RECENT ONE VIEW DEVELOPMENTS

C. Valensi, E. Oseret, M. Tribalat, H. Bolloré, S. Ibnamar, K. Camus, M. Hoffer, A. Delval +

W. Jalby

University of Versailles Saint Quentin/University of Paris Saclay

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

- MAQAO is a performance analyzis and optimisation framework operating at binary level developed at UVSQ since 2004
  - Complementary modules, each of them focusing on one aspect of performance analysis: profiler, static analyzer, value profiler, simplified simulators, decremental analyzer, …
  - Support for Intel/AMD x86-64, Xeon Phi and ARM (ongoing)
  - http://maqao.exascale-computing.eu

- 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

OneView Target objectives

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

- No interest in overwhelming them with low level performance data such as raw performance counter measurements or more generally with hardware centric metrics..
- Their main knobs are at code source level:
  - Change compiler flags and runtime settings
  - Change loop/parallel construct body (remove dependencies, simplify control flow, …), insert pragmas …
  - Restructure arrays
- Guide code developer through optimization process (insert pragmas, to restructure loop/parallel constructs….):
  - Provide them with potential performance gain associated with a transformation or more generally with a bad code characteristic
  - Give an estimate of effort required.
  - Perform comparative analysis between different code versions/compilers/hardware

OneView Recent Developments

Five main directions (still in progress):

- Improve analysis of parallel OMP codes : integrate OMPT in MAQAO.
- Support new architectures (AMD, ARM, ……..GPU): the user will have the same interface and logic across different architectures
- Integrate further Hardware component usage analysis: Saturation/Intensity method (D. Kuck/INTEL)
- Enhance comparative analysis
- Simplify end user interface: make a synthesis of all of the information to offer a better guidance through the optimization “maze”

FOCUS IN THIS TALK ON THE LAST TWO POINTS

PRINCIPLES OF COMPARATIVE ANALYSIS

Basic principles: run different “code versions” and compare them on “appropriate levels”.

TRIAL AND ERROR and comparison are fundamental techniques in scientific approach.

- Different “code versions”
  - Different runtime settings (on different number of cores, etc..)
  - Different compilers
  - Different hardware (X86, ARM, …) with same or different ISA
  - Different code versions
- “Appropriate levels”:
  - ISOBINARY: the same binary is compared in different settings
  - ISOSOURCE: the same source is compared
  - ISOFUNCTION STRUCTURE: the source code can be different but the function structure is preserved.
  - Generic: much harder to compare

NOT VERY SOPHISTICATED AT FIRST BUT VERY USEFULL AND IMPLEMENTATION IS A BIT SUBTLE

EXPERIMENTAL SETUP

- Hardware
  - SKL: Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz
  - KBL: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
  - ZEN-2: AMD EPYC 7H12 64-Core Processor
  - ZEN-3: AMD EPYC 7763 64-Core Processor
- Compilers:
  - INTEL ICC/IFC and ICX/IFX
  - GCC 11
- Codes
  - Lulesh
  - MiniQMC: proxyapp for QMCPACK (An Open Source Quantum Monte Carlo Package for the electronic structure of atoms, molecules and solids)
  - CHAMP: The Cornell-Holland Ab-initio Materials Package (CHAMP) is a quantum Monte Carlo package for electronic structure calculations of molecular systems
  - Jastrow: Quantum Montecarlo Computation (QMCKL library).
  - MAHYCO (Arcane): hydrodynamics code

TYPICAL ONE VIEW GLOBAL TAB

Global Metrics		
Total Time (s)		63.86
Profiled Time (s)		61.31
Time in analyzed loops (%)		61.6
Time in analyzed innermost loops (%)		61.2
Time in user code (%)		61.6
Compilation Options		OK
Perfect Flow Complexity		1.01
Iterations Count		1.00
Array Access Efficiency (%)		88.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	7
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	11
Data In L1 Cache	Potential Speedup	1.59
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.16
	Nb Loops to get 80%	11

FOCUS: on transformations and impact at the application level MiniQMC (proxy for QMCPACK) code running on SKL and ICC 19.

