ASSESSING CPU CODE QUALITY: COMPILER COMPARISON

Presenter: William JALBY

UVSQ/UPSaclay: E. Oseret, K. Camus, C. Valensi, H. Bollore,

Machine time donation: MEGWARE, CALMIP, IT4I, AWS

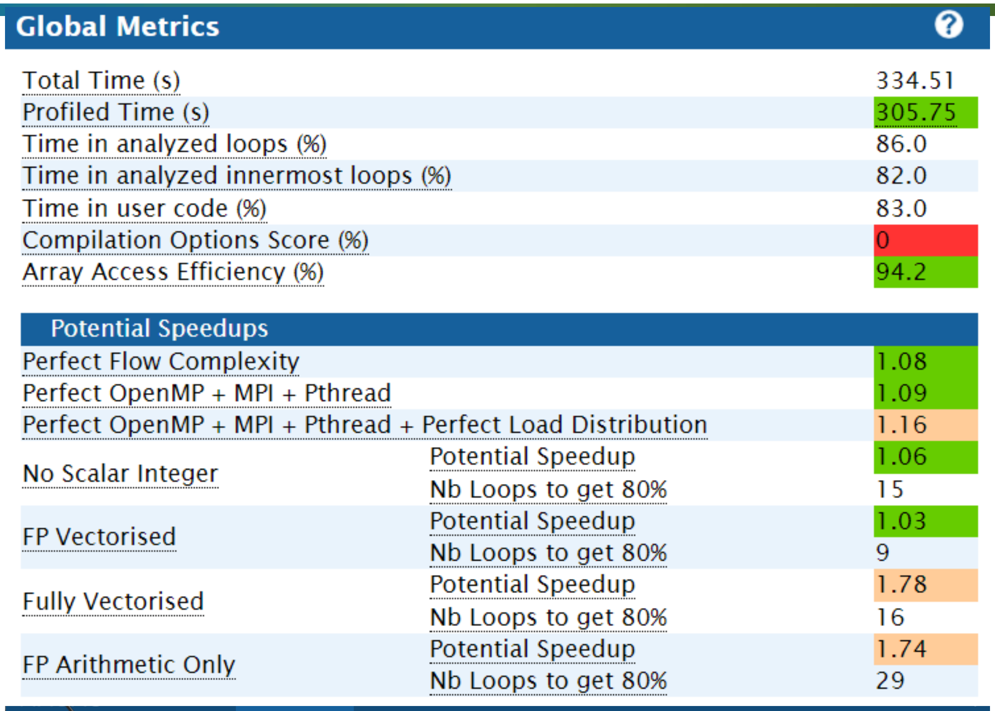
MAQAO: Modular Assembly Quality Analyzer and Optimizer

- Objectives:
  - Characterizing performance of HPC applications
  - **Guiding users** through optimization process
  - Estimating return of investment (**R.O.I.**)
- Characteristics:
  - Support for **Intel / AMD x86-64** and **AArch64** (beta version)
    - Work in progress on GPU Support: integrating other tools output or building on primitives (HSA)
  - **Modular tool** offering complementary views
  - LGPL3 Open Source software
  - Binary release available as **static executable**
- Operating principle: 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

➔ What You Analyse Is What You Run

MAQAO ONE View: Performance View Aggregator

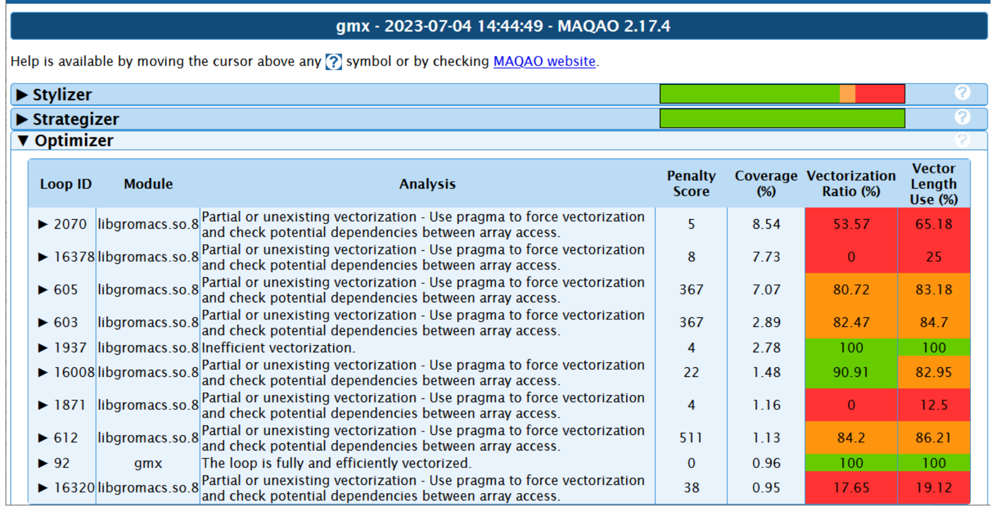
- High level summary
  - Checking analysis validity
  - Detecting performance issues
  - Assessing optimisation complexity
  - Estimating gain for addressing each performance issue
- Code quality metrics at global and loop level
- Hints for improving performance with associated expected speedup
- Detailed analyses results



Global Metrics

Total Time (s)	334.51
Profiled Time (s)	305.75
Time in analyzed loops (%)	86.0
Time in analyzed innermost loops (%)	82.0
Time in user code (%)	83.0
Compilation Options Score (%)	0
Array Access Efficiency (%)	94.2

Potential Speedups		
Perfect Flow Complexity		1.08
Perfect OpenMP + MPI + Pthread		1.09
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.16
No Scalar Integer	Potential Speedup	1.06
	Nb Loops to get 80%	15
FP Vectorised	Potential Speedup	1.03
	Nb Loops to get 80%	9
Fully Vectorised	Potential Speedup	1.78
	Nb Loops to get 80%	16
FP Arithmetic Only	Potential Speedup	1.74
	Nb Loops to get 80%	29



gmx - 2023-07-04 14:44:49 - MAQAO 2.17.4

Help is available by moving the cursor above any ? symbol or by checking MAQAO website.

▶ Stylizer
▶ Strategizer
▼ Optimizer

Loop ID	Module	Analysis	Penalty Score	Coverage (%)	Vectorization Ratio (%)	Vector Length Use (%)
▶ 2070	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	5	8.54	53.57	65.18
▶ 16378	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	8	7.73	0	25
▶ 605	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	367	7.07	80.72	83.18
▶ 603	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	367	2.89	82.47	84.7
▶ 1937	libgromacs.so.8	Inefficient vectorization.	4	2.78	100	100
▶ 16008	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	22	1.48	90.91	82.95
▶ 1871	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	4	1.16	0	12.5
▶ 612	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	511	1.13	84.2	86.21
▶ 92	gmx	The loop is fully and efficiently vectorized.	0	0.96	100	100
▶ 16320	libgromacs.so.8	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	38	0.95	17.65	19.12

OUTLINE

- Motivating example
- Identifying potential compiler issues
- A few examples

IMPORTANT: Input/comments are welcome. Project is still very flexible

MOTIVATING EXAMPLE CONTEXT

- TARGET CODE: HACC MK (Livermore: LLNL)
- Target hardware: AMD EPYC 9654 96-Core Processor (2 x 96 cores) provided by MEGWARE
- Two compilers:
  - AMD clang version 16.0.3 (CLANG: AOCC_4.1.0-Build#270 2023_07_10)
  - GNU C17 13.2.0 -march=znver4 ….
  - clang based Intel(R) oneAPI DPC++/C++ Compiler 2024.0.0 (2024.0.0.20231017)
- For each compiler 16 flags were tested.
- Linux 5.14.0-427.18.1.el9_4.x86_64 #1 SMP PREEMPT_DYNAMIC Tue May 28 06:27:02 EDT 2024
- Systematic test/benchmarking effort carried out in QaaS project (Quality as a Service)

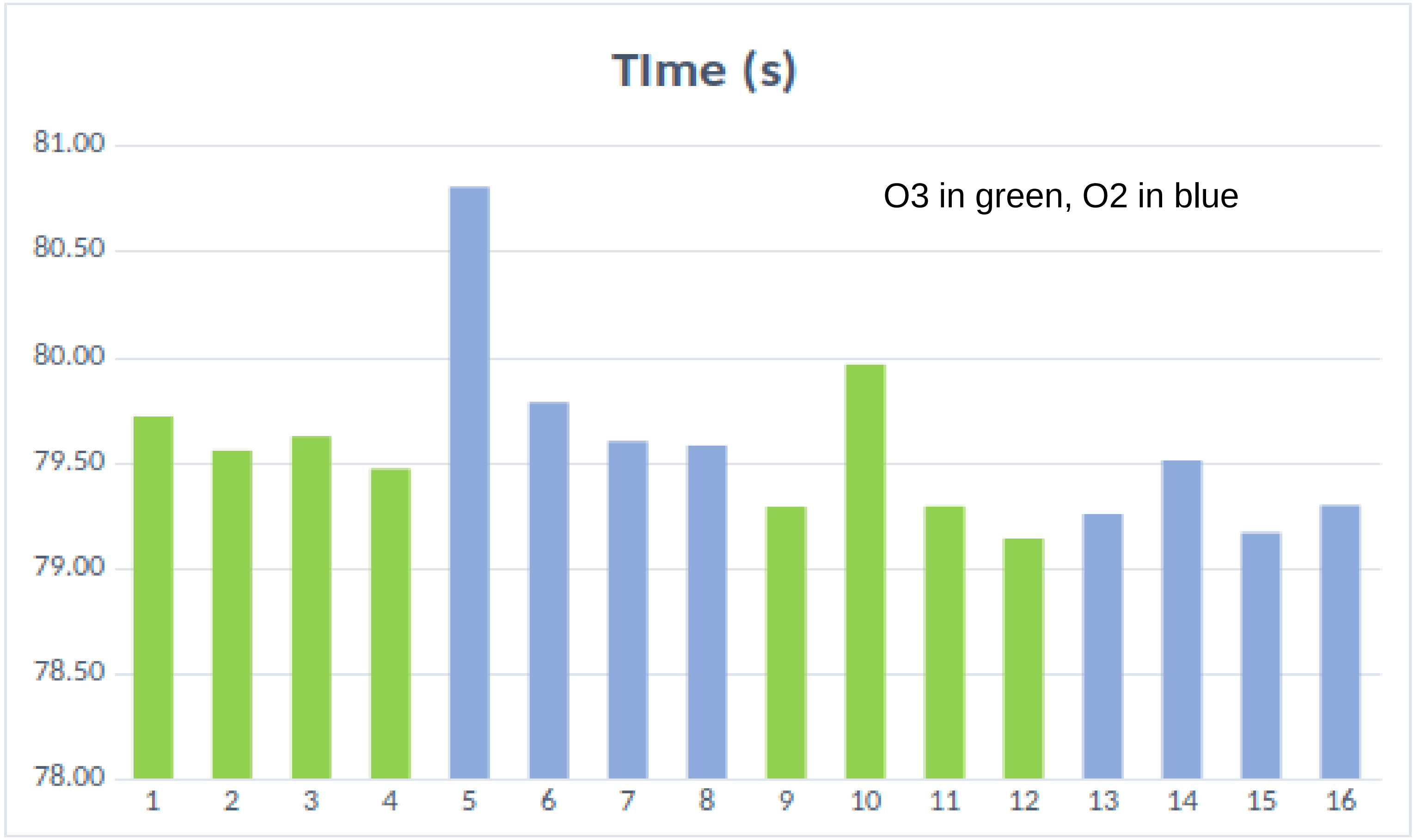
COMPILER OPTIONS FOR AOCC

16 “standard” compiler options with different optimization levels (O2 and O3), different vector lengths, ….
Orthogonal choice

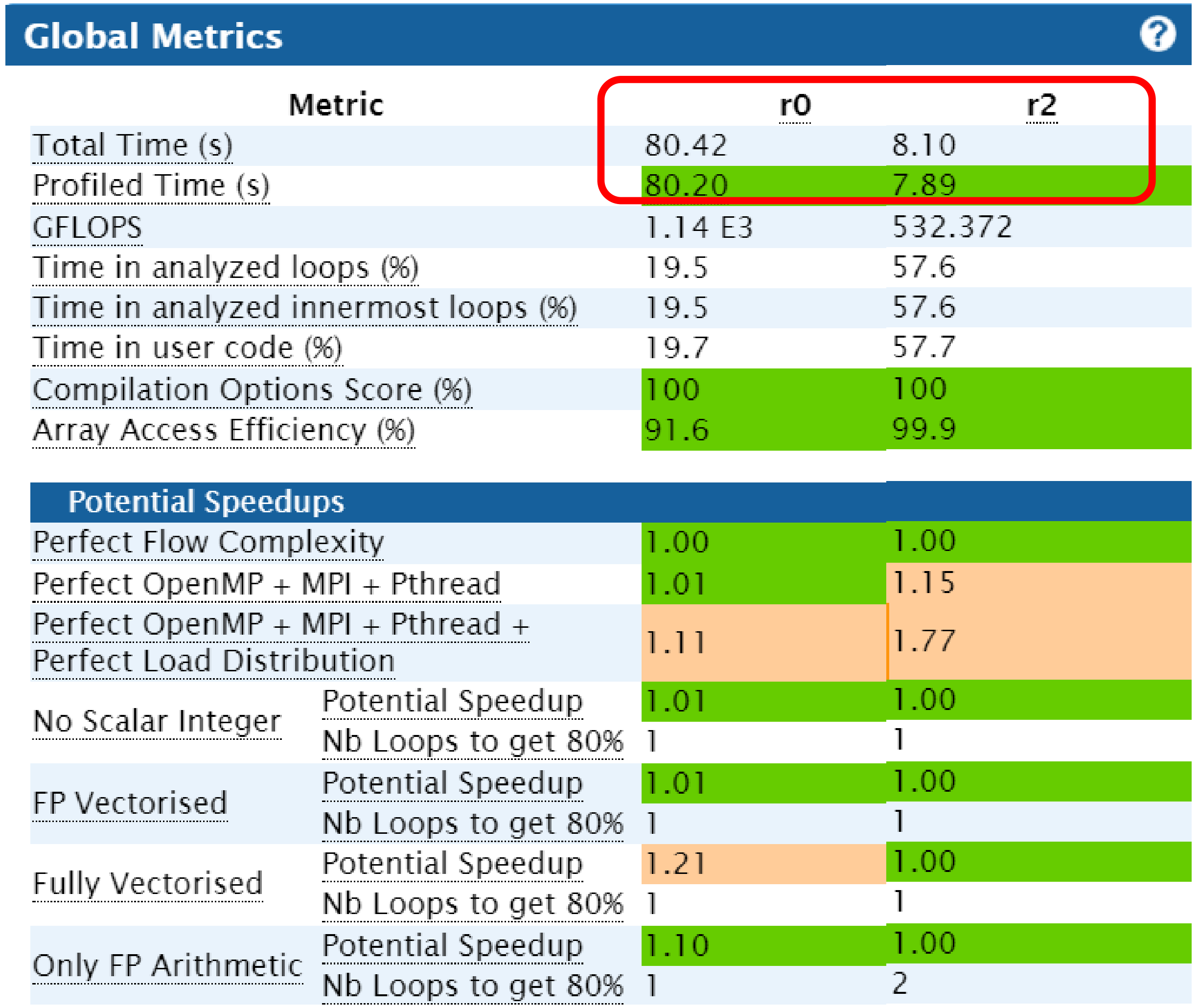
option #	flags	
1	O3 -march=znver4	**Reference O3 option pattern (essentially on Vectorization)**
2	O3 -march=znver4 -mprefer-vector-width=512	
3	O3 -march=znver4 -mprefer-vector-width=256	
4	O3 -march=znver4 -fno-vectorize -fno-slp-vectorize -fno-openmp-simd	
5	O2 -march=znver4	**Reference O2 option pattern (essentially on Vectorization)**
6	O2 -march=znver4 -mprefer-vector-width=512	
7	O2 -march=znver4 -mprefer-vector-width=256	
8	O2 -march=znver4 -fno-vectorize -fno-slp-vectorize -fno-openmp-simd	
9	O3 -march=znver4 -flto	**Reference O3 option pattern + FLTO**
10	O3 -march=znver4 -mprefer-vector-width=512 -flto	
11	O3 -march=znver4 -mprefer-vector-width=256 -flto	
12	O3 -march=znver4 -fno-vectorize -fno-slp-vectorize -fno-openmp-simd -flto	
13	O2 -march=znver4 -flto	**Reference O2 option pattern + FLTO**
14	O2 -march=znver4 -mprefer-vector-width=512 -flto	
15	O2 -march=znver4 -mprefer-vector-width=256 -flto	
16	O2 -march=znver4 -fno-vectorize -fno-slp-vectorize -fno-openmp-simd -flto	

