ASSESSING CPU CODE QUALITY

William Jalby (UVSQ)

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

Machine time donation: MEGWARE

OUTLINE

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

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

MOTIVATING EXAMPLE CONTEXT

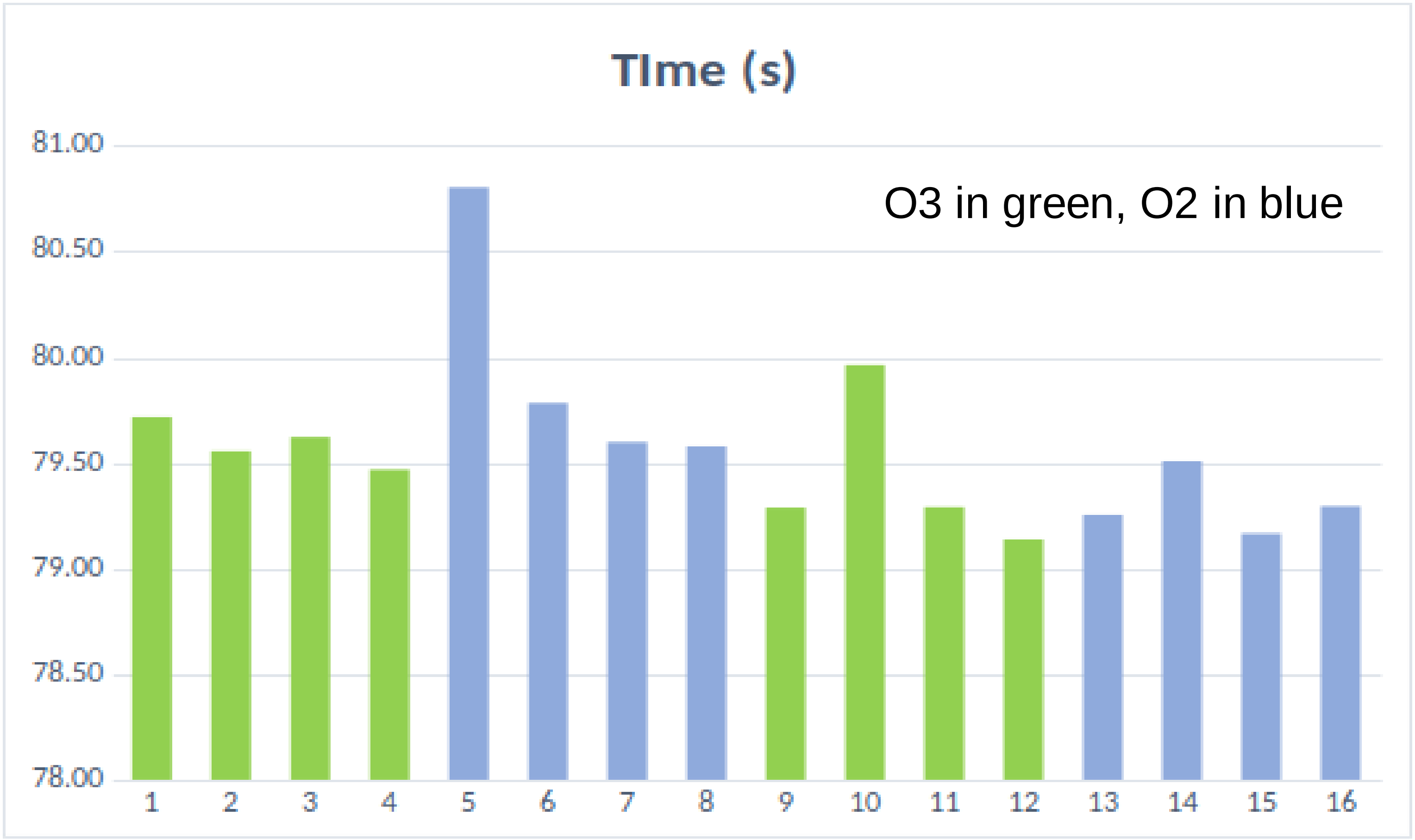
- TARGET CODE: HACC MK (Livermore: LLNL)
- Target hardware: AMD EPYC 9654 96-Core Processor (2 x 96 cores) provided by MEGWARE
- Three compilers:
  - AMD clang version 16.0.3 (CLANG: AOCC_4.1.0-Build#270 2023_07_10)
  - GNU C17 13.2.0 -march=znver4 ….
  - clang based Intel(R) oneAPI DPC++/C++ Compiler 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

