Optimizing beyond vectorization and parallelization : A case study on QMCPACK

Cédric Valensi, Emmanuel Oseret, William Jalby, Mathieu Tribalat, Kevin Camus, Youenn Lebras (UVSQ/ECR)
cedric.valensi@uvsq.fr , emmanuel.oseret@uvsq.fr, william.jalby@uvsq.fr, mathieu.tribalat@uvsq.fr, kevin.camus@uvsq.fr
Othman Bouizi, Jeongnim Kim, David Wong, David Kuck, Victor Lee (INTEL)
othman.bouizi@intel.com, jeongnim.kim@intel.com, david.c.wong@intel.com, david.kuck@intel.com, victor.w.lee@intel.com

http://www.maqao.org

With ECR, INTEL, CEA and UVSQ support

OUR VIEW: SYST = Arch + App CENTRIC VIEW

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

STARTING POINT: the user has at his disposal a 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, binary transforms.
- **How to apply transformations ??** Through the use of compiler directives or code restructuring (ASSIST) but under user responsibility

OUR VIEW: *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…

OUR VIEW:ADDITIONAL CHARACTERISTICS

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.

Hardware performance counters can be very helpful but hard to use

THE WRONG WAY OF USING HW COUNTERS: provide the user with useless, symptomatic not causal info/metrics such as Instructions per Cycle or Cache misses

THE GOOD WAY OF USING COUNTERS: aggregate counters to build meaningful metrics which can be related to source code such as cache and memory traffic

Our Approach : Analysis at 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 You Analyze Is What You Run**

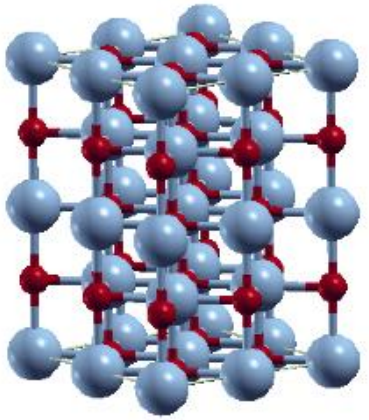
Talk organization

- Description of two key MAQAO modules (CQA and DECAN)
- Experimental results obtained on QMCPACK
- QMCPACK Optimization

EXPERIMENTAL SETUP

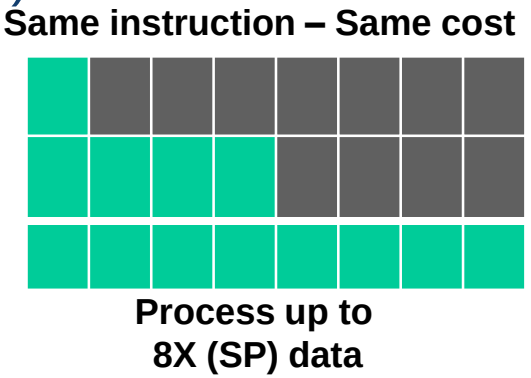
- TARGET APPLICATION QMCPACK:

an open-source, C++, high-performance electronic structure code that implements numerous Quantum Monte Carlo algorithms.
http://www.qmcpack.org



- NiO ECP Benchmark Suite
- COMPILER: Intel 19.0.1.144
- MKL: Intel 2019.1.144
- Unicore runs except when specified.
- HARDWARE:
  - Haswell: Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz
  - Skylake, Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz

- Relies on simplified CPU model
  - Allows faster analyses
- Machine model:
  - Execution pipeline
  - Port throughput
  - L1 data access additional L2/L3/RAM models)
  - Buffer size assumed infinite
- Key performance levers for core level efficiency:
  - Vectorizing
  - Avoiding high latency instructions if possible
  - Having the compiler generate efficient code
  - Reorganizing memory layout
- More realistic performance model with UFS: precise internal architecture including buffer size.



