

Assessing CPU Code Quality Compiler Comparison

Hugo Bolloré, Cédric Valensi, Emmanuel Oseret, Mathieu Tribalat,
William Jalby

Université de Versailles Saint-Quentin en Yvelines

Machine time donation: MEGWARE, CALMIP, IT4I, AWS

Outline

- Motivating example
- Identifying potential compiler issues
- Case Study
- Conclusion

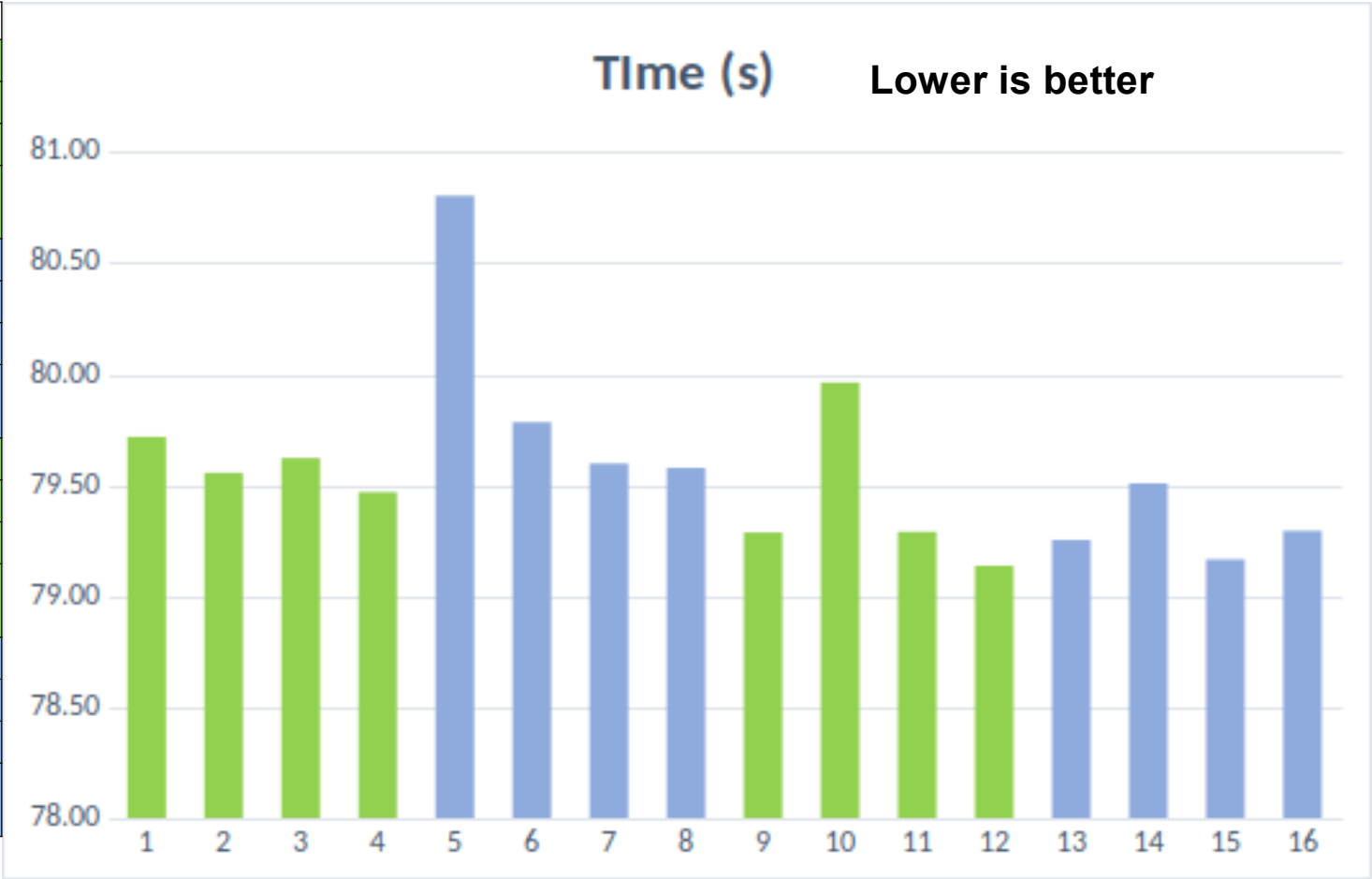
MOTIVATING EXAMPLE

- TARGET CODE: HACC MK (Livermore: LLNL)
- Target hardware: AMD EPYC 9654 96-Core Processor (2 x 96 cores)
 - Provided by MEGWARE
- Three compilers:
 - **AOCC**: AMD clang version 16.0.3
 - **GCC**: GNU C17 13.2.0
 - **ICX**: Intel(R) oneAPI DPC++/C++ Compiler 2024.0.0 (clang based)
- For each compiler 16 flags were tested.
- Systematic test/benchmarking effort carried out in QaaS project (Quality as a Service)