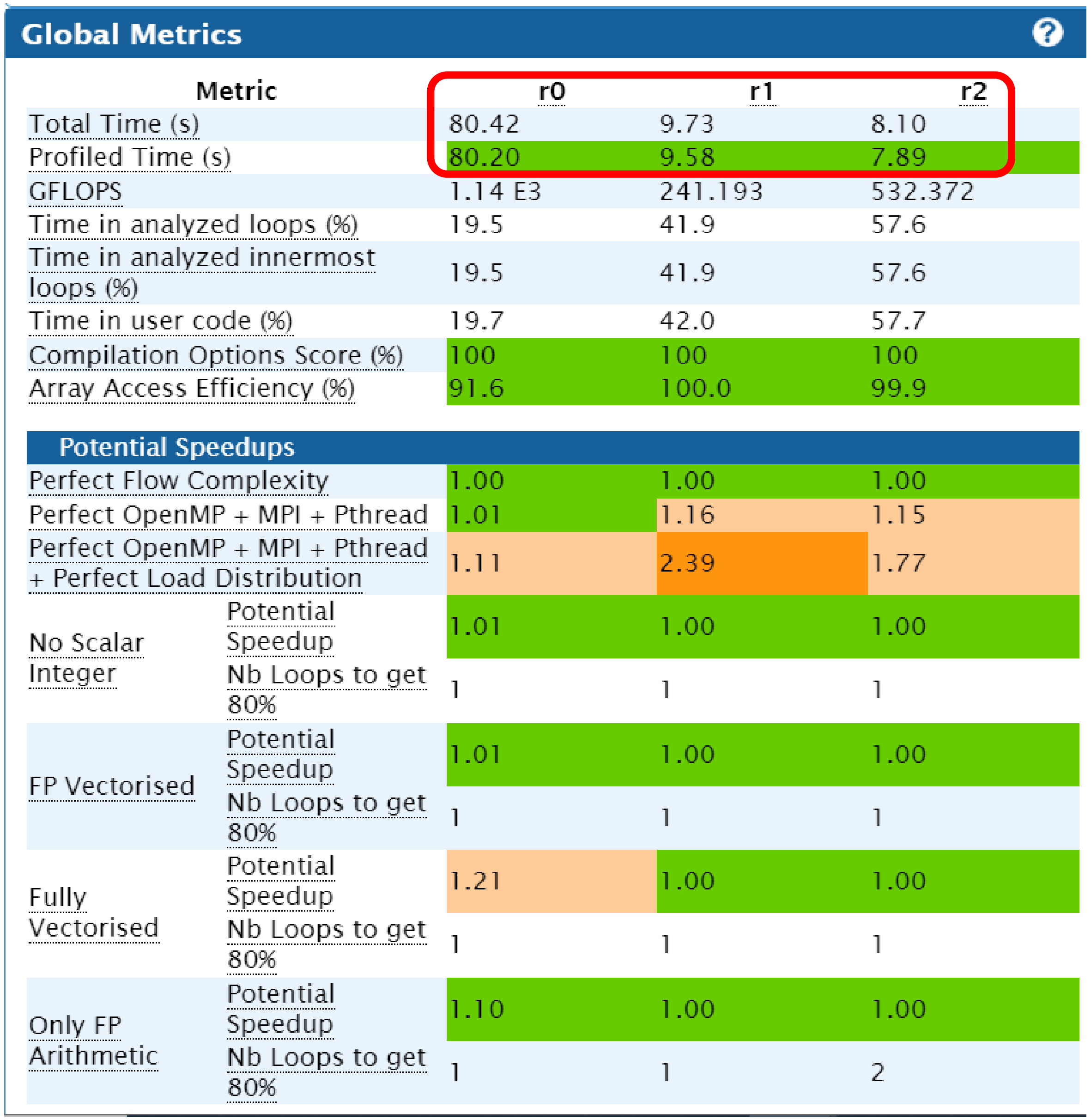
16 "standard" compiler options with different optimization levels (O2 and O3), different vector lengths, ….
Orthogonal choice