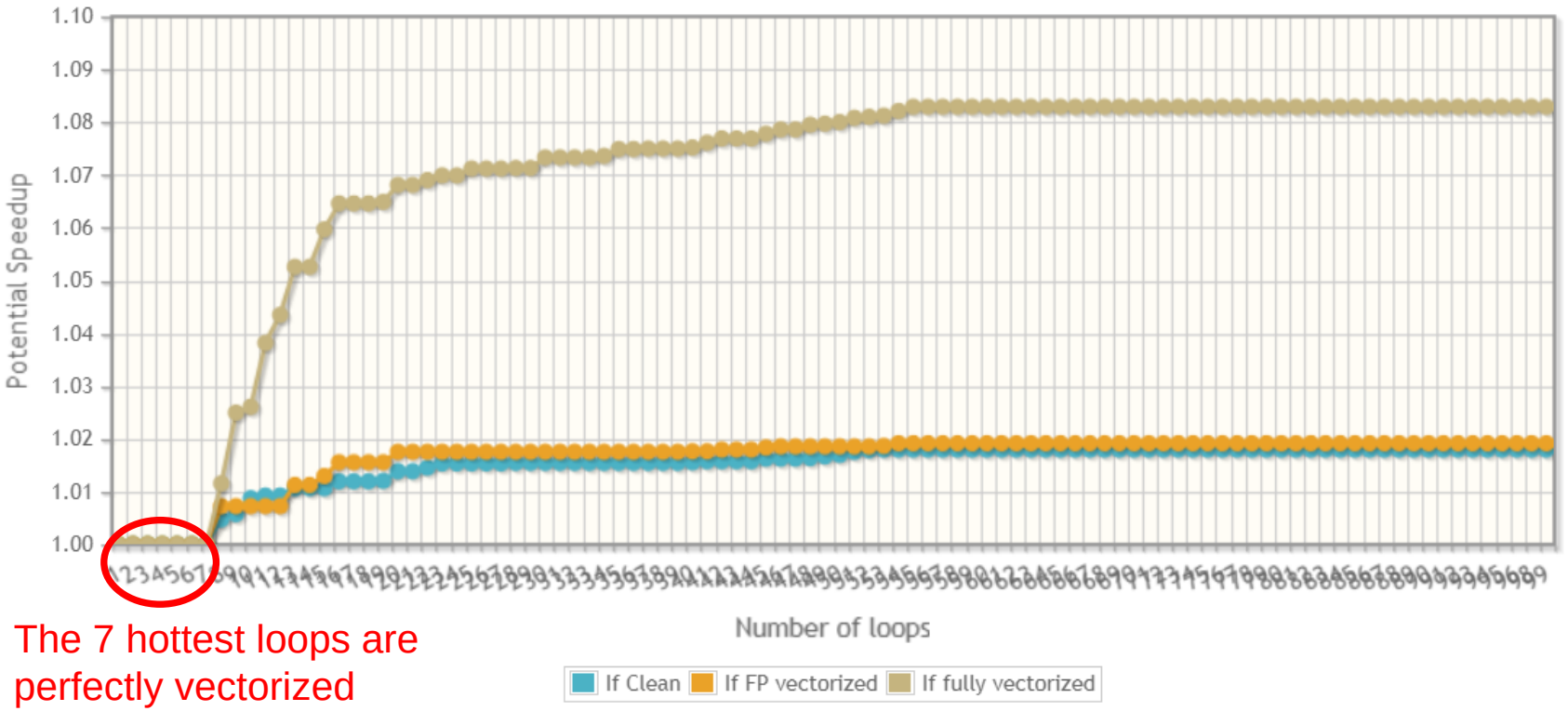
"WHAT IF" SCENARIOS: VECTORIZATION

- Code "Clean"
  - Generate an Assembly "Clean" 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.

ONE VIEW : GLOBAL APPLICATION VIEW (3)

On the X Axis, loops are sorted by coverage.
The Y AXIS represents cumulative speedup on the whole application