PERFORMANCE RESULTS FOR AOCC

LOWER IS BETTER



PERFORMANCE RESULTS FOR MULTIPLE COMPILERS



Global Metrics

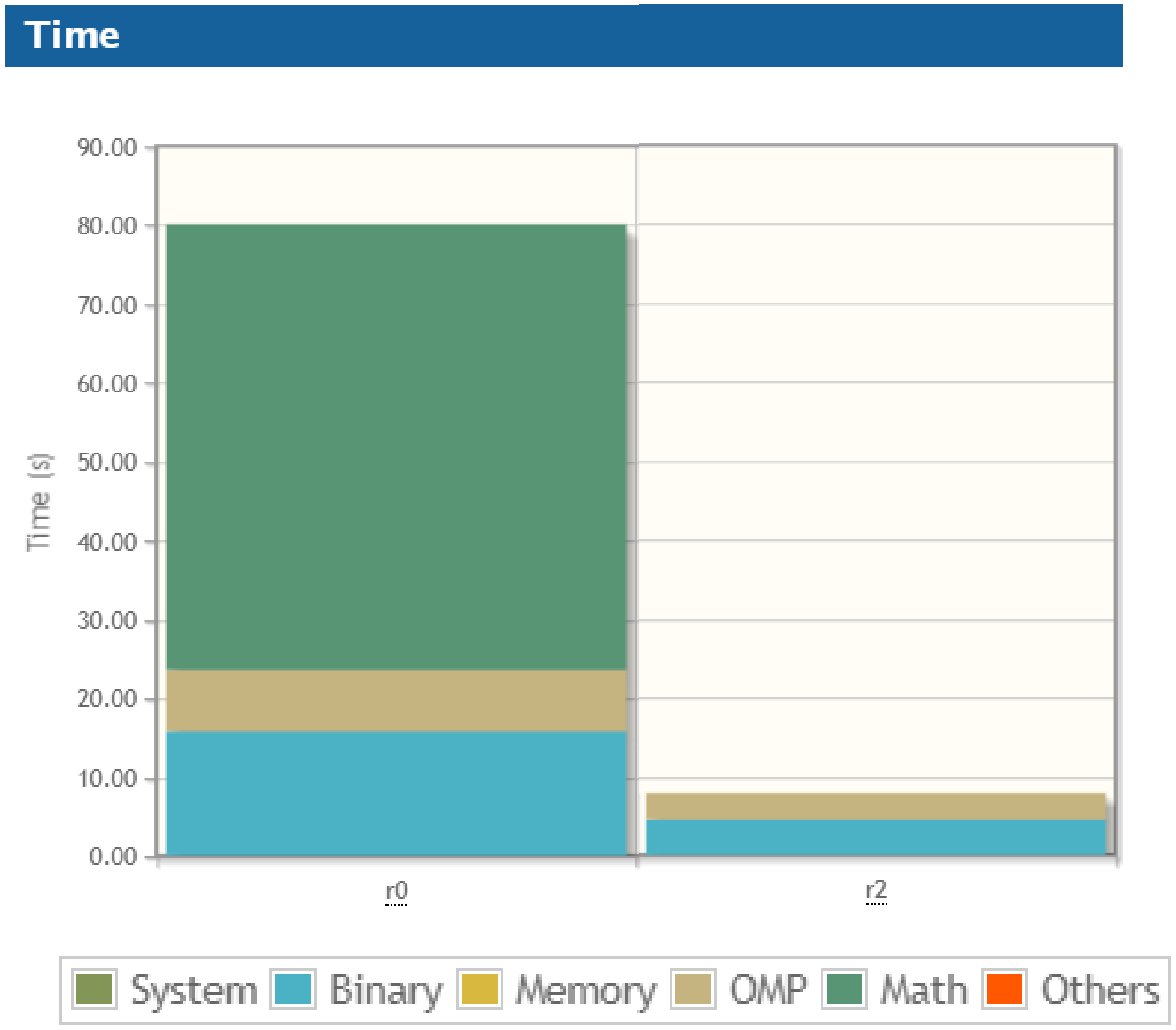
Metric		r0	r2
Total Time (s)		80.42	8.10
Profiled Time (s)		80.20	7.89
GFLOPS		1.14 E3	532.372
Time in analyzed loops (%)		19.5	57.6
Time in analyzed innermost loops (%)		19.5	57.6
Time in user code (%)		19.7	57.7
Compilation Options Score (%)		100	100
Array Access Efficiency (%)		91.6	99.9
Potential Speedups			
Perfect Flow Complexity		1.00	1.00
Perfect OpenMP + MPI + Pthread		1.01	1.15
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.11	1.77
No Scalar Integer	Potential Speedup	1.01	1.00
	Nb Loops to get 80%	1	1
FP Vectorised	Potential Speedup	1.01	1.00
	Nb Loops to get 80%	1	1
Fully Vectorised	Potential Speedup	1.21	1.00
	Nb Loops to get 80%	1	1
Only FP Arithmetic	Potential Speedup	1.10	1.00
	Nb Loops to get 80%	1	2

▼ Compared Reports

- r2: icx_3
- r0: aocc_12

AOCC 10x slower than ICX !!

PERFORMANCE RESULTS FOR MULTIPLE COMPILERS



WHAT'S HAPPENING WITH AOCC

PROBLEMS WITH AOCC

- Big performance difference is linked with math library use
- AOCC by default uses the libm installed and not its own libalm (AMD Libm). This library is installed separately from the compiler.

FIRST FIX: we forced libalm use: **unfortunately no performance gain.**

SECOND FIX: since ICX did not show any time spent in math library, try to suppress math library use: in fact add fastmath flag to AOCC so it will force inlining and suppressing lib calls.

ICX uses –fastmath by default.

This time it worked, performance very similar between AOCC and ICX

MAIN LESSONS

- **LESSON 1 (old news):** try **multiple compiler options.**
- **LESSON 2: try multiple compilers** and back port some optimizations from the best performing compilers (for example compiler flags or compiler directives).
- **LESSON 3: THE REAL ONE:** try to analyze and detect compiler failures systematically.

For using LESSON3, you need tools…..

ANALYZING COMPILER OUTPUTS: GOALS (1)

Why analyzing compiler outputs? evaluating code quality

First target: code developer: Overall goal improve Application performance and to a lesser extent application performance portability

- ➢ Analyze impact of compiler versions: provides a better answer than a simple GO/NOGO
- ➢ Analyze impact of compiler switches and identify best compiler switches
- ➢ Analyze differences in compiler behavior (across different compilers)
  - Select the best compiler
  - Import optimization from one compiler to the other one: for example compiler B has vectorized a loop which was not vectorized by the user preferred compiler A. Insert vectorization directive to the corresponding loop, working around probably a deficient data dependence analysis.
- ➢ Perform a global compiler comparison across multiple compilers: perform performance portability analysis. This might be very useful for ISV and library developers.

ANALYZING COMPILER OUTPUTS: GOALS (2)

Why analyzing compiler outputs? evaluating code quality

Second target: compiler developer. Overall goal improve compiler quality and help platform migration

- Analyze impact of compiler versions (same as application developer)
- Analyze impact of compiler switches and identify best compiler switches
- Analyze differences in compiler behavior (across different compilers)
  - Perform a competitive positioning
  - Import optimization from one compiler to the other one
  - Facilitate migration, application port by identifying strength weakness of compilers: can help benchmarking teams

Third target: benchmark (before sales), after sales support. Overall goal improve application performance (same as application developer

- Help the app developer and/or the user to fully exploit system capabilities
- Globally: very similar to the app developer but with less knowledge on the application and more knowledge on the software stack

ANALYZING COMPILER OUTPUTS: METHODOLOGY (1)

How to assess code quality ?

“The proof is in eating the cake”: use time as a main figure of merit to assess code quality: the faster is the better.

Three levels of comparison are useful and necessary!:

1. At the whole application
2. At the function level
3. At the loop level

Levels 2 and 3 complement level 1 because two compilers can achieve similar performance level at the whole application level but with very different performance at the function/loop level (compensation effect).

LIMITATIONS OF TIMING ANALYSIS (1)

Main limitations of timing analysis:

1) Timing analysis requires runs on the same machine: otherwise, comparison is unfair and is impacted by architecture differences

2) Timings are measured at the binary level and therefore are associated with binary code fragments. Main difficulty in comparison is matching functions and loops:

- ➢ Make sure that that 2 function names (generated by two different compilers) correspond to the same function at the source code level.
- ➢ Same issues with loops except that additionally the same source code loop might have multiple binary versions

Performing timing at the source level would require probe insertion at the source code level which would distort potentially code behavior.

RESOLVING FUNCTION/LOOP MATCHING

For both functions and loops, the main idea/technique is very similar. Let us focus on the more complex one i.e. loops.

- Connecting ASM code and source code
  - Relying on compiler info –g option allows to establish link between binary and source code.
- Dealing with multiple code versions
  - Goal: Grouping all of the versions together
  - Use ASM/source code connection (cf above).
  - Allow some approximation in source line numbers: code section from lines 72 to 81 is probably equivalent to code section from lines 71 to 82.
  - Multiple ASM pointing to the same code section are likely to correspond to multiple versions

REMARK 1: approximation in source line numbers does not work with very short size loops (cf array statements)

REMARK 2 : for matching functions, going through source code is easier and more efficient because in general there are not multiple versions of the same function

LIMITATIONS OF TIMING ANALYSIS (2)

Main limitations of timing analysis (follow up):

3) The “Proof is in eating the cake” does not tell you anything about the cook or the recipe. Often you need to understand why there is a timing difference.

Using other metrics (stalls, cache level access rate) than time will provide additional interesting info but will suffer from the same problem.

Comparing Compiler Optimization report is very promising and it provides some info about the code generation process.

However, Compiler Optimization reports:

- ➢ Don’t give, in general, details on failures/shortcomings of the compilation process
- ➢ Are proprietary in particular for proprietary compilers
- ➢ Are not standardized therefore extremely difficult to compare compilers.

ANALYZING CODE QUALITY (1)

Focus on loops: innermost/in between/outermost

- Focus on assembly code (main compiler output)..
- Evaluate ASM using CQA (Code Quality Analysis) included in MAQAO.
- Generic topics of interest
  - Port / FU usage
  - Vectorization
  - Instruction set use
  - Vectorization Roadblocks
  - Data access
- Use simplified simulation tools (such as CQA/UFS) to get performance estimations; critical for comparing ASM versions

By looking directly at ASM, both compiler mistakes but also source code issues will be taken into account.

ANALYZING CODE QUALITY (2)

Two level analysis

- **Static** at the ASM level denoted (**SA**) in next slides
- **Dynamic** requiring measurement at execution denoted (**DT**) in the sequel
- All static metrics/issues are detected by CQA (Code Quality Analysis included in MAQAO) while dynamic rely on MAQAO instrumentation at binary level
- Dynamic profiling is also essential to assess loop relative cost.

ANALYZING CODE QUALITY (3)

Classify performance issues into 5 main categories

1. **Loop computation:** issues related to the computation organization.
2. **Control Flow:** issues relevant to control
3. **Data access:** issues essentially related to memory operations
4. **Vectorization roadblocks:** issues preventing vectorization
5. **Inefficient vectorization:** issues related to vectorization quality

LOOP COMPUTATION ISSUES

ISSUES
Presence of reductions dependency cycles (SA)
Presence of expensive FP instructions: div/sqrt, sin/cos, exp/log, etc…(SA)
Presence of special convert instructions: moving between different FP format (SA)
Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA (SA)
Large loop body: over micro-op cache size (SA)
Presence of a large number of scalar integer instructions: more than 1.1 x speedup when suppressing scalar integer instructions (SA)
Bottleneck in the front end (SA)
Low iteration count (DT)
Highly variable Cycle per Iteration across loop instances (DT)

CONTROL FLOW ISSUES

ISSUES
Presence of calls (SA)
Presence of 2 to 4 paths (SA)
Presence of more than 4 paths (SA)
Non-innermost loop (SA)
Low iteration count (DT)

VECTORIZATION ROADBLOCKS

ISSUES
Presence of calls (SA)
Presence of 2 to 4 paths (SA)
Presence of more than 4 paths (SA)
Presence of reductions dependency cycles (SA)
Presence of constant non unit stride data access (SA)
Presence of indirect access (SA)
Non innermost loop (SA)

VECTORIZATION EFFICIENCY ISSUES

ISSUES
Partial or unexisting vectorization (SA)
Presence of expensive instructions : scatter/gather (SA)
Presence of special instructions executing on a single port (SA): typically all data restructuring instructions, expand, pack, unpack, etc…
Use of shorter than available vector length (SA)
Use of masked instructions (SA)
Time spent in peel/tail loop greater than time spent in main loop (DT)

CASE STUDY 1

SPMXV: A VERY SIMPLE CODE

```
#pragma omp parallel for schedule(static, CHUNK_SIZE) num_threads(nth)
for (i = 0; i < tInput->stNumRows; i++) {
   double sum = 0.0;
   int  rowbeg = Arow[i];
   int  rowend = Arow[i+1];
   int  nz;
   for (nz = rowbeg; nz < rowend; nz++) {
      int  col =  Acol[nz];
      sum += Aval[nz] * x[ col ];
   }
   y[i] = sum;
 }
```

Test code developed by RWTH Aachen.

Essential kernel for iterative method.

A few key characteristics:

- Outermost loop (on i)
  - Fully parallel
  - Very large iteration count
- Innermost loop (on nz):
  - Reduction
  - Low iteration (typically less than 15)
  - Regular sweep on Array A and indirect access on array x

SPMXV TESTS : HARD / SOFT CONFIGURATION

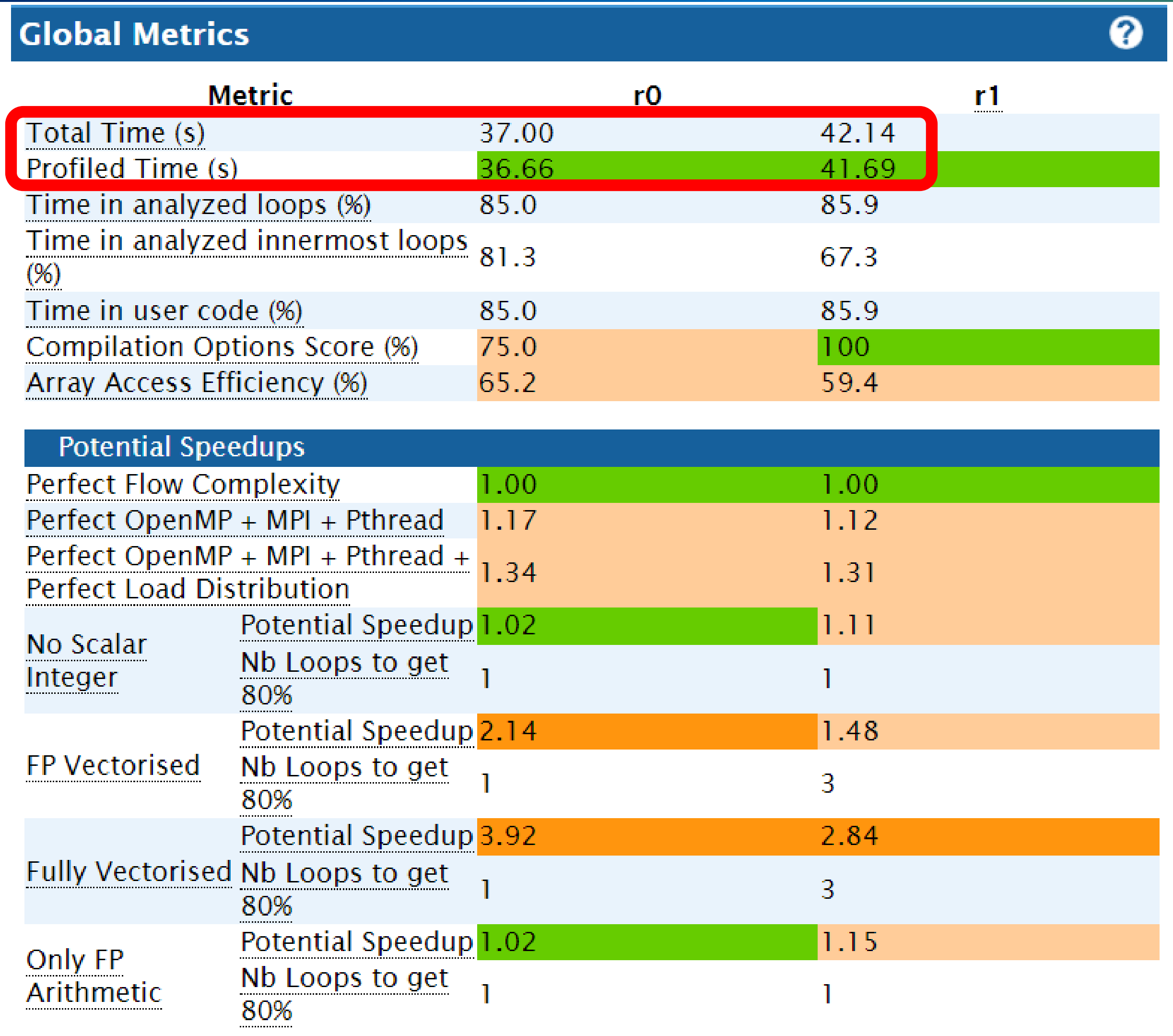
HARDWARE CONFIGURATION

- Skylake : Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz3)
- 2 x 26 Cores
- 2.1 GHz

SOFTWARE CONFIGURATION

- OPENMP parallel: 52 threads
- Linux 6.10.10-arch1-1 #1 SMP PREEMPT_DYNAMIC
- GCC: GNU C++17 14.2.1 20240910 -march=skylake-avx512 ....
- ICX: clang based Intel(R) oneAPI DPC++/C++ Compiler 2024.2.1 (2024.2.1.20240711) -g -fiopenmp -march=native -O3 ......