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



PERFORMANCE RESULTS FOR MULTIPLE COMPILERS



Global Metrics

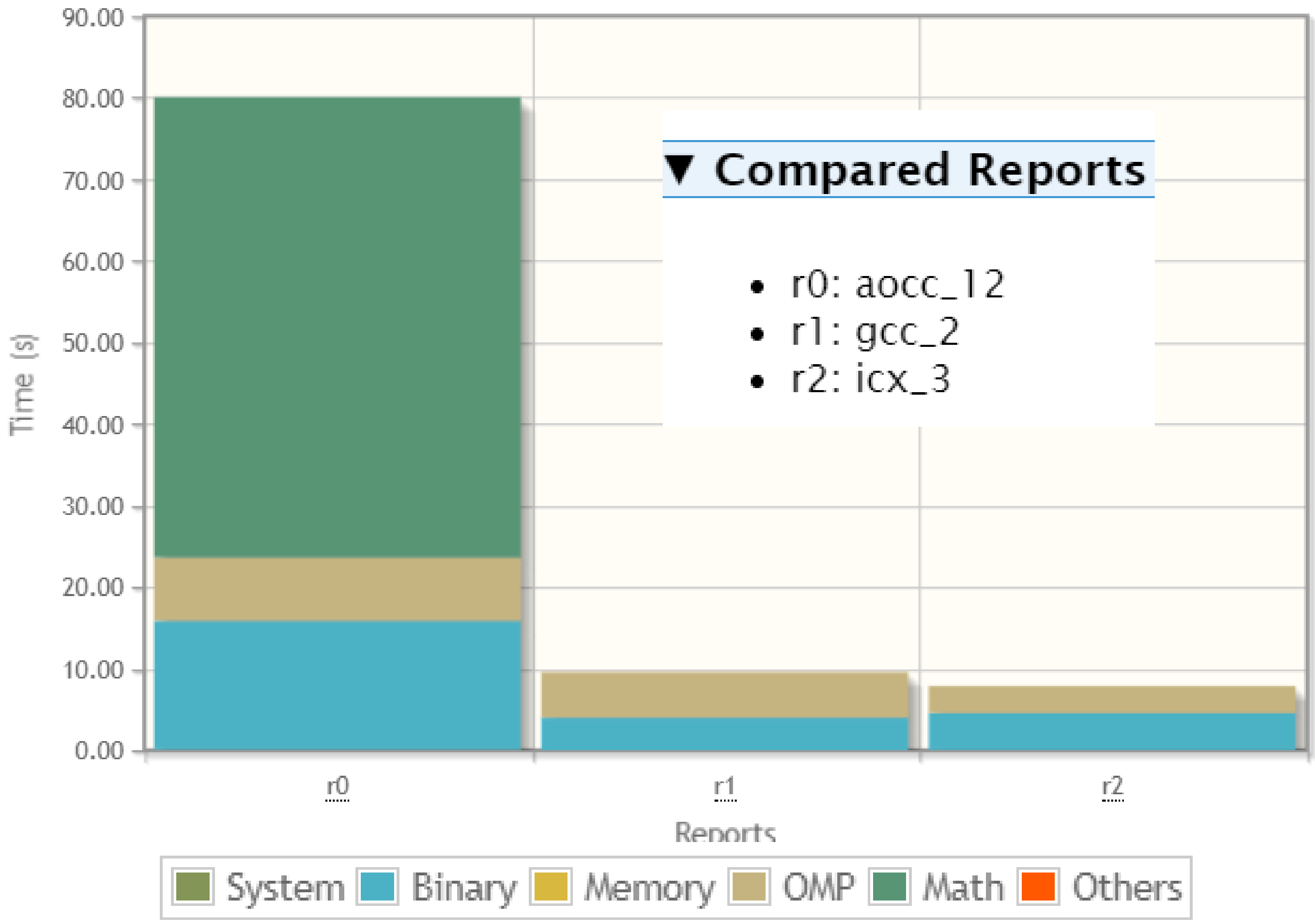
Metric		r0	r1	r2
Total Time (s)		80.42	9.73	8.10
Profiled Time (s)		80.20	9.58	7.89
GFLOPS		1.14 E3	241.193	532.372
Time in analyzed loops (%)		19.5	41.9	57.6
Time in analyzed innermost loops (%)		19.5	41.9	57.6
Time in user code (%)		19.7	42.0	57.7
Compilation Options Score (%)		100	100	100
Array Access Efficiency (%)		91.6	100.0	99.9
Potential Speedups				
Perfect Flow Complexity		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread		1.01	1.16	1.15
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.11	2.39	1.77
No Scalar Integer	Potential Speedup	1.01	1.00	1.00
	Nb Loops to get 80%	1	1	1
FP Vectorised	Potential Speedup	1.01	1.00	1.00
	Nb Loops to get 80%	1	1	1
Fully Vectorised	Potential Speedup	1.21	1.00	1.00
	Nb Loops to get 80%	1	1	1
Only FP Arithmetic	Potential Speedup	1.10	1.00	1.00
	Nb Loops to get 80%	1	1	2

▼ Compared Reports

- r0: aocc_12
- r1: gcc_2
- r2: icx_3

AOCC 10x slower than ICX !!

