AbdelHafid Mazouz Ph.D.(Intel)
- Jeongnim Kim (Intel US)
- Aurélien Delval, Ph.D. Student (SiPearl)

Past Team Members
- Denis Barthou, Prof. (Huawei)
- Andrés S. Charif-Rubial, Ph.D. (†)
- Jean-Thomas Acquaviva, Ph.D. (DDN)
- Souad Koliaï, Ph.D. (South Pole)
- Zakaria Bendifallah, Ph.D. (Eviden)
- Jean-Baptiste Le Reste, M.Sc.Eng. (AnotherBrain)
- Sylvain Henry, Ph.D. (InputOutput)
- Aleksandre Vardoshvili, M.Sc.Eng.
- Romain Pillot, Eng
- Youenn Lebras, Ph.D. (Noxant)
- Jäsper Salah Ibnamar, M.Sc.Eng.
- Max Hoffer, Eng. (ILL)

Past Collaborators
- Stéphane Zuckerman, Ph.D. (ENSEA)
- Julien Jaeger, Ph.D. (CEA DAM)
- Tipp Moseley, Ph.D. (Google)
- Jean-Christophe Beyler, Ph.D. (Google)
- José Noudohouenou, Ph.D. (AMD)

More on MAQAO

MAQAO website: www.maqao.org

- Mirror: maqao.liparad.uvsq.fr
- Documentation: www.maqao.org/documentation.html
  - Tutorials for ONE View, LProf and CQA
  - Lua API documentation
- Latest release: http://www.maqao.org/download.html
  - Binary releases (2-3 per year)
  - Source code
- Publications around MAQAO: http://www.maqao.org/publications.html
- Repository of MAQAO analyses: http://datafront.maqao.org/public/
  - Mirror: http://datafront.liparad.uvsq.fr/public/
- Email: contact@maqao.org

Useful notions

SIMD/Vectorization/Data Parallelism

- Scalar pattern: a[i] = b[i] + c[i]
- Vector pattern: a(i, i + 8) = b(i, i + 8) + c(i, i + 8)
- Benefits : increases memory bandwidth and **IPC**
- Example implementations :
  - ARM : Neon, SVE
  - x86 : SSE, AVX, AVX512

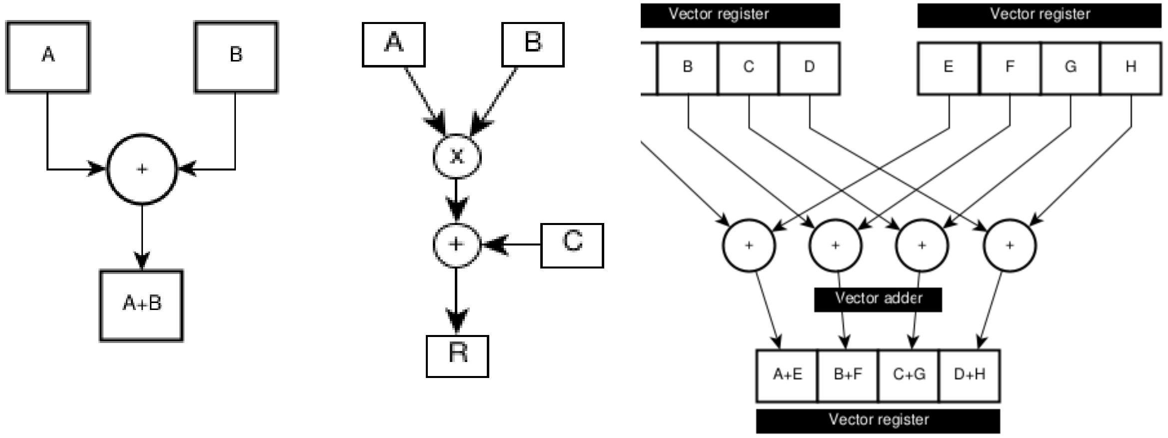
FMA/MAC

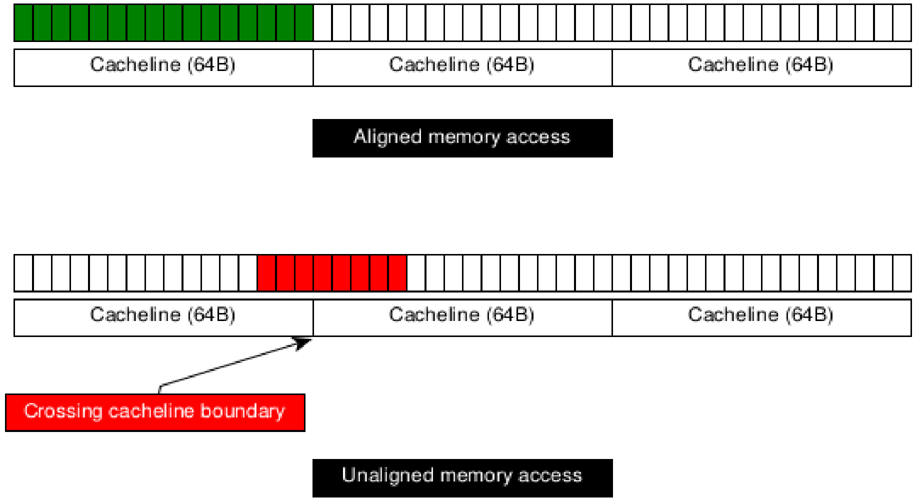
- Fused-Multiply-Add
- Multiply-Accumulate

Memory and caches

- Computations are in general faster than memory accesses
- Alignment/Contiguity of memory (x86) : **posix_memalign**, **aligned_alloc**, …
- Caches: L1, L2, L3, …

Scalar addition | FMA | Vector addition





MAQAO Main Features

Binary layer
- Builds internal representation from binary
  - Construct high level structures (CFG, DDG, SSA, …)
  - Links binary instructions to source code
  - ⚠ A single source loop can be compiled as multiple assembly loops → Affecting unique identifiers to loops
- Allows patching through binary rewriting

Profiling
- LProf: Lightweight sampling-based Profiler operating at process, thread, function and loops level

Static analysis
- CQA (Code Quality Analyzer): Evaluates the quality of the binary code and offers hints for improving it

Performance view aggregation module: ONE View
- Goal: Guiding the user through the analysis & optimization process.
- Synthesizes information provided by different MAQAO modules
- Automatizes execution of experiments invoking other MAQAO modules and aggregates their results to produce high-level reports in HTML or XLSX format

MAQAO LProf: Lightweight Profiler

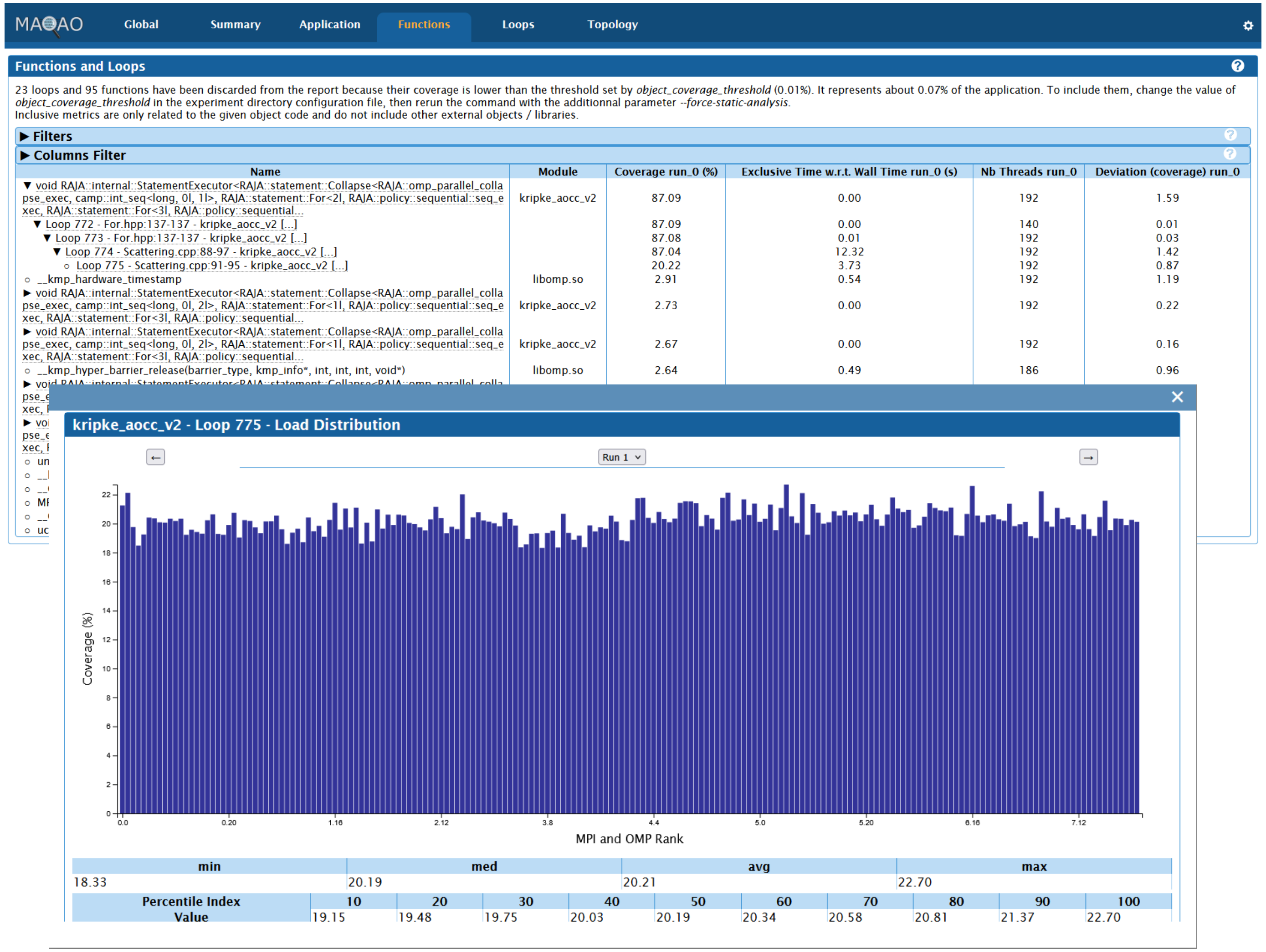
Goal: Lightweight localization of application hotspots

Features:

- Lightweight
- Sampling based
- Access to hardware counters
- Analysis at function and loop granularity

Strengths:

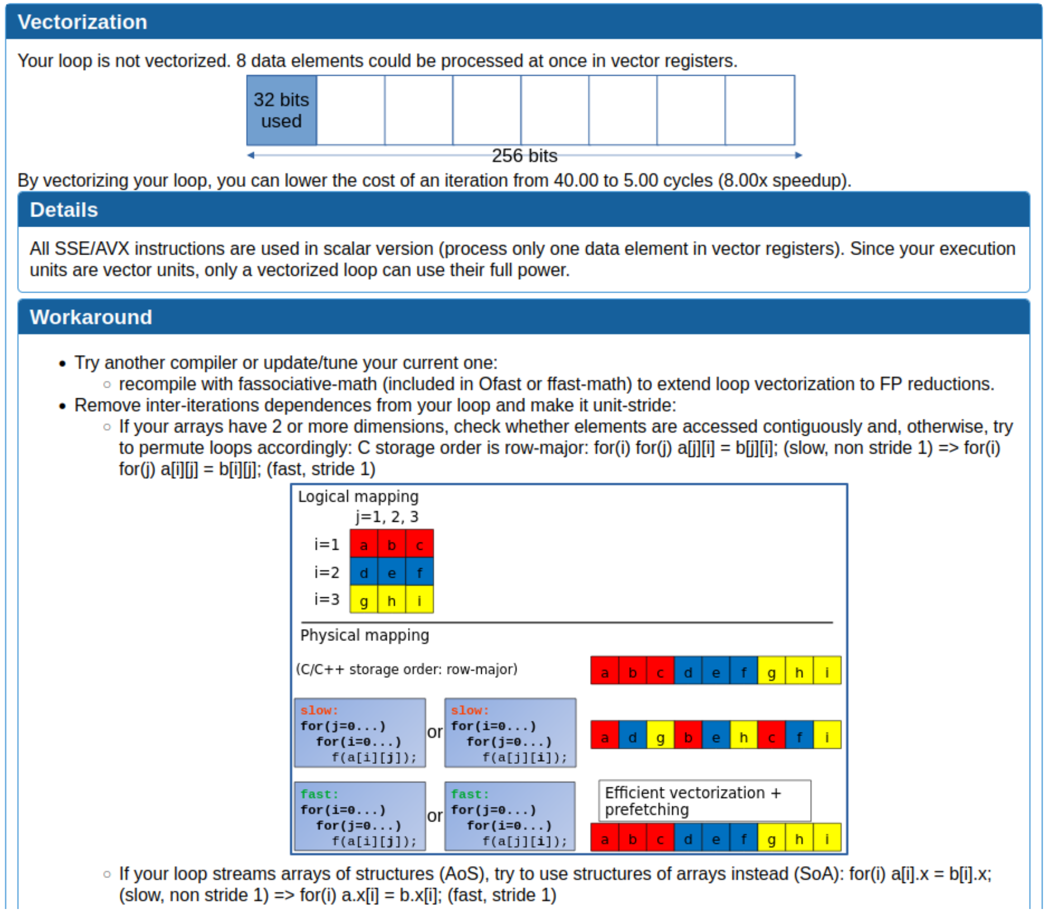
- Non intrusive: No recompilation necessary
- Low overhead
- Agnostic with regard to parallel runtime



MAQAO CQA: Code Quality Analyzer

Goal: Assist developers in improving code performance

Features:
- Static analysis: no execution of the application
- Allows cross-analysis of/on multiple architectures
- Evaluates the quality of compiler generated code
- Proposes hints and workarounds to improve quality/performance
- Loops centric
  - In HPC, loops cover most of the processing time
- Targets compute-bound codes



CQA Performance Predictions: "What If" Scenarios

Objective: Provide optimistic speedups if a given optimization was applied to a loop

- For each optimization, CQA will generate the corresponding ideal assembly code and compute its speedup compared to the original
- These "What If Scenarios" are generated in a fully static manner.