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

Iteration count: evaluate the impact of having all loop iteration count over 100

Array Access Efficiency: Percentage of Unit Stride access

TYPICAL ONE VIEW GLOBAL TAB

Global Metrics		
Total Time (s)		63.86
Profiled Time (s)		61.31
Time in analyzed loops (%)		61.6
Time in analyzed innermost loops (%)		61.2
Time in user code (%)		61.6
Compilation Options		OK
Perfect Flow Complexity		1.01
Iterations Count		1.00
Array Access Efficiency (%)		88.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	7
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	11
Data In L1 Cache	Potential Speedup	1.59
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.16
	Nb Loops to get 80%	11

FOCUS: on transformations and impact at the application level MiniQMC (proxy for QMCPACK) code running on SKL and ICC 19.

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

TYPICAL ONE VIEW GLOBAL TAB

Global Metrics		
Total Time (s)		63.86
Profiled Time (s)		61.31
Time in analyzed loops (%)		61.6
Time in analyzed innermost loops (%)		61.2
Time in user code (%)		61.6
Compilation Options		OK
Perfect Flow Complexity		1.01
Iterations Count		1.00
Array Access Efficiency (%)		88.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	7
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	1.04
	Nb Loops to get 80%	11
Data In L1 Cache	Potential Speedup	1.59
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.16
	Nb Loops to get 80%	11

FOCUS: on transformations and impact at the application level MiniQMC (proxy for QMCPACK) code running on SKL and ICC 19.

Data in L1 cache: Performance gain if all of the operands are coming from L1

ISO BINARY: SCALABILITY RUNS

CHAMP: Weak Scalability Analysis

r0 : 1 core r1: 2 cores r2: 4 cores r3: 8 cores r4: 16 cores r5: 26 cores

Global Metrics

Metric		r0	r1	r2	r3	r4	r5
Total Time (s)		21.23	21.61	21.28	21.43	21.91	22.94
Profiled Time (s)		20.86	20.80	20.96	21.12	21.51	22.49
Time in analyzed loops (%)		79.8	78.4	78.5	77.3	76.8	75.9
Time in analyzed innermost loops (%)		42.6	41.6	41.5	41.3	40.6	40.6
Time in user code (%)		89.8	88.8	88.2	87.7	87.4	86.8
Compilation Options		OK					
Perfect Flow Complexity		1.00	1.00	1.00	1.00	1.00	1.00
Array Access Efficiency (%)		70.5	70.9	71.3	71.2	71.0	71.1
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.02	1.02	1.02	1.02
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.01	1.02	1.03	1.03	1.03
No Scalar Integer	Potential Speedup	1.20	1.20	1.20	1.19	1.19	1.19
	Nb Loops to get 80%	10	10	10	10	10	10
FP Vectorised	Potential Speedup	1.12	1.12	1.11	1.11	1.11	1.11
	Nb Loops to get 80%	5	5	5	5	5	5
Fully Vectorised	Potential Speedup	1.81	1.77	1.77	1.76	1.76	1.75
	Nb Loops to get 80%	26	26	26	26	27	27
Only FP Arithmetic	Potential Speedup	1.65	1.63	1.63	1.62	1.62	1.60
	Nb Loops to get 80%	25	25	25	25	26	27
Scalability - Gap		1.00	1.02	1.00	1.01	1.03	1.08