PERFORMANCE RESULTS FOR MULTIPLE COMPILERS



▼ Compared Reports

- r0: aocc_12
- r1: gcc_2
- r2: icx_3

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 and GCC did not show any time spent in math library, try to suppress math library use (more details will be given later).

This time it worked, performance very similar to GCC 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 (1)

Evaluate current compiler capabilities:

- Starting with ASM produced.
- 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
- We will take into account both compiler mistakes but also source code issues.

ANALYZING CODE QUALITY

Focus on loops: innermost/in between/outermost

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

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.

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)

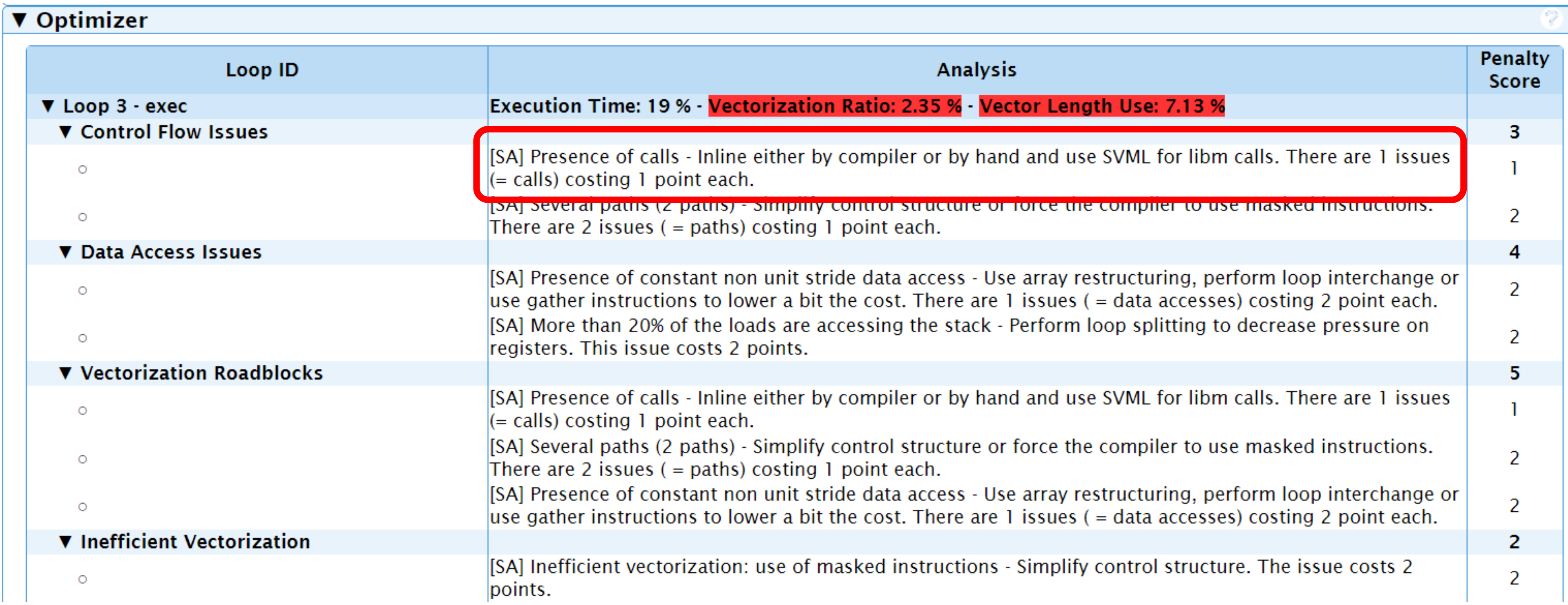
ANALYZING CODE QUALITY (3)

Dealing with more complex issues:

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

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

ANALYZING CODE QUALITY (4)

Dealing with multiple compiler outputs on the same code.