SPMXV COMPARE: GCC VERSUS ICX



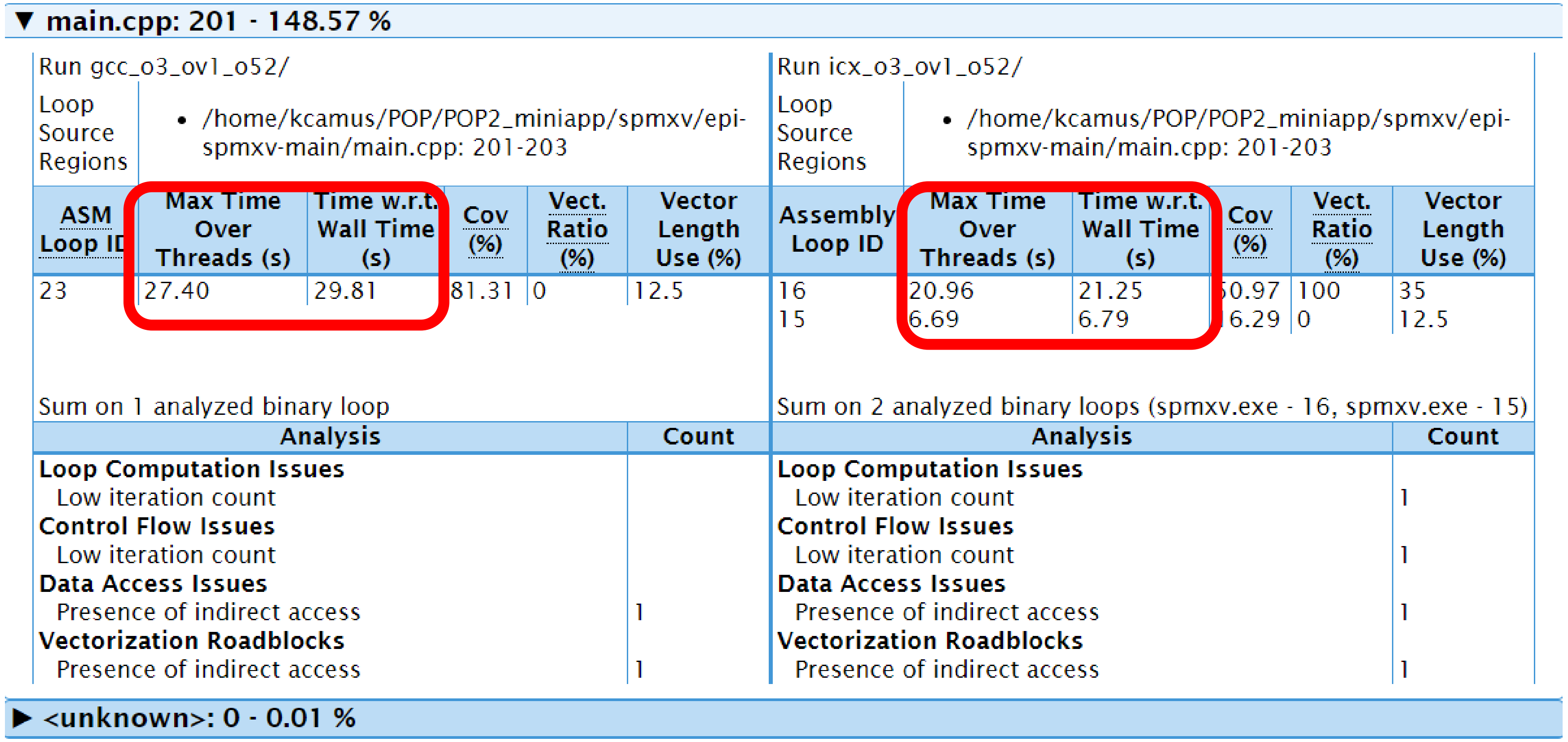
Global Metrics

Metric		r0	r1
Total Time (s)		37.00	42.14
Profiled Time (s)		36.66	41.69
Time in analyzed loops (%)		85.0	85.9
Time in analyzed innermost loops (%)		81.3	67.3
Time in user code (%)		85.0	85.9
Compilation Options Score (%)		75.0	100
Array Access Efficiency (%)		65.2	59.4
Potential Speedups			
Perfect Flow Complexity		1.00	1.00
Perfect OpenMP + MPI + Pthread		1.17	1.12
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.34	1.31
No Scalar Integer	Potential Speedup	1.02	1.11
	Nb Loops to get 80%	1	1
FP Vectorised	Potential Speedup	2.14	1.48
	Nb Loops to get 80%	1	3
Fully Vectorised	Potential Speedup	3.92	2.84
	Nb Loops to get 80%	1	3
Only FP Arithmetic	Potential Speedup	1.02	1.15
	Nb Loops to get 80%	1	1

▼ Compared Reports

- r0: gcc_o3_ov1_o52/
- r1: icx_o3_ov1_o52/

INNERMOST LOOP COMPARISON



▼ main.cpp: 201 - 148.57 %

Run gcc_o3_ov1_o52/

Loop Source Regions
- /home/kcamus/POP/POP2_miniapp/spmxv/epi-spmxv-main/main.cpp: 201-203

ASM Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)
23	27.40	29.81	81.31	0	12.5

Sum on 1 analyzed binary loop

Analysis	Count
Loop Computation Issues	
Low iteration count	
Control Flow Issues	
Low iteration count	
Data Access Issues	
Presence of indirect access	1
Vectorization Roadblocks	
Presence of indirect access	1

Run icx_o3_ov1_o52/

Loop Source Regions
- /home/kcamus/POP/POP2_miniapp/spmxv/epi-spmxv-main/main.cpp: 201-203

Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)
16	20.96	21.25	50.97	100	35
15	6.69	6.79	16.29	0	12.5

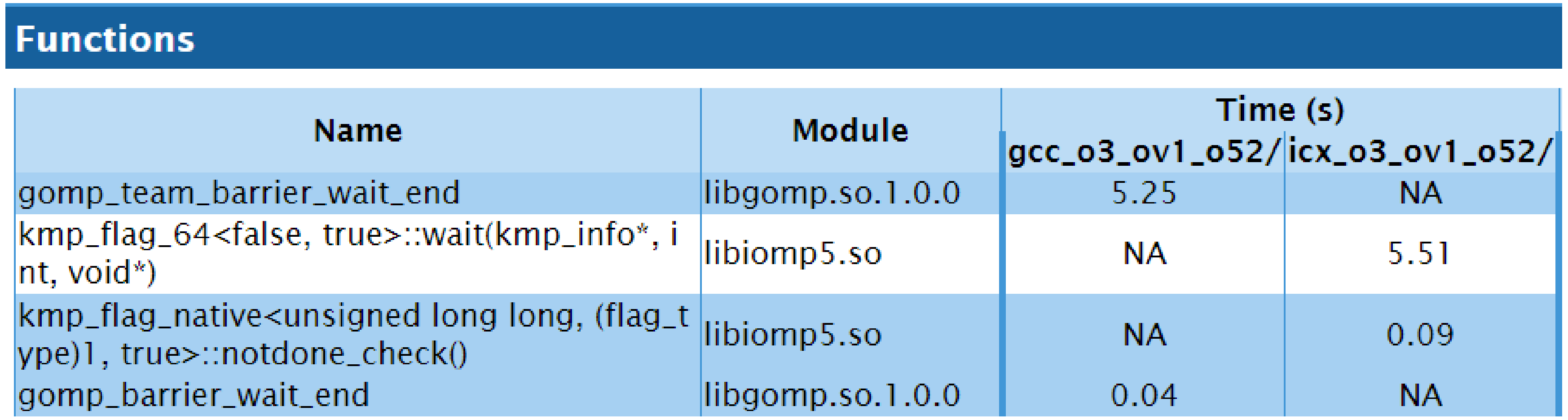
Sum on 2 analyzed binary loops (spmxv.exe - 16, spmxv.exe - 15)

Analysis	Count
Loop Computation Issues	
Low iteration count	1
Control Flow Issues	
Low iteration count	1
Data Access Issues	
Presence of indirect access	1
Vectorization Roadblocks	
Presence of indirect access	1

▶ <unknown>: 0 - 0.01 %

- GCC remains scalar
- ICX vectorizes
- All in all, very similar loop timings

OPENMP COMPARISON

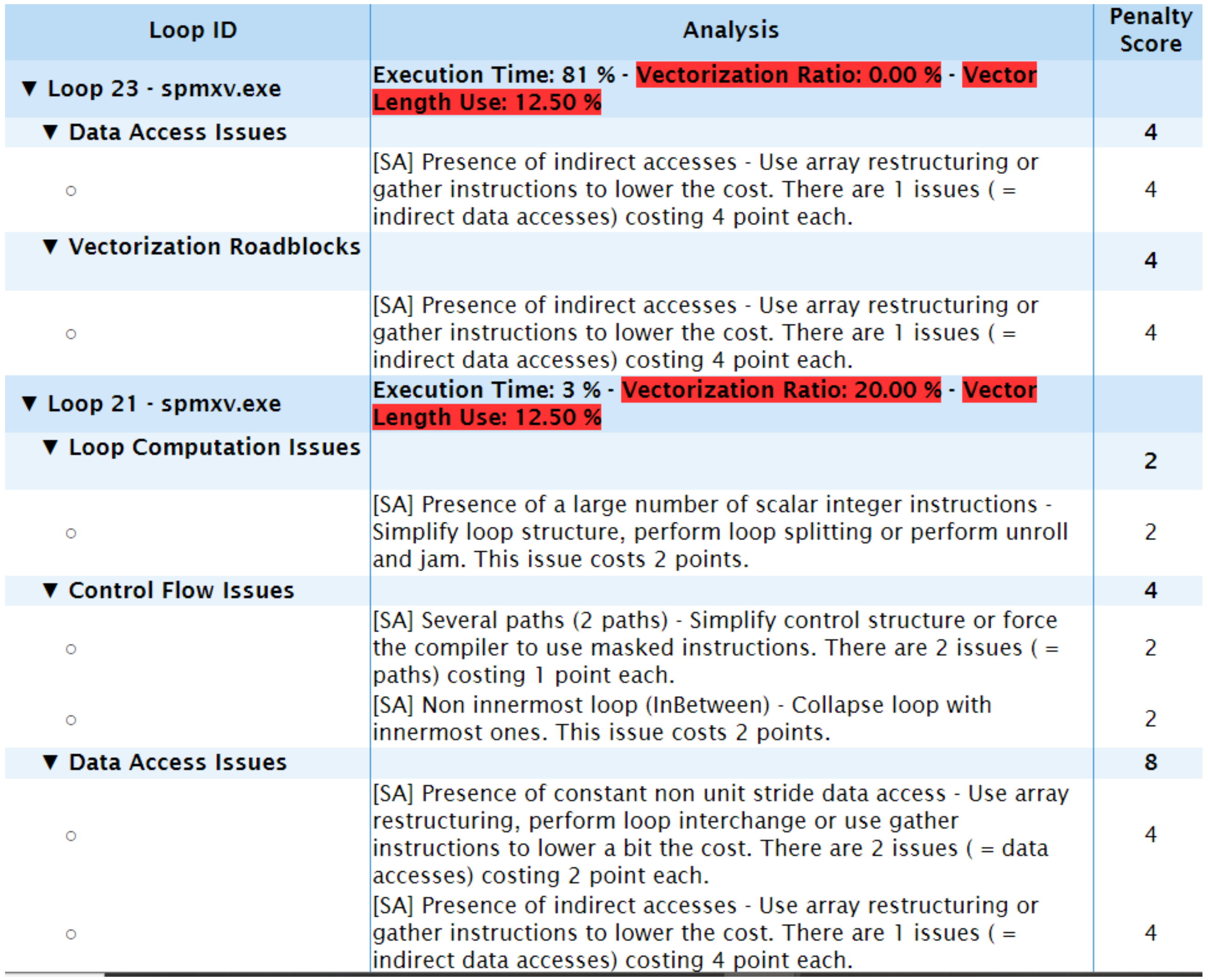


Functions

Name	Module	Time (s)	
		gcc_o3_ov1_o52/	icx_o3_ov1_o52/
gomp_team_barrier_wait_end	libgomp.so.1.0.0	5.25	NA
kmp_flag_64<false, true>::wait(kmp_info*, int, void*)	libiomp5.so	NA	5.51
kmp_flag_native<unsigned long long, (flag_type)1, true>::notdone_check()	libiomp5.so	NA	0.09
gomp_barrier_wait_end	libgomp.so.1.0.0	0.04	NA

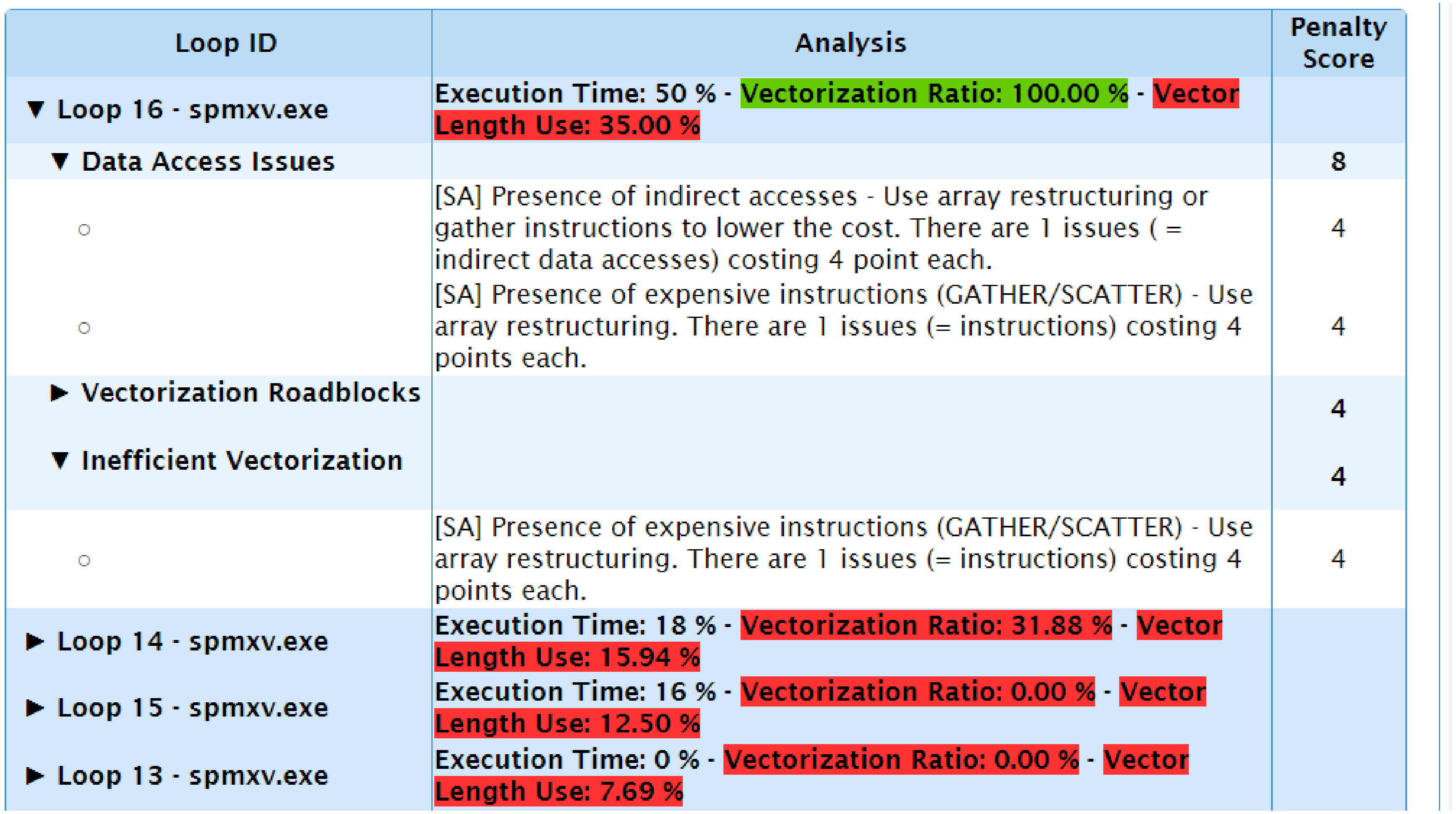
- OpenMP overheads are very similar in both cases
- Matching OpenMP libraries between 2 different compilers is not easy.

