The MAQAO performance analysis and optimization framework

Subtitle:

Yet another Performance Evaluation Tool: what does MAQAO bring to the table

C. Valensi, H. Bollore, E. Oseret, M. Tribalat, T. Dionisi, L. Neto, K. Camus

W. Jalby

UVSQ/UPSaclay

TALK ORGANIZATION

- Some motivation
- MAQAO/ONEVIEW principles
- A quick tour of MAQAO OV Functionalities
- Focus on two distinctive MAQAO functionnlaities
- Wrapup

The **Standard Performance Evaluation Corporation (SPEC)** is an American non-profit organization that aims to "produce, establish, maintain and endorse a standardized set" of performance benchmarks for computers (Wikipedia)

SPEC was founded in 1988. SPEC benchmarks are widely used to evaluate the performance of computer systems; the test results are published on the SPEC website (www.spec.org).

Members are hardware/software companies, a few Universities and research center

GOAL: provide an unbiased way of comparing systems performance using a precise protocol and several reference programs representative of “real” workload. Public database available

REPRESENTATIVE FOR A USER: OK if his favorite program is included in the reference list, otherwise representativeness can be argued for ever.

A CLOSE LOOK AT A DECADE (2007 - 2016)

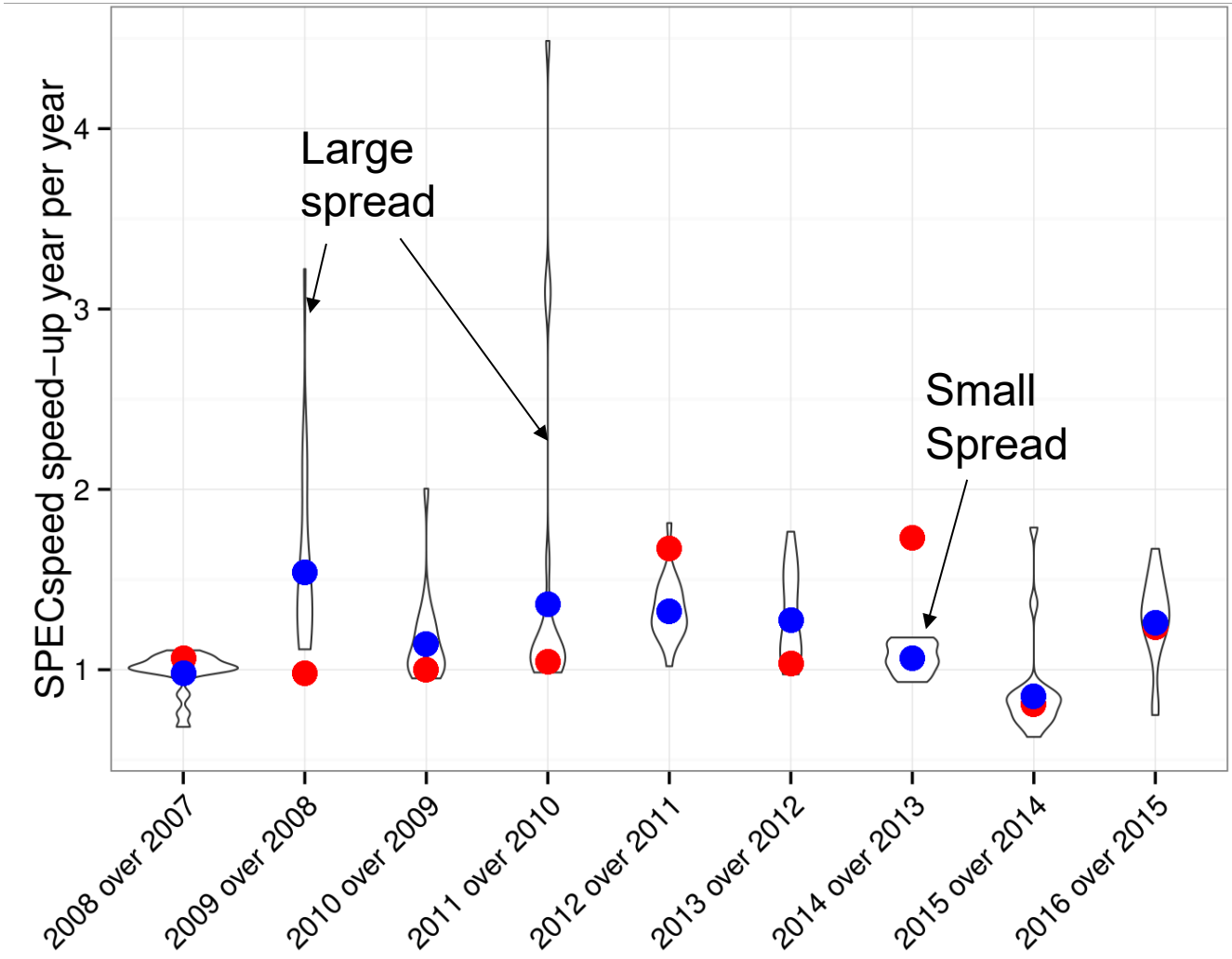
	System	CPU	Year	Core DP (Gflops)	Freq (GHz)	# Cores	L3 (MB)
	Harpertown (45 nm)	X5482	2007	12.80	3.20	8	0
	Harpertown (45 nm)	X5492	2008	13.60	3.40	8	0
	Gainestown (45 nm)	W5590	2009	13,32	3,33	8	8
	Westmere-EP (32 nm)	X5680	2010	13,32	3,33	12	12
	Westmere-EP (32 nm)	X5690	2011	13,88	3,47	12	12
AVX introduction →	Sandy Bridge-EP (32 nm)	E5-2690	2012	23.20	2.90	16	20
	Ivy Bridge-EP (22 nm)	E5-2690 v2	2013	24.00	3.00	20	25
FMA Introduction →	Haswell-EP (22 nm)	E5-2690 v3	2014	41.60	2.60	24	30
	Haswell-EP (22 nm)	E5-4650	2015	33.60	2.10	48	30
	Broadwell-EP (14 nm)	E5-2690 v4	2016	41.60	2.60	28	35