ISO FUNCTION STRUCTURE: FUNCTION LEVEL

CHAMP Unicore on SKL

Functions

Name	Module	Time (s)				
		champ_01apr_ov3_energy_15k	champ_26apr_ov3_energy_15k	champ_27apr_ov3_energy_15k	champ_29apr_ov3_energy_15k	champ_11may_ov3_energy_15k
multideterminante	vmc.mov1	7.37	4.6	4.07	3.45	3.55
basis_fns	vmc.mov1	2.19	1.85	2.09	1.96	1.93
compute_ymat	vmc.mov1	6.01	0.13	0.13	0.08	3.6
orbitals	vmc.mov1	1.49	1.56	1.47	1.43	1.48
nonloc	vmc.mov1	1.37	1.28	1.38	1.32	1.44
multideterminante_grad	vmc.mov1	1.09	1.09	1.11	1.11	1.19
multideterminant_hpsi	vmc.mov1	1.29	0.9	0.76	0.7	0.82
orbitalse	vmc.mov1	0.79	0.87	0.8	0.81	0.86
matinv	vmc.mov1	0.85	0.94	0.93	0.56	0.7
__powr8i4	vmc.mov1	0.62	0.76	0.71	0.68	0.7
idiff	vmc.mov1	0.65	0.65	0.7	0.66	0.66
splfit	vmc.mov1	0.56	0.55	0.51	0.58	0.61
detsav	vmc.mov1	1.31	0.56	0.5	0.2	0.21
__intel_avx_rep_memset	vmc.mov1	0.12	0.57	0.47	0.53	0.5
__intel_avx_rep_memcpy	vmc.mov1	0.25	0.14	0.42	0.46	0.82
determinante_psit	vmc.mov1	0.49	0.3	0.36	0.32	0.55
update_ymat	vmc.mov1	0.54	0.3	0.24	0.23	0.31
__libm_log_l9	vmc.mov1	0.24	0.31	0.25	0.23	0.23
psinl	vmc.mov1	0.13	0.16	0.14	0.14	0.17
slm	vmc.mov1	0.15	0.16	0.15	0.12	0.13
multideterminants_define	vmc.mov1	0.11	0.13	0.07	0.11	0.12
__libm_exp_l9	vmc.mov1	0.13	0.1	0.09	0.11	0.09
jastrow4e	vmc.mov1	0.14	0.06	0.07	0.05	0.07
compute_determinante_grad	vmc.mov1	0.07	0.04	0.07	0.05	0.07

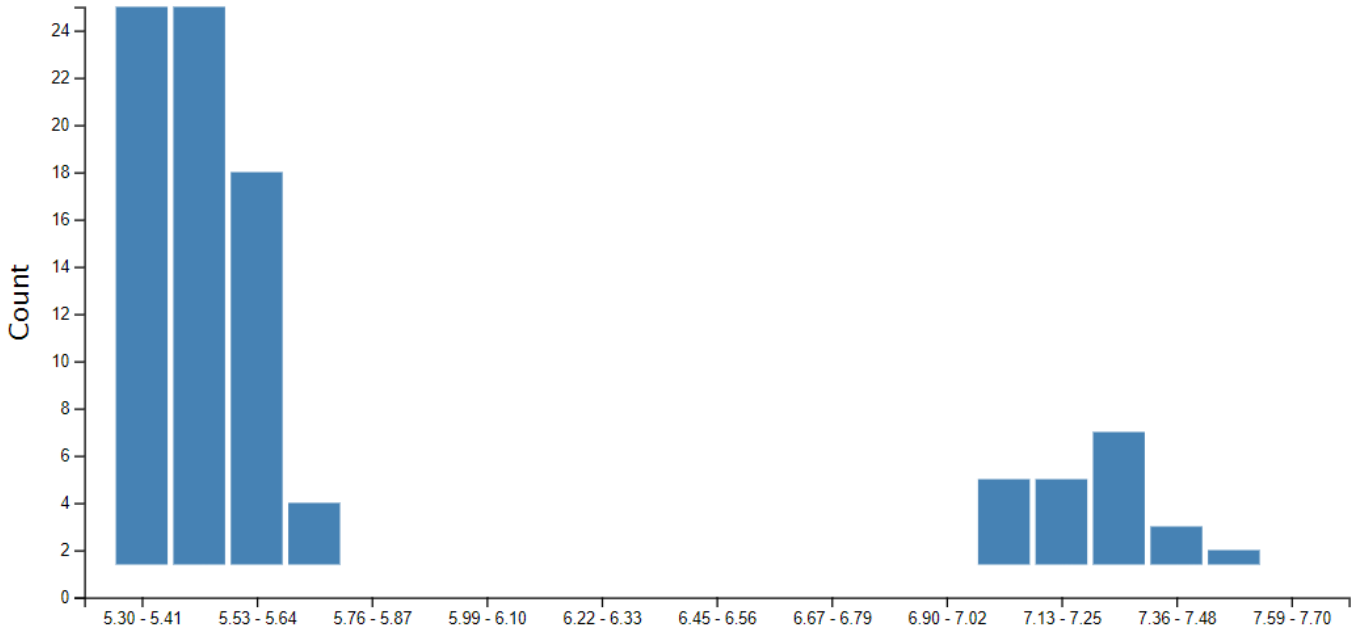
ISOBINARY: STABILITY RUNS (1)