No Scalar Integer: keep only FP Arithmetic and Memory operations, suppress all others

- Scenario: Perfect data access (no address computations)

FP Vectorised: only replace scalar FP Arithmetic by Vector FP Arithmetic equivalent. Generate additional instructions to fill in Vector Registers.

- Scenario: All FP operations successfully vectorised

Fully Vectorised: replace both scalar FP Arithmetic and FP Load/Store by their Vector equivalent.

- Scenario: All operations successfully vectorised

Application to Motivating Example

Issues identified by CQA

6) Variable number of iterations

2) Non-unit stride accesses

```
do j = ni + nvalue1, nato
   nj1 = ndim3d*j + nc ; nj2 = nj1 + nvalue1 ; nj3 = nj2 + nvalue1
   u1 = x11 – x(nj1) ; u2 = x12 – x(nj2) ; u3 = x13 – x(nj3)
   rtest2 = u1*u1 + u2*u2 + u3*u3 ; cnij = eci*qEold(j)
   rij = demi*(rvwi + rvwalc1(j))
   drtest2 = cnij/(rtest2 + rij) ; drtest = sqrt(drtest2)
   Eq = qq1*qq(j)*drtest
   ntj = nti + ntype(j)
   Ed = ceps(ntj)*drtest2*drtest2*drtest2
   Eqc = Eqc + Eq ; Ephob = Ephob + Ed
   gE = (c6*Ed + Eq)*drtest2 ; virt = virt + gE*rtest2
   u1g = u1*gE ; u2g = u2*gE ; u3g = u3*gE
   g1c = g1c –u1g ; g2c = g2c – u2g ; g3c = g3c –u3g
   gr(nj1, thread_num) = gr(nj1, thread_num) + u1g
   gr(nj2, thread_num) = gr(nj2, thread_num) + u2g
   gr(nj3, thread_num) = gr(nj3, thread_num) + u3g
end do
```

1) High number of statements

7) Vector vs scalar

4) DIV/SQRT

3) Indirect accesses

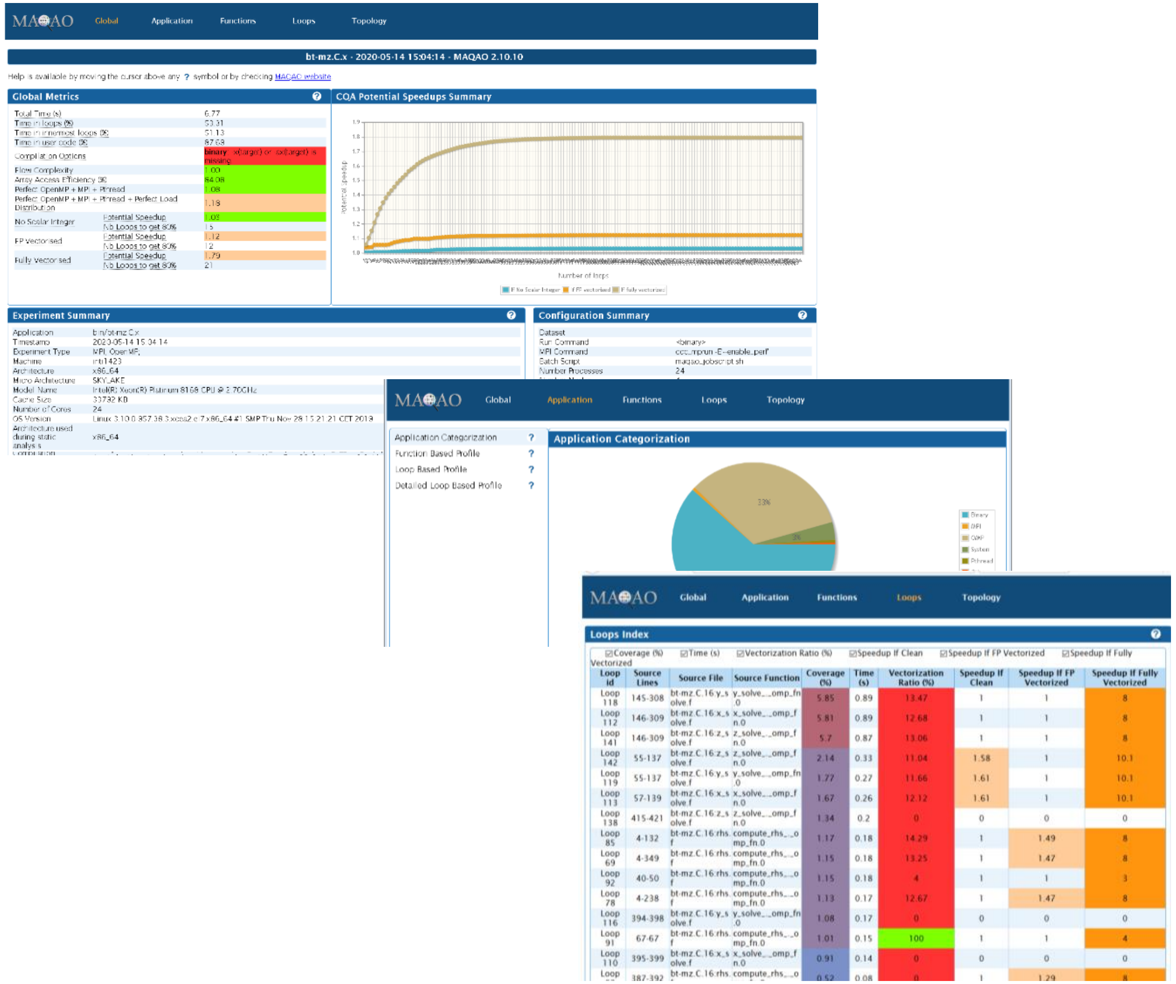
5) Reductions

2) Non-unit stride accesses

CQA can detect and provide hints to resolve most of the identified issues:

1) **High number of statements**
2) **Non-unit stride accesses**
3) **Indirect accesses**
4) **DIV/SQRT**
5) Reductions
6) Variable number of iterations
7) **Vector vs scalar**

MAQAO ONE View: Performance View Aggregator



Automating the whole analysis process

- Takes care of invoking successive MAQAO modules
- Generate aggregated performance views
- Generate a report in HTML format

Main steps:

- Invokes LProf to identify hotspots
- Invokes CQA to analyse hotspots

Available results:

- Speedup predictions
- High-level summary
- Global performance metrics
- Hints for improving performance
- Parallel efficiency analysis

ONE View Reports Levels

ONE View ONE
- Requires a single run of the application
- Profiling of the application using LProf
- Static analysis using CQA

Scalability mode
- Multiple executions with varying parallel configurations
- Allows to evaluate scalability or parallel behaviour of applications

Comparison mode
- Comparison of multiple runs (iso-binary or iso-source)
- Allows to compare performance across different datasets, compilers, or hardware platforms

Stability mode
- Multiple runs with identical parameters
- Allows to assess the stability of execution time

Analysing an application with MAQAO

ONE View execution

- Provide all parameters necessary for executing the application
  - Parameters can be passed on the command line or as a configuration file
  - Parameters include executable name, MPI commands, dataset directory, …

```
$ maqao oneview --create-report=one --mpi_command="mpirun -n 16" -- bt-mz.C.x
```

OR

```
$ maqao oneview --create-report=one --config=my_config.json"
```

- ONE View can reuse an existing experiment directory to perform further analyses
- Results available in HTML format by default
  - XLS spreadsheets and textual output generation are also available

Online help is available:

```
$ maqao oneview --help
```

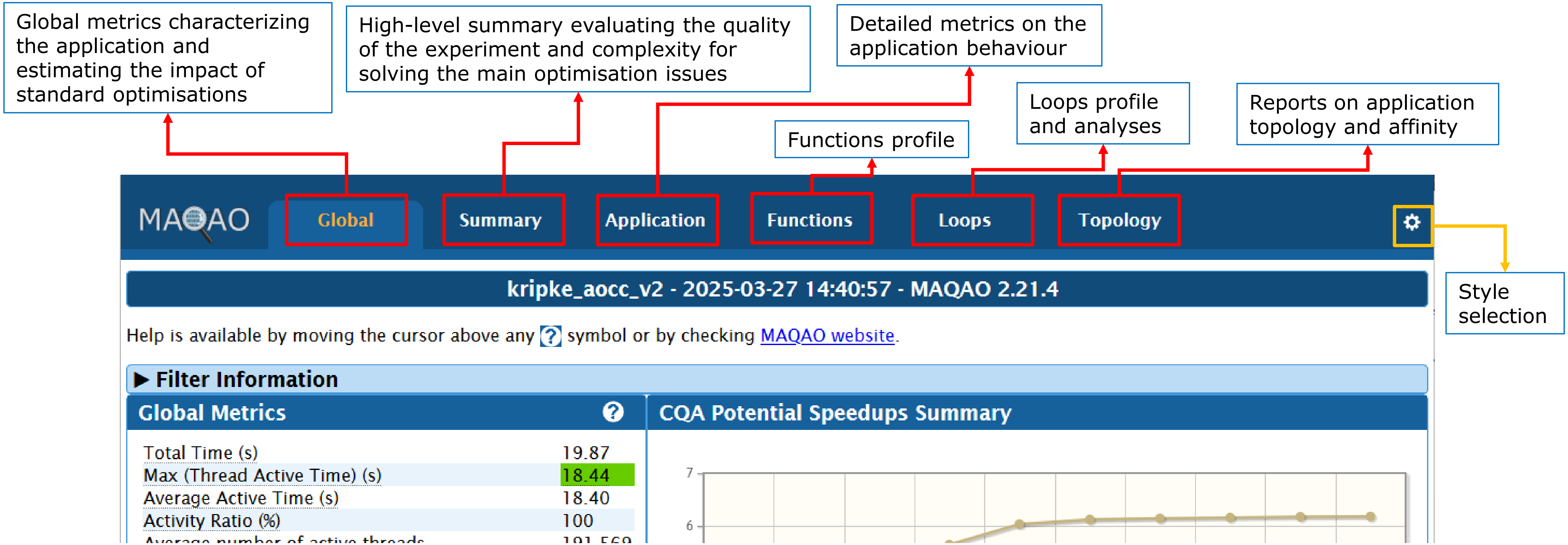
Thank you for your attention !

Questions ?

Navigating ONE View Reports

ONE View main tabs

- ONE View reports are organised among tabs, each regrouping different sets of information

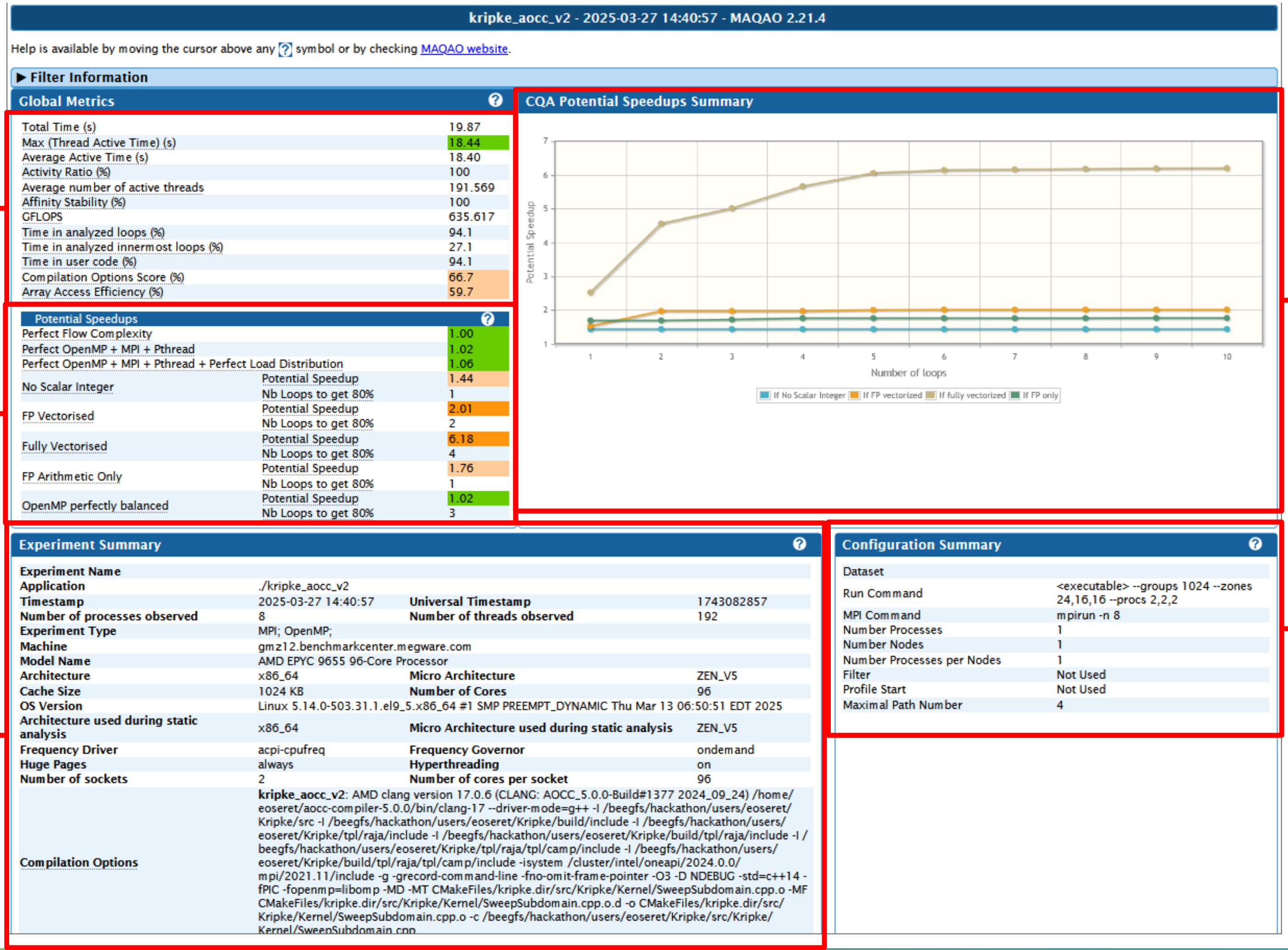


Global tab

Global metrics on the application general behaviour

Maximal speed-ups at application level for various ideal optimisations

Experimental condition: system characteristics, parallelisation, application compilation flags, …



Graphs associated to some of the metrics or speed-ups from the two leftmost panels.