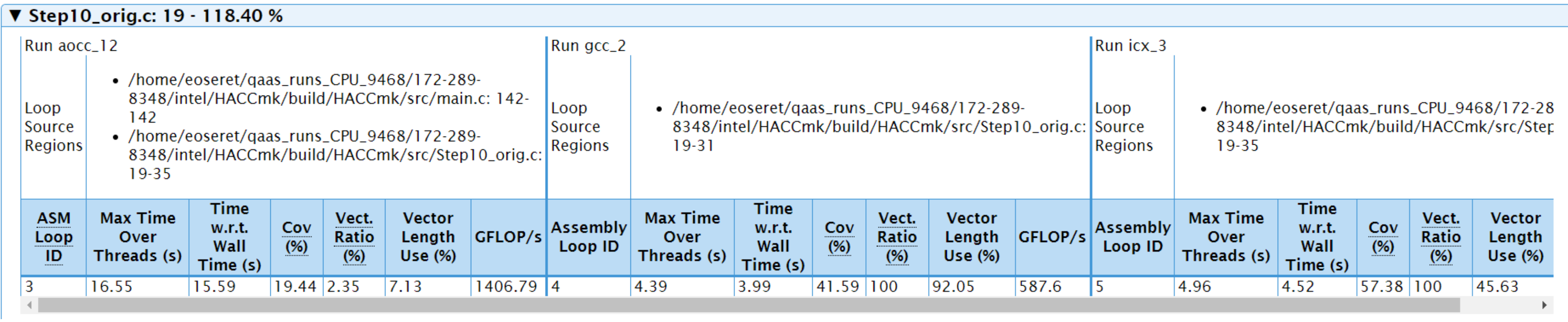
Very similar to detect multiple code versions.

- For each compiler connect ASM code and source code
  - Relying on compiler info –g option allows to establish link between binary and source code.
- Dealing with multiple compiler outputs
  - 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 the same source code