Same code is run a large number of times to study measurement stability. Standard statistics (deciles) are computed. An extra xlsx file is produced.
KBL 4 cores. Lulesh2.0

Total Time

min	med	avg	max
5.3	5.53	5.93	7.59

Percentile Index	10	20	30	40	50	60	70	80	90	100
Value	5.37	5.4	5.43	5.47	5.52	5.57	5.67	7.03	7.26	7.59



ISOBINARY: STABILITY RUNS (2)

Same code is run a large number to study measurement stability.
Going at the function level allows to quickly identify delinquent function
Min, max, avg, med are computed over the 100 runs.

Name	Module	min (Max Time Over Threads) (s)	avg (Max Time Over Threads) (s)	med (Max Time Over Threads) (s)	max (Max Time Over Threads) (s)
○ omp_get_num_procs	libgomp.so.1.0.0	0.71	1.14	0.9	2.24
○ std::vector >::operator[](unsigned long)	lulesh2.0	0.41	0.51	0.51	0.62
► CalcFBHourglassForceForElems(Domain&, double*, double*, double*, double*, double*, double*, double*, double, int, int) [clone ._omp_fn.7]	lulesh2.0	0.34	0.42	0.41	0.55
► EvalEOSForElems(Domain&, double*, int, int*, int) [clone ._omp_fn.17]	lulesh2.0	0.28	0.35	0.35	0.46
► CalcElemFBHourglassForce(double*, double*, double*, double (*) [4], double, double*, double*, double*)	lulesh2.0	0.19	0.25	0.24	0.33
► CalcEnergyForElems(double*, double*, double*, double*, double*, double*, double*, double*, double*, double*, double*, double*, double, double, double, double, double*, double*, double, double, int, int*) [clone ._omp_fn.20]	lulesh2.0	0.15	0.20	0.2	0.27
► CalcMonotonicQGradientsForElems(Domain&) [clone ._omp_fn.14]	lulesh2.0	0.16	0.20	0.2	0.26

ISO SOURCE: DIFFERENT HARDWARE

All runs were unicore and used the same compiler GNU 11
Code MAHYCO (Arcane framework)

r0: SKL r1: ZEN_2 r2: ZEN_3

Global Metrics

Metric		r0	r1	r2
Total Time (s)		916.81	738.02	592.37
Profiled Time (s)		915.78	734.50	590.03
Time in analyzed loops (%)		72.8	69.3	68.2
Time in analyzed innermost loops (%)		41.7	40.1	41.4
Time in user code (%)		87.7	86.6	85.9
Compilation Options		OK	OK	OK
Perfect Flow Complexity		1.36	1.26	1.28
Array Access Efficiency (%)		64.0	62.1	61.7
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.26	1.17	1.16
	Nb Loops to get 80%	5	5	5
FP Vectorised	Potential Speedup	1.41	1.31	1.29
	Nb Loops to get 80%	5	7	7
Fully Vectorised	Potential Speedup	2.26	1.91	1.88
	Nb Loops to get 80%	16	14	14
Only FP Arithmetic	Potential Speedup	1.46	1.42	1.33
	Nb Loops to get 80%	7	6	6