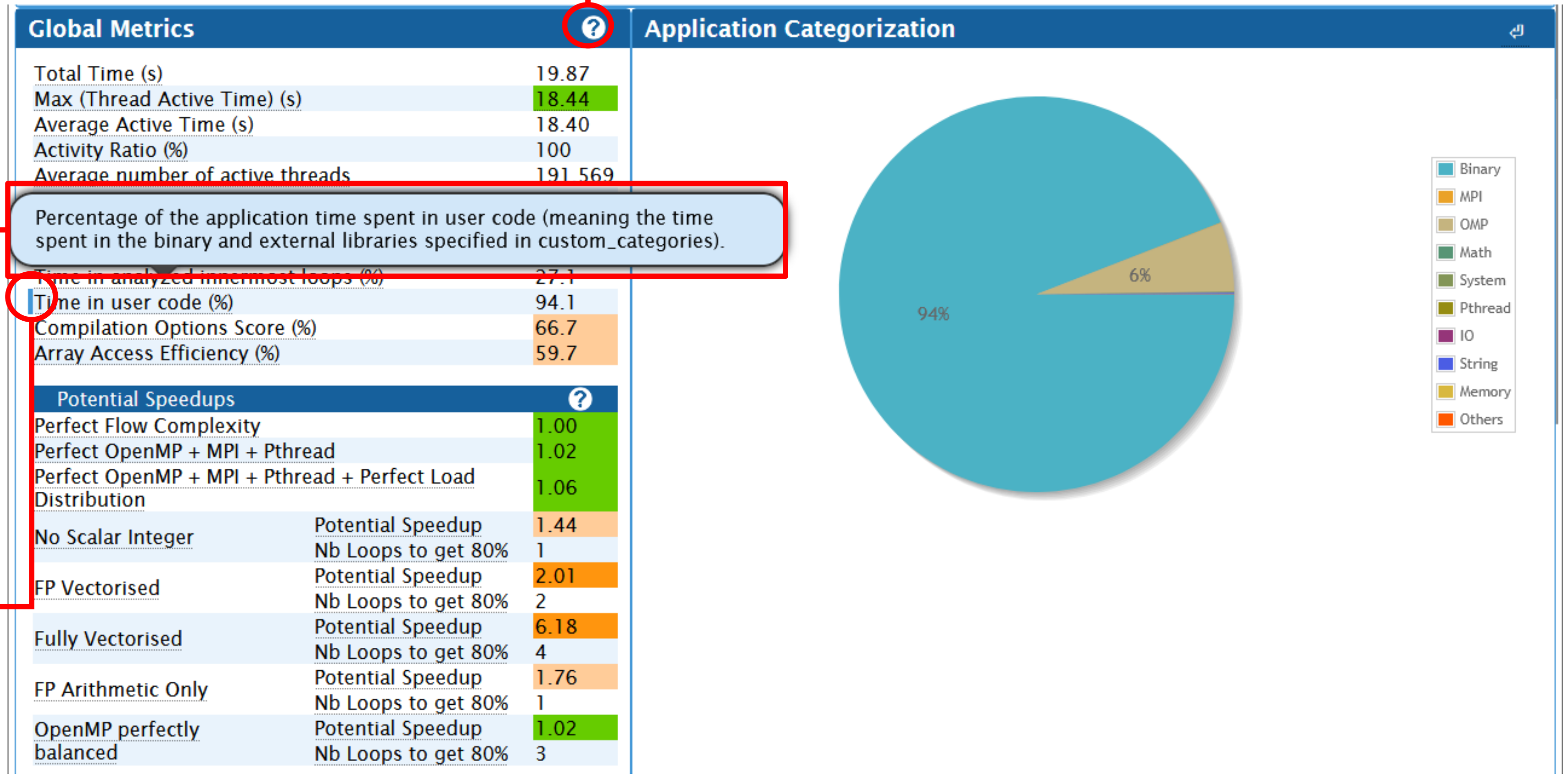
Recap of ONE View configuration

Navigating Global Tab

Hovering over a question mark in a panel header displays a tooltip describing the content of the panel

Hovering over a metric name displays a tooltip describing the metric

When hovering over a metric, a small blue bar appearing on the left means that clicking on the metric will cause a graph to appear on the rightmost panel



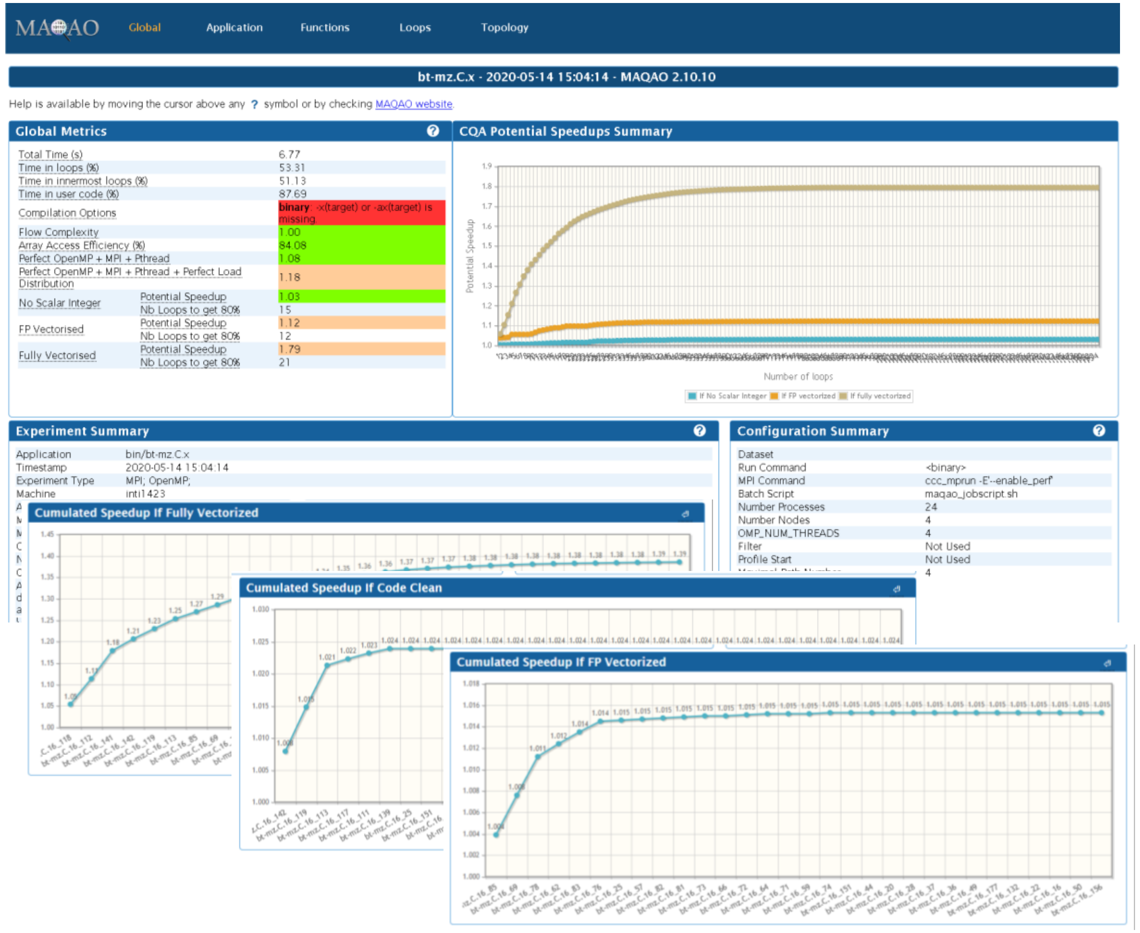
Global Tab Metrics

Experiment summary

- Machine characteristics and configuration

Global metrics

- General quality metrics derived from MAQAO analyses
- Global speedup predictions
  - Speedup prediction depending on the number of vectorised loops
  - Ordered speedups to identify the loops to optimise first

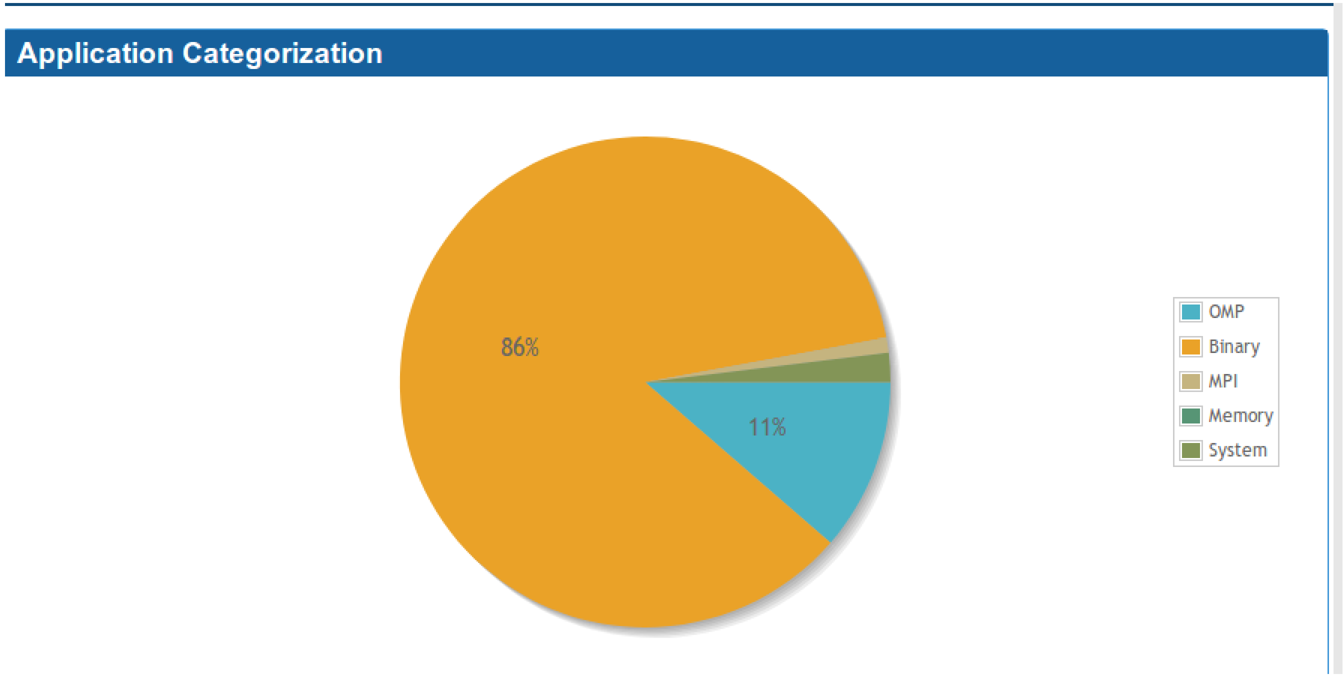


Time Categorisation

Available from the Global Tab and Application Tab

Objective: Identifying at a glance where time is spent

- Application
  - Main executable
- Parallelization
  - Threads
  - OpenMP
  - MPI
- System libraries
  - I/O operations
  - String operations
  - Memory management functions
- External libraries
  - Specialised libraries such as libm / libmkl
  - Application code in external libraries



Summary Tab

Displays an evaluation of the quality of the analysis performed by ONE View. Red items signal issues that could require a rerun of the analysis with the suggested new parameters

Displays an evaluation of the overall optimisation of the code and the expected pitfalls that will be encountered when trying to optimise it.

- The first two tabs will be open by default only if they contain at least one red item

Displays for the hottest loops a list of performance issues and the estimated complexity of their resolutions

Summary Tab: Loop optimisation

Description of a performance issue with associated hint to fix

Estimation of the complexity of fixing this issue (lower is easier)

▼ Optimizer

Loop ID	Analysis	Penalty Score
▼ Loop 774 - kripke_aocc_v2	Execution Time: 66 % - Vectorization Ratio: 16.25 % - Vector Length Use: 14.14 %	
▼ Loop Computation Issues		6
○	[SA] Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA - Reorganize arithmetic expressions to exhibit potential for FMA. This issue costs 4 points.	4
○	[SA] Presence of a large number of scalar integer instructions - Simplify loop structure, perform loop splitting or perform unroll and jam. This issue costs 2 points.	2
▶ Control Flow Issues	+	4
▶ Data Access Issues		8
▶ Vectorization Roadblocks		10
▶ Loop 775 - kripke_aocc_v2	Execution Time: 20 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 983 - kripke_aocc_v2	Execution Time: 2 % - Vectorization Ratio: 100.00 % - Vector Length Use: 25.00 %	
▶ Loop 659 - kripke_aocc_v2	Execution Time: 2 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 1309 - kripke_aocc_v2	Execution Time: 1 % - Vectorization Ratio: 14.71 % - Vector Length Use: 14.34 %	
▶ Loop 1088 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 981 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 773 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 658 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 11.89 %	
▶ Loop 1308 - kripke_aocc_v2	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.28 %	