Performance results for AOCC

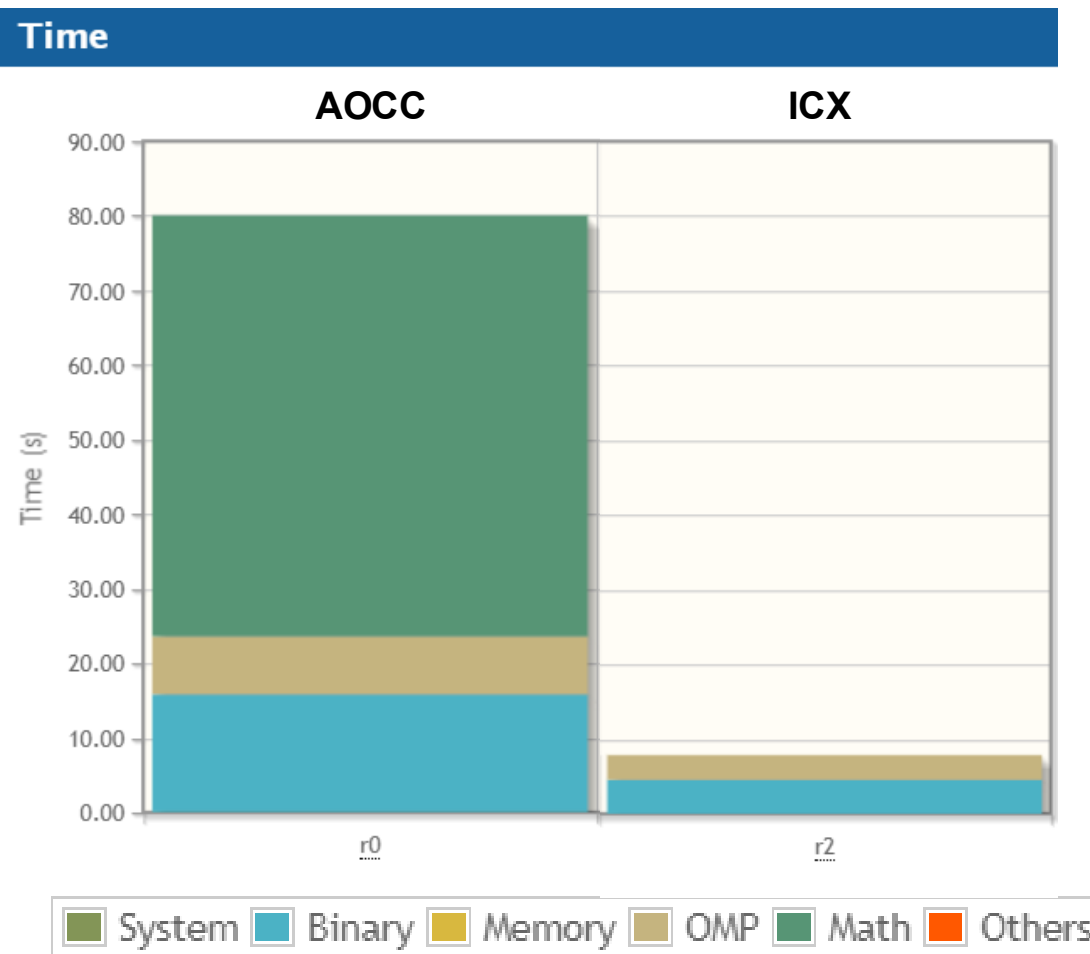
option	flags
1	O3 -march=znver4
2	O3 -march=znver4 -mprefer-vector-width=512
3	O3 -march=znver4 -mprefer-vector-width=256
4	O3 -march=znver4 -fno-vectorize -fno-slp-vectorize -fno-openmp-simd
5	O2 -march=znver4
6	O2 -march=znver4 -mprefer-vector-width=512
7	O2 -march=znver4 -mprefer-vector-width=256
8	O2 -march=znver4 -fno-vectorize -fno-slp-vectorize -fno-openmp-simd
9	O3 -march=znver4 -flto
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
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