ISO SOURCE: DIFFERENT COMPILERS

TARGET CODE/PROC: QMCKL Jastrow on SKL

Compilers: r1: Intel 2021.5.0 r2: Intel 22.0-1069

Global Metrics

Metric		r1	r2
Total Time (s)		8.03	7.98
Profiled Time (s)		8.03	7.98
Time in analyzed loops (%)		26.1	26.0
Time in analyzed innermost loops (%)		24.1	25.5
Time in user code (%)		26.2	26.0
Compilation Options		OK	**bench_jastrow**: -march=(target) is missing. -funroll-loops is missing. **libqmckl.so.0**: -g is missing, it is needed to have more accurate reports. -O2, -O3 or -Ofast is missing. -x(target) or -ax(target) is missing.
Perfect Flow Complexity		1.00	1.00
Array Access Efficiency (%)		98.2	83.1
Perfect OpenMP + MPI + Pthread		1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00
No Scalar Integer	Potential Speedup	1.01	1.01
	Nb Loops to get 80%	5	3
FP Vectorised	Potential Speedup	999 E-3	1.01
	Nb Loops to get 80%	1	3
Fully Vectorised	Potential Speedup	1.19	1.17
	Nb Loops to get 80%	7	7
Only FP Arithmetic	Potential Speedup	1.16	1.16
	Nb Loops to get 80%	7	6

ISO FUNCTION STRUCTURE:

Global Metrics

Metric		r0	r1	r2	r3	r4
Total Time (s)		29.12	22.53	21.32	19.63	21.80
Profiled Time (s)		28.78	22.18	20.92	19.25	21.49
Time in analyzed loops (%)		87.3	81.1	79.7	79.8	78.7
Time in analyzed innermost loops (%)		37.8	47.1	43.8	51.4	51.7
Time in user code (%)		94.6	90.7	90.0	88.8	88.5
Compilation Options		OK	OK	OK	OK	OK
Perfect Flow Complexity		1.00	1.05	1.00	1.00	1.00
Iterations Count		1.04	1.02	1.03	1.03	1.02
Array Access Efficiency (%)		79.6	81.9	71.8	70.3	71.3
Perfect OpenMP + MPI + Pthread		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
No Scalar Integer	Potential Speedup	1.23	1.20	1.19	1.17	1.16
	Nb Loops to get 80%	12	13	10	12	11
FP Vectorised	Potential Speedup	1.18	1.27	1.27	1.29	1.27
	Nb Loops to get 80%	14	14	14	17	18
Fully Vectorised	Potential Speedup	3.69	2.86	2.73	2.63	2.58
	Nb Loops to get 80%	41	41	41	41	41
Only FP Arithmetic	Potential Speedup	2.01	1.65	1.65	1.59	1.69
	Nb Loops to get 80%	26	29	28	35	37
Data In L1 Cache	Potential Speedup	1.05	1.06	1.07	1.10	1.08
	Nb Loops to get 80%	5	4	5	6	6

5 successive code versions of CHAMP

Unicore runs SKL

Regular gains except for the last one!!

Simplify user interface (3)

Sort out performance issues:

LEVEL 0 (Stylizer): Is your run worth analysing ? Lack of important flags, too short execution times, … all of such issues deeply the interest of performing detailed analysis.

LEVEL 1 (Strategizer): Globally how difficult the optimization process will be ? We analyse the main profile characteristics in particular identifying importance of main types of codes: loops, innermost, outermost, in between, library use, etc….

LEVEL 2 (Optimizer): at loop level (innermost/inbetween/outermost) detect performance issues (in a predefined list) and then display only the ones which are present.