GCC DETAILED LOOP ANALYSIS



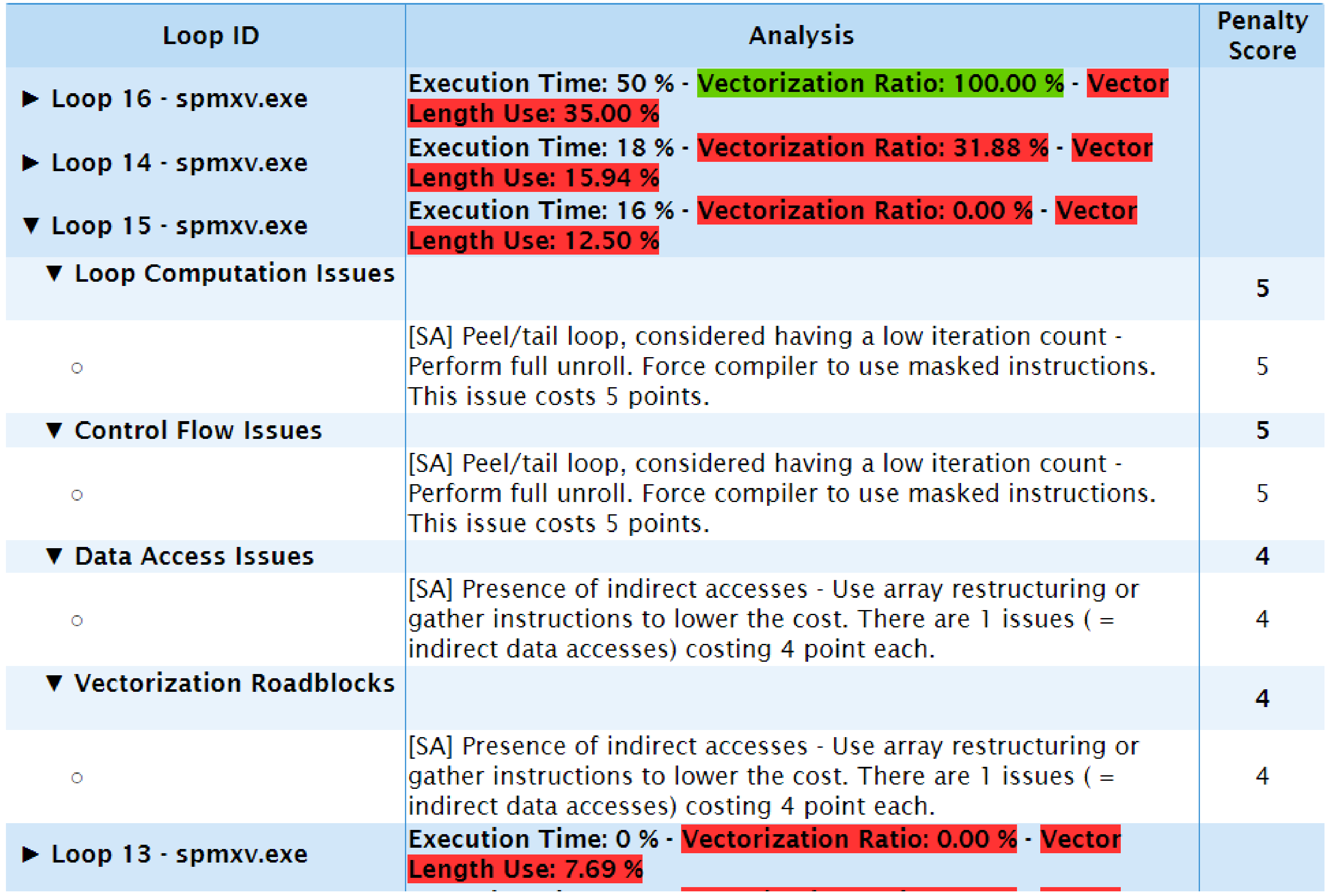
Loop ID	Analysis	Penalty Score
▼ **Loop 23 - spmxv.exe**	**Execution Time: 81 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %**	
▼ **Data Access Issues**		**4**
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 1 issues (= indirect data accesses) costing 4 point each.	4
▼ **Vectorization Roadblocks**		**4**
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 1 issues (= indirect data accesses) costing 4 point each.	4
▼ **Loop 21 - spmxv.exe**	**Execution Time: 3 % - Vectorization Ratio: 20.00 % - Vector Length Use: 12.50 %**	
▼ **Loop Computation Issues**		**2**
○	[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**
○	[SA] Several paths (2 paths) - Simplify control structure or force the compiler to use masked instructions. There are 2 issues (= paths) costing 1 point each.	2
○	[SA] Non innermost loop (InBetween) - Collapse loop with innermost ones. This issue costs 2 points.	2
▼ **Data Access Issues**		**8**
○	[SA] Presence of constant non unit stride data access - Use array restructuring, perform loop interchange or use gather instructions to lower a bit the cost. There are 2 issues (= data accesses) costing 2 point each.	4
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 1 issues (= indirect data accesses) costing 4 point each.	4

ICX DETAILED LOOP ANALYSIS (1)



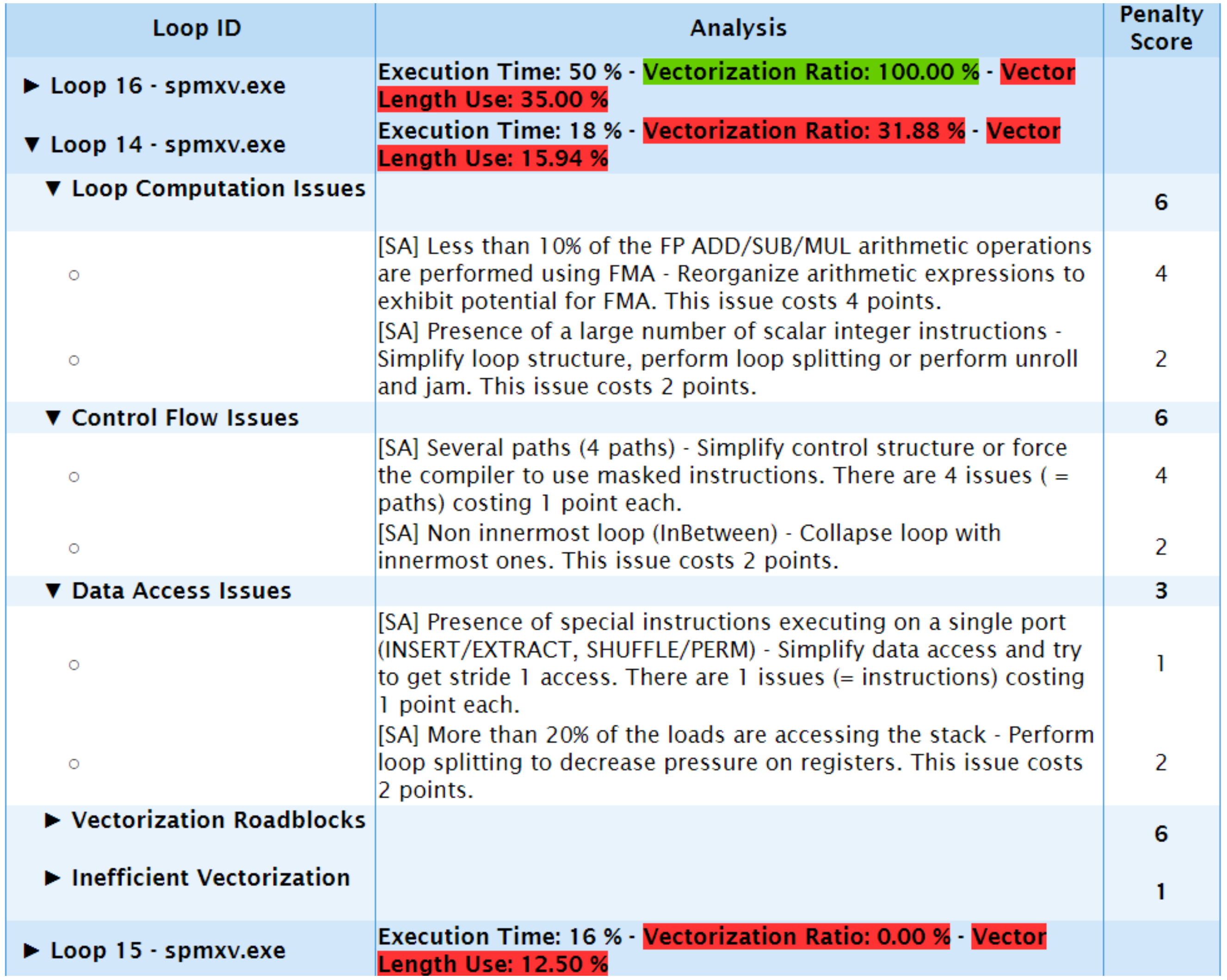
Loop ID	Analysis	Penalty Score
▼ Loop 16 - spmxv.exe	Execution Time: 50 % - Vectorization Ratio: 100.00 % - Vector Length Use: 35.00 %	
▼ Data Access Issues		8
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 1 issues (= indirect data accesses) costing 4 point each.	4
○	[SA] Presence of expensive instructions (GATHER/SCATTER) - Use array restructuring. There are 1 issues (= instructions) costing 4 points each.	4
▶ Vectorization Roadblocks		4
▼ Inefficient Vectorization		4
○	[SA] Presence of expensive instructions (GATHER/SCATTER) - Use array restructuring. There are 1 issues (= instructions) costing 4 points each.	4
▶ Loop 14 - spmxv.exe	Execution Time: 18 % - Vectorization Ratio: 31.88 % - Vector Length Use: 15.94 %	
▶ Loop 15 - spmxv.exe	Execution Time: 16 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 13 - spmxv.exe	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 7.69 %	

ICX DETAILED LOOP ANALYSIS (2)



Loop ID	Analysis	Penalty Score
▶ Loop 16 - spmxv.exe	Execution Time: 50 % - Vectorization Ratio: 100.00 % - Vector Length Use: 35.00 %	
▶ Loop 14 - spmxv.exe	Execution Time: 18 % - Vectorization Ratio: 31.88 % - Vector Length Use: 15.94 %	
▼ Loop 15 - spmxv.exe	Execution Time: 16 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▼ Loop Computation Issues		**5**
○	[SA] Peel/tail loop, considered having a low iteration count - Perform full unroll. Force compiler to use masked instructions. This issue costs 5 points.	5
▼ Control Flow Issues		**5**
○	[SA] Peel/tail loop, considered having a low iteration count - Perform full unroll. Force compiler to use masked instructions. This issue costs 5 points.	5
▼ Data Access Issues		**4**
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 1 issues (= indirect data accesses) costing 4 point each.	4
▼ Vectorization Roadblocks		**4**
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 1 issues (= indirect data accesses) costing 4 point each.	4
▶ Loop 13 - spmxv.exe	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 7.69 %	

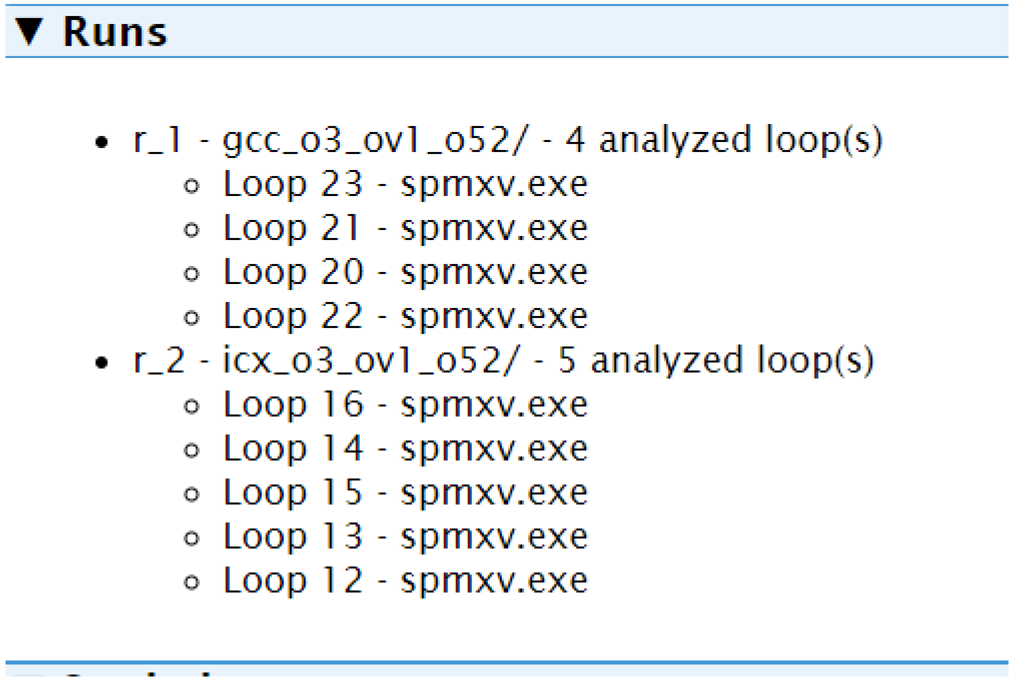
ICX DETAILED LOOP ANALYSIS (3)



Loop ID	Analysis	Penalty Score
▶ Loop 16 - spmxv.exe	Execution Time: 50 % - Vectorization Ratio: 100.00 % - Vector Length Use: 35.00 %	
▼ Loop 14 - spmxv.exe	Execution Time: 18 % - Vectorization Ratio: 31.88 % - Vector Length Use: 15.94 %	
▼ 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		6
○	[SA] Several paths (4 paths) - Simplify control structure or force the compiler to use masked instructions. There are 4 issues (= paths) costing 1 point each.	4
○	[SA] Non innermost loop (InBetween) - Collapse loop with innermost ones. This issue costs 2 points.	2
▼ Data Access Issues		3
○	[SA] Presence of special instructions executing on a single port (INSERT/EXTRACT, SHUFFLE/PERM) - Simplify data access and try to get stride 1 access. There are 1 issues (= instructions) costing 1 point each.	1
○	[SA] More than 20% of the loads are accessing the stack - Perform loop splitting to decrease pressure on registers. This issue costs 2 points.	2
▶ Vectorization Roadblocks		6
▶ Inefficient Vectorization		1
▶ Loop 15 - spmxv.exe	Execution Time: 16 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	

SPMXV: AGGREGATE RESULTS (1)

Single aggregation is performed: first for each compiler, various issues are aggregated across key loops (see below)
Double aggregation is also performed: issues are aggregated between compilers.

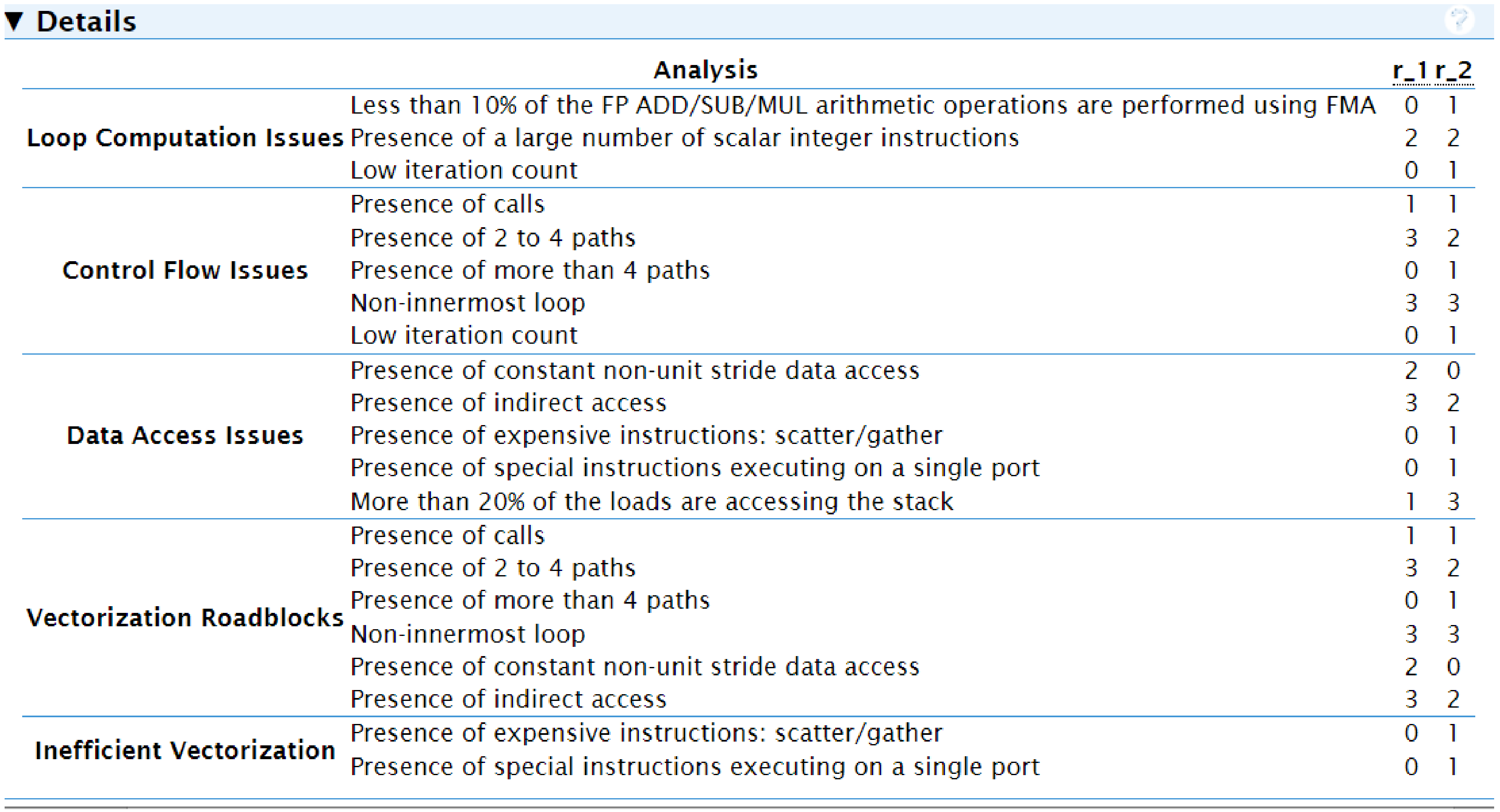


▼ Runs

- r_1 - gcc_o3_ov1_o52/ - 4 analyzed loop(s)
  - Loop 23 - spmxv.exe
  - Loop 21 - spmxv.exe
  - Loop 20 - spmxv.exe
  - Loop 22 - spmxv.exe
- r_2 - icx_o3_ov1_o52/ - 5 analyzed loop(s)
  - Loop 16 - spmxv.exe
  - Loop 14 - spmxv.exe
  - Loop 15 - spmxv.exe
  - Loop 13 - spmxv.exe
  - Loop 12 - spmxv.exe