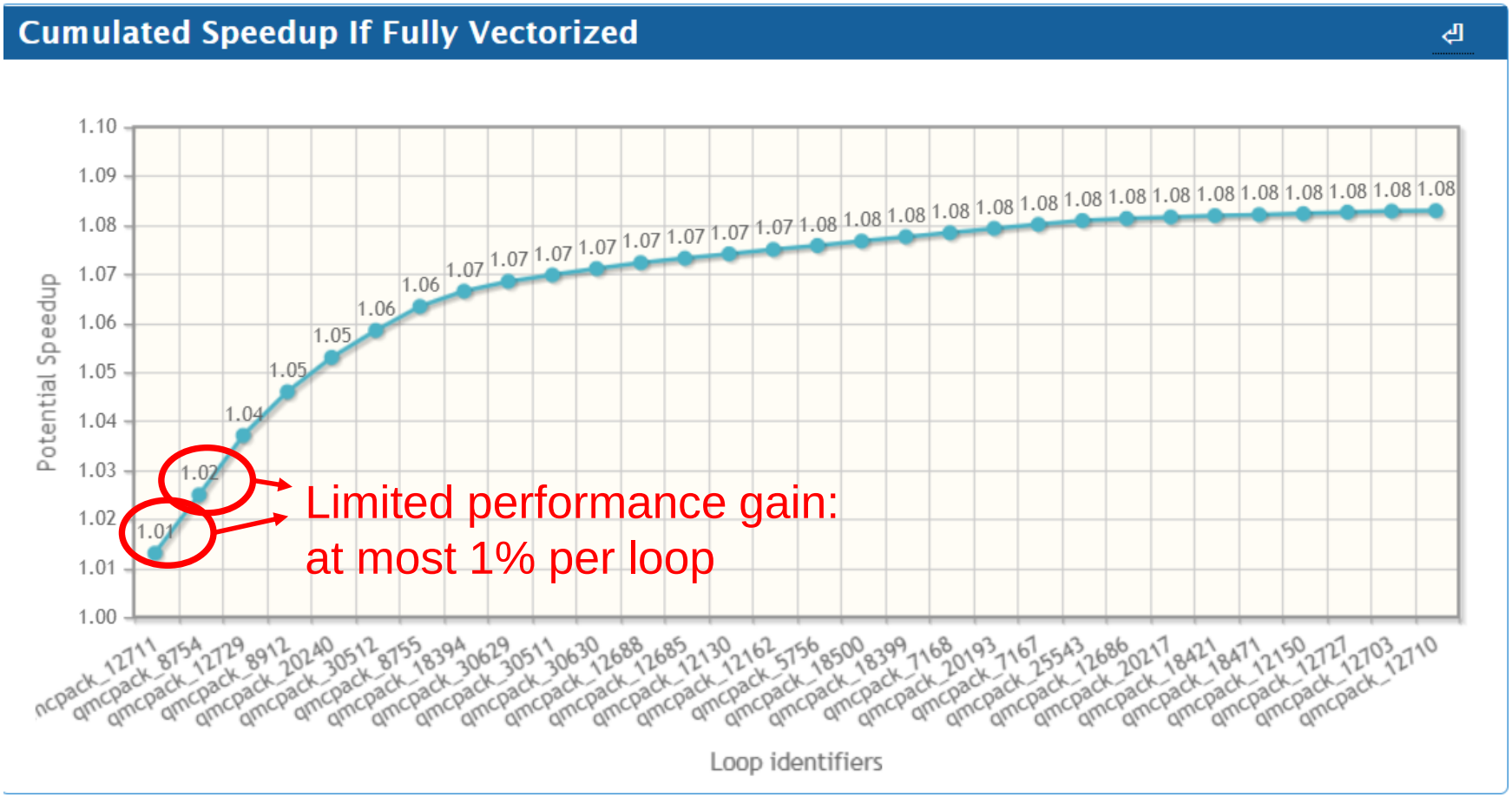
CQA Potential Speedups Summary



The 7 hottest loops are perfectly vectorized

DETAILED ANALYSIS OF WHAT IF FULL VECTORIZED

On the X Axis, loops are sorted by potential gain on the whole application.
The Y AXIS represents cumulative speedup on the whole application



MAQAO DECAN: Decremental Analysis

- Goal: modify the application to
  - Identify causes of bottlenecks
  - Measure associated ROI (Return On Investment)

- Differential analysis:
  - Target innermost loops
  - Transform loops: generate different binary variants of the original loops
  - Measure and compare original loop performance with modified variants

- Transformations
  - Remove or modify groups of instructions
  - Target memory accesses or computations
  - Binary variants generate “wrong results”: we don’t care, and writes systematically protected to preserve memory state