HIGH LEVEL VIEW OF SUMMARY TAB

https://datafront.maqao.exascale-computing.eu/public_html/oneview2020/miniqmc/OMPoffload-bfb1b0/base/skl/ov3/miniqmc_OMPoffload-bfb1b0_base_skl_o1_m1_c1_ov3_g422-n5-N1-b_icc-avx512/summary.html

QMCPACK Unicore on SKL (ICC)

MAQAO | Summary | Global | Application | Functions | Loops | Topology

miniqmc - 2022-06-07 14:48:04 - MAQAO 2.15.5

elp is available by moving the cursor above any [?] symbol or by checking MAQAO website.

▸ **Stylizer**

▸ **Strategizer**

▾ **Optimizer**

Loop ID	Module	Analysis	Score	Coverage (%)
▸ 1675	miniqmc	The loop is fully and efficiently vectorized.	0	25.17
▸ 1688	miniqmc	Inefficient vectorization.	28	12.25
▸ 2634	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	4	8.47
▸ 2633	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	4	3.02
▸ 1587	miniqmc	Inefficient vectorization.	28	2.33
○ 1710	miniqmc	Partial or unexisting vectorization - No issue detected	0	0.78
▸ 2692	miniqmc	Inefficient vectorization.	70	0.74
▸ 887	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	6	0.63
○ 2108	miniqmc	Partial or unexisting vectorization - No issue detected	0	0.62
▸ 2632	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	4	0.54

STYLIZER: DETAILS (1)

QMCPACK Unicore on SKL (ICC)

▼ Stylizer

[4 / 4] Application profile is long enough (61.48 s)

To have good quality measurements, it is advised that the application profiling time is greater than 10 seconds.

[3 / 3] Optimization level option -O3 is used

To have better performances, it is advised to help the compiler by using a proper optimization level (-O3)

[3 / 3] Helper debug compilation options -g and -fno-omit-frame-pointer are used

-g option gives access to debugging informations, such are source locations and -fno-omit-frame-pointer improve the accuracy of callchains found during the application profiling.

STYLIZER: DETAILS (2)

QMCPACK Unicore on SKL (ICC)

[3 / 3] Architecture specific option -xCORE is used

[2 / 2] Application is correctly profiled ("Others" category represents 0.11 % of the execution time)

To have a representative profiling, it is advised that the category "Others" represents less than 20% of the execution time in order to analyze as much as possible of the user code

The Optimizer

LEVEL 2 (Optimizer): at loop level (innermost/inbetween/outermost) detect performance issues (in a predefined list) and then display only the ones which are present: standard automobile dashboard

- For each performance issue associate a penalty score (higher is worse) and an estimate of performance gain (Work in Progress).
- For each issue list one or several potential roadblock
- 4 major categories
  1. Vectorization roadblocks including data access (no dependence analysis)
  2. Inefficient vectorization
  3. Advanced optimizations
  4. Parallelism
- Based on static analysis and also on dynamic analysis

SCORE is a penalty score: lower is better

QMCPACK (using icx) https://datafront.maqao.exascale-computing.eu/public_html/oneview2020/miniqmc/OMPoffload-bfb1b0/base/skl/ov3/miniqmc_OMPoffload-bfb1b0_base_skl_o1_m1_c1_ov3_g422-n5-N1-b_icx-avx512/summary.html

▼ Optimizer

Loop ID	Module	Analysis	Score	Coverage (%)
▶ 824	miniqmc	Inefficient vectorization.	33	14.36
▶ 1156	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	4	12.27
▶ 815	miniqmc	Inefficient vectorization.	13	6.94
▶ 809	miniqmc	Inefficient vectorization.	12	6.54
▶ 811	miniqmc	Inefficient vectorization.	13	6.18
▶ 813	miniqmc	Inefficient vectorization.	12	6.15
▶ 292	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	85	1.3
▶ 817	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	6	1.25
▶ 1472	miniqmc	Inefficient vectorization.	2	0.98
▶ 315	miniqmc	Inefficient vectorization.	10	0.68