SPMXV: SINGLE AGGREGATION (2)

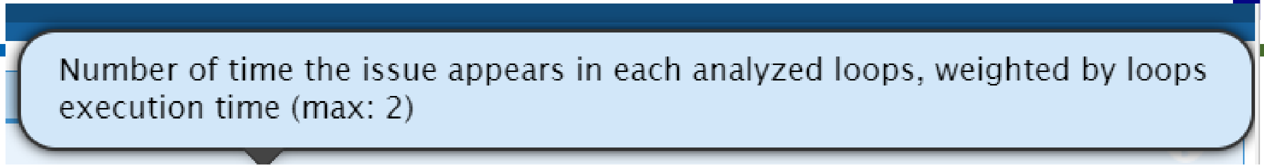
- r_1 - gcc_o3_ov1_o52/ - 4 analyzed loop(s)
- r_2 - icx_o3_ov1_o52/ - 5 analyzed loop(s)

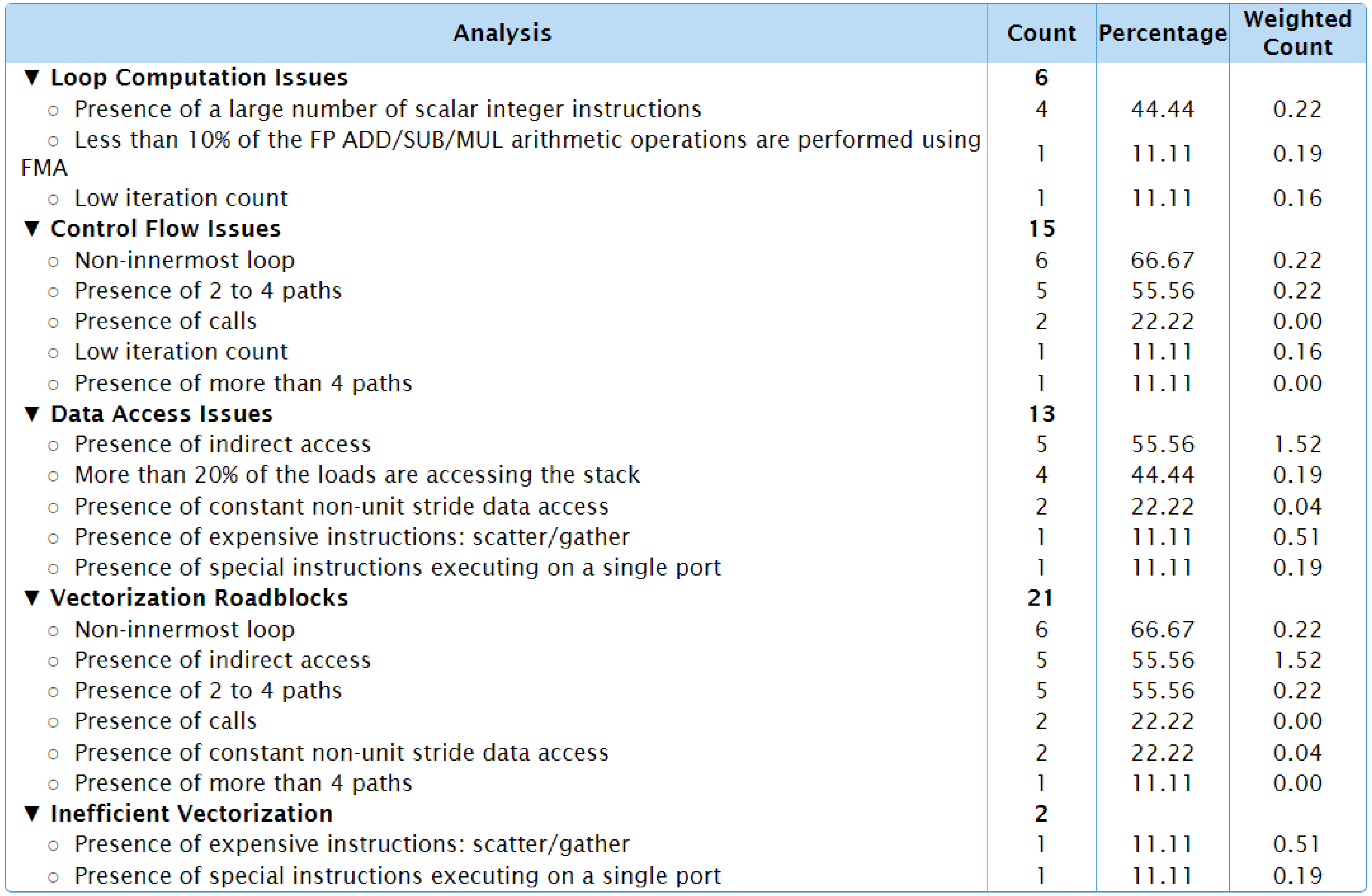


▼ Details

Analysis		r_1	r_2
Loop Computation Issues	Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	0	1
	Presence of a large number of scalar integer instructions	2	2
	Low iteration count	0	1
Control Flow Issues	Presence of calls	1	1
	Presence of 2 to 4 paths	3	2
	Presence of more than 4 paths	0	1
	Non-innermost loop	3	3
	Low iteration count	0	1
Data Access Issues	Presence of constant non-unit stride data access	2	0
	Presence of indirect access	3	2
	Presence of expensive instructions: scatter/gather	0	1
	Presence of special instructions executing on a single port	0	1
	More than 20% of the loads are accessing the stack	1	3
Vectorization Roadblocks	Presence of calls	1	1
	Presence of 2 to 4 paths	3	2
	Presence of more than 4 paths	0	1
	Non-innermost loop	3	3
	Presence of constant non-unit stride data access	2	0
	Presence of indirect access	3	2
Inefficient Vectorization	Presence of expensive instructions: scatter/gather	0	1
	Presence of special instructions executing on a single port	0	1

SPMXV: DOUBLE AGGREGATION (3)


Number of time the issue appears in each analyzed loops, weighted by loops execution time (max: 2)



Analysis	Count	Percentage	Weighted Count
▼ **Loop Computation Issues**	**6**		
○ Presence of a large number of scalar integer instructions	4	44.44	0.22
○ Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	1	11.11	0.19
○ Low iteration count	1	11.11	0.16
▼ **Control Flow Issues**	**15**		
○ Non-innermost loop	6	66.67	0.22
○ Presence of 2 to 4 paths	5	55.56	0.22
○ Presence of calls	2	22.22	0.00
○ Low iteration count	1	11.11	0.16
○ Presence of more than 4 paths	1	11.11	0.00
▼ **Data Access Issues**	**13**		
○ Presence of indirect access	5	55.56	1.52
○ More than 20% of the loads are accessing the stack	4	44.44	0.19
○ Presence of constant non-unit stride data access	2	22.22	0.04
○ Presence of expensive instructions: scatter/gather	1	11.11	0.51
○ Presence of special instructions executing on a single port	1	11.11	0.19
▼ **Vectorization Roadblocks**	**21**		
○ Non-innermost loop	6	66.67	0.22
○ Presence of indirect access	5	55.56	1.52
○ Presence of 2 to 4 paths	5	55.56	0.22
○ Presence of calls	2	22.22	0.00
○ Presence of constant non-unit stride data access	2	22.22	0.04
○ Presence of more than 4 paths	1	11.11	0.00
▼ **Inefficient Vectorization**	**2**		
○ Presence of expensive instructions: scatter/gather	1	11.11	0.51
○ Presence of special instructions executing on a single port	1	11.11	0.19

CASE STUDY 2

OPENRADIOSS AND RADIOSS

Altair® Radioss® & OpenRadioss™
The Industry Standard Open Platform for Crash & Impact

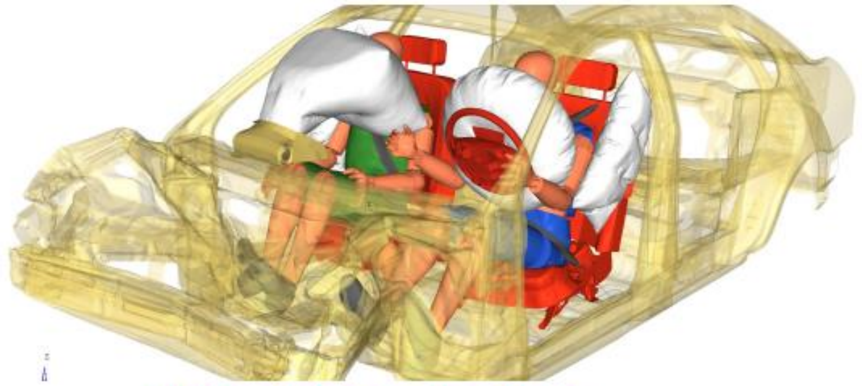
OpenRadioss™ Open-Source Version

- Source code publicly accessible from: https://github.com/OpenRadioss
- Upstream version, with contributions from a fast-growing worldwide community
- Precompiled Linux & Windows executables to run latest builds with no license check
- Support from the community, via forum

www.openradioss.org

Altair® Radioss® Commercial Version

- Commercial releases with extensive QA, professional support, documentation and maintenance priority
- Available under Altair Units license
- Encrypted models for dummies & barriers
- Channels valuable community contributions into industrial release



www.altair.com/radioss

ALTAIR

OpenRadioss TESTS : HARD / SOFT CONFIGURATION

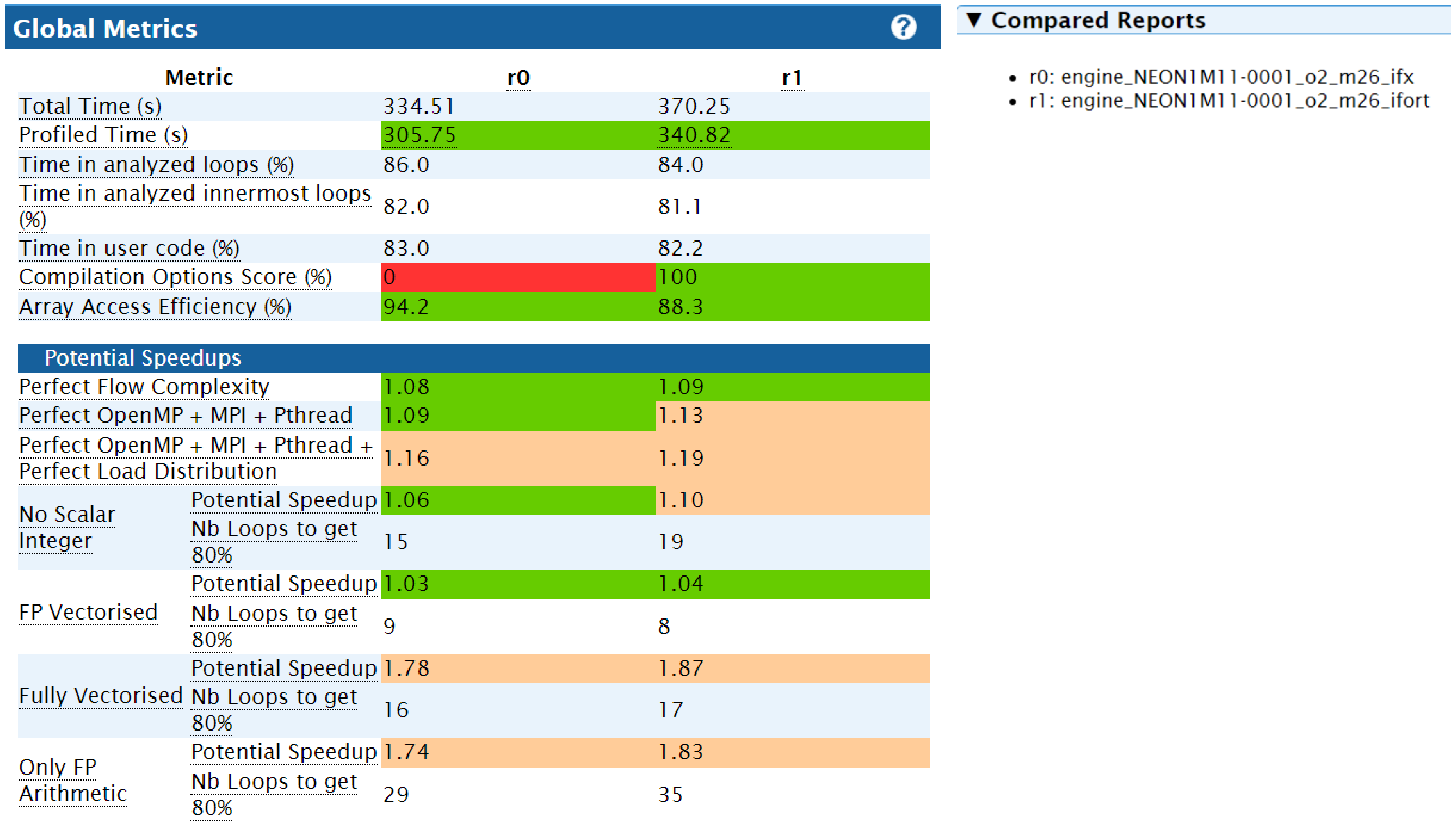
HARDWARE CONFIGURATION

- Skylake : Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz3)
- 2 x 26 Cores
- 2.1 GHz

SOFTWARE CONFIGURATION

- MPI 26 Process OPENMP 2 Threads
- Linux 6.10.10-arch1-1 #1 SMP PREEMPT_DYNAMIC
- Intel(R) Fortran 24.0-1693
- Intel(R) Fortran Intel(R) 64 Compiler Classic for applications running on Intel(R) 64, Version 2021.13.1 Build 20240703_000000

OpenRadioss COMPARE IFORT VERSUS IFX



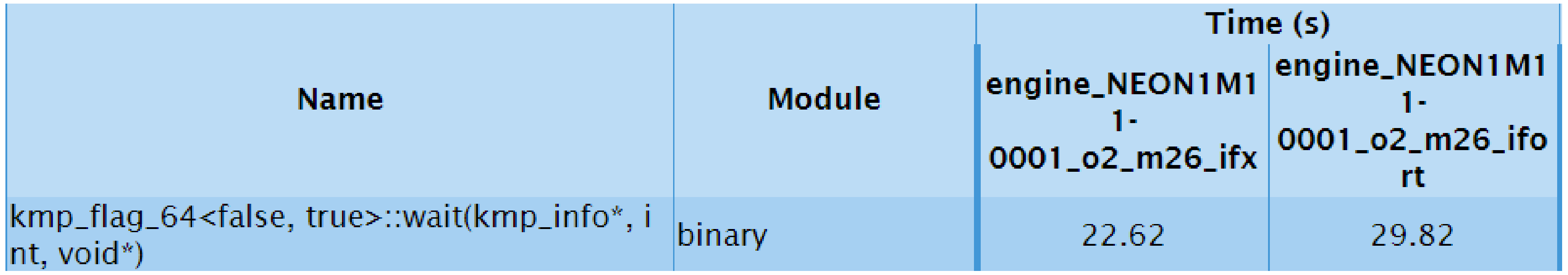
Global Metrics

Metric		r0	r1
Total Time (s)		334.51	370.25
Profiled Time (s)		305.75	340.82
Time in analyzed loops (%)		86.0	84.0
Time in analyzed innermost loops (%)		82.0	81.1
Time in user code (%)		83.0	82.2
Compilation Options Score (%)		0	100
Array Access Efficiency (%)		94.2	88.3
Potential Speedups			
Perfect Flow Complexity		1.08	1.09
Perfect OpenMP + MPI + Pthread		1.09	1.13
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.16	1.19
No Scalar Integer	Potential Speedup	1.06	1.10
	Nb Loops to get 80%	15	19
FP Vectorised	Potential Speedup	1.03	1.04
	Nb Loops to get 80%	9	8
Fully Vectorised	Potential Speedup	1.78	1.87
	Nb Loops to get 80%	16	17
Only FP Arithmetic	Potential Speedup	1.74	1.83
	Nb Loops to get 80%	29	35