MAQAO DECAN Transformations

Typical transformations/variants:

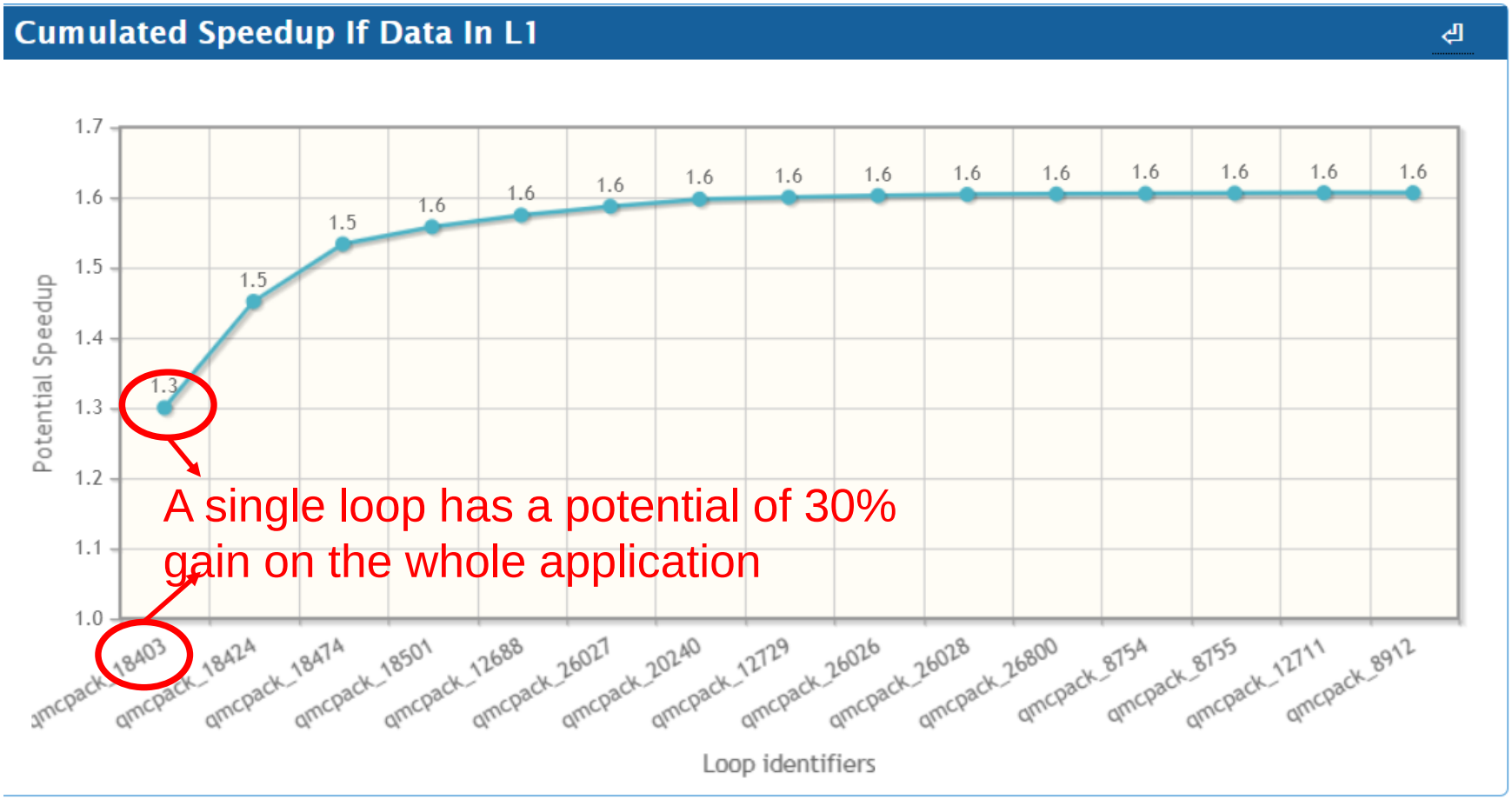
- **FP:** only FP arithmetic instructions are preserved
  - => loads and stores are removed
- **LS:** only loads and stores are preserved
  - => compute instructions are removed