- 10 INTEL reference architectures from 2007 to 2016
- AVX then FMA (Fused Multiply Add) introductions

4

Performance Progress from 2007 to 2016 (1)

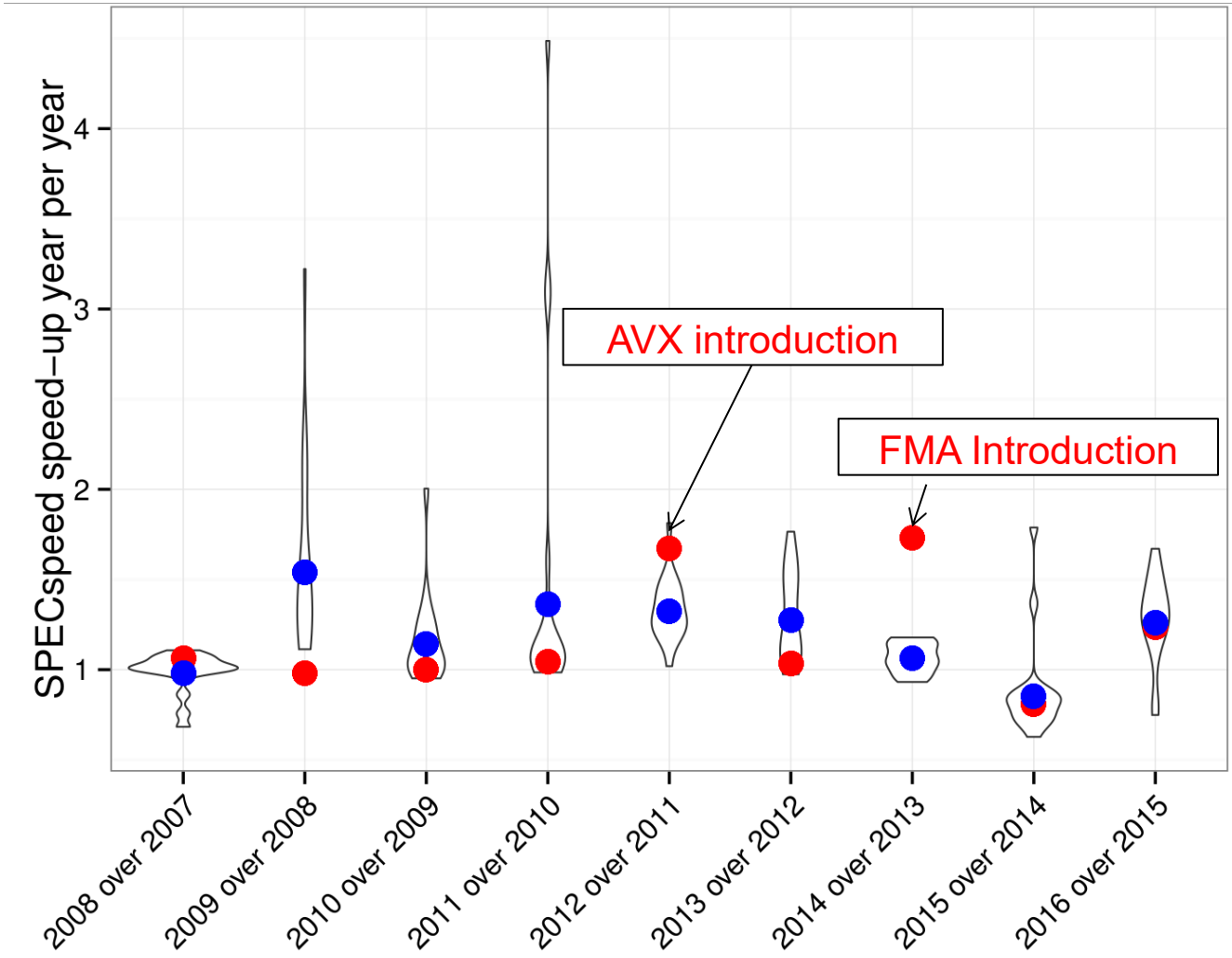
- Speedup from one year to the next one
- SPEC FP 2006 : Full set of “real” scientific applications
- Violin: distribution for the full set of SPECFP
- Architecture/compilers vary from year to year BUT Source code are invariant
- Unicore measurements
- Baseline compiler options but standard – O3 options