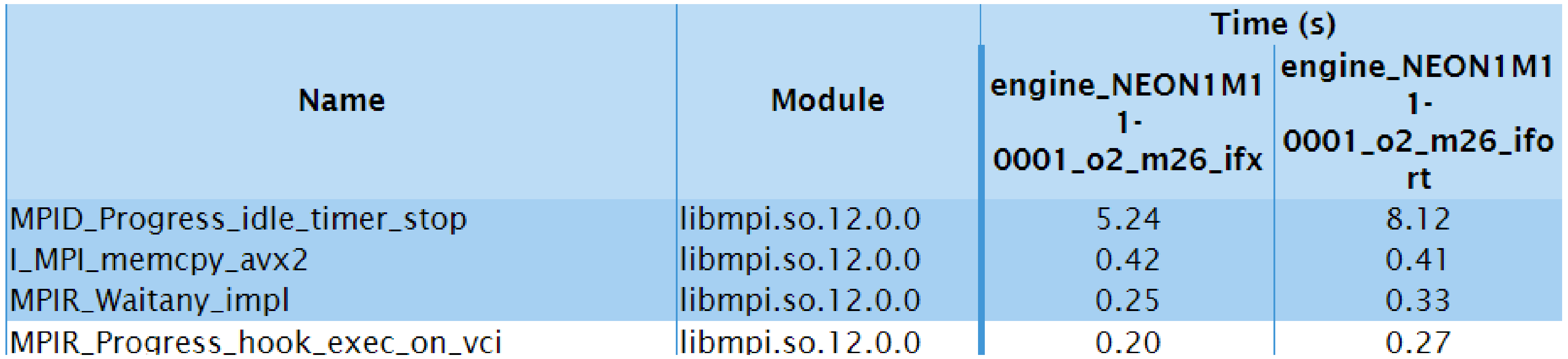
▼ **Compared Reports**

- r0: engine_NEON1M11-0001_o2_m26_ifx
- r1: engine_NEON1M11-0001_o2_m26_ifort

MPI + OPENMP COMPARISON



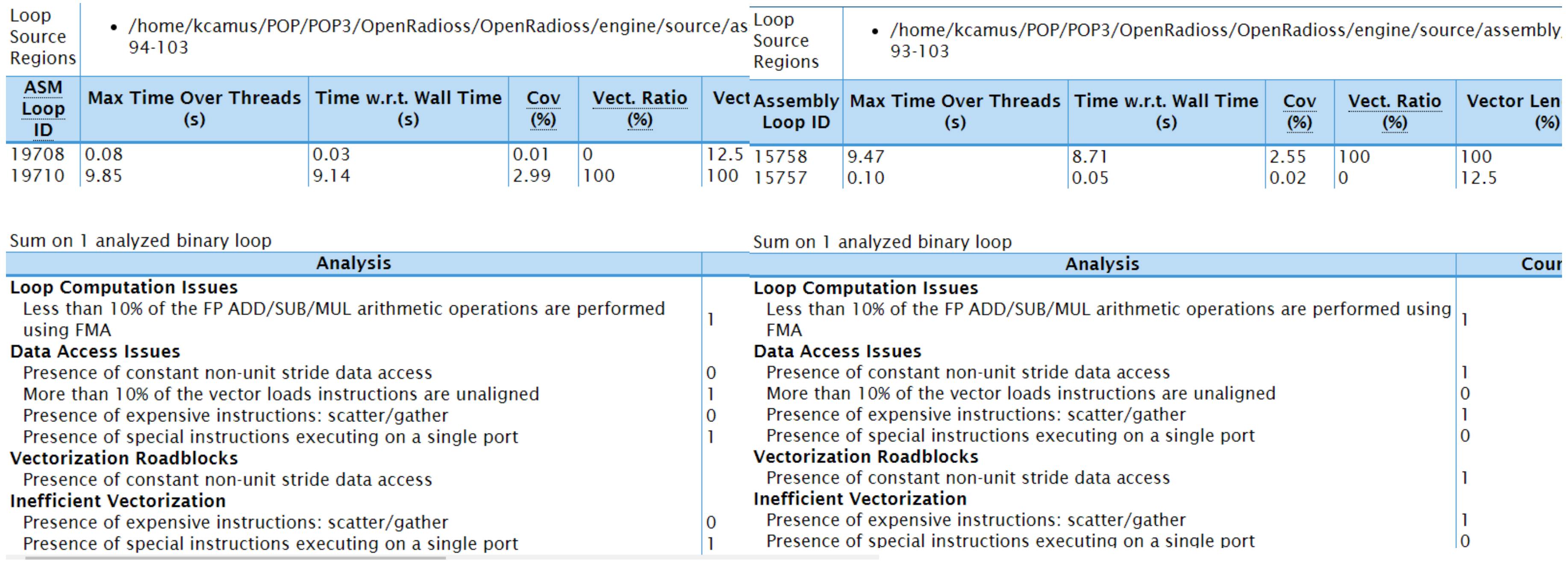
Name	Module	Time (s)	
		engine_NEON1M11-0001_o2_m26_ifx	engine_NEON1M11-0001_o2_m26_ifort
kmp_flag_64<false, true>::wait(kmp_info*, int, void*)	binary	22.62	29.82



Name	Module	Time (s)	
		engine_NEON1M11-0001_o2_m26_ifx	engine_NEON1M11-0001_o2_m26_ifort
MPID_Progress_idle_timer_stop	libmpi.so.12.0.0	5.24	8.12
I_MPI_memcpy_avx2	libmpi.so.12.0.0	0.42	0.41
MPIR_Waitany_impl	libmpi.so.12.0.0	0.25	0.33
MPIR_Progress_hook_exec_on_vci	libmpi.so.12.0.0	0.20	0.27

- OpenMP difference : 7.3 sec
- MPI difference : around 2.7 sec
- OpenMP + MPI account for 10sec difference still far from the whole app level difference: 35 sec

OpenRadioss : LOOP COMPARISON



Loop Source Regions

- /home/kcamus/POP/POP3/OpenRadioss/OpenRadioss/engine/source/as
  94-103

ASM Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vect
19708	0.08	0.03	0.01	0	12.5
19710	9.85	9.14	2.99	100	100

Sum on 1 analyzed binary loop

Analysis	
Loop Computation Issues	
Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	1
Data Access Issues	
Presence of constant non-unit stride data access	0
More than 10% of the vector loads instructions are unaligned	1
Presence of expensive instructions: scatter/gather	0
Presence of special instructions executing on a single port	1
Vectorization Roadblocks	
Presence of constant non-unit stride data access	
Inefficient Vectorization	
Presence of expensive instructions: scatter/gather	0
Presence of special instructions executing on a single port	1

Loop Source Regions

- /home/kcamus/POP/POP3/OpenRadioss/OpenRadioss/engine/source/assembly
  93-103

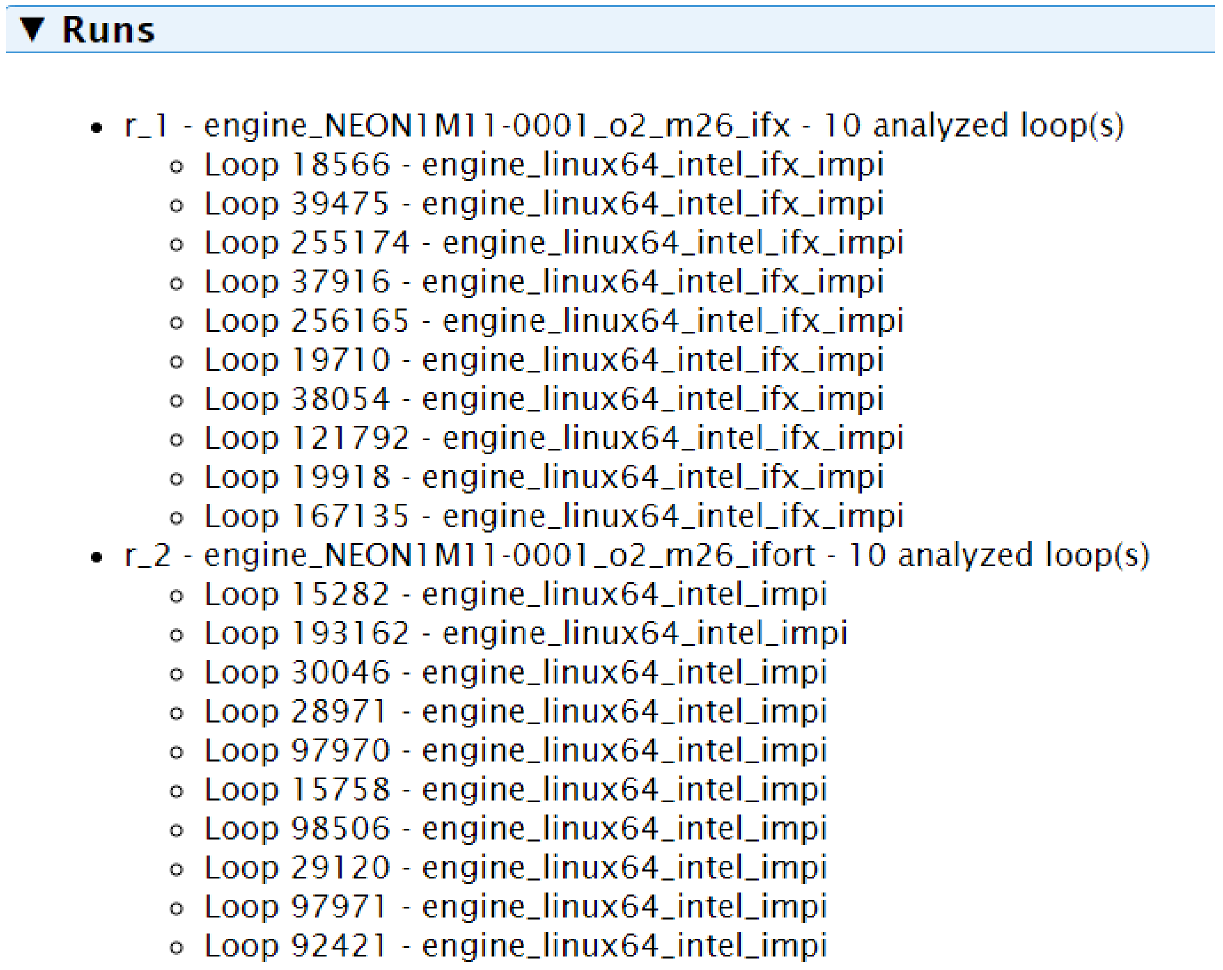
Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Len (%)
15758	9.47	8.71	2.55	100	100
15757	0.10	0.05	0.02	0	12.5

Sum on 1 analyzed binary loop

Analysis	Cour
Loop Computation Issues	
Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	1
Data Access Issues	
Presence of constant non-unit stride data access	1
More than 10% of the vector loads instructions are unaligned	0
Presence of expensive instructions: scatter/gather	1
Presence of special instructions executing on a single port	0
Vectorization Roadblocks	
Presence of constant non-unit stride data access	1
Inefficient Vectorization	
Presence of expensive instructions: scatter/gather	1
Presence of special instructions executing on a single port	0

OpenRADIOSS: AGGREGATE RESULTS (1)

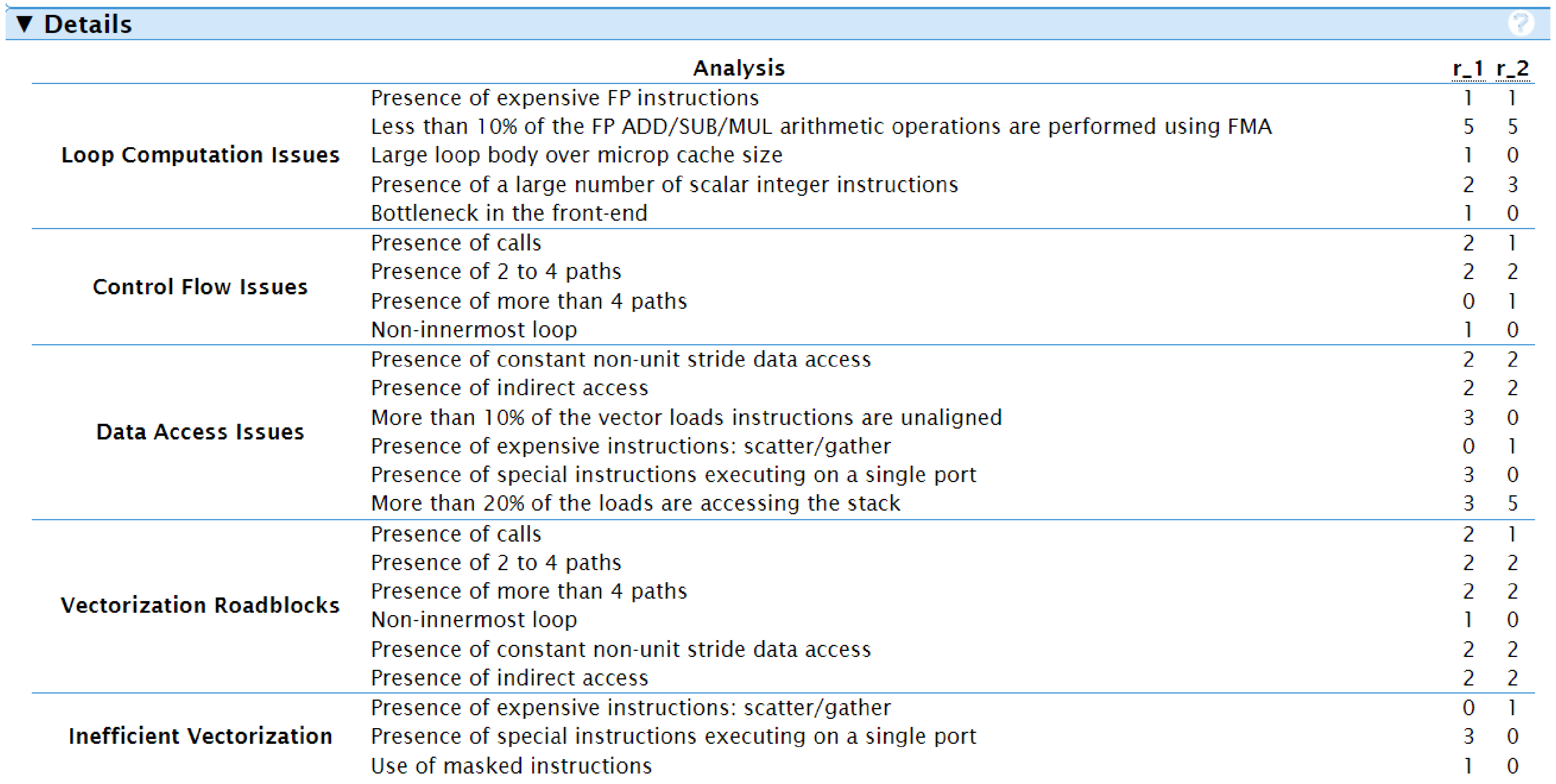
Single aggregation is performed: first for each compiler, various issues are aggregated across key loops (see below)
Double aggregation is also performed: issues are aggregated between compilers.