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

COMPARING LOOP ASSEMBLY CODES

Comparing AOCC, GCC and ICX

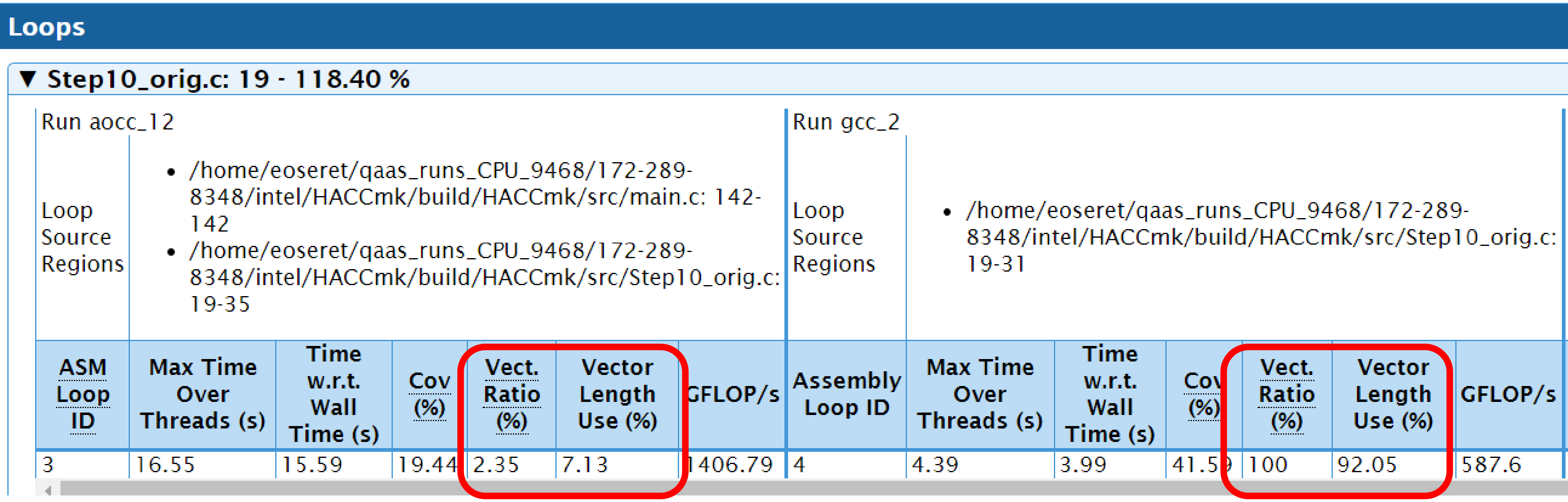


▼ Step10_orig.c: 19 - 118.40 %

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



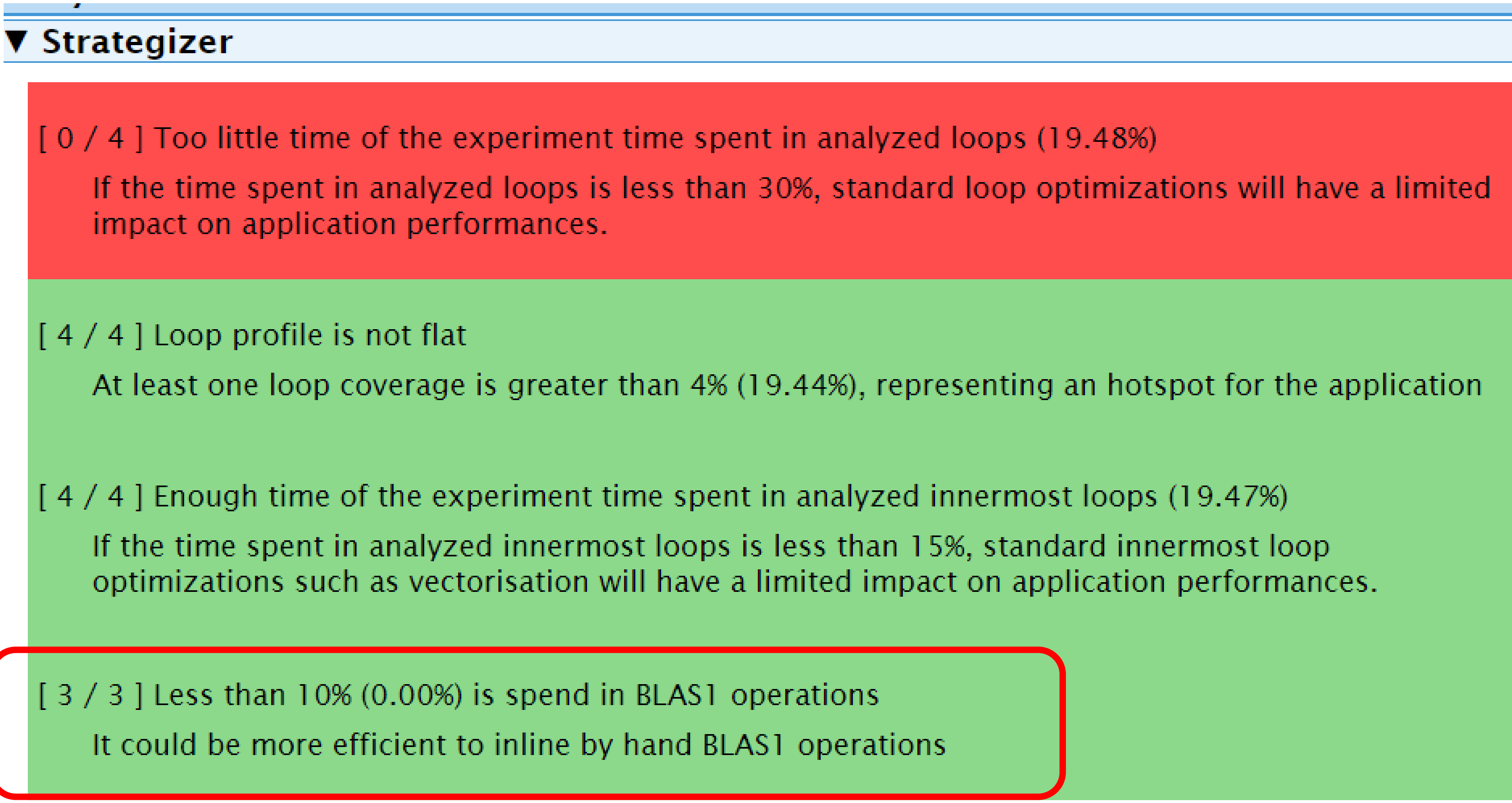
Loops

▼ **Step10_orig.c: 19 - 118.40 %**

Run aocc_12							Run gcc_2						
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					
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	41.5	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