5

Performance Progress from 2007 to 2016 (2)

- Speedup from one year to the next
- Red Dots: speedup of peak FP performance: most of the time, speedup around one.
- Blue Dots: geometric mean of Speedups
- Unicore measurements
- Baseline



6

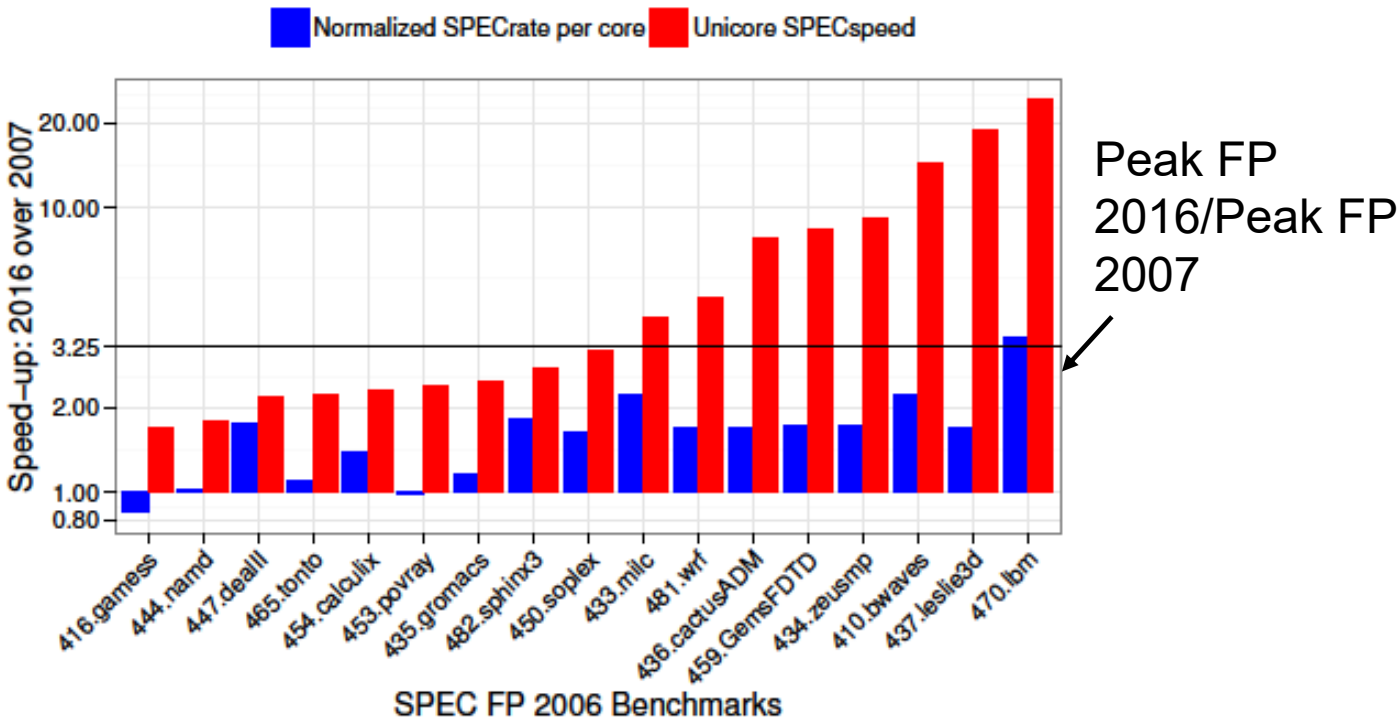
Unicore versus throughput

- Unicore measurements: only one core accessing the whole memory system: unrealistic, in real systems all of the cores will be active and share the memory system
- Launch multiple copies of the same program, one per core: each core will access a fair share of the system and therefore performance measured will be more realistic: SPECrate measurements.