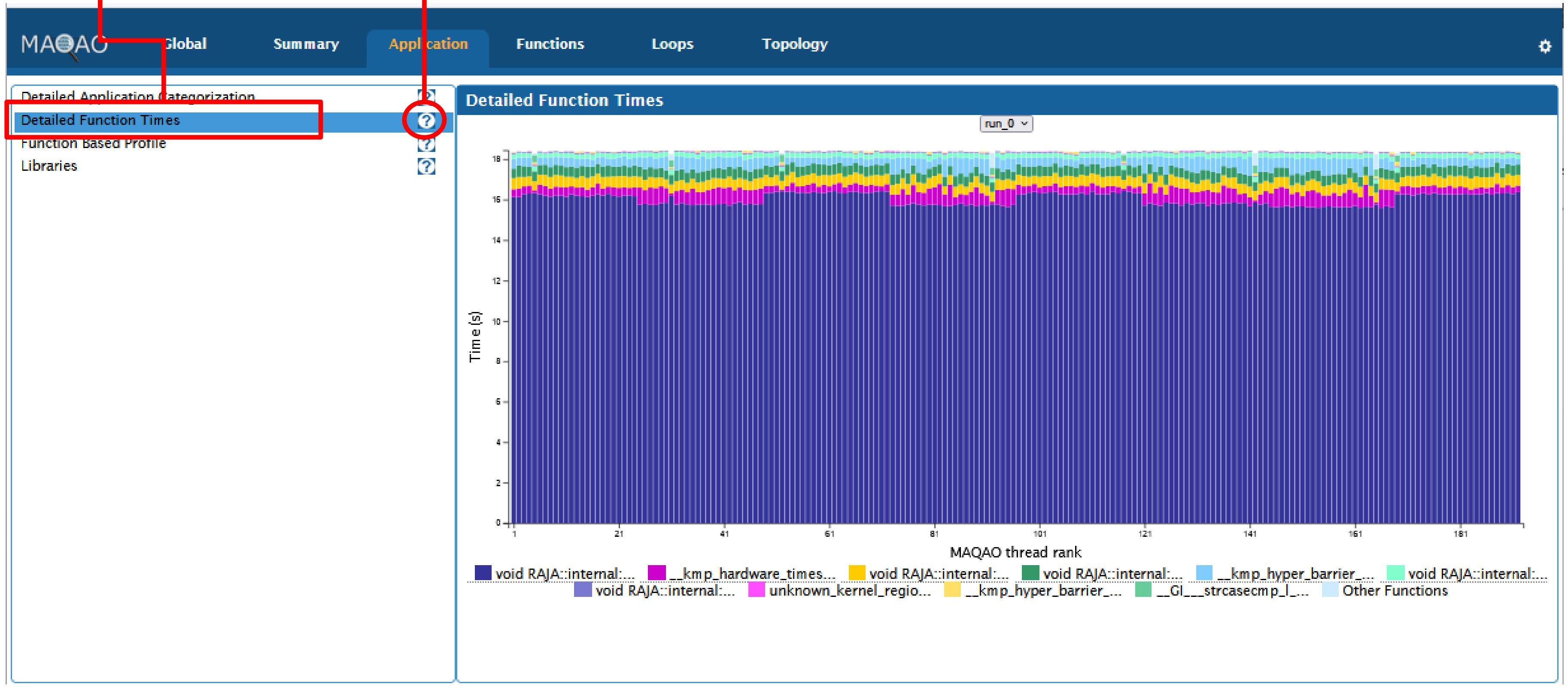
Click the triangle to expand and show all performance issue for a loop

A “+” sign appears when hovering over a line, which allows to expand all at once

Application Tab

Click on an option to display the corresponding graph on the rightmost panel

Hovering over a question mark near an option displays a tooltip describing the associated information

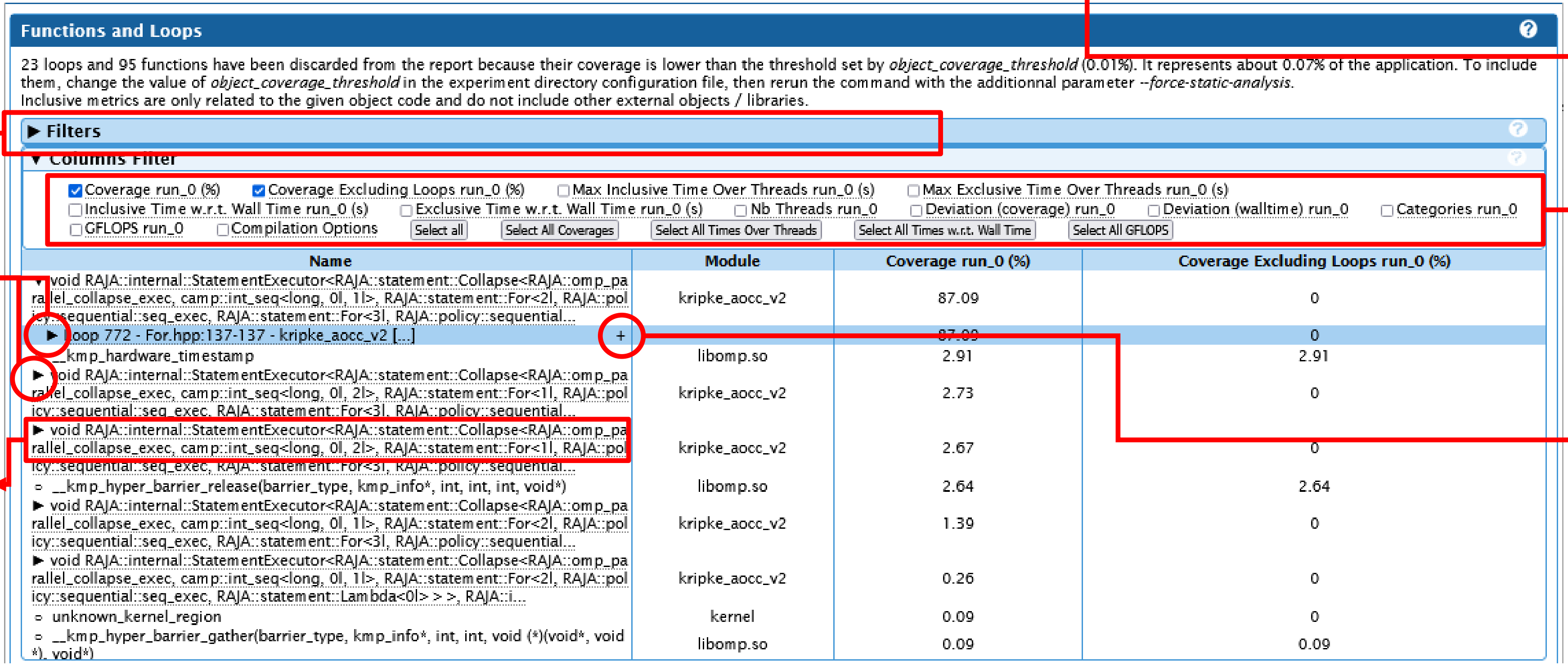


Functions Tab

Click the triangle to show the loops contained in a function or a loop

Expand this tab to view options for restricting the displayed functions by the library containing them

Check the boxes to select which columns to display. The buttons allow to select/deselect groups of similar boxes at once



Right-click on a line to get a contextual menu allowing to display load balancing graph and callchains for this function or loop

A "+" sign appears when hovering over a line, which allows to expand the full loop nest contained in the function

Functions Tab: Functions and Loops Profiling

Identifying hotspots

- Exclusive coverage
- Load balancing across threads
- Loops nests by functions

▼ matmul_sub
- ○ Loop 230 - solve_subs.f:71-175 - bt-mz.C.16
- ○ Loop 231 - solve_subs.f:71-175 - bt-mz.C.16 — Single

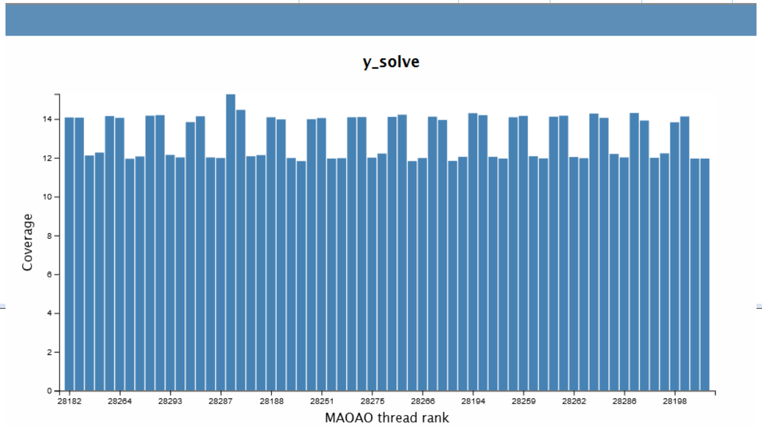
▼ z_solve
- ▼ Loop 232 - z_solve.f:53-423 - bt-mz.C.16 — Outermost
  - ▼ Loop 233 - z_solve.f:54-423 - bt-mz.C.16 — Inbetween
    - ▼ Loop 236 - z_solve.f:54-423 - bt-mz.C.16 — Inbetween
      - ○ Loop 239 - z_solve.f:146-308 - bt-mz.C.16 — Innermost
      - ○ Loop 235 - z_solve.f:55-137 - bt-mz.C.16
      - ○ Loop 234 - z_solve.f:415-423 - bt-mz.C.16

MAQAO | Global | Application | Functions | Loops | Topology

Functions and Loops

▶ Filters

Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)
○ gomp_team_barrier_wait_end	libgomp.so.1.0.0	21.34	3.26	64	4.47
○ binvcrhs	bt-mz.C.16	16.06	2.45	64	1.10
▶ z_solve._omp_fn.0	bt-mz.C.16	9.84	1.5	64	0.52
○ matmul_sub	bt-mz.C.16	9.52	1.45	64	0.68
▼ y_solve._omp_fn.0	bt-mz.C.16	9.09	1.39	64	0.68
▼ Loop 114 - y_solve.f:4-398 - bt-mz.C.16		8.82	1.35		
▼ Loop 115 - y_solve.f:4-398 - bt-mz.C.16		8.82	1.35		
○ Loop 118 - y_solve.f:145-308 - bt-mz.C.16		5.85	0.89		
○ Loop 119 - y_solve.f:55-137 - bt-mz.C.16		1.77	0.27		
○ Loop 116 - y_solve.f:394-398 - bt-mz.C.16		1.08	0.17		
○ Loop 117 - y_solve.f:337-360 - bt-mz.C.16		0.12	0.02		
▶ x_solve._omp_fn.0	bt-mz.C.16	8.68	1.32	64	0.64
○ gomp_barrier_wait_end	libgomp.so.1.0.0	8.26	1.26	64	4.91
▶ compute_rhs._omp_fn.0	bt-mz.C.16	7.57	1.16	64	0.46
○ mca_btl_vader_component_progress	mca_btl_vader.so	3.62	0.55	16	1.76
○ matvec_sub	bt-mz.C.16	2.73	0.42	64	0.20
○ lhsinit	bt-mz.C.16	0.54	0.08	64	0.06
○ opal_progress					0.18
▶ copy_x_face._omp_fn.2					0.06
▶ add._omp_fn.0					0.05
○ binvrhs					0.04
○ ompi_coll_libnbc_progress					0.13
▶ copy_y_face._omp_fn.0					0.05
○ opal_timer_linux_get_cycles_sys_timer					0.09
○ exact_solution					0.03
▶ copy_x_face._omp_fn.3					0.02
▶ copy_y_face._omp_fn.1					0.02
○ gomp_team_barrier_wait_final					0.02
○ exact_rhs._omp_fn.0					0.02
○ opal_progress@plt					0.03
▶ initialize._omp_fn.0					0.01
○ ompi_request_default_wait_all					0.03
○ gomp_thread_start					0.01
○ Unknown kernel region					0.01



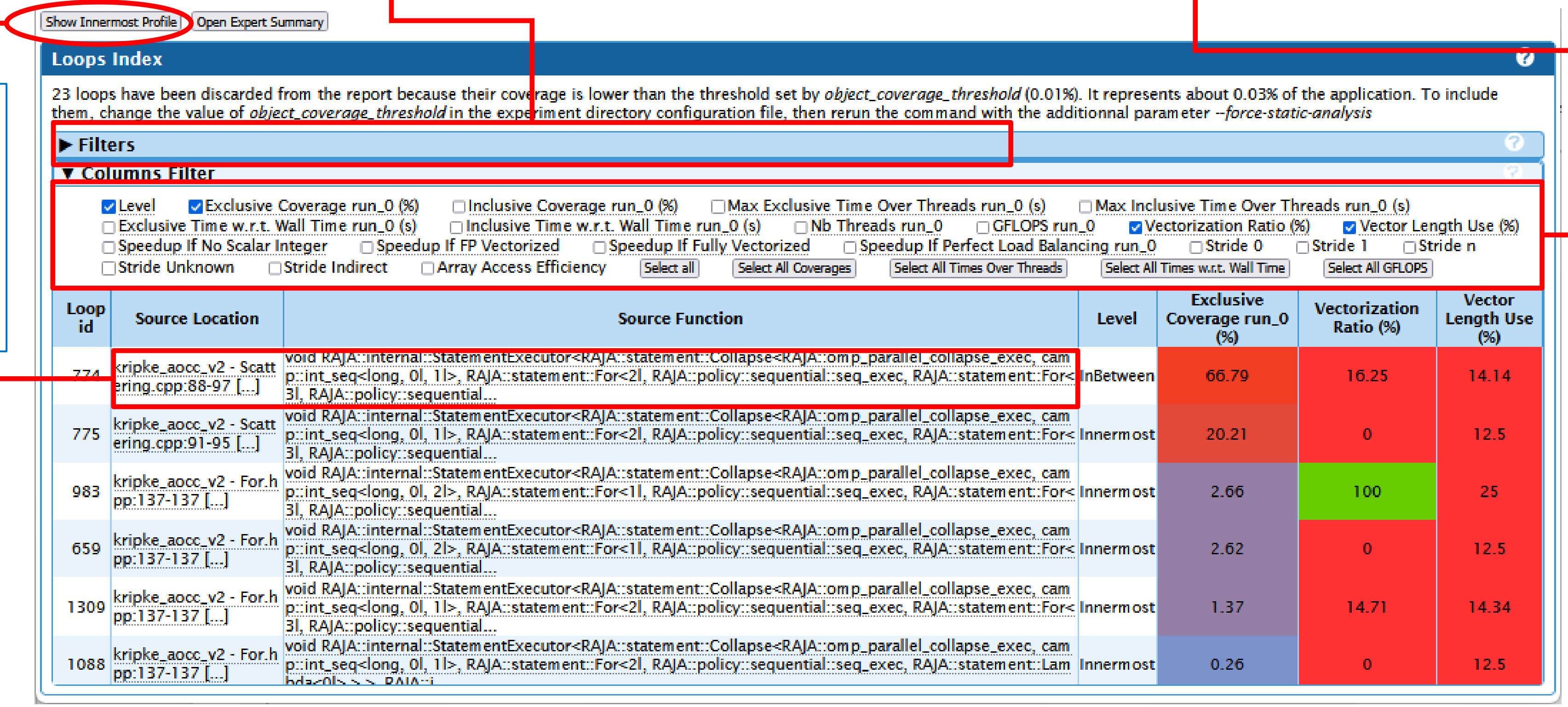
Loops Tab

Click this button to restrict the display to innermost/single loops

Expand this tab to view options for restricting the displayed loops by the library containing them