Comparing T(FP) (Time spent in FP variant) and T(LS) (Time spent in LS variant) allows us to quantify how much a loop is CPU bound versus data access bound

- **DL1:** memory references replaced with constant memory address
  - => for loops, data now accessed from L1: precise impact of perfect blocking
- **PSOR/PVLOR:** add software prefetch instructions on vector loads/stores
- **S2L:** stores are replaced by loads
  - => for loops, no more coherency actions so evaluation of coherency cost blocking

DETAILED ANALYSIS OF WHAT IF L1 BLOCKING

On the X Axis, loops are sorted by potential gain on the whole application.
The Y AXIS represents cumulative speedup on the whole application



ONE VIEW: EXPERT VIEW, DECAN VARIANTS (1)

COLOR CODE: RED : duration less than 250 Cycles, ORANGE: Duration less than 500 cycles

STABILITY ANALYSIS: STA = (Median – Min)/Min

Expert Summary

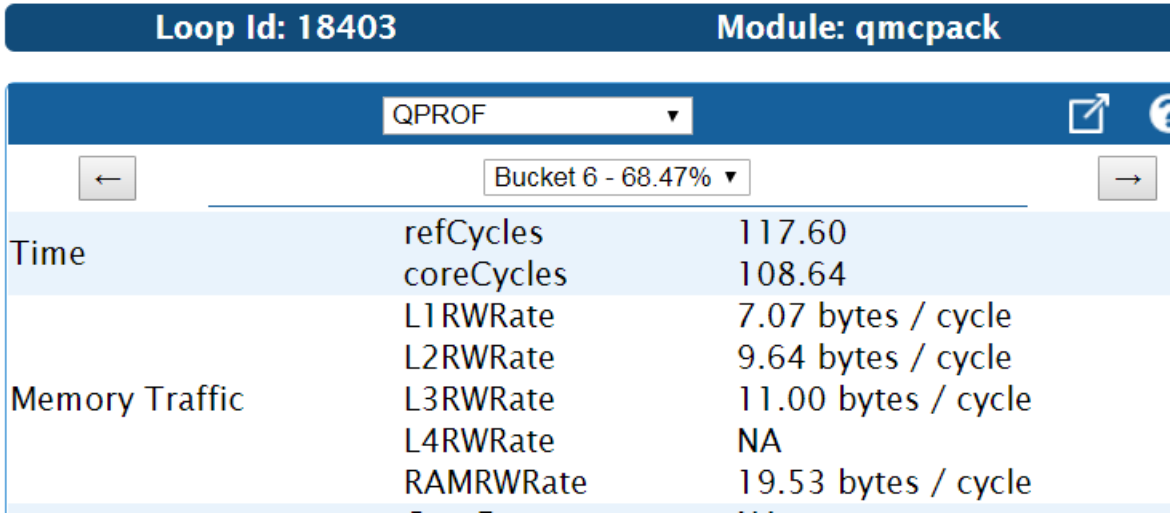
☑ Analysis ☐ CQA speedup if clean ☐ CQA speedup if FP arith vectorized ☐ CQA speedup if fully vectorized ☑ Number of paths ☑ ORIG / DL1
☐ Saturation ratio (MAX(DL1,LS)/REF) ☐ Saturation ☐ FP/CQA(FP) ☐ DL1/CQA(DL1) ☐ FP/LS ☐ Frequency Impact ☑ ORIG (cycles per iteration) ☑ STA (ORIG)
☑ REF (cycles per iteration) ☑ STA (REF) ☑ FP (cycles per iteration) ☑ STA (FP) ☑ LS (cycles per iteration) ☑ STA (LS) ☑ DL1 (cycles per iteration) ☑ STA (DL1)
☑ FES (cycles per iteration) ☑ STA (FES) ☑ CQA cycles ☑ CQA cycles if clean ☑ CQA cycles if FP arith vectorized ☑ CQA cycles if fully vectorized ☑ Iteration count
☑ Function ☑ Source ☑ Nb FP_ADD / CPI ☑ Nb FP_MUL / CPI ☑ CAP(FP) ☑ BW(FP) ☑ SAT(FP) ☑ CAP(L1R) ☑ BW(L1R) ☑ SAT(L1R) ☑ CAP(L1W)
☑ BW(L1W) ☑ SAT(L1W) ☑ CAP(L2) ☑ BW(L2) ☑ SAT(L2) ☑ CAP(L3) ☑ BW(L3) ☑ SAT(L3) ☑ CAP(RAM_R) ☑ CAP(RAM_W) ☑ Select all