O3 in green, O2 in blue



➔ 2,5% difference between best and worst case for the whole application execution time

Performance results for multiple compilers



- **AOCC 10x slower than ICX**
- Main difference due to Math library

- PROBLEMS WITH AOCC
 - Main 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: Forcing libalm use → **unfortunately no performance gain.**
 - SECOND FIX: since ICX did not show any time spent in math library, trying to suppress math library use by adding `fastmath` flag to AOCC to 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 backport some optimizations from the best performing compilers (for example compiler flags or compiler directives).
- **LESSON 3: THE REAL ONE:** try to analyse and detect compiler shortcomings systematically.
→ We need tools for that...

IDENTIFYING POTENTIAL COMPILER ISSUES

Analysing compiler generated code: goals

- Evaluating **code quality**: a set of key code characteristics with a large impact on performance
- Targets:
 - **Code developer**: Improve application performance and performance portability
 - **Compiler developer**: Improve compiler quality and assist in platform migration
- Analyses:
 - Impact of **compiler versions**: provides a better answer than a simple GO/NOGO
 - Impact of **compiler switches**: identify best compiler switches
 - Differences in compiler behaviour across different compilers
 - Perform **performance portability** analysis through global comparison across multiple compilers.
- Outcomes:
 - **Select** the best compiler for the application
 - **Import optimization** from one compiler to another (e.g. inserting vectorisation directives)
 - Facilitate application migration and porting by identifying **compilers strengths and weaknesses**
 - Help developers and/or users to fully exploit system capabilities

- 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).

- Timing analysis requires runs on the same machine: otherwise, comparison is impacted by architecture differences
- Timings are measured at the binary level and therefore are associated with binary code fragments. Main difficulty in comparison is matching functions and loops.
- The “Proof is in eating the cake” does not tell you anything about the cook or the recipe: we need to understand why there is a timing difference.
- Other metrics such as stalls, cache level access rate, ... will provide additional interesting info but will suffer from the same problem as timings.
- Compiler reports provide useful info about the code generation process. However, these reports:
 - Don't give, in general, details on failures/shortcomings of the compilation process
 - Are not standardized therefore making extremely difficult to compare compilers.

Analyzing code quality

- Focus on loops: single/innermost/in between/outermost
- Focus on assembly code: compiler shortcomings but also source code issues will both be taken into account.
- Using our performance analysis framework MAQAO (www.maqao.org)
 - Modular framework operating at binary level
- Static analysis (SA)
 - Evaluating assembly code quality using MAQAO module CQA (Code Quality Analysis).
 - Generic topics of interest: Port / FU usage, vectorization & vectorization roadblocks, instruction set use, data access, ...
 - Generating performance estimations necessary for comparing ASM versions
- Dynamic analysis (DA)
 - Sampling profiling using MAQAO module LProf to assess loop relative cost.
 - Value profiling and decremental analysis based on binary instrumentation

Performance issues categories

- **Loop computation:** issues related to the computation organization.
- **Control flow:** issues relevant to control
- **Data access:** issues essentially related to memory operations
- **Vectorization roadblocks:** issues preventing vectorization
- **Inefficient vectorization:** issues related to vectorization quality
- Some issues can be present into multiple categories

Performance issues (1/3)

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

Performance issues (2/3)

ISSUES	Category
Presence of calls (SA)	Control flow, Vectorisation roadblock
Presence of 2 to 4 paths (SA)	Control flow, Vectorisation roadblock
Presence of more than 4 paths (SA)	Control flow, Vectorisation roadblock
Non-innermost loop (SA)	Control flow, Vectorisation roadblock
Out of user code (SA)	Vectorisation roadblock
Presence of constant non unit stride data access (SA)	Vectorisation roadblock, Data access
Presence of indirect access (SA)	Vectorisation roadblock, Data access
High secondary DTLB miss rate (DA)	Data access
More than 10% of the vector loads instructions are unaligned (SA)	Data access
More than 20% of the loads are accessing the stack (SA).	Data access
Ratio ORIG/DL1 greater than 3x (DA)	Data access