Check the boxes to select which columns to display. The buttons allow to select/deselect groups of similar boxes at once

Hover over a loop Source Location (resp. Source Function) to display in a tooltip the full list of its sources (resp. full name of the function containing it)



Loops Tab: Profiling Summary

Identifying loop hotspots

- Vectorisation information
- Potential speedup by optimisation
  - No scalar integer: Removing address computations
  - FP Vectorised: Vectorising floating-point computations
  - Fully Vectorised: Vectorising floating-point computations and memory accesses
  - Perfect Load Balancing: Optimal balance across all threads

MAQAO Global Application Functions Loops Topology

Show Full Profile | Open Expert Summary

Loops Index

73 loops have been discarded from the report because their coverage is lower than the threshold set by *object_coverage_threshold* (0.01%). It represents about 0% of the application. To include them, change the value of *object_coverage_threshold* in the experiment directory configuration file, then rerun the command with the additionnal parameter *--force-static-analysis*

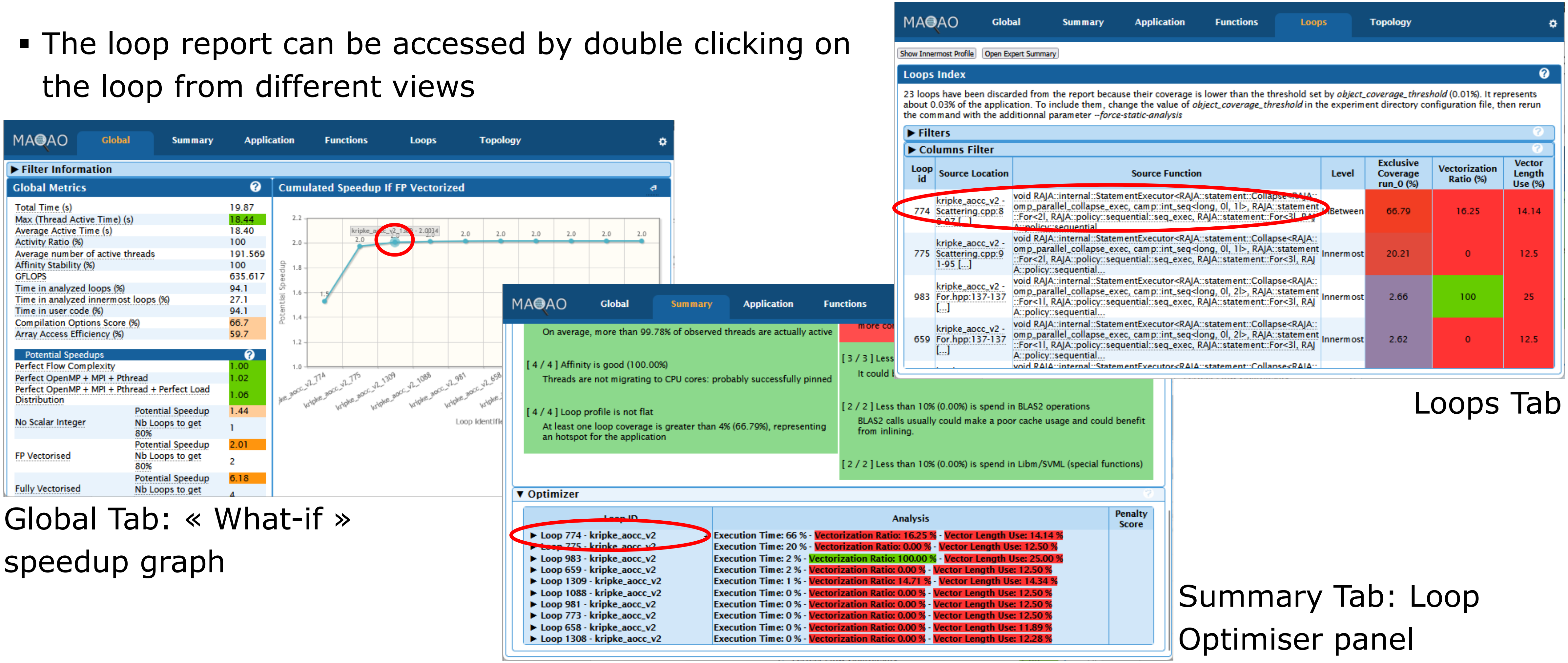
▶ **Filters**

☑Coverage (%) ☑Level ☑Time (s) ☑Vectorization Ratio (%) ☑Speedup If No Scalar Integer ☑Speedup If FP Vectorized ☑Speedup If Fully Vectorized ☑Speedup If Perfect Load Balancing | Select none

Loop id	Source Location	Source Function	Coverage (%)	Level	Time (s)	Vectorization Ratio (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing
179	bt-mz_C.8 - x_solve.f:146-309	x_solve_._omp_fn.0	7.67	Innermost	1.29	5.02	1.04	1	2.06	1.22
207	bt-mz_C.8 - z_solve.f:146-309	z_solve_._omp_fn.0	7.67	Innermost	1.29	5.31	1.02	1	2.06	1.15
185	bt-mz_C.8 - y_solve.f:145-308	y_solve_._omp_fn.0	7.35	Innermost	1.24	5.17	1.03	1	2.06	1.22
208	bt-mz_C.8 - z_solve.f:55-137	z_solve_._omp_fn.0	3.48	Innermost	0.59	7.09	1	1.13	2.26	1.17
180	bt-mz_C.8 - x_solve.f:57-139	x_solve_._omp_fn.0	3.09	Innermost	0.52	7.04	1	1.11	2.23	1.25
186	bt-mz_C.8 - y_solve.f:55-137	y_solve_._omp_fn.0	3.06	Innermost	0.52	7.09	1	1.11	2.23	1.21
156	bt-mz_C.8 - rhs.f:40-50	compute_rhs_._omp_fn.0	2.41	Innermost	0.41	0	1	2	2	1.15
133	bt-mz_C.8 - rhs.f:4-349	compute_rhs_._omp_fn.0	1.84	Innermost	0.31	0	1	1.65	3.41	1.29
150	bt-mz_C.8 - rhs.f:4-132	compute_rhs_._omp_fn.0	1.77	Innermost	0.3	0	1	1.71	3.68	1.27
142	bt-mz_C.8 - rhs.f:4-238	compute_rhs_._omp_fn.0	1.76	Innermost	0.3	0	1	1.65	3.41	1.27
204	bt-mz_C.8 - z_solve.f:415-421	z_solve_._omp_fn.0	1.7	Innermost	0.29	0	1	1	2.83	1.17

Loop Report

- The loop report can be accessed by double clicking on the loop from different views



Global Tab: « What-if » speedup graph

Summary Tab: Loop Optimiser panel

Loops Tab

Navigating a Loop Report

Source code of the loop. If Assembly Code is selected is the other panel, selecting a line will highlight the corresponding assembly lines

Assembly code of the loop. If Source Code is selected in the other panel, selecting a line will highlight the corresponding source lines

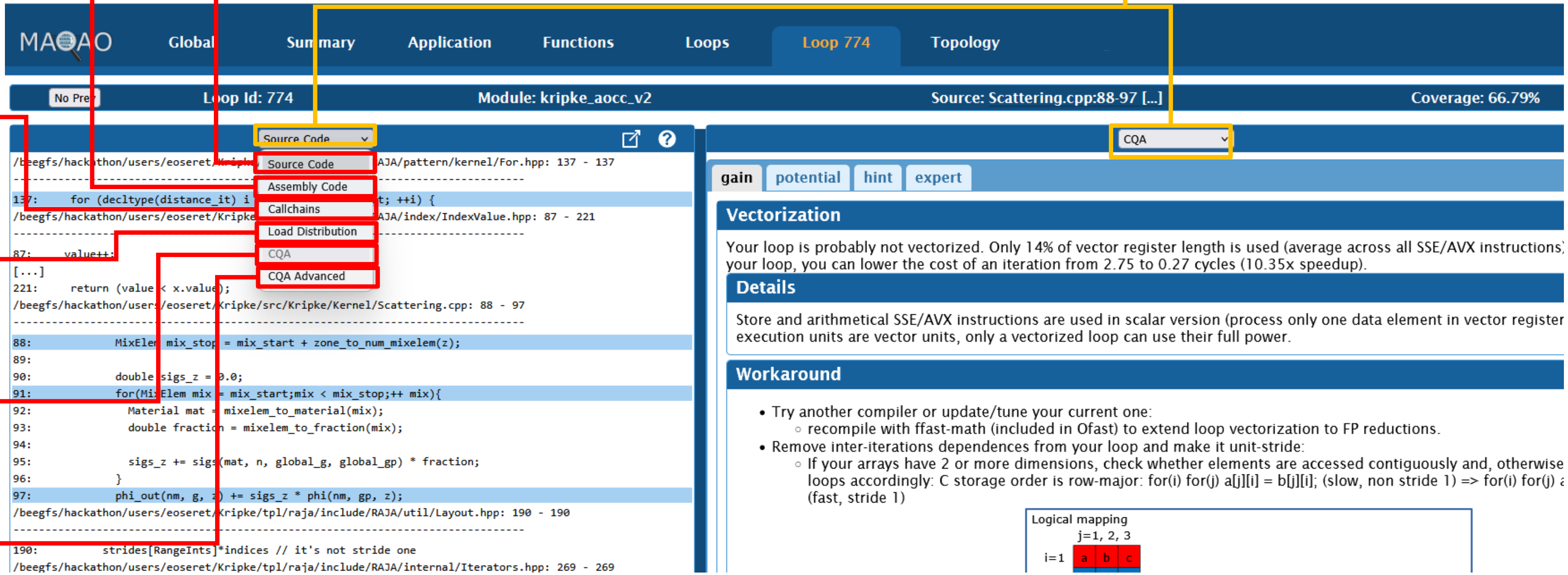
Use the scrolling lists on both panels to choose the content of the panel

Call stacks leading to this loop, with associated % of occurrence

Percentage of time spent in this loop across the threads

High-level static analysis report with hints for improving performance

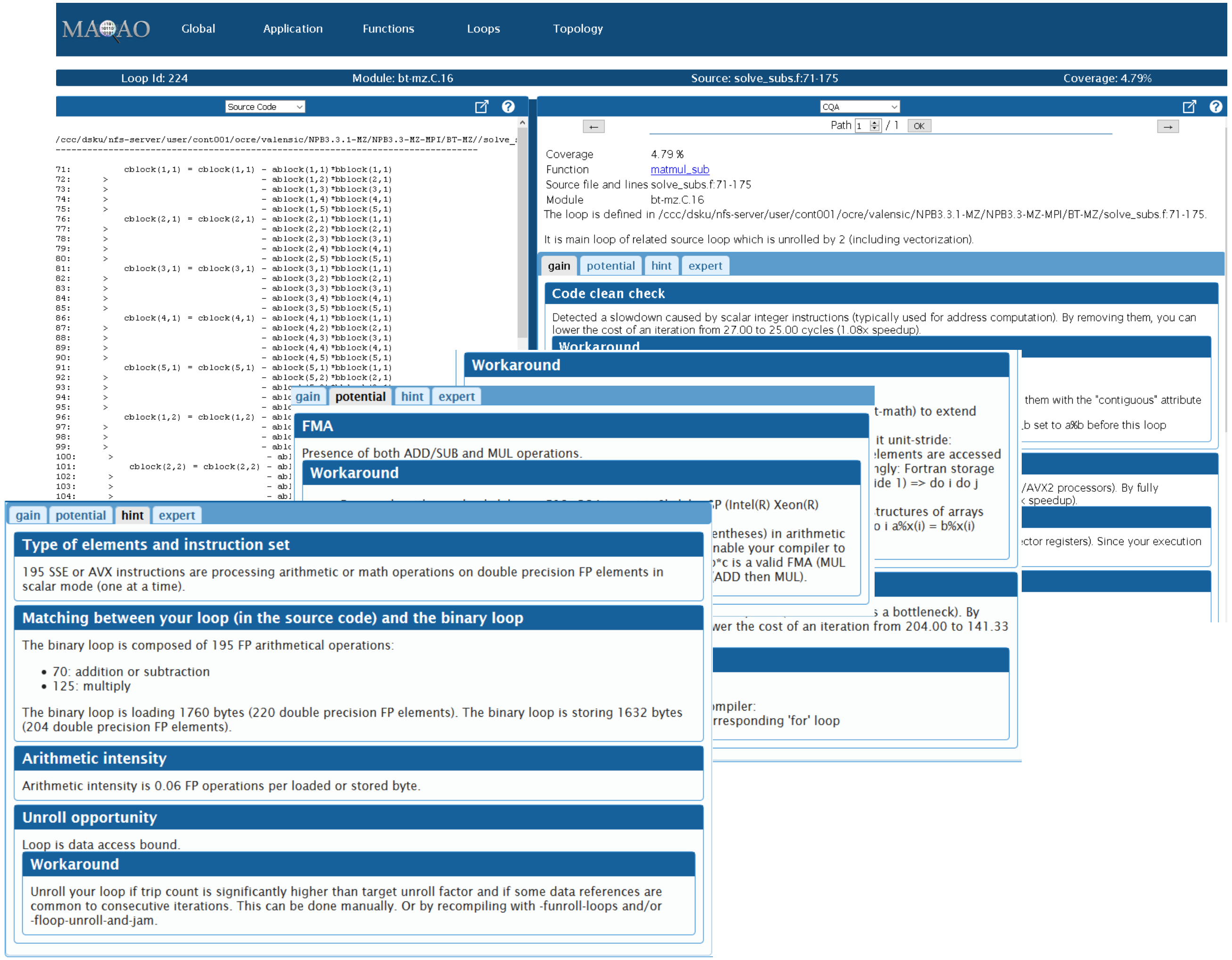
Detailed metrics generated by the static analysis



Loop Report: Static Analysis

High level reports

- Reference to the source code
- Bottleneck description
- Hints for improving performance
- Reports categorized by probability that applying hints will yield predicted gain
  - Gain: Good probability
  - Potential gain: Average probability
  - Hints: Lower probability