7

A more realistic view on performance progress (1)

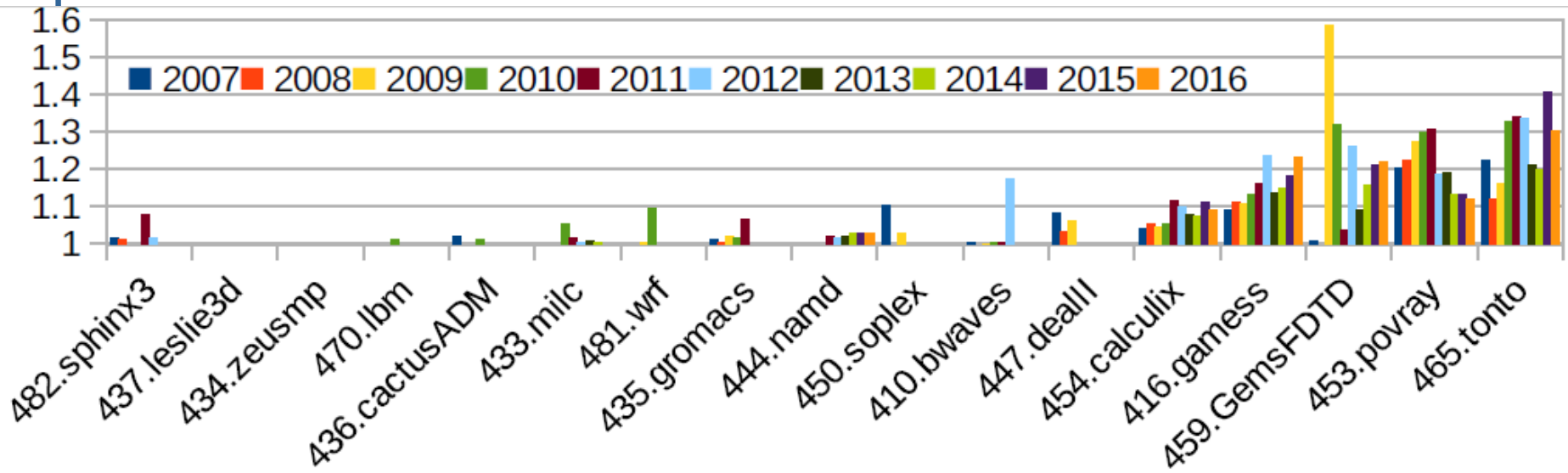
Ratio of 2016 performance over 2007



- Baseline numbers (no specific hand tuning but standard –O3)
- Performance gains highly dependant upon applications
- Red bars much higher than blue bars
- Blue bars always under the ratio of peak FP…..

8

Peak versus baseline



- Baseline: standard flags (typical –O3)
- Peak: hand picked flags
- Y axis: speedup of peak versus baseline
- X axis: sorted first by SPEC FP 2006 codes and second by year (same reference architectures)
- Profile Guided Optimization (PGO) is the most profitable (rightmost part of the digram)

9

GORDON BELL AWARD 2007 - 2016

Year	Machine	Cores	Tflops obtained	Tflops peak	Obtained over Peak (%)	Computations
2007	BlueGene/L	131K	0,11	0,28	39	Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability
2008	Cray XT4	31K	0,2	0,26	77	Simulations of disorder effects in high-Tc superconductors
2009	Cray XT5	147K	1,03	1,36	76	Ab initio computation of free energies
2010	Cray XT5-HE	200K	0,7	2,3	30	Direct numerical simulation of blood flow
2011	K computer	442K	3,08	7,07	44	First-principles calculations of electron states of a silicon nanowire
2012	K computer	663K	4,45	10,6	42	Astrophysical N -Body Simulation
2013	Sequoia	1.6M	11	20,1	55	Cloud Cavitation Collapse
2014	Anton 2	NA	NA	NA	NA	Molecular dynamics
2015	Sequoia	1.6M	0,69	20,1	3	Implicit Solver for Complex PDEs
2016	Sunway TaihuLight	10.56M	7,85	125	6	Fully-Implicit Solver for Nonhydrostatic Atmospheric Dynamics

- Very impressive results: high efficiency except for the last 2 years
- Very different results from SPEC evolution
- Codes have been fully customized: using (??) top of the line performance evaluation tools ☺
- Are these codes more than Proof of Concept ?? Impact on standard apps ??

10

SPEC numbers over 2007 - 2016 are completely depressing.

REMARK: similar analysis was pursued on real apps and with different architectures for the period beyond 2017 and similar “depressing” results were obtained.

On real applications, the gap between peak (nominal) performance and observed performance is increasing!!

Two possible choices:

1. Improve Compiler Autotuning: OK but will provide at best 10 to 20% perf improvement: limited gain but still worth getting
2. Rewrite applications: OK but might be very costly and might be difficult for most application fields

A FEW DIRECTIONS

More powerful approaches

- Split performance optimizations into 3 subproblems
  - Identify performance issue (diagnostic)
  - Identify potential remedies
  - Implement code transformations (directives, pragma or rewriting)
- Use performance tools to guide application restructuring identifying fruitful code transformations .
- Identify good compiler options but go firther and compare compilers output: import optimizations from one compiler to the other one.

12

MAQAO/ONEVIEW PRINCIPLES

Target audience/usage for performance tools

Performance tools can be used by different people with different goals:

- Application developers to get high performance codes on different compilers/architectures
  - Perform (major) code refactoring to improve performance
  - Understand/analyze impact of new compilers/libraries/hardware and perform corresponding code “adjustments”
  - Analyze impact of minor code changes: performance maintenance every week/every month
- Compiler/Runtime/Library developers
- Hardware designers
- And some more …(not an exhaustive list)

All of these different audiences need different information and different formatting.