OPTIMIZER DETAILS

Loop ID	Module	Analysis	Score	Coverage (%)
▼ 824	miniqmc	Inefficient vectorization.	33	14.36
○		Inefficient vectorization: more than 10% of the vector loads instructions are unaligned - When allocating arrays, don't forget to align them. There are 12 issues (= arrays) costing 2 points each	24	
○		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	
○		The ratio FP/LS (floating point / memory accesses) is smaller than 0.8 (0.17) - Focus on optimizing data accesses.	0	
○		Ratio time (ORIG)/time (DL1) is greater than 3 - Perform blocking. Perform array restructuring. There are 1 issues (= non unit stride or indirect memory access) costing 2 point each, with an additional malus of 5 points due to the ORIG/DL1 ratio.	7	
▼ 1156	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	4	12.27
○		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	
▼ 815	miniqmc	Inefficient vectorization.	13	6.94
○		The ratio FP/LS (floating point / memory accesses) is smaller than 0.8 (0.06) - Focus on optimizing data accesses.	0	
○		Ratio time (ORIG)/time (DL1) is greater than 3 - Perform blocking. Perform array restructuring. There are 0 issues (= non unit stride or indirect memory access) costing 2 point each, with an additional malus of 13 points due to the ORIG/DL1 ratio.	13	
▼ 809	miniqmc	Inefficient vectorization.	12	6.54
○		The ratio FP/LS (floating point / memory accesses) is smaller than 0.8 (0.06) - Focus on optimizing data accesses.	0	
○		Ratio time (ORIG)/time (DL1) is greater than 3 - Perform blocking. Perform array restructuring. There are 0 issues (= non unit stride or indirect memory access) costing 2 point each, with an additional malus of 12 points due to the ORIG/DL1 ratio.	12	
▶ 811	miniqmc	Inefficient vectorization.	13	6.18
▶ 813	miniqmc	Inefficient vectorization.	12	6.15
▶ 292	miniqmc	Partial or unexisting vectorization - Use pragma to force vectorization and check potential dependencies between array access.	85	1.3

CONCLUSION AND FUTURE DIRECTIONS

- 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
- Extensions: support for more complex performance issues, improve detection
- Perform automatic or semi automatic “issues” repair: improve automatically style and perform some optimizations
- Build a database of issues

BACKUP SLIDES

ROADBLOCK ISSUES (1)

Performance Issue	Cost for removing perf issue	Optimization
ROADBLOCK FOR VECTORIZATION		
Presence of calls (SA)	+1 per call	Inline either by compiler or by hand. For libm calls use SVML
Presence of 2 to 4 paths (SA)	+1 per path	Simplify control structure (might be difficult). Force the compiler to use masked instructions
Presence of more than 4 paths (SA)	+4 + 1 per path	Simplify control structure (might be very difficult)
Presence of reductions dependency cycles (SA)	+1 per reduction	Use appropriate compiler flags or directives (for example OMP SIMD reduction) for vectorization reductions
Presence of constant non unit stride data access (SA)	+2 per non unit stride data access	Use array restructuring. Perform loop interchange Using gather instructions will lower a bit the cost.
Presence of indirect access (SA)	+4per indirect access	Use array restructuring. Using gather instructions will lower the cost.
Non innermost loop (SA)	+2	Collapse loop with innermost one

ROADBLOCK ISSUES

VECTORIZATION		
Partial or unexisting vectorization (SA)	Sum of the roadblocks above	Get rid of the roadblocks (see above) Use pragma to force vectorization Check potential dependenciesbetween array access
Inefficient vectorization: use of shorter than available vector length (SA)	+2	Force compiler to use proper vector length CAUTION: use of 512 bits vectors could be more expensive than 256 bits on some processors Use intrinsics (costly and not portable)
Inefficient vectorization:use of masked instructions (SA)	+2	Simplify control structure
Inefficient vectorization: more than 10% of the vector loads instructions are unaligned (SA)	+2 per array	Align array access. When allocating arrays, don't forget to align them.