Loop Report – Expert View

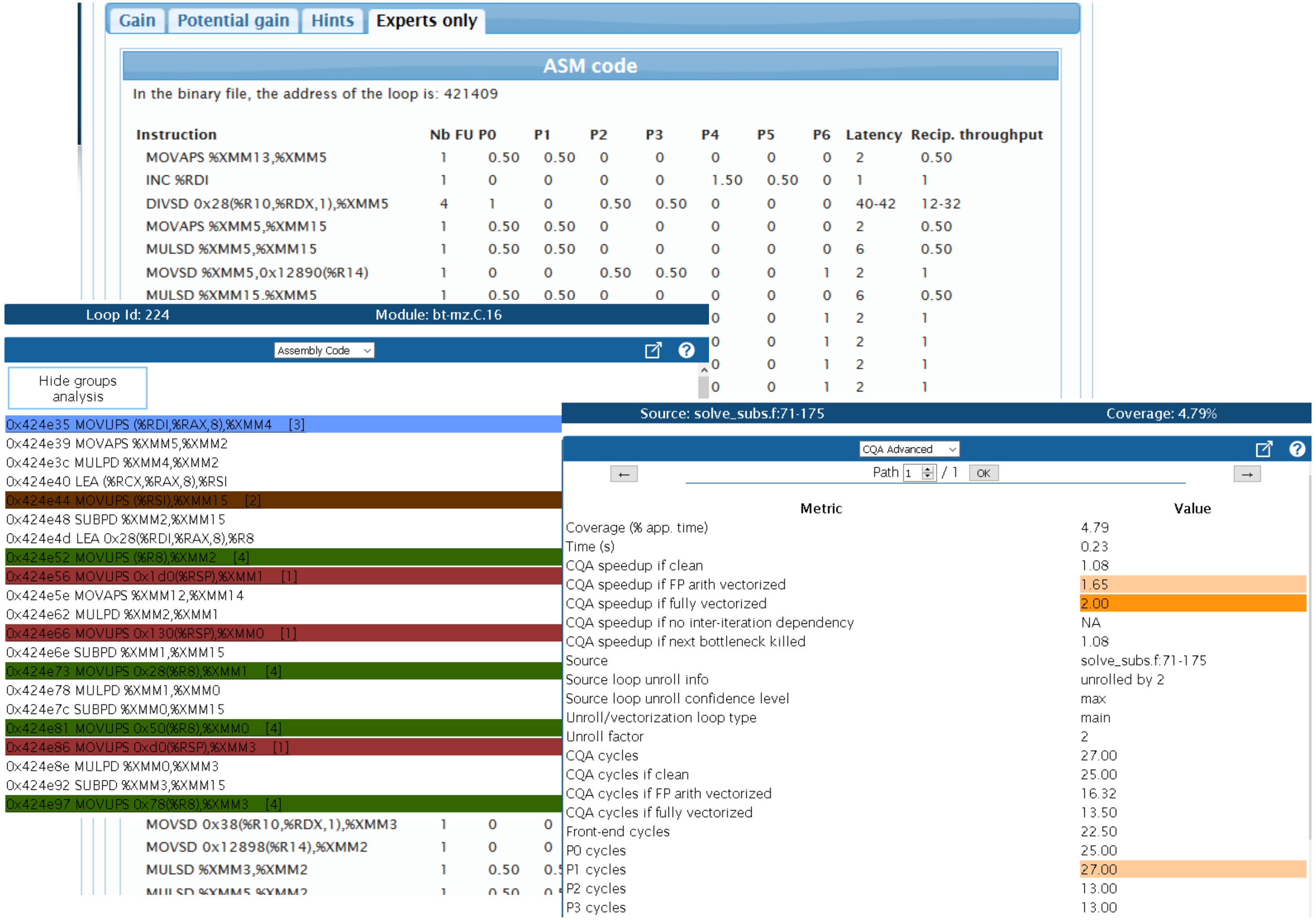
Low level reports for performance experts

- Assembly-level
- Instructions cycles costs
- Instructions dispatch predictions
- Memory access analysis

Assembly code

- Highlights groups of instructions accessing the same memory addresses

CQA low level metrics



Topology Tab

Click the triangle to show the processes (resp. threads) depending on a node (resp. process)

Double click on a thread to display the Functions Profile (similar to the Functions tab) restricted to that thread

A "+" sign appears when hovering over a line, which allows to expand the item and all its children

Software Topology

Number processes: 1 **Number nodes:** 1 **Number processes per node:** 1
Run Command: <executable> --groups 1024 --zones 24,16,16 --procs 2,2,2 **MPI Command:** mpirun -n 8 **Run Directory:** .
OMP_NUM_THREADS: 24 **OMP_PROC_BIND:** spread **I_MPI_PIN_DOMAIN:** auto **OMP_PLACES:** threads

ID	Observed Processes	Observed Threads	Time(s)	Elapsed Time (s)	Active Time (%)	Start (after process) (s)	End (before process) (s)	Maximum Time on the Same CPU (s)
▼ Node gmz12.benchmarkcenter.megware.com	8	192	18.44					
▶ MPI # 0 (PID 19442) +		24	18.44					
▼ MPI # 1 (PID 19458)		24	18.41					
○ MPI # 1 - OMP # 1.0 (TID 19458)			18.33	19.83	92.42	0.00	0.00	18.62
○ MPI # 1 - OMP # 1.1 (TID 19512)			18.41	18.67	98.58	1.15	0.01	18.63
○ MPI # 1 - OMP # 1.2 (TID 19515)			18.40	18.67	98.56	1.15	0.01	18.62
○ MPI # 1 - OMP # 1.3 (TID 19523)			18.41	18.68	98.57	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.4 (TID 19536)			18.40	18.68	98.52	1.15	0.00	18.62
○ MPI # 1 - OMP # 1.5 (TID 19547)			18.41	18.68	98.61	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.6 (TID 19565)			18.41	18.67	98.58	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.7 (TID 19580)			18.41	18.67	98.58	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.8 (TID 19595)			18.40	18.67	98.53	1.15	0.00	18.63
○ MPI # 1 - OMP # 1.9 (TID 19679)			18.40	18.66	98.61	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.10 (TID 19695)			18.40	18.66	98.64	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.11 (TID 19709)			18.41	18.66	98.67	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.12 (TID 19725)			18.40	18.66	98.64	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.13 (TID 19741)			18.41	18.66	98.70	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.14 (TID 19757)			18.40	18.66	98.64	1.17	0.00	18.63
○ MPI # 1 - OMP # 1.15 (TID 19769)			18.41	18.66	98.67	1.17	0.01	18.63
○ MPI # 1 - OMP # 1.16 (TID 19776)			18.40	18.66	98.65	1.17	0.01	18.63

MAQAO ONE View Thread/Process View

Software Topology
- List of nodes
- Processes by node
- Thread by process

View by thread
- Function profile at the thread or process level

MAQAO | Global | Application | Functions | Loops | Topology

Software Topology

ID	Processes	Threads	Time(s)
▼ Node c251-109.wrangler.tacc.utexas.edu	8	32	5.34
▼ Process 145897		4	5.34
○ Thread 145897			5.34
○ Thread 145933			5.32
○ Thread 145952			5.32
○ Thread 145969			5.3
▶ Process 145899		4	5.34
▶ Process 145901		4	5.34
▶ Process 145903		4	5.34
▶ Process 145898		4	5.34
▶ Process 145900		4	5.34
▶ Process 145895		4	5.34
▶ Process 145896		4	5.34
▶ Node c251-110.wrangler.tacc.utexas.edu	8	32	5.36
○ AVERAGE			5.36

MAQAO | Global

Profiling node c251-109.wrangler.tacc.utexas.edu - process 145897 - thread 145897

Name	Module	Coverage (%)	Time (s)
○ binvcrhs	bt-mz_B.16	24.34	1.3
○ _INTERNAL_25_______src_kmp_barrier_cpp_fa608613::__kmp_hyper_barrier_gather(barrier_type, kmp_info*, int, int, void (*)(void*, void*), void*)	libiomp5.so	17.6	0.94
▶ matmul_sub	bt-mz_B.16	12.73	0.68
▶ y_solve	bt-mz_B.16	7.87	0.42
▶ compute_rhs	bt-mz_B.16	7.49	0.4
▶ x_solve	bt-mz_B.16	7.12	0.38
▶ z_solve	bt-mz_B.16	6.74	0.36

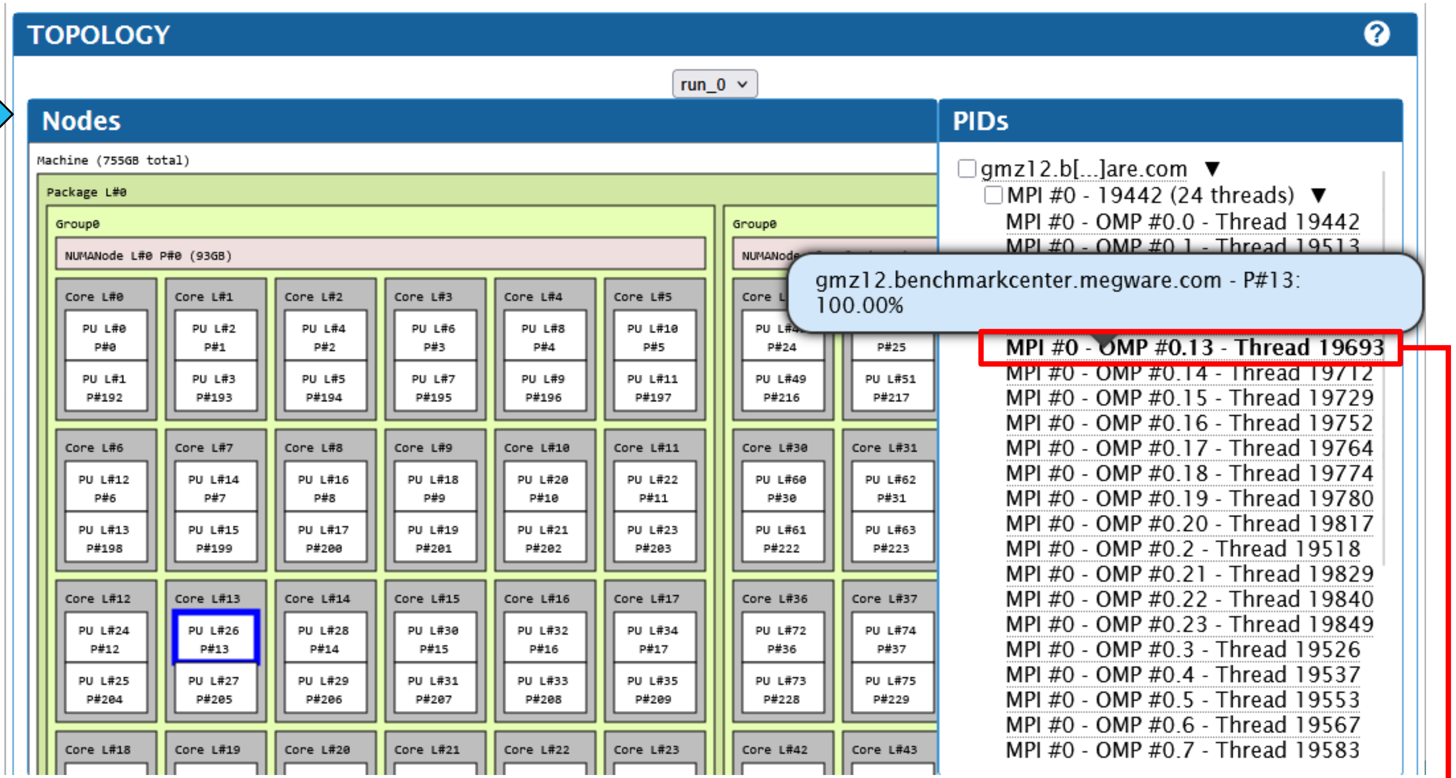
Topology Tab: lstopo View

- Hover above the « Topology » tab name to see the scrolling list allowing to select additional views to display

Displays a graphical representation of the threads affinities and the cores on which they ran

Displays for each core a weighted list of threads that executed on this core

Displays for each thread a weighted list of the cores on which the thread executed

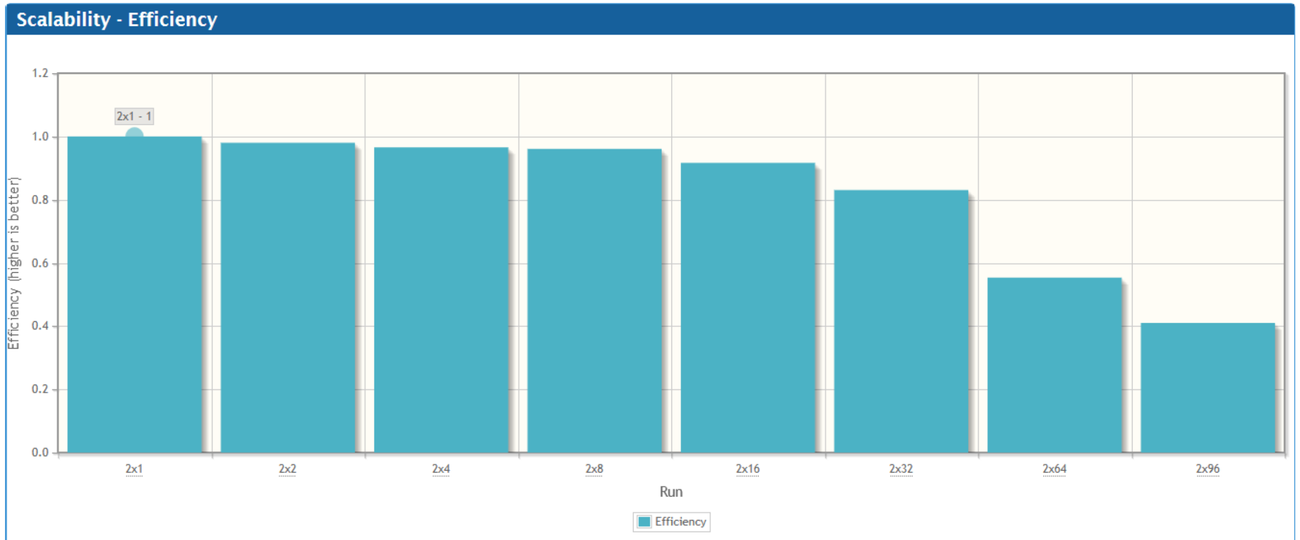


Hover over a thread to highlight the core(s) on which it ran

MAQAO ONE View Scalability Reports

Goal: Provide a view of the application scalability

- Profiles with different numbers of threads/processes
- Displays efficiency metrics for application