ID	Module	Coverage (% app. time)	Analysis	Number of paths	ORIG / DL1	ORIG (cycles per iteration)	STA (ORIG)	REF (cycles per iteration)	STA (REF)	FP (cycles per iteration)	STA (FP)	LS (cycles per iteration)	STA (LS)	DL1 (cycles per iteration)	STA (DL1)	FES (cycles per iteration)	STA (FES)
▶ Loop 18403	binary	26.3	RAM bound	1	8.61	81.92	1.03	77.52	0.27	6.56	0.24	71.84	0.43	9.52	0.18	9.04	0.30
▶ Loop 26027	binary	11.86	Balanced workload (back-end starvation)	128	1.00	149.88	0.14	156.83	0.09	156.58	0.11	148.06	0.16	150.23	0.11	135.50	0.08
▶ Loop 18424	binary	10.64	RAM bound	1	3.74	71.33	0.30	77.61	0.23	32.16	0.02	80.12	0.34	19.10	0.05	17.18	0.02
▶ Loop 18474	binary	4.79	RAM bound	1	4.04	75.96	0.35	80.67	0.34	32.12	0.01	123.61	1.06	18.82	0.02	17.33	0.04
▶ Loop 26026	binary	3.04	Balanced workload (back-end starvation)	128	0.99	173.71	0.14	181.81	0.12	175.88	0.06	173.17	0.07	176.04	0.14	163.92	0.07
▶ Loop 26028	binary	2.66	Balanced workload (back-end starvation)	128	0.97	171.54	0.19	180.85	0.11	174.73	0.06	170.67	0.07	176.56	0.08	163.50	0.14
▶ Loop 18501	binary	1.4	RAM bound	1	4.43	332.00	0.12	261.18	0.16	79.00	0.01	240.45	0.10	74.91	0.02	65.55	0.00

Indicates major bottleneck: CPU bound versus data access bound

From measurements of specific HW events, QPLOT

- Builds the traffic for each memory level (L1, L2, L3, RAM)
- Compares measurement with static info gathered by CQA
- Computes and plots Data rate access versus Mflops rate. Generates Intensity (Bytes per flop) values for each codelet and for each memory level
- Performs codelet classification according to intensity to drive/suggest potential code optimization.



QMCPACK: OUR ASSESSMENT

qmcpack

Help is available by moving the cursor above any ? symbol or by checking MAC

Global Metrics

Total Time (s)		25
Time in loops (%)		76.53
Time in innermost loops (%)		75.07
Compilation Options		OK
Flow Complexity		1.40
Array Access Efficiency (%)		84.69
Clean	Potential Speedup	1.02
	Nb Loops to get 80%	11
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	5
Fully Vectorised	Potential Speedup	1.08
	Nb Loops to get 80%	8
Data In L1 Cache	Potential Speedup	1.61
	Nb Loops to get 80%	3

Control Flow needs optimization

Very good quality of the generated code

Excellent Vectorization: limited potential performance gain

Data access needs optimization

QMCPACK: LOOKING AT BRANCHES

Expert Summary

Analysis | CQA speedup if clean | CQA speedup if FP arith vectorized | CQA speedup if fully vectorized | Number of paths | ORIG / DL
Saturation ratio (MAX(DL1,LS)/REF) | Saturation | FP/CQA(FP) | DL1/CQA(DL1) | FP/LS | Frequency Impact | ORIG (cycles per iterat
REF (cycles per iteration) | STA (REF) | FP (cycles per iteration) | STA (FP) | LS (cycles per iteration) | STA (LS) | DL1 (cycles per itera
FES (cycles per iteration) | STA (FES) | CQA cycles | CQA cycles if clean | CQA cycles if FP arith vectorized | CQA cycles if fully vectoriz
Function | Source | Nb FP_ADD / CPI | Nb FP_MUL / CPI | CAP(FP) | BW(FP) | SAT(FP) | CAP(L1R) | BW(L1R) | SAT(L1R)
BW(L1W) | SAT(L1W) | CAP(L2) | BW(L2) | SAT(L2) | CAP(L3) | BW(L3) | SAT(L3) | CAP(RAM_R) | CAP(RAM_W) | Sele