ADDITIONAL OPTIMIZATIONS (1)

ADDITIONAL OPTIMIZATIONS		
Presence of expensive FP instructions : div/sqrt, sin/cos, exp/log, etc…(SA + DT)	+4 per expensive operations	Perform hoisting Change algorithm Use SVML or proper numerical library. Perform value profiling (count the number of distinct input values).
Presence of expensive instructions : scatter/gather (SA)	+4 per scatter gather	Use array restructuring
Presence of special instructions executing on a single port (SA): typically all data restructuring instructions, expand, pack, unpack, etc…	+1 per special instruction	Simplify data access: try to get stride 1 access
Less than 10% of the FP ADD/SUB/MUL arithmetic operations are performed using FMA (SA)	+4	Reorganize arithmetic expressions to exhibit potential for FMA.

ADDITIONAL OPTIMIZATIONS (2)

Large loop body: over microp cache size	+2 per chunk of 50 instructions beyond the first chunk of 50	Perform loop splitting Reduce unrolling.
More than 20% of the loads are accessing the stack (SA).	+2	Perform loop splitting
Presence of a large number of scalar integer instructions: more than 1.1 x speedup when suppressing scalar integer instructions (SA)	+2	Simplify loop structure Perform loop splitting Perform unroll and jam
Bottleneck in the front end (SA)	+2	If loop size is very small (rare occurrences), perform unroll and jam If loop size is large, perform loop splitting
Low iteration count (DT)	+1	Perform full unroll Use compiler pragmas Use PGO/FDO compiler options Force compiler to use masked instructions
Ratio ORIG/DL1 greater than 3x (DT)	+2 per non unit stride or indirect access	Perform blocking Perform array restructuring
Highly variable Cycle per Iteration across loop instances (DT)	+4	Loop execution is sensitive to different contexts or/and call chain: try to determine such contexts and use loop specialization Try FDO/PGO compiler options
High secondary DTLB miss rate (DT)	+4	Perform array restructuring Perform blocking Activate huge pages.

PARALLELIZATION ISSUES

Performance Issue	Cost for removing perf issue	Optimization
More than 10% of the total execution time is spent in serial execution (DP).	+6	Increase parallelism: parallelize more loops Change algorithm
For a parallel loop/region construct, more than 10% of the execution time is spent in waiting (DT).	+2 per loop/region	Change scheduling using: in particular Test Guided. Try to improve load balancing at the algorithm level
For a parallel loop construct, more than 10% of the spent in pure synchronization primitives (barrier, locks) (DT)	+2 per special instruction	Change synchronization operations
For a parallel loop, highly variable behavior (synchronization time, waiting time etc…) across loop instances (DT)	+4	Loop execution is sensitive to different contexts or/and call chain: try to determine such contexts and use loop specialization
For a parallel loop, ratio of ORIG over S2L is greater than 1,2x .	+4	Try to reduce write on shared structures. Changes variable layout. Check for false sharing
For a parallel throughput execution of the code, performance loss with respect to ideal speedup is greater than 1,2X: high contention on shared resources (L3, RAM)	+8	Reorganize algorithm to decrease use of shared resources.

STRATEGIZER: DETAILED VIEW (1)

▼ Strategizer

[4 / 4] Enough time of the experiment time spent in analyzed loops (62.04%)

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% (25.17%), representing an hotspot for the application

[4 / 4] Enough time of the experiment time spent in analyzed innermost loops (61.66%)

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% (0.01%) is spend in BLAS1 operations

It could be more efficient to inline by hand BLAS1 operations

STRATEGIZER: DETAILED VIEW (2)

[3 / 3] Cumulative Outermost/In between loops coverage (0.38%) lower than cumulative innermost loop coverage (61.66%)

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

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

[2 / 2] Less than 10% (2.66%) is spend in BLAS2 operations

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