Our primary target is code developers (not performance experts) so we need to simplify their code optimization process.

A secondary (close to the first one) is benchmarkers, performance experts but not code experts

- A few basic ingredients: latency/bandwidth/size of different (functional and storage) units, dependencies,

- A very large number of possible combinations
  - Quantitative nature of Basic ingredients: cache miss ratio can vary incrementally between 0 and 100%
  - Raw values of basic ingredients might be unusable: number of stalls due to a buffer full)
  - Raw values are useless only combinations are meaning full: for example cache miss ratio by itself is not sufficient only cache miss x average time for miss is meaningful

- The user is often lost by hardware counters and to some extent he does not care because they are not accurate enough: within a loop, you need to precisely identify arrays which are causing trouble because it is where you will have to focus your efforts.

NOBODY WANTS PROBLEMS EVERYBODY WANTS SOLUTIONS ☺

- It is nice to correlate source line numbers with hardware counters values but this is not enough because the user cannot change hardware in general ☺
- We should concentrate on the knobs that the code developer has at his disposal:
  - Better compiler options
  - Code restructuring
  - Data restructuring
- More precisely, MAQAO will
  - Identify well known issues: small trip count, complex control flow, lack of vectorization, poor vectorization etc…..
  - Use what if methods to predict (more or less accurately) the impact of removing the issue
  - Also Use what if methods to predict the impact of standard transformations: partial of full vectorization etc….
  - Use comparison to get a better understanding of code behaviour

OUR APPROACH: APPLICATION CENTRIC VIEW (1)

Instead of only pinpointing problems, try to guide the user towards a few solutions.

STARTING POINT: the user has at his disposal a limited given number of code transformations.

OUR VIEW:

- **What type of problems are we facing ??** CPU or data access problems
- **What transformations to apply??** a few key transformations are targeted: compiler switches, partial/full vectorization, loop blocking/array restructuring, if removal, full unroll, binary tranforms (prefetch) etc…..

OUR APPROACH: APPLICATION CENTRIC VIEW (2)

OUR VIEW:

- **Where to apply transformations** ?? Find the most rewarding loops and issues to be fixed.

A simple example

- **Loop A: 40%** total time, expected **10%** speedup
  - ➔ TOTAL IMPACT: **4%** speedup
- **Loop B: 20%** total time, expected **50%** speedup
  - ➔ TOTAL IMPACT: **10%** speedup

=> Need for tools capable of evaluating performance gains related to transformations: evaluation of What if Scenarios.

- **The user wants to optimize for several data sets, configurations, parallelization parameters (number of ranks/threads), and find tradeoffs:** automatically run and aggregate performance numbers with varying number of cores, different datasets, etc…

How much can the user trust our recommendations ??

Provide the user with quality indicators on measurement

- **For example, measuring too short durations within an OoO machine is not meaningful:** any duration measured under 500 cycles is subject to caution. Any duration measured under 250 cycles is close to noise.

ANALYSIS PERFORMED AT THE BINARY LEVEL

- Advantages of binary analysis:
  - Compiler optimizations increase the distance between the executed code and the source
  - Source code instrumentation may prevent the compiler from applying some transformations
- We want to evaluate the “real” executed code: **What We Analyze Is What You Run**

A QUICK TOUR OF MAQAO/OV FUNCTIONALITIES

MAQAO/ONE VIEW: KEY TOOLS/MODULES

MAQAO relies on various tools for analyzing code behavior:

- Lprof: light weight profiler operating at process, thread, function and loops
- Vprof: value profiler: probes inserted at binary level to measure duration, iteration count, etc…
- CQA: Code Quality Analysis: analyzes statically code, compute various metrics including vectorization and performs very simple simulation (assuming all operands in L1) and infinite buffer size.
- UFS: more realistic simulator taking into account buffer size
- DECAN : modifies loop binaries reinsert them for execution and measures modified versus original
- PRoMPT: exploits OMPT standard to analyze parallel loop behavior

ONE View: Performance View Aggregator module

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

MAQAO "WHAT IF" SCENARIOS: UNICORE (1)

- Generate performance estimates for different transformations
- Work at the innermost loop level (ASM)
- First modify ASM to take into account code transformation: "Clean" version (only FP operations are kept), "FP Vector" (only vectorizes FP arithmetic), "Full Vector" (vectorizes all FP operations), "DL1" (forces all of the operands to come from L1, ……
- Second generate performance estimates
  - Either using static more or less simplified simulators
  - Or embed modified ASM in the real code and measure it.
- Many more what if scenario can be derived: suppressing branches, suppressing costly FP operations (Div/SQRT)
- All of these metrics will be reorganized and aggregated through the Oneview module

"WHAT IF" SCENARIOS: VECTORIZATION

- Code "No scalar integer"
  - Generate an Assembly "No scalar integer" variant : keep only FP Arithmetic and Memory operations, suppress all other
  - Generate a CQA Performance estimate on the "Clean" Variant
- Code "FP Vector"
  - Generate an Assembly "FP Vector" variant : only replace scalar FP Arithmetic by Vector FP Arithmetic equivalent. Generate additional instructions to fill in Vector Registers.
  - Generate a CQA Performance estimate
- Code "Full Vector"
  - Generate an Assembly "Full Vector" variant : replace both scalar FP Arithmetic and FP Load/Store by their Vector equivalent.
  - Generate a CQA Performance estimate
- All of these "What If Scenarios" are generated in a fully static manner.