▼ Runs

- r_1 - engine_NEON1M11-0001_o2_m26_ifx - 10 analyzed loop(s)
  - Loop 18566 - engine_linux64_intel_ifx_impi
  - Loop 39475 - engine_linux64_intel_ifx_impi
  - Loop 255174 - engine_linux64_intel_ifx_impi
  - Loop 37916 - engine_linux64_intel_ifx_impi
  - Loop 256165 - engine_linux64_intel_ifx_impi
  - Loop 19710 - engine_linux64_intel_ifx_impi
  - Loop 38054 - engine_linux64_intel_ifx_impi
  - Loop 121792 - engine_linux64_intel_ifx_impi
  - Loop 19918 - engine_linux64_intel_ifx_impi
  - Loop 167135 - engine_linux64_intel_ifx_impi
- r_2 - engine_NEON1M11-0001_o2_m26_ifort - 10 analyzed loop(s)
  - Loop 15282 - engine_linux64_intel_impi
  - Loop 193162 - engine_linux64_intel_impi
  - Loop 30046 - engine_linux64_intel_impi
  - Loop 28971 - engine_linux64_intel_impi
  - Loop 97970 - engine_linux64_intel_impi
  - Loop 15758 - engine_linux64_intel_impi
  - Loop 98506 - engine_linux64_intel_impi
  - Loop 29120 - engine_linux64_intel_impi
  - Loop 97971 - engine_linux64_intel_impi
  - Loop 92421 - engine_linux64_intel_impi

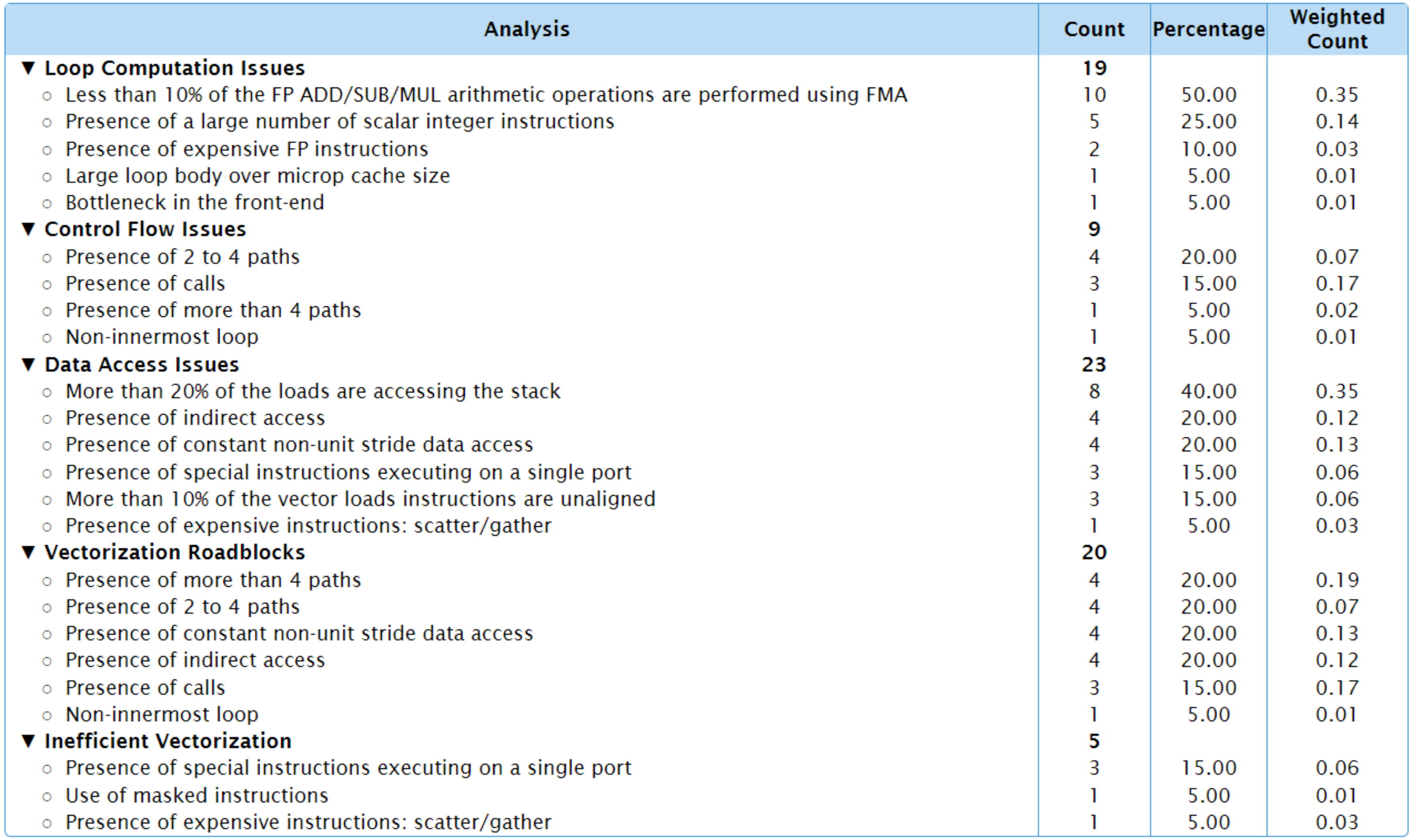
OpenRadioss: SINGLE AGGREGATION (2)



▼ Details

	Analysis	r_1	r_2
Loop Computation Issues	Presence of expensive FP instructions	1	1
	Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	5	5
	Large loop body over microp cache size	1	0
	Presence of a large number of scalar integer instructions	2	3
	Bottleneck in the front-end	1	0
Control Flow Issues	Presence of calls	2	1
	Presence of 2 to 4 paths	2	2
	Presence of more than 4 paths	0	1
	Non-innermost loop	1	0
Data Access Issues	Presence of constant non-unit stride data access	2	2
	Presence of indirect access	2	2
	More than 10% of the vector loads instructions are unaligned	3	0
	Presence of expensive instructions: scatter/gather	0	1
	Presence of special instructions executing on a single port	3	0
	More than 20% of the loads are accessing the stack	3	5
Vectorization Roadblocks	Presence of calls	2	1
	Presence of 2 to 4 paths	2	2
	Presence of more than 4 paths	2	2
	Non-innermost loop	1	0
	Presence of constant non-unit stride data access	2	2
	Presence of indirect access	2	2
Inefficient Vectorization	Presence of expensive instructions: scatter/gather	0	1
	Presence of special instructions executing on a single port	3	0
	Use of masked instructions	1	0

OpenRadioss: DOUBLE AGGREGATION (3)



Analysis	Count	Percentage	Weighted Count
▼ **Loop Computation Issues**	**19**		
○ Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	10	50.00	0.35
○ Presence of a large number of scalar integer instructions	5	25.00	0.14
○ Presence of expensive FP instructions	2	10.00	0.03
○ Large loop body over microp cache size	1	5.00	0.01
○ Bottleneck in the front-end	1	5.00	0.01
▼ **Control Flow Issues**	**9**		
○ Presence of 2 to 4 paths	4	20.00	0.07
○ Presence of calls	3	15.00	0.17
○ Presence of more than 4 paths	1	5.00	0.02
○ Non-innermost loop	1	5.00	0.01
▼ **Data Access Issues**	**23**		
○ More than 20% of the loads are accessing the stack	8	40.00	0.35
○ Presence of indirect access	4	20.00	0.12
○ Presence of constant non-unit stride data access	4	20.00	0.13
○ Presence of special instructions executing on a single port	3	15.00	0.06
○ More than 10% of the vector loads instructions are unaligned	3	15.00	0.06
○ Presence of expensive instructions: scatter/gather	1	5.00	0.03
▼ **Vectorization Roadblocks**	**20**		
○ Presence of more than 4 paths	4	20.00	0.19
○ Presence of 2 to 4 paths	4	20.00	0.07
○ Presence of constant non-unit stride data access	4	20.00	0.13
○ Presence of indirect access	4	20.00	0.12
○ Presence of calls	3	15.00	0.17
○ Non-innermost loop	1	5.00	0.01
▼ **Inefficient Vectorization**	**5**		
○ Presence of special instructions executing on a single port	3	15.00	0.06
○ Use of masked instructions	1	5.00	0.01
○ Presence of expensive instructions: scatter/gather	1	5.00	0.03

WRAP UP/CONCLUSIONS (1)

- Code quality generated by the compiler is of primary importance: this quality is highly dependent upon compiler and compiler options. Looking for the best compiler options can be extremely expensive in particular through brute force search
- Performing systematic exploration of compiler flags and compilers is mandatory but not enough.

We need to go further

WRAP UP/CONCLUSIONS (2)

We need to go further

- Performing a detailed assessment of code quality is therefore very important
- CQA/MAQAO/ONEVIEW (www.maqao.org) provides an efficient way of assessing code quality by
  - identifying compiler shortcomings/failures
  - Comparing between options and compilers
- These tools and methodology will be very useful for
  - Helping code developers finding the right compiler directives
  - Helping compiler developer improving/fixing their software

Website & resources

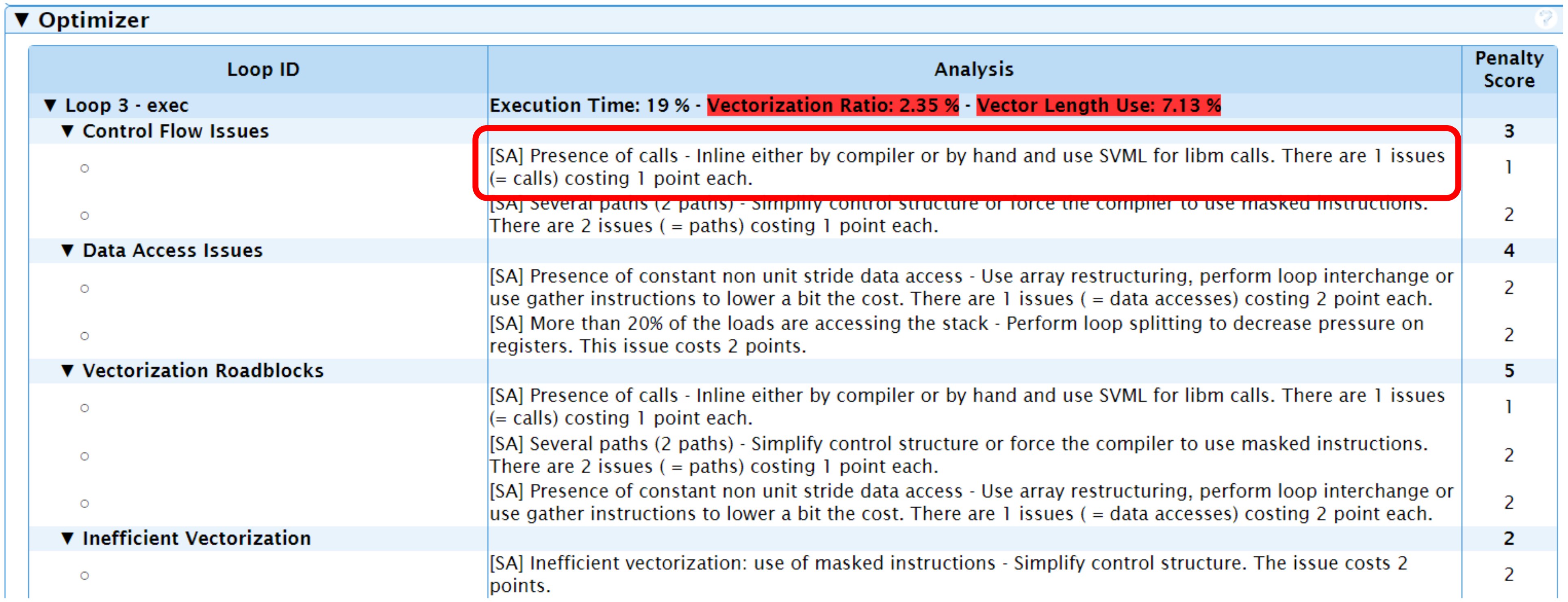
- MAQAO website: www.maqao.org
    - Mirror: maqao.exascale-computing.eu
- Documentation: www.maqao.org/documentation.html
    - Tutorials for ONE View, LProf and CQA
    - Lua API documentation
- Latest release: www.maqao.org/downloads.html
    - Binary releases (2-3 per year)
    - Core sources
- Publications: www.maqao.org/publications.html
- Email: contact@maqao.org
- Results: http://datafront.exascale-computing.eu/public/

THANKS FOR YOUR ATTENTION

Questions ?

BACKUP SLIDES

OPTIMIZATION SUMMARY HACC MK COMPILED WITH AOCC

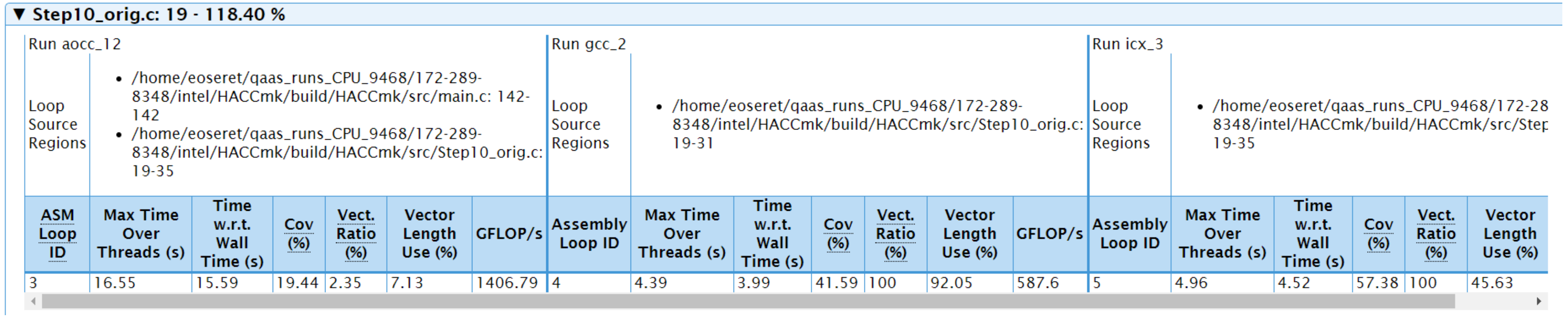


▼ Optimizer

Loop ID	Analysis	Penalty Score
▼ Loop 3 - exec	Execution Time: 19 % - Vectorization Ratio: 2.35 % - Vector Length Use: 7.13 %	
▼ Control Flow Issues		3
○	[SA] Presence of calls - Inline either by compiler or by hand and use SVML for libm calls. There are 1 issues (= calls) costing 1 point each.	1
○	[SA] Several paths (2 paths) - Simplify control structure or force the compiler to use masked instructions. There are 2 issues (= paths) costing 1 point each.	2
▼ Data Access Issues		4
○	[SA] Presence of constant non unit stride data access - Use array restructuring, perform loop interchange or use gather instructions to lower a bit the cost. There are 1 issues (= data accesses) costing 2 point each.	2
○	[SA] More than 20% of the loads are accessing the stack - Perform loop splitting to decrease pressure on registers. This issue costs 2 points.	2
▼ Vectorization Roadblocks		5
○	[SA] Presence of calls - Inline either by compiler or by hand and use SVML for libm calls. There are 1 issues (= calls) costing 1 point each.	1
○	[SA] Several paths (2 paths) - Simplify control structure or force the compiler to use masked instructions. There are 2 issues (= paths) costing 1 point each.	2
○	[SA] Presence of constant non unit stride data access - Use array restructuring, perform loop interchange or use gather instructions to lower a bit the cost. There are 1 issues (= data accesses) costing 2 point each.	2
▼ Inefficient Vectorization		2
○	[SA] Inefficient vectorization: use of masked instructions - Simplify control structure. The issue costs 2 points.	2

COMPARING LOOP ASSEMBLY CODES

Comparing AOCC, GCC and ICX

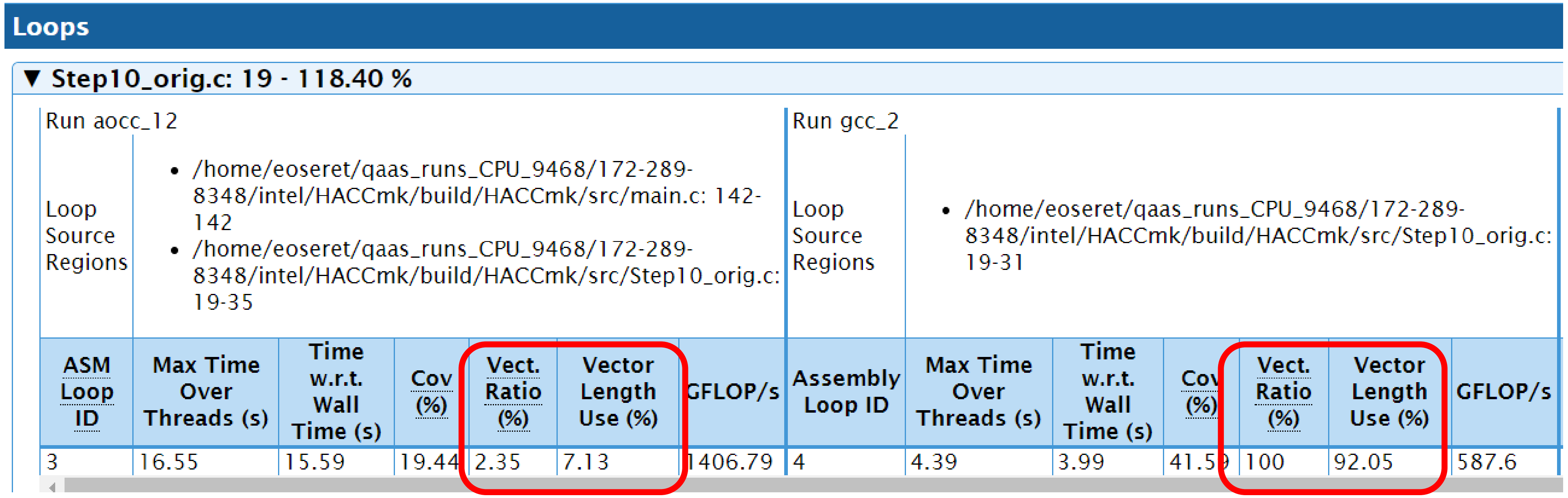


▼ Step10_orig.c: 19 - 118.40 %

Run aocc_12							Run gcc_2							Run icx_3					
Loop Source Regions	• /home/eoseret/qaas_runs_CPU_9468/172-289-8348/intel/HACCmk/build/HACCmk/src/main.c: 142-142 • /home/eoseret/qaas_runs_CPU_9468/172-289-8348/intel/HACCmk/build/HACCmk/src/Step10_orig.c: 19-35						Loop Source Regions	• /home/eoseret/qaas_runs_CPU_9468/172-289-8348/intel/HACCmk/build/HACCmk/src/Step10_orig.c: 19-31						Loop Source Regions	• /home/eoseret/qaas_runs_CPU_9468/172-28 8348/intel/HACCmk/build/HACCmk/src/Step 19-35				
ASM Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	GFLOP/s	Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	GFLOP/s	Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)
3	16.55	15.59	19.44	2.35	7.13	1406.79	4	4.39	3.99	41.59	100	92.05	587.6	5	4.96	4.52	57.38	100	45.63

LOOKING AT LOOP ASSEMBLY CODES (ZOOM)