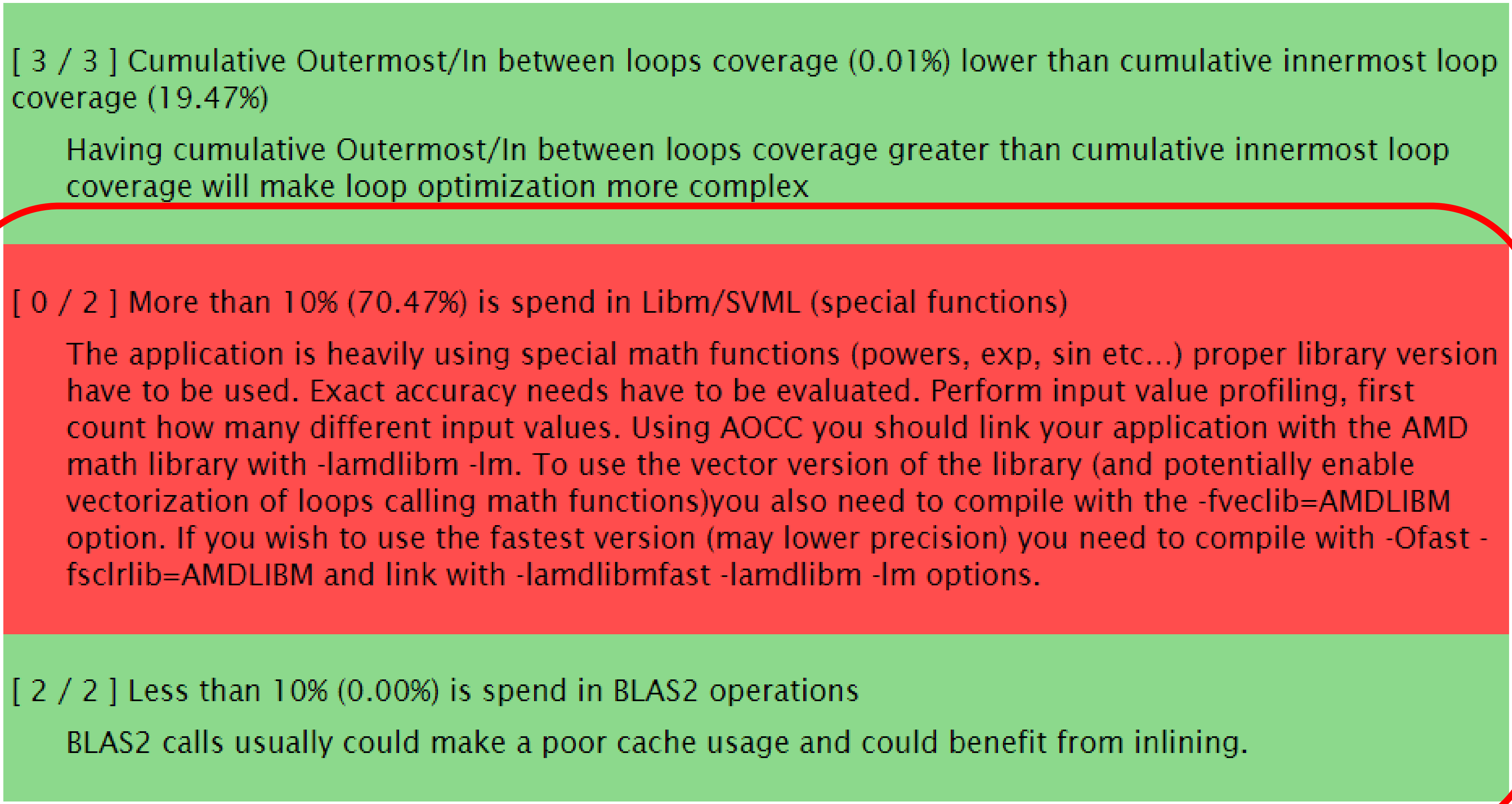
[4 / 4] Enough time of the experiment time spent in analyzed innermost loops (19.47%)