TYPICAL ONE VIEW GLOBAL TAB (1)

RUN ON AWS G4 96 Cores
OpenRadioss: 48 MPI x 2 OpenMP
Compiler: Arm F90 F90 Flang - 1.5 2017-05-01

Global Metrics

Total Time (s)	1.12 E3
Profiled Time (s)	1.09 E3
Time in analyzed loops (%)	71.0
Time in analyzed innermost loops (%)	67.5
Time in user code (%)	75.8
Compilation Options Score (%)	0
Array Access Efficiency (%)	78.6

Potential Speedups

Perfect Flow Complexity		1.04
Perfect OpenMP + MPI + Pthread		1.14
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.27
No Scalar Integer	Potential Speedup	1.14
	Nb Loops to get 80%	23
FP Vectorised	Potential Speedup	1.09
	Nb Loops to get 80%	22
Fully Vectorised	Potential Speedup	1.58
	Nb Loops to get 80%	41
FP Arithmetic Only	Potential Speedup	1.40
	Nb Loops to get 80%	34

Direct measurements and analysis:

Array Access Efficiency:
Percentage of Unit Stride access

TYPICAL ONE VIEW GLOBAL TAB (2)

FOCUS: on transformations and impact at the application level

Global Metrics

Total Time (s)		1.12 E3
Profiled Time (s)		1.09 E3
Time in analyzed loops (%)		71.0
Time in analyzed innermost loops (%)		67.5
Time in user code (%)		75.8
Compilation Options Score (%)		0
Array Access Efficiency (%)		78.6

Potential Speedups

Perfect Flow Complexity		1.04
Perfect OpenMP + MPI + Pthread		1.14
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.27
No Scalar Integer	Potential Speedup	1.14
	Nb Loops to get 80%	23
FP Vectorised	Potential Speedup	1.09
	Nb Loops to get 80%	22
Fully Vectorised	Potential Speedup	1.58
	Nb Loops to get 80%	41
FP Arithmetic Only	Potential Speedup	1.40
	Nb Loops to get 80%	34

Perfect flow complexity: evaluate performance gain if innermost loops had no branches

FP vectorized: Performance gain if all the FP arithmetic operations were vectorized

Fully vectorized: Performance gain if all the FP arithmetic operations+ Load/Store instructions were vectorized

"WHAT IF" SCENARIOS MULTICORE (1)

- Evaluate the performance impact of code transformation.
- Rely on performance measurement (either profiling or tracing (OMPT))
- Generate performance estimate:
  - Perfect OpenMP + MPI + Pthread: for each thread, get rid of time spent in OpenMP, MPI, Thread libraries and then takes max
  - Perfect OpenMP + MPI + Pthread + Perfect Load Distribution: for each thread, get rid of time spent in OpenMP, MPI, Thread libraries and then takes average
- Performance estimate generates either globally or at the loop level

WHAT IF MUTICORE SCENARIO

Global Metrics

Metric		r0	r1	r2	r3	r4
Total Time (s)		862.11	442.46	373.20	341.43	351.33
Profiled Time (s)		820.72	403.01	336.03	298.46	293.46
Time in analyzed loops (%)		93.4	91.1	89.7	86.3	81.5
Time in analyzed innermost loops (%)		91.5	88.2	86.3	82.1	77.1
Time in user code (%)		51.5	63.7	73.2	84.4	86.8
Compilation Options Score (%)		0	0	0	0	0
Array Access Efficiency (%)		96.3	95.2	94.9	94.2	94.4
Potential Speedups						
Perfect Flow Complexity		1.09	1.08	1.07	1.08	1.08
Perfect OpenMP + MPI + Pthread		1.71	1.25	1.16	1.07	1.03
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.87	1.49	1.30	1.14	1.11
Scalability - Gap		1.00	0.51	0.43	0.40	0.41
No Scalar Integer	Potential Speedup	1.18	1.09	1.07	1.06	1.06
	Nb Loops to get 80%	1	9	13	17	18
FP Vectorised	Potential Speedup	1.24	1.09	1.05	1.03	1.03
	Nb Loops to get 80%	1	3	6	10	11
Fully Vectorised	Potential Speedup	2.84	2.35	2.07	1.79	1.62
	Nb Loops to get 80%	12	13	14	18	18
Only FP Arithmetic	Potential Speedup	2.13	1.95	1.87	1.76	1.68
	Nb Loops to get 80%	19	24	26	32	33

▼ Compared Reports

- r0: m1o52
- r1: m2o26
- r2: m4o13
- r3: m26o2
- r4: m52o1

mX: number of MPI ranks

oY number of OpenMP threads

OpenRadioss running on a 2x 26 cores Skylake using IFX.