ID	Module	Coverage (% app. time)	Analysis	CQA speedup if clean	CQA speedup if FP arith vectorized	CQA speedup if fully vectorized	Number of paths	ORIG / DL1	FP/CQA(FP)	DL1/CQA(DL1)	FP/LS	Fr
▶ Loop 18403	binary	26.3	RAM bound	1.00	1.00	1.00	1	8.61	0.82	1.19	0.09	
▶ Loop 26027	binary	11.86	Balanced workload (back-end starvation)	1.00	1.00	1.00	128	1.00	2.65	2.78	1.06	
▶ Loop 18424	binary	10.64	RAM bound	1.00	1.00	1.00	1	3.74	1.69	1.01	0.40	
▶ Loop 18474	binary	4.79	RAM bound	1.00	1.00	1.00	1	4.04	1.69	0.99	0.26	
▶ Loop 26026	binary	3.04	Balanced workload (back-end starvation)	1.00	1.00	1.00	128	0.99	3.01	3.26	1.02	
▶ Loop 26028	binary	2.66	Balanced workload (back-end starvation)	1.00	1.00	1.00	128	0.97	2.96	3.27	1.02	
▶ Loop 18501	binary	1.44	RAM bound	1.02	1.00	1.01	1	4.43	1.09	1.61	0.33	

ONE View_LiPara....pptx | MAQAO_Cenaero....pptx

Loops with a very large number of paths: issues with control flow/branch

OPTIMIZING BRANCHES

ISSUE: CQA detected a large number of paths in a few loops. These loop were perfectly vectorized but the compiler generated a very complex control flow around the vector instructions. The source code contained a loop nest (7 iterations) annotated with a full unroll directive. Unfortunately, the compiler did not use this directive and generated a very complex control flow.

SOLUTION (Variant called FU: Full Unroll): Instead of relying on a compiler directive, fully unroll by hand the loop nest that caused trouble for the compiler.

PERFORMANCE GAIN:

- On Haswell , between 1,11x and 1,51x speedup at loop level and between 1,05x and 1,08x at application level
- On Skylake, between 1,47x and 1,75x speedup at loop level and between 1,07x and 1,09x at application level

ISSUE: CQA detected: large number of stack access, unbalanced port usage due to the presence of “special” instructions, partial vectorization. In fact the loop body was too large and overwhelmed compiler optimization capacities.

SOLUTION (Variant called SPLIT): Split the loop to reduce the loop complexity to a level which could be managed by the compiler.

PERFORMANCE GAIN:

- On Haswell , between 1,13x and 1,35x speedup at loop level and 1,01x at full application level
- On Skylake, between 1,25x and 1,58x speedup at loop level and 1,01x at full application level

OPTIMIZING DATA ACCESS

ISSUE: QPLOT detected: large amount of L1 traffic. DECAN indicated strong potential benefit for traffic reduction.
SOLUTION (Variant called FUME): A surrounding loop provided some data reuse which was exploited by Unroll and Merge.

PERFORMANCE GAIN:

- On Haswell, between 1,6x and 2,51x speedup at loop level and 1,06x and 1,08x at application level.

- On Skylake, between 2,6x and 2,9x speedup at loop level and around 1,2x at application level

AGGREGATING ALL OPTIMIZATIONS

Machine	Data Set	ORIG	FU	FU + SPLIT	FU + SPLIT + FUME
Haswell	Small	1	1.08	1.09	1.17
Haswell	Medium	1	1.05	1.06	1.12
Skylake	Small	1	1.09	1.10	1.33
Skylake	Medium	1	1.07	1.08	1.29

OPTIMIZING FURTHER

Using DECAN to explore impact of software prefetch instructions.