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

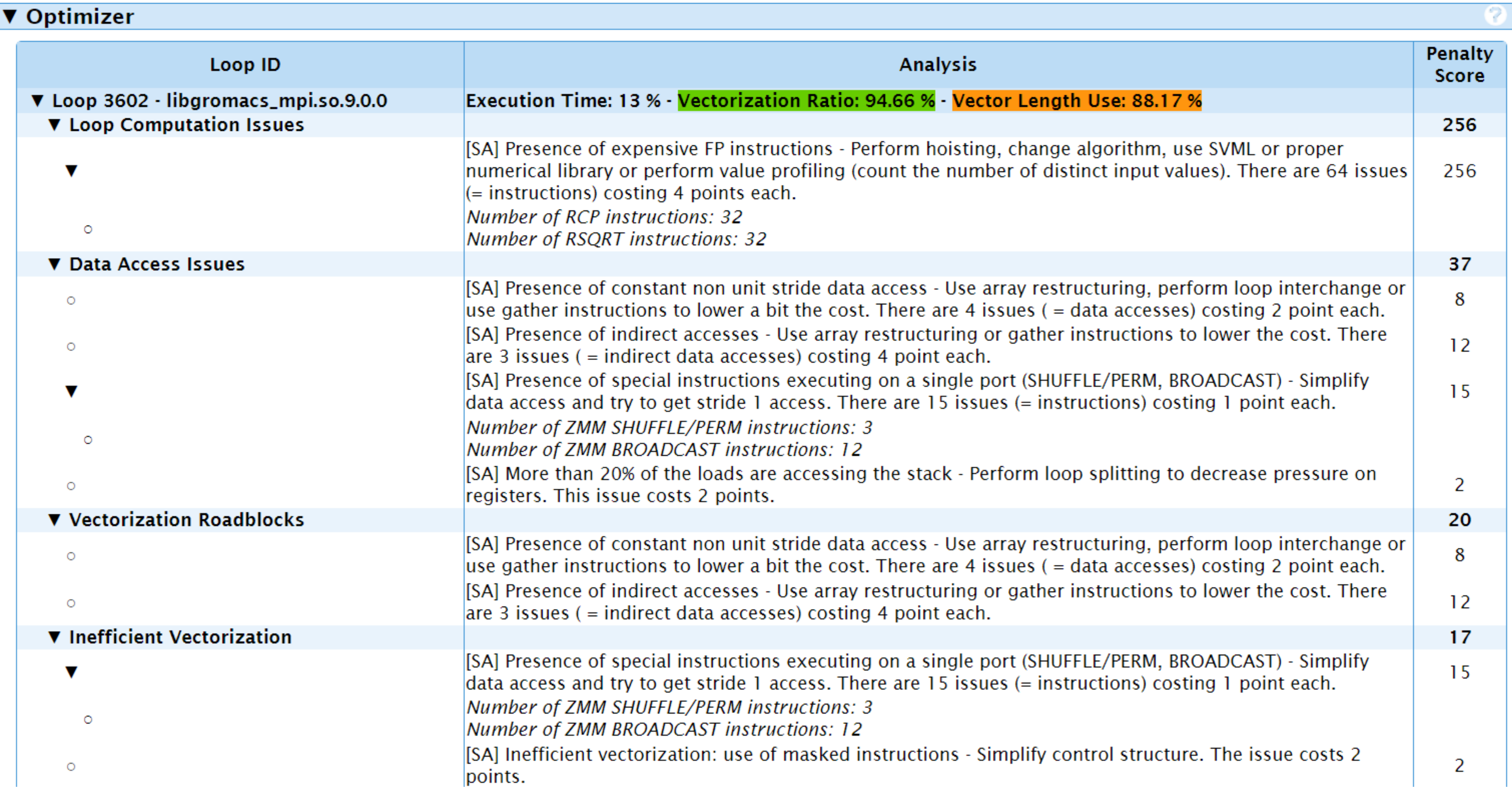
[0 / 2] More than 10% (70.47%) is spend in Libm/SVML (special functions)

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

IMPROVING COMPILERS

Gather all of previous metrics from the Optimization summaries and provides them :

- to code developers for optimization
- to compiler developers to fix weaknesses

IMPORTANT ISSUE : cannot compare directly compiler optimization reports (lack of standards)

COMPARING AND IMPROVING COMPILERS: detect good features of a compiler and reinject them in another one.

WRAP UP/CONCLUSIONS

- Code quality generated by the compiler is of primary importance: this quality highly dependent upon compiler and compiler options. Looking for the best compiler options can be extremely expensive
- Assessing 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
  - Helping code developers finding the right compiler directives
  - Helping compiler developer improving/fixing their software
- Moving to GPU ? Needs access to documentation on ASM
  - Will start with AMD

THANKS FOR YOUR ATTENTION

Questions ?

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**
- 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

➔ What You Analyse Is What You Run

SITE: www.maqao.org

MAQAO Ecosystem

- Historical partnerships
  - CEA (French Department of Energy) Since 1990 and first MAQAO version on Itanium and long term partneship on application analysis and optimization and on tools
  - ATOS: since 1990: compilers, performance tools and applications benchmarking and optimization
- 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, POP3
- Partner of the VI-HPS consortium
- Past projects: H4H, COLOC, PerfCloud, ELCI, MB3