Code quality analysis

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

Similar issues as with application developer except it should be more generic.

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 OUTPUT

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

Main limitations of timing analysis:

1) Timing analysis requires runs on the same machine

2) 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)

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)

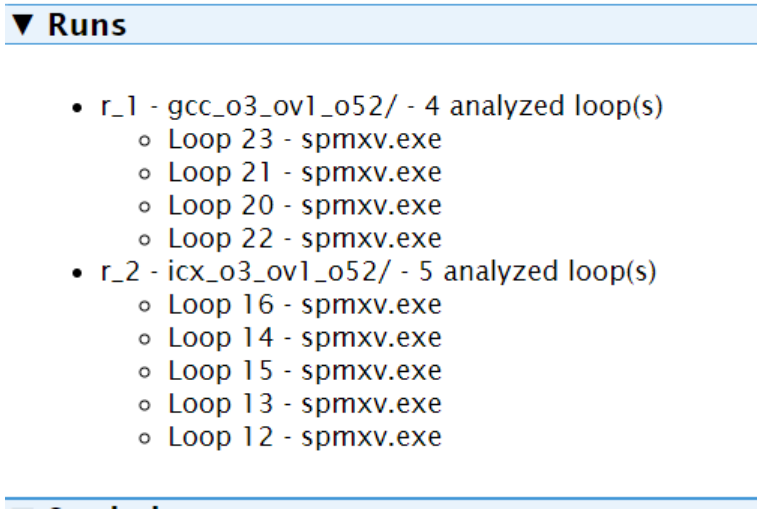
SA: Static Analysis
DT: Dynamic Timing

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

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.



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

OUR APPROACH

- Performing a detailed assessment of code quality is therefore very important
- 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

WRAP UP ON MAQAO/OV

- Optimizing (complex) for complex recent architectures is becoming more and more difficult
- We need a new generation of performance tools to guide the code/developer through that task

MAQAO/ONE VIEW provides a new approach

- Provides an application centric view
- Provides synthetic/aggregated view meaningful for the user
- Provides performance estimates of potential gains (what if scenarios)
- The compare mode with its different flavors (ISO BINARY, ISO SOURCE, ISO FUNCTION STRUCTURE) is very efficient for the code developer to track progress and to detect quickly problems.
- The “summary” approach provides another way of interacting with code developer: more direct and focused, and efficient guidance through optimization maze

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/

BACKUP SLIDES

QaaS Characteristics (1)

QaaS is an **Open Source collaborative** project aiming at providing a software environment for benchmarking applications/systems with the following key characteristics:

- Automation: pushed as much as possible. Large amount of data automatically generated
- Systematic exploration of key parameters space:
  - hardware platforms (AMD, ARM (Ampere, Grace, G3/G4), INTEL, etc…)
  - Compilers and compiler settings
  - number of cores and core mapping, more generally runtime and run time settings
  - More generally: software stack parameters

QaaS Characteristics (2)

- Uniform exploration: to enable systematic comparison
- Reliability: built in repetition mechanisms to ensure reliable results
- Detailed level of comparison: use of MAQAO/OneView) to provide several profiling levels : whole application, functions, loops and key application characteristics.
- Flexible search and formatting through results
- Two levels of optimizations:
  - automatic (compilers/compiler switches/transformations),
  - recommendations for “manual” optimizations.

MAJOR PARTNERS: INTEL US + UVSQ: a major release targeted for SC24

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)

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	0.97	100	35
15	6.69	6.79	6.29	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)

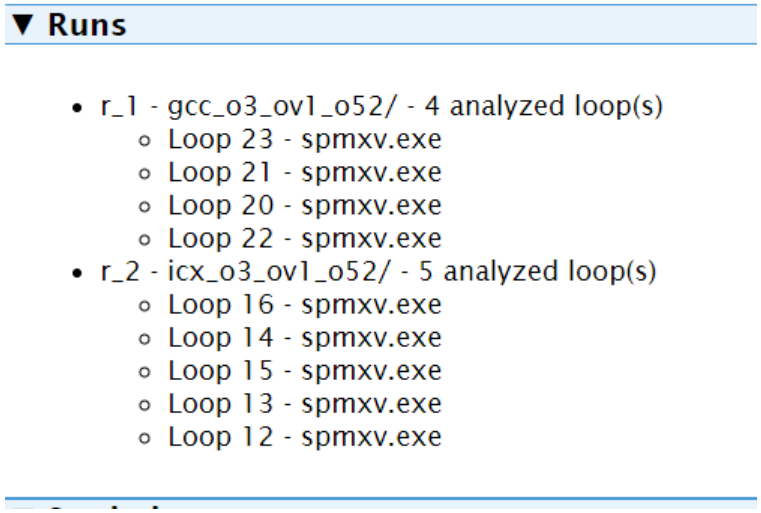
Loop ID	Analysis	Penalty Score
▶ Loop 16 - spmxv.exe	Execution Time: 50 % - Vectorization Ratio: 100.00 % - Vector Length Use: 35.00 %	
▶ Loop 14 - spmxv.exe	Execution Time: 18 % - Vectorization Ratio: 31.88 % - Vector Length Use: 15.94 %	
▼ Loop 15 - spmxv.exe	Execution Time: 16 % - Vectorization Ratio: 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.