Performance issues (3/3)

ISSUES	Category
Partial or non-existent vectorization (SA)	Vectorisation efficiency
Presence of expensive instructions : scatter/gather (SA)	Vectorisation efficiency, Data access
Presence of special instructions executing on a single port (SA): typically all data restructuring instructions, expand, pack, unpack, etc...	Vectorisation efficiency, Data access
Use of shorter than available vector length (SA)	Vectorisation efficiency
Use of masked instructions (SA)	Vectorisation efficiency
Time spent in peel/tail loop greater than time spent in main loop (DA)	Vectorisation efficiency

SPMXV: Sparse Matrix Vector operations

CASE STUDY

Spmxv: a very simple code

- 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 count (typically less than 15)
 - Regular sweep on Array A and indirect access on Array x

```
#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;
}
```

Spmxv tests : hard / soft configuration

- **HARDWARE CONFIGURATION**
 - Skylake : Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz
 - 2 x 26 Cores
 - 2.1 GHz
- **SOFTWARE CONFIGURATION**
 - OPENMP parallel: 52 threads
 - GCC 14.2.1
 - ICX 2024.2.1
 - Compiler Options:
 - Architecture specialisation (-march=skylake-avx512 / -march=native)
 - Optimisation (-O3 / -Ofast, ...)
 - OpenMP (-fiopenmp / -fopenmp)

Innermost loop comparison

▼ main.cpp: 201 - 148.57 %											
Run gcc_o3_ov1_o52/ GCC						Run icx_o3_ov1_o52/ ICX					
Loop Source Regions		• /home/kcamus/POP/POP2_miniapp/spmxv/epi-spmxv-main/main.cpp: 201-203				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 (%)	Assembly 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	16	20.96	21.25	50.97	100	35
						15	6.69	6.79	16.29	0	12.5
Sum on 1 analyzed binary loop						Sum on 2 analyzed binary loops (spmxv.exe - 16, spmxv.exe - 15)					
Analysis					Count	Analysis					Count
Loop Computation Issues						Loop Computation Issues					
Low iteration count						Low iteration count					1
Control Flow Issues						Control Flow Issues					
Low iteration count						Low iteration count					1
Data Access Issues						Data Access Issues					
Presence of indirect access					1	Presence of indirect access					1
Vectorization Roadblocks						Vectorization Roadblocks					
Presence of indirect access					1	Presence of indirect access					1
► <unknown>: 0 - 0.01 %											

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

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

ICX detailed loop analysis

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 %	

- Single aggregation: for each compiler, issues are aggregated across key loops
- Double aggregation: 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: results aggregation

- 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

CONCLUSION

Wrap up/conclusion

- Code quality is a set of key code characteristics with a large impact on performance
 - 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.
 - ➔ Performing a detailed assessment of code quality is therefore very important
- MAQAO 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

Questions ?

THANKS FOR YOUR ATTENTION

BACKUP SLIDES

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	

Resolving function/loop matching

- 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 functions, going through source code is easier and more efficient because in general there are not multiple versions of the same function

Spmxv compare: gcc versus icx

▼ Compared Reports

- r0: gcc_o3_ov1_o52/
- r1: icx_o3_ov1_o52/

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

→ gcc faster than icx

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
FP Vectorised	Potential Speedup 2.14	1.48
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup 3.92	2.84
	Nb Loops to get 80%	1
Only FP Arithmetic	Potential Speedup 1.02	1.15
	Nb Loops to get 80%	1

→ But icx global vectorisation is better than gcc

Spmxv: 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.

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: double aggregation

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

MAQAO: Modular Assembly Quality Analyzer and Optimizer



- Objectives:
 - Characterizing performance of HPC applications
 - Guiding users through optimization process
 - Estimating optimization 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
 - **Modular** tool offering complementary views
- Operating principle: Analysis at Binary Level
 - Compiler optimizations increase the distance between the executed code and the source code
 - ➔ What You Analyse Is What You Run
- High level reports
 - Checking analysis validity
 - Detecting performance issues
 - Assessing optimisation complexity
 - Code quality metrics at global and loop level
 - Hints for improving performance with associated expected speedup

Website & resources

- MAQAO website: <https://www.maqao.org/>
 - Mirrors: maqao.exascale-computing.eu, maqao.liparad.uvsq.fr
- 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: datafront.maqao.org/public/