▼ Detailed Efficiency

Run	Number of observed threads	Efficiency (ideal is 1)
2x1	2	1
2x2	4	0.98
2x4	8	0.97
2x8	16	0.96
2x16	32	0.92
2x32	64	0.83
2x64	128	0.55
2x96	192	0.41

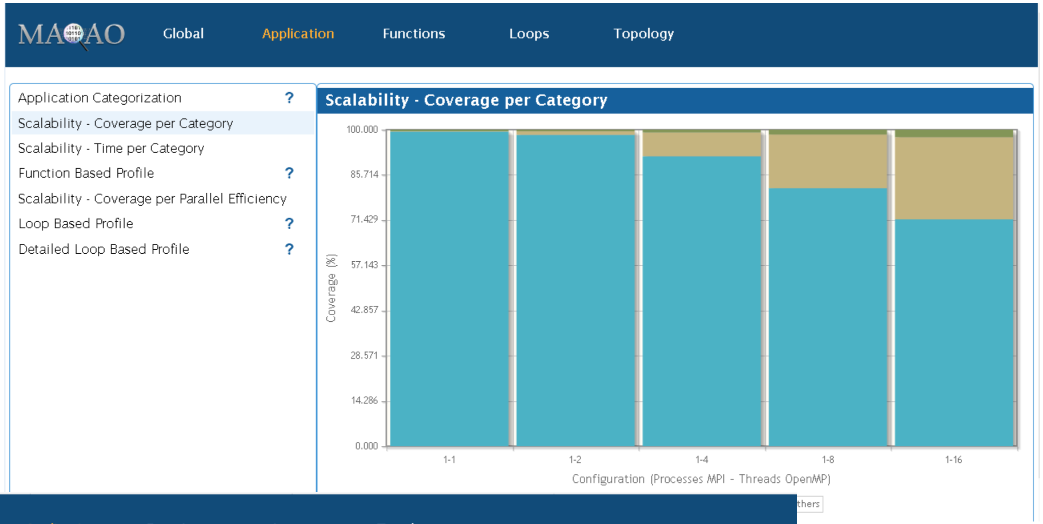
MAQAO ONE View Scalability Reports – Application View

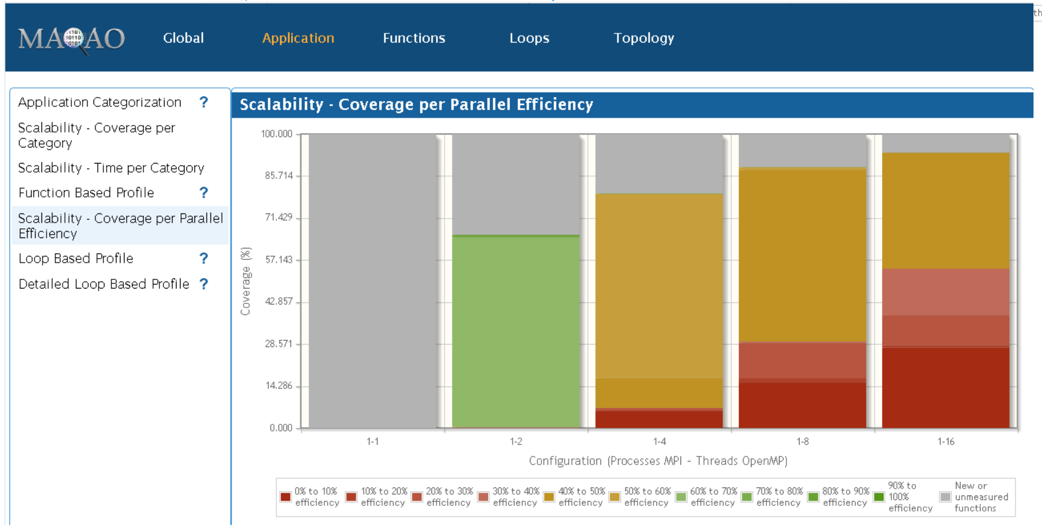
Coverage per category

- Comparison of categories for each run

Coverage per parallel efficiency

- $Efficiency = \frac{T_{sequential}}{T_{parallel} * N_{threads}}$
  - Distinguishing functions only represented in parallel or sequential
- Displays efficiency by coverage

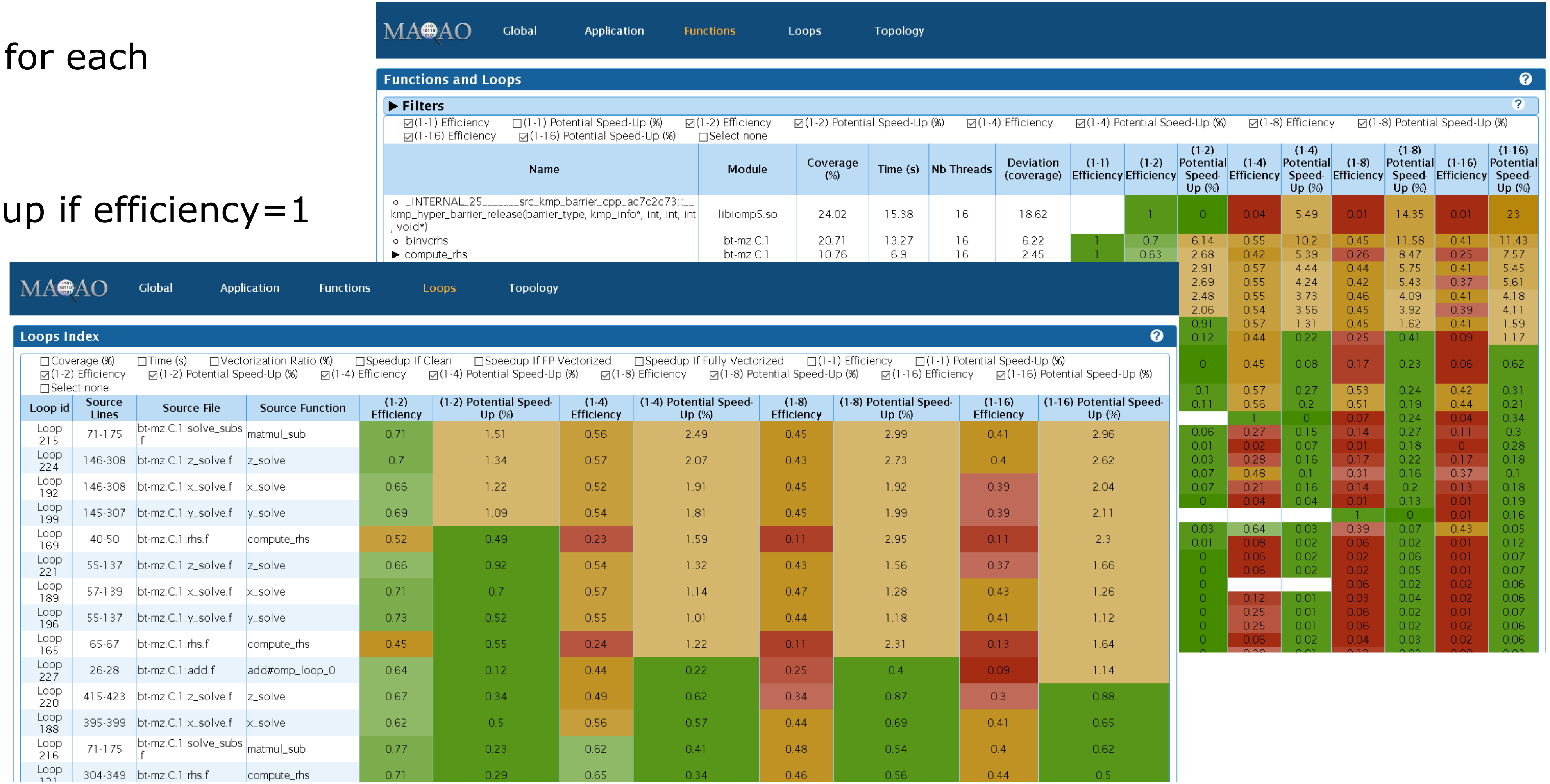




MAQAO ONE View Scalability Reports – Functions and Loops Views

Displays metrics for each function/loop

- Efficiency
- Potential speedup if efficiency=1



Name	Module	Coverage (%)	Time (s)	Nb Threads	Deviation (coverage)	(1-1) Efficiency	(1-2) Efficiency	(1-2) Potential Speed-Up (%)	(1-4) Efficiency	(1-4) Potential Speed-Up (%)	(1-8) Efficiency	(1-8) Potential Speed-Up (%)	(1-16) Efficiency	(1-16) Potential Speed-Up (%)
o _INTERNAL_25______src_kmp_barrier_cpp_ac7c2c73::__kmp_hyper_barrier_release(barrier_type, kmp_info*, int, int, int, void*)	libiomp5.so	24.02	15.38	16	18.62		1	0	0.04	5.49	0.01	14.35	0.01	23
o binvcrhs	bt-mz.C.1	20.71	13.27	16	6.22	1	0.7	6.14	0.55	10.2	0.45	11.58	0.41	11.43
▶ compute_rhs	bt-mz.C.1	10.76	6.9	16	2.45	1	0.63	2.68	0.42	5.39	0.26	8.47	0.25	7.57
								2.91	0.57	4.44	0.44	5.75	0.41	5.45
								2.69	0.55	4.24	0.42	5.43	0.37	5.61
								2.48	0.55	3.73	0.46	4.09	0.41	4.18
								2.06	0.54	3.56	0.45	3.92	0.39	4.11
								0.91	0.57	1.31	0.45	1.62	0.41	1.59
								0.12	0.44	0.22	0.25	0.41	0.09	1.17
								0	0.45	0.08	0.17	0.23	0.06	0.62
								0.1	0.57	0.27	0.53	0.24	0.42	0.31
								0.11	0.56	0.2	0.51	0.19	0.44	0.21
									1	0	0.07	0.24	0.04	0.34
								0.06	0.27	0.15	0.14	0.27	0.11	0.3
								0.01	0.02	0.07	0.01	0.18	0	0.28
								0.03	0.28	0.16	0.17	0.22	0.17	0.18
								0.07	0.48	0.1	0.31	0.16	0.37	0.1
								0.07	0.21	0.16	0.14	0.2	0.13	0.18
								0	0.04	0.04	0.01	0.13	0.01	0.19
											1	0	0.01	0.16
								0.03	0.64	0.03	0.39	0.07	0.43	0.05
								0.01	0.08	0.02	0.06	0.02	0.01	0.12
								0	0.06	0.02	0.02	0.06	0.01	0.07
								0	0.06	0.02	0.02	0.05	0.01	0.07
								0			0.06	0.02	0.02	0.06
								0	0.12	0.01	0.03	0.04	0.02	0.06
								0	0.25	0.01	0.06	0.02	0.01	0.07
								0	0.25	0.01	0.06	0.02	0.02	0.06
								0	0.06	0.02	0.04	0.03	0.02	0.06

Loop id	Source Lines	Source File	Source Function	(1-2) Efficiency	(1-2) Potential Speed-Up (%)	(1-4) Efficiency	(1-4) Potential Speed-Up (%)	(1-8) Efficiency	(1-8) Potential Speed-Up (%)	(1-16) Efficiency	(1-16) Potential Speed-Up (%)
Loop 215	71-175	bt-mz.C.1:solve_subs.f	matmul_sub	0.71	1.51	0.56	2.49	0.45	2.99	0.41	2.96
Loop 224	146-308	bt-mz.C.1:z_solve.f	z_solve	0.7	1.34	0.57	2.07	0.43	2.73	0.4	2.62
Loop 192	146-308	bt-mz.C.1:x_solve.f	x_solve	0.66	1.22	0.52	1.91	0.45	1.92	0.39	2.04
Loop 199	145-307	bt-mz.C.1:y_solve.f	y_solve	0.69	1.09	0.54	1.81	0.45	1.99	0.39	2.11
Loop 169	40-50	bt-mz.C.1:rhs.f	compute_rhs	0.52	0.49	0.23	1.59	0.11	2.95	0.11	2.3
Loop 221	55-137	bt-mz.C.1:z_solve.f	z_solve	0.66	0.92	0.54	1.32	0.43	1.56	0.37	1.66
Loop 189	57-139	bt-mz.C.1:x_solve.f	x_solve	0.71	0.7	0.57	1.14	0.47	1.28	0.43	1.26
Loop 196	55-137	bt-mz.C.1:y_solve.f	y_solve	0.73	0.52	0.55	1.01	0.44	1.18	0.41	1.12
Loop 165	65-67	bt-mz.C.1:rhs.f	compute_rhs	0.45	0.55	0.24	1.22	0.11	2.31	0.13	1.64
Loop 227	26-28	bt-mz.C.1:add.f	add#omp_loop_0	0.64	0.12	0.44	0.22	0.25	0.4	0.09	1.14
Loop 220	415-423	bt-mz.C.1:z_solve.f	z_solve	0.67	0.34	0.49	0.62	0.34	0.87	0.3	0.88
Loop 188	395-399	bt-mz.C.1:x_solve.f	x_solve	0.62	0.5	0.56	0.57	0.44	0.69	0.41	0.65
Loop 216	71-175	bt-mz.C.1:solve_subs.f	matmul_sub	0.77	0.23	0.62	0.41	0.48	0.54	0.4	0.62
Loop 121	304-349	bt-mz.C.1:rhs.f	compute_rhs	0.71	0.29	0.65	0.34	0.46	0.56	0.44	0.5

Backup Slides

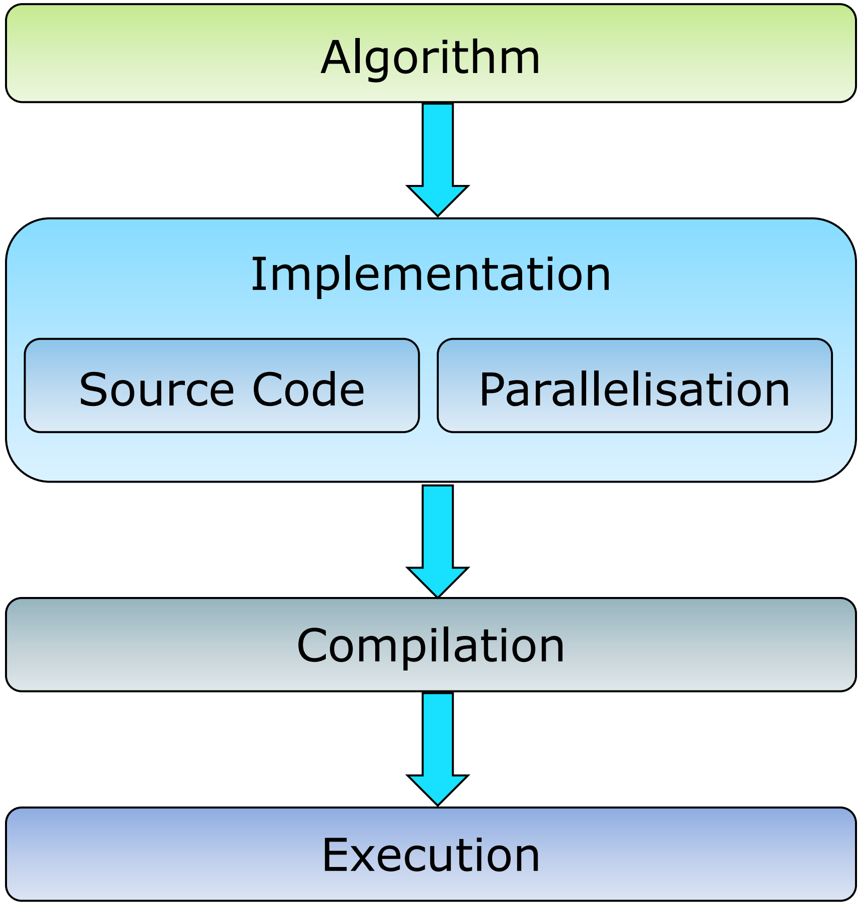
Performance analysis and optimisation

Where is the application spending most execution time and resources?