PERFORMANCE GAIN:

On Haswell, additional 4% performance gain.
On Skylake, additionnal 3% performance gain.

ONE VIEW: ANALYZING MPI + OPENMP

PERFORM AUTOMATICALLY MULTIPLE RUNS: variation on the number of threads and computes efficiency.
STRONG SCALING AND WEAK SCALING
QMCPACK: Monte Carlo nature of the algorithm induces "Embarrassingly Parallel" code. Efficiency loss is due to contention.

Loops Index

☑ Coverage (%) ☑ Level ☐ Time (s) ☐ Vectorization Ratio (%) ☐ Speedup If Clean ☐ Speedup If FP Vectorized ☐ Speedup If Fully Vectorized ☑ (1-2) Efficiency
☐ (1-2) Potential Speed-Up (%) ☑ (1-4) Efficiency ☐ (1-4) Potential Speed-Up (%) ☑ (1-8) Efficiency ☐ (1-8) Potential Speed-Up (%) ☑ (1-16) Efficiency
☐ (1-16) Potential Speed-Up (%) ☑ (1-32) Efficiency ☐ (1-32) Potential Speed-Up (%) ☑ (1-52) Efficiency ☐ (1-52) Potential Speed-Up (%) ☑ Select all

Loop id	Source Location	Source Function	Coverage (%)	Level	(1-2) Efficiency	(1-4) Efficiency	(1-8) Efficiency	(1-16) Efficiency	(1-32) Efficiency	(1-52) Efficiency
18403	qmcpack:MultiBsplineValue.hpp:56-57	qmcplusplus::BsplineSet >::evaluate	23.96	Innermost	1	1.03	0.97	0.83	0.51	0.31
18424	qmcpack:MultiBsplineVGLH.hpp:187-207	qmcplusplus::BsplineSet >::evaluate	8.85	Innermost	1	0.99	0.98	0.95	0.84	0.6
26027	qmcpack:ParticleBConds3DSoa.h:231-310	qmcplusplus::SoaDistanceTableAA::moveOnSphere	7.05	Single	1	0.99	0.97	0.97	0.96	0.88
18474	qmcpack:MultiBsplineVGLH.hpp:187-207	qmcplusplus::BsplineSet >::evaluate_notranspose	4.19	Innermost	1	1	0.98	0.91	0.66	0.43
26026	qmcpack:ParticleBConds3DSoa.h:231-310	qmcplusplus::SoaDistanceTableAA::evaluate	3.67	Single	1	0.99	0.97	0.94	0.85	0.71
26028	qmcpack:ParticleBConds3DSoa.h:231-310	qmcplusplus::SoaDistanceTableAA::move	2.74	Single	1	0.98	0.98	0.99	0.97	0.88
30512	qmcpack::	__intel_avx_rep_memset	1.32	Single	1	1.02	0.95	0.91	0.85	0.58
18501	qmcpack:SplineC2RAdoptor.h:325-373	void qmcplusplus::SplineC2RSoA::assign_vgl >, qmcplusplus::Vector , std::allocator > > >	1.31	Single	1	1.01	0.98	0.99	0.92	0.75
8754	qmcpack:stl_vector.h:798-798	qmcplusplus::CoulombPBCAA::evalSR	1.21	Innermost	1	1.03	1.02	0.99	1.01	0.99
12711	qmcpack:BsplineFunctor.h:690-695	qmcplusplus::J2OrbitalSoA >::ratioGrad	0.93	Innermost	1	1.01	0.99	1.01	0.99	0.9
12688	qmcpack:BsplineFunctor.h:639-643	qmcplusplus::J2OrbitalSoA >::ratio	0.74	Innermost	1	1	1.01	0.96	0.96	0.87
8912	qmcpack:CoulombPBCAB.cp	qmcplusplus::CoulombPBCAB::evalSR	0.72	Innermost	1	0.97	0.97	0.96	0.97	0.94

WRAP UP

- 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)
- Main lesson learned through QMCPACK optimization: it is of major importance of fully understand the full code structure not only vectorization/parallelization.
- More analyses to be done : in particular, more what if scenarios for studying parallelism (MPI, OpenMP), branch impact,

Thanks for your attention!

Questions ?