Focussing on AOCC versus GCC



Loops

▼ Step10_orig.c: 19 - 118.40 %

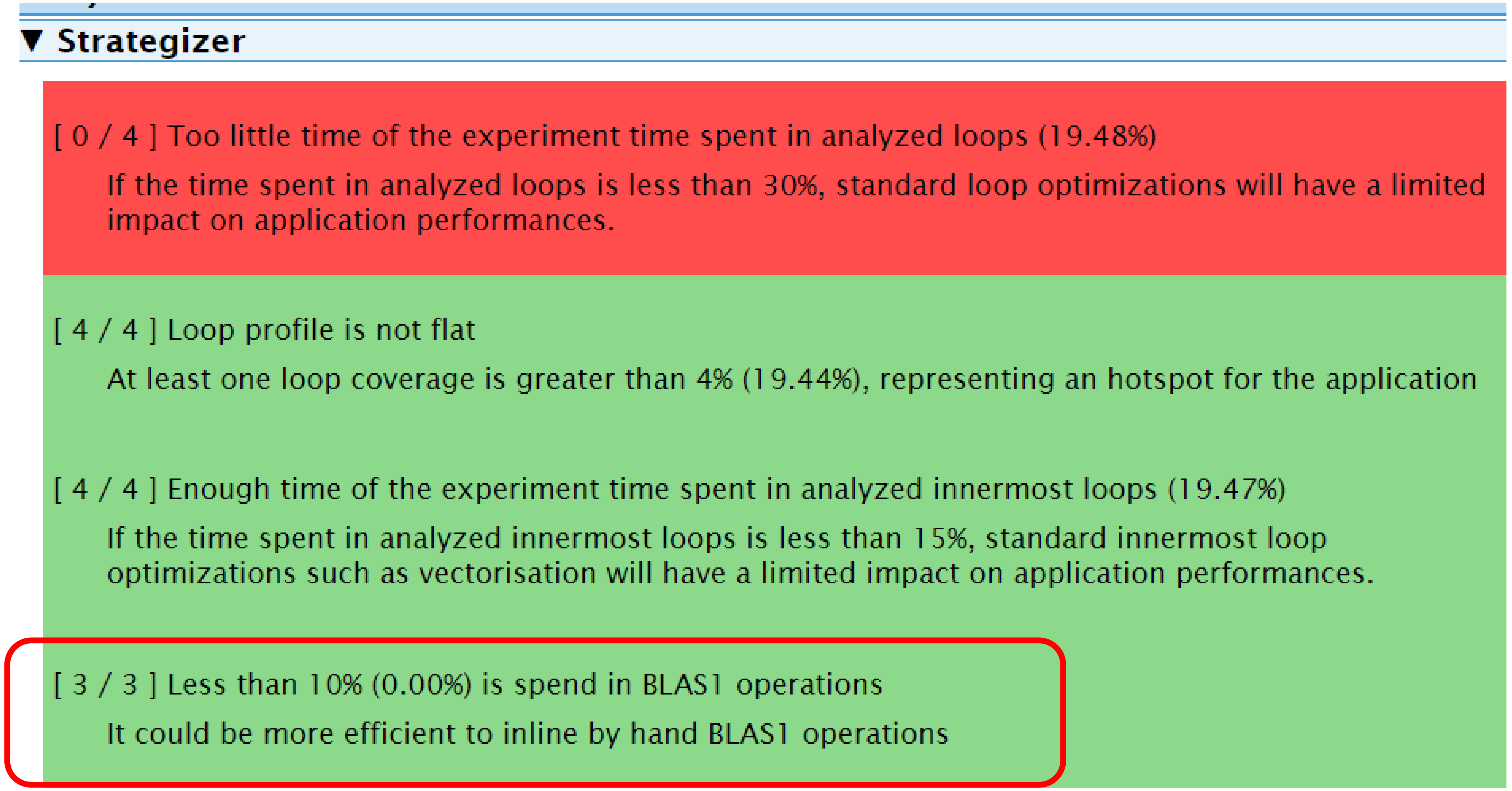
Run aocc_12	
Loop Source Regions	• /home/eoseret/qaas_runs_CPU_9468/172-289-8348/intel/HACCmk/build/HACCmk/src/main.c: 142-142 • /home/eoseret/qaas_runs_CPU_9468/172-289-8348/intel/HACCmk/build/HACCmk/src/Step10_orig.c: 19-35

Run gcc_2	
Loop Source Regions	• /home/eoseret/qaas_runs_CPU_9468/172-289-8348/intel/HACCmk/build/HACCmk/src/Step10_orig.c: 19-31

ASM Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	GFLOP/s	Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	GFLOP/s
3	16.55	15.59	19.44	2.35	7.13	1406.79	4	4.39	3.99	41.59	100	92.05	587.6

DIVING INTO LIBRARY USE (1)

Library use can be easily monitored and analyzed.



▼ Strategizer

[0 / 4] Too little time of the experiment time spent in analyzed loops (19.48%)

If the time spent in analyzed loops is less than 30%, standard loop optimizations will have a limited impact on application performances.

[4 / 4] Loop profile is not flat

At least one loop coverage is greater than 4% (19.44%), representing an hotspot for the application

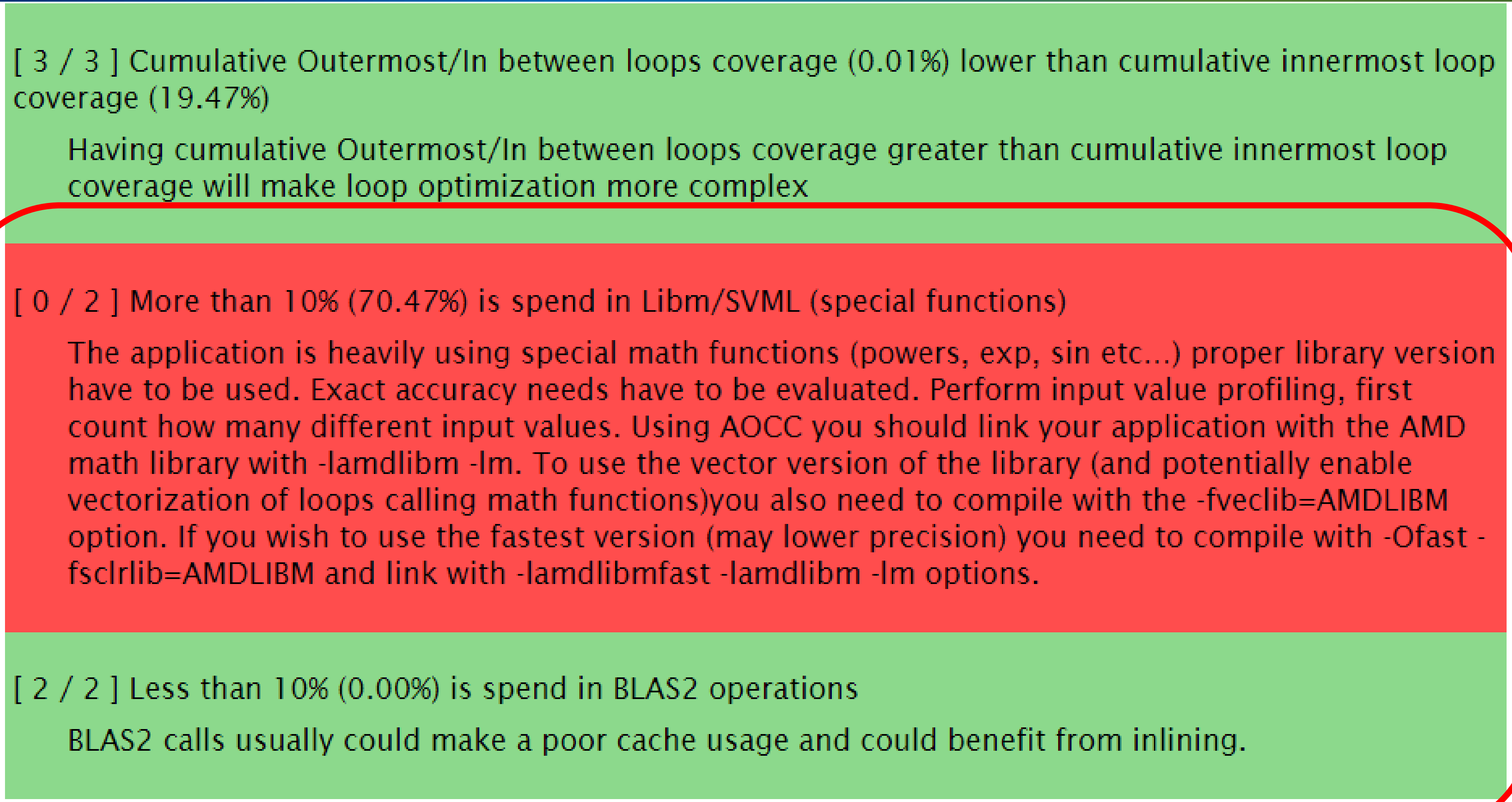
[4 / 4] Enough time of the experiment time spent in analyzed innermost loops (19.47%)

If the time spent in analyzed innermost loops is less than 15%, standard innermost loop optimizations such as vectorisation will have a limited impact on application performances.

[3 / 3] Less than 10% (0.00%) is spend in BLAS1 operations

It could be more efficient to inline by hand BLAS1 operations

DIVING INTO LIBRAY USE (2)



[3 / 3] Cumulative Outermost/In between loops coverage (0.01%) lower than cumulative innermost loop coverage (19.47%)

Having cumulative Outermost/In between loops coverage greater than cumulative innermost loop coverage will make loop optimization more complex

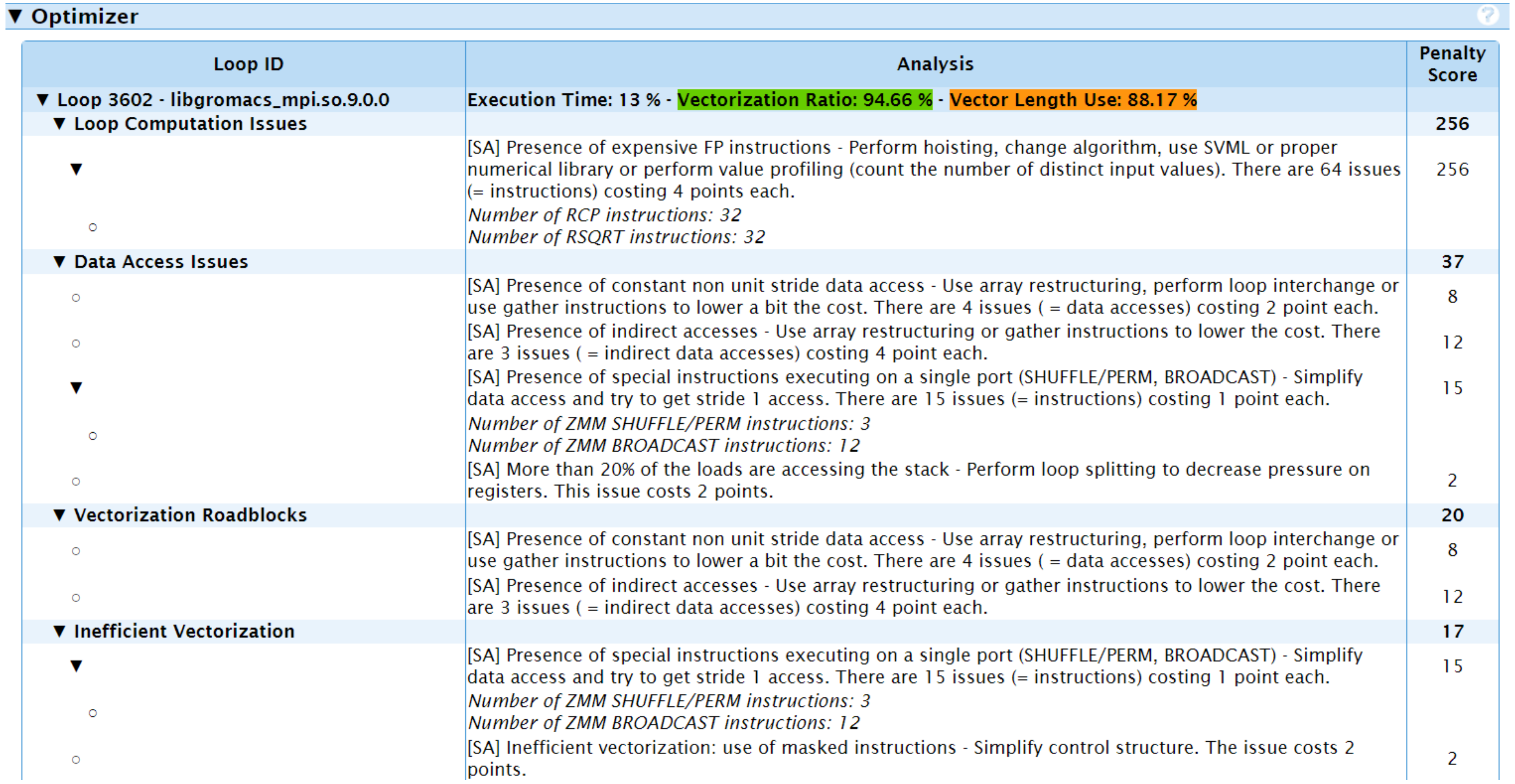
[0 / 2] More than 10% (70.47%) is spend in Libm/SVML (special functions)

The application is heavily using special math functions (powers, exp, sin etc...) proper library version have to be used. Exact accuracy needs have to be evaluated. Perform input value profiling, first count how many different input values. Using AOCC you should link your application with the AMD math library with -lamdlibm -lm. To use the vector version of the library (and potentially enable vectorization of loops calling math functions)you also need to compile with the -fveclib=AMDLIBM option. If you wish to use the fastest version (may lower precision) you need to compile with -Ofast -fsclrlib=AMDLIBM and link with -lamdlibmfast -lamdlibm -lm options.

[2 / 2] Less than 10% (0.00%) is spend in BLAS2 operations

BLAS2 calls usually could make a poor cache usage and could benefit from inlining.

DEALING WITH FULL APPLICATIONS: GROMACS



▼ Optimizer

Loop ID	Analysis	Penalty Score
▼ Loop 3602 - libgromacs_mpi.so.9.0.0	Execution Time: 13 % - Vectorization Ratio: 94.66 % - Vector Length Use: 88.17 %	
▼ Loop Computation Issues		**256**
▼	[SA] Presence of expensive FP instructions - Perform hoisting, change algorithm, use SVML or proper numerical library or perform value profiling (count the number of distinct input values). There are 64 issues (= instructions) costing 4 points each.	256
○	*Number of RCP instructions: 32* *Number of RSQRT instructions: 32*	
▼ Data Access Issues		**37**
○	[SA] Presence of constant non unit stride data access - Use array restructuring, perform loop interchange or use gather instructions to lower a bit the cost. There are 4 issues (= data accesses) costing 2 point each.	8
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 3 issues (= indirect data accesses) costing 4 point each.	12
▼	[SA] Presence of special instructions executing on a single port (SHUFFLE/PERM, BROADCAST) - Simplify data access and try to get stride 1 access. There are 15 issues (= instructions) costing 1 point each.	15
○	*Number of ZMM SHUFFLE/PERM instructions: 3* *Number of ZMM BROADCAST instructions: 12*	
○	[SA] More than 20% of the loads are accessing the stack - Perform loop splitting to decrease pressure on registers. This issue costs 2 points.	2
▼ Vectorization Roadblocks		**20**
○	[SA] Presence of constant non unit stride data access - Use array restructuring, perform loop interchange or use gather instructions to lower a bit the cost. There are 4 issues (= data accesses) costing 2 point each.	8
○	[SA] Presence of indirect accesses - Use array restructuring or gather instructions to lower the cost. There are 3 issues (= indirect data accesses) costing 4 point each.	12
▼ Inefficient Vectorization		**17**
▼	[SA] Presence of special instructions executing on a single port (SHUFFLE/PERM, BROADCAST) - Simplify data access and try to get stride 1 access. There are 15 issues (= instructions) costing 1 point each.	15
○	*Number of ZMM SHUFFLE/PERM instructions: 3* *Number of ZMM BROADCAST instructions: 12*	
○	[SA] Inefficient vectorization: use of masked instructions - Simplify control structure. The issue costs 2 points.	2

ONE View Reports Levels

ONE View ONE

- Requires a single run of the application
- Profiling of the application and static analysis on loop hotspots
- Allows to identify main bottlenecks, estimate complexity of resolution and expected associated speedup

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 perform detailed comparative analysis across different datasets, compilers, hardware platforms, runtimes, …

Stability mode

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

MAQAO Ecosystem

- Historical partnerships
  - CEA (French Department of Energy) Since 1990 and first MAQAO version on Itanium and long term partnership on application analysis, optimization and tools
  - ATOS: since 1990: compilers, performance tools and applications benchmarking and optimization
  - INTEL: since 2000: compilers, numerical libraries and performance tools
- Recent partnerships
  - AWS
  - SiPearl
- Current Projects
  - Exascale Computing Research (ECR): UVSQ, Intel (2005-2020) and CEA
  - EMOPASS (European Processor Initiative)
  - European Centers of Excellence : TREX, POP2, POP3
- Partner of the VI-HPS consortium
- Past projects: H4H, COLOC, PerfCloud, ELCI, MB3, ...

Performance Optimisation and Productivity 3

A Centre of Excellence in HPC

Contact:

- https://www.pop-coe.eu
- pop@bsc.es
- @POP_HPC
- youtube.com/POPHPC

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101143931. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, France, Portugal and the Czech Republic.