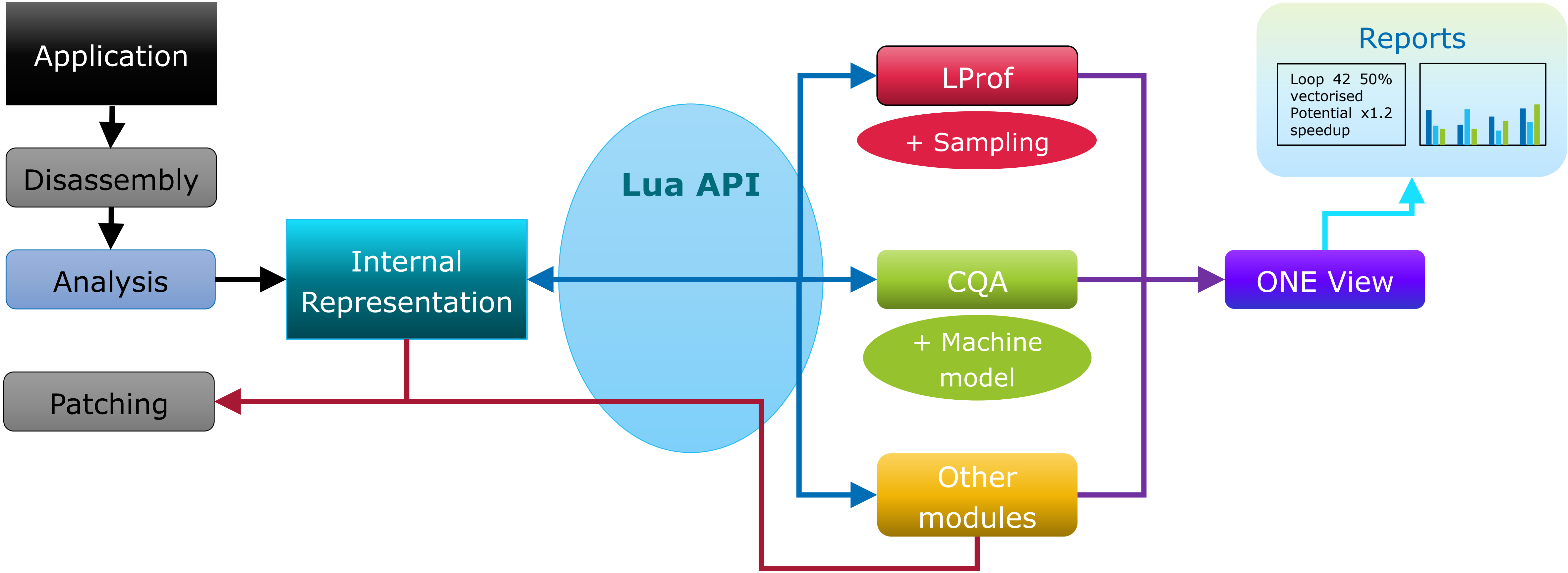
Why is the application spending time there?
- Algorithm, implementation, runtime or hardware?
- Data access or computation?

How to improve the application?
- At which step(s) of the workflow or dev process?
- What additional information is needed?

How much gain can be expected?
- At what cost?



MAQAO Main structure



MAQAO CQA: Main Concepts

Applications exploit at best 5 to 10% of the peak performance.

Main elements of analysis:

- Peak performance
- Execution pipeline
- Resources/Functional units

Same instruction – Same cost

Process up to 8X data

Key performance levers for core level efficiency:

- Vectorization
- Avoiding high latency instructions if possible (DIV/SQRT)
- Guiding the compiler code optimization
- Reorganizing memory and data structures layout

MAQAO CQA: Guiding the compiler and implementation hints

Compilers can be driven using flags, pragmas, and keywords:

- Ensuring full use of architecture capabilities (e.g. using flag -xHost on AVX capable machines)
- Forcing optimizations (unrolling, vectorization, alignment, …)
- Bypassing conservative behaviour when possible (e.g. 1/X precision)

Hints for implementation changes:

- Improve data access patterns
  - Memory alignment
  - Loop interchange
  - Changing loop strides
  - Reshaping arrays of structures
- Avoid instructions with high latency (SQRT, DIV, GATHER, SCATTER, …)

Analysing an application with MAQAO

MAQAO modules can be invoked separately for advanced analyses

- LProf
  - Profiling

```
$ maqao lprof xp=exp_dir --mpi-command="mpirun -n 16 -ppn 4" ppn=4 -- ./bt-mz.C.16
```

  - Display functions profile

```
$ maqao lprof xp=exp_dir -df
```

  - Displaying the results from a ONE View run

```
$ maqao lprof xp=oneview_xp_dir/tools/lprof_npsu -df
```

- CQA

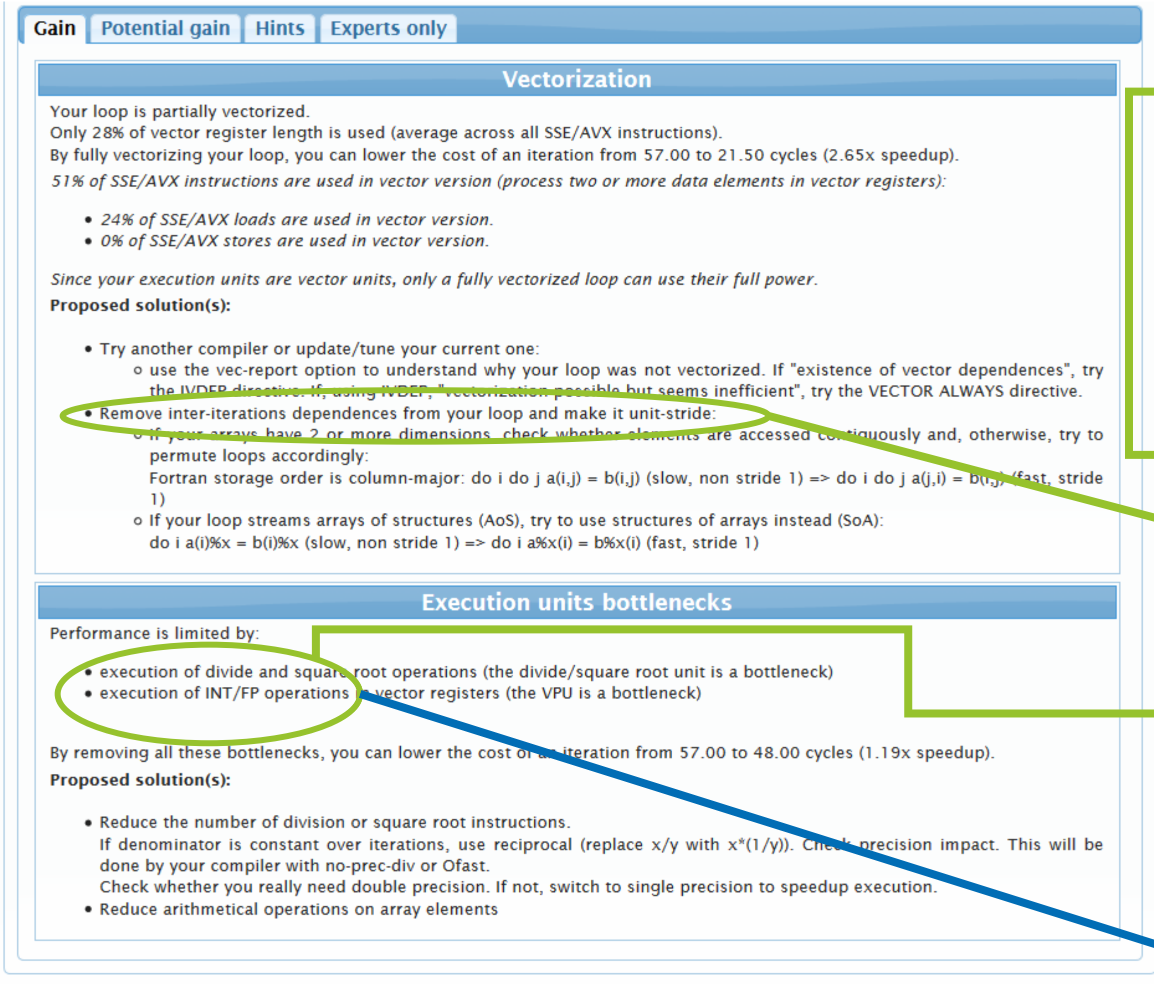
```
$ maqao cqa loop=42 bt-mz.C.16
```

Online help is available:

```
$ maqao lprof --help
```

```
$ maqao cqa --help
```

Application to Motivating Example



Gain | Potential gain | Hints | Experts only

Vectorization

Your loop is partially vectorized.
Only 28% of vector register length is used (average across all SSE/AVX instructions).
By fully vectorizing your loop, you can lower the cost of an iteration from 57.00 to 21.50 cycles (2.65x speedup).
51% of SSE/AVX instructions are used in vector version (process two or more data elements in vector registers):

- *24% of SSE/AVX loads are used in vector version.*
- *0% of SSE/AVX stores are used in vector version.*

Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:
    Fortran storage order is column-major: do i do j a(i,j) = b(i,j) (slow, non stride 1) => do i do j a(j,i) = b(i,j) (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
    do i a(i)%x = b(i)%x (slow, non stride 1) => do i a%x(i) = b%x(i) (fast, stride 1)

Execution units bottlenecks

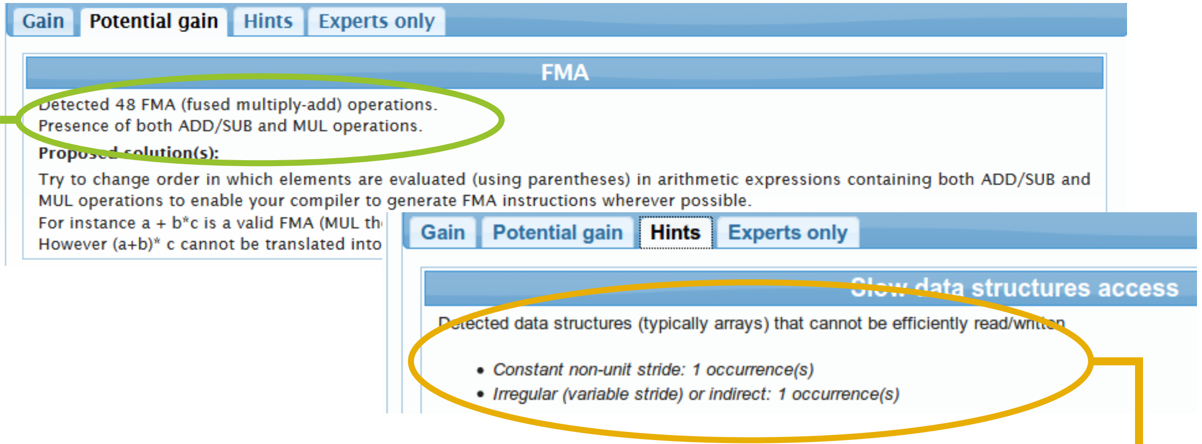
Performance is limited by:

- execution of divide and square root operations (the divide/square root unit is a bottleneck)
- execution of INT/FP operations on vector registers (the VPU is a bottleneck)

By removing all these bottlenecks, you can lower the cost of an iteration from 57.00 to 48.00 cycles (1.19x speedup).

Proposed solution(s):

- Reduce the number of division or square root instructions.
  If denominator is constant over iterations, use reciprocal (replace x/y with x*(1/y)). Check precision impact. This will be done by your compiler with no-prec-div or Ofast.
  Check whether you really need double precision. If not, switch to single precision to speedup execution.
- Reduce arithmetical operations on array elements



Gain | Potential gain | Hints | Experts only

FMA

Detected 48 FMA (fused multiply-add) operations.
Presence of both ADD/SUB and MUL operations.

Proposed solution(s):

Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible.
For instance a + b*c is a valid FMA (MUL th…
However (a+b)* c cannot be translated into…

Gain | Potential gain | Hints | Experts only

Slow data structures access

Detected data structures (typically arrays) that cannot be efficiently read/written

- *Constant non-unit stride: 1 occurrence(s)*
- *Irregular (variable stride) or indirect: 1 occurrence(s)*

1) High number of statements
2) Non-unit stride accesses
3) Indirect accesses
4) DIV/SQRT
5) Reductions
6) Variable number of iterations
7) Vector vs scalar