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

Comparing at the loop level

Run engine_NEON1M11-0001_o2_m48_gcc

Loop Source Regions
- /home/kcamus/pop3/OpenRadioss/OpenRadioss/engine/source/interfaces/intsc 84-100

ASM Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vecti
29120	38.27	24.52	1.72	11.11	27.08

Sum on 1 analyzed binary loop (engine_linuxa64_gf_ompi - 29120)

Analysis	
Loop Computation Issues	
Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	1
Presence of a large number of scalar integer instructions	1
Control Flow Issues	
Presence of 2 to 4 paths	1
Data Access Issues	
Presence of constant non-unit stride data access	1
Presence of indirect access	0
Vectorization Roadblocks	
Presence of 2 to 4 paths	1
Presence of constant non-unit stride data access	1
Presence of indirect access	0

Run engine_NEON1M11-0001_o2_m48_acfl

Loop Source Regions
- /home/kcamus/pop3/OpenRadioss/OpenRadioss/engine/source/interfaces/in 85-103

Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Ve
36824	33.51	18.04	1.66	10.61	25.1

Sum on 1 analyzed binary loop (engine_linuxa64_ompi - 36824)

Analysis	
Loop Computation Issues	
Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA	1
Presence of a large number of scalar integer instructions	1
Control Flow Issues	
Presence of 2 to 4 paths	1
Data Access Issues	
Presence of constant non-unit stride data access	0
Presence of indirect access	1
Vectorization Roadblocks	
Presence of 2 to 4 paths	1
Presence of constant non-unit stride data access	0
Presence of indirect access	1

"WHAT" IF MULTICORE SCENARIO

Global Metrics

Metric		r0	r1	r2	r3	r4	r5	r6	r7
Total Time (s)		2.41 E3	1.26 E3	648.52	355.62	203.22	110.41	78.51	94.06
Profiled Time (s)		2.41 E3	1.26 E3	647.50	354.44	202.11	109.40	77.22	92.69
Time in analyzed loops (%)		28.6	29.9	30.3	32.1	35.1	33.1	30.7	30.4
Time in analyzed innermost loops (%)		21.6	22.5	22.6	23.9	26.7	26.3	25.0	25.0
Time in user code (%)		96.6	96.5	96.0	95.3	93.5	94.7	94.4	94.9
Compilation Options Score (%)		66.3	66.1	65.7	65.4	64.7	65.3	64.8	64.4
Perfect Flow Complexity		1.00	1.00	1.00	1.00	1.01	1.01	1.00	1.00
Array Access Efficiency (%)		72.4	71.3	71.6	71.6	70.6	69.7	69.4	71.4
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.01
No Scalar Integer	Potential Speedup	1.17	1.16	1.16	1.15	1.14	1.13	1.10	1.05
	Nb Loops to get 80%	12	13	13	13	14	14	14	14
FP Vectorised	Potential Speedup	1.02	1.02	1.02	1.03	1.03	1.03	1.03	1.02
	Nb Loops to get 80%	6	7	7	7	8	9	8	9
Fully Vectorised	Potential Speedup	1.06	1.08	1.08	1.10	1.14	1.13	1.14	1.18
	Nb Loops to get 80%	19	21	22	22	21	19	17	10
Only FP Arithmetic	Potential Speedup	1.26	1.26	1.26	1.27	1.28	1.25	1.22	1.15
	Nb Loops to get 80%	17	21	22	23	26	26	27	22
OpenMP perfectly balanced	Potential Speedup	1.00	1.03	1.02	1.04	1.06	1.08	1.14	1.39
	Nb Loops to get 80%	4	2	5	6	9	10	10	9
Scalability - Gap		1.00	1.05	1.08	1.18	1.35	1.46	2.08	4.99

r0 (resp. r1 , r2, etc…) denotes runs on 1 (resp. 2, 4 etc…) cores

GROMACS running on a 2x 64 cores AMD EPYC7H12:. The dataset was chosen small enough to show scaling issues with more than 16 cores .

- Objectives:
  - Characterizing performance of HPC applications
  - **Guiding users** through optimization process
  - Estimating return of investment (**R.O.I.**) for transformation
- Characteristics:
  - Support for **Intel / AMD x86-64** and **AArch64**
    - 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**
  - Project started in 2004 on Itanium
- 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 We Analyse Is What You Run**

59

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, 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, MontBlanc3, ...