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
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:

- 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.

**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
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
MAQAO CQA Main Concepts

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

The 7 hottest loops are perfectly 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.

Limited performance gain: at most 1% per loop.
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
Typical transformations/variants:

- **FP:** only FP arithmetic instructions are preserved
  - \( \Rightarrow \) loads and stores are removed

- **LS:** only loads and stores are preserved
  - \( \Rightarrow \) 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
  - \( \Rightarrow \) 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
  - \( \Rightarrow \) for loops, no more coherency actions so evaluation of coherency cost 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.

A single loop has a potential of 30% gain on the whole application.
COLOR CODE: **RED** : duration less than 250 Cycles, **ORANGE** : Duration less than 500 cycles

STABILITY ANALYSIS: $STA = \frac{(Median - Min)}{Min}$

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.

<table>
<thead>
<tr>
<th>Loop Id: 18403</th>
<th>Module: qmcpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPROF</td>
<td></td>
</tr>
<tr>
<td>Bucket 6 - 68.47%</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>refCycles</td>
<td>117.60</td>
</tr>
<tr>
<td>coreCycles</td>
<td>108.64</td>
</tr>
<tr>
<td>L1RWRate</td>
<td>7.07 bytes / cycle</td>
</tr>
<tr>
<td>L2RWRate</td>
<td>9.64 bytes / cycle</td>
</tr>
<tr>
<td>L3RWRate</td>
<td>11.00 bytes / cycle</td>
</tr>
<tr>
<td>L4RWRate</td>
<td>NA</td>
</tr>
<tr>
<td>RAMRWRate</td>
<td>19.53 bytes / cycle</td>
</tr>
<tr>
<td>Memory Traffic</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>refCycles</td>
<td>117.60</td>
</tr>
<tr>
<td>coreCycles</td>
<td>108.64</td>
</tr>
<tr>
<td>L1RWRate</td>
<td>7.07 bytes / cycle</td>
</tr>
<tr>
<td>L2RWRate</td>
<td>9.64 bytes / cycle</td>
</tr>
<tr>
<td>L3RWRate</td>
<td>11.00 bytes / cycle</td>
</tr>
<tr>
<td>L4RWRate</td>
<td>NA</td>
</tr>
<tr>
<td>RAMRWRate</td>
<td>19.53 bytes / cycle</td>
</tr>
</tbody>
</table>

MQAQO Performance Analysis and Optimization Tool
## QMCPACK: OUR ASSESSMENT

**MAQAO Performance Analysis and Optimization Tool**

### Global Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time (s)</td>
<td>25</td>
</tr>
<tr>
<td>Time in loops (%)</td>
<td>76.53</td>
</tr>
<tr>
<td>Time in innermost loops (%)</td>
<td>75.07</td>
</tr>
<tr>
<td>Compilation Options</td>
<td>OK</td>
</tr>
<tr>
<td>Flow Complexity</td>
<td>1.40</td>
</tr>
<tr>
<td>Array Access Efficiency (%)</td>
<td>84.69</td>
</tr>
</tbody>
</table>

**Potential Speedup**

- **Clean**
  - Nb Loops to get 80%: 11
  - Potential Speedup: 1.02
- **FP Vectorised**
  - Nb Loops to get 80%: 5
  - Potential Speedup: 1.02
- **Fully Vectorised**
  - Nb Loops to get 80%: 8
  - Potential Speedup: 1.08
- **Data In L1 Cache**
  - Nb Loops to get 80%: 3
  - Potential Speedup: 1.61

**Notes**

- **Control Flow needs optimization**
- **Very good quality of the generated code**
- **Excellent Vectorization: limited potential performance gain**
- **Data access needs optimization**
Loops with a very large number of paths: issues with control flow/branch

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

OPTIMIZING FURTHER
Using DECAN to explore impact of software prefetch instructions.

PERFORMANCE GAIN:
On Haswell, additional 4% performance gain.
On Skylake, additional 3% performance gain.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Data Set</th>
<th>ORIG</th>
<th>FU</th>
<th>FU + SPLIT</th>
<th>FU + SPLIT + FUME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell</td>
<td>Small</td>
<td>1</td>
<td>1.08</td>
<td>1.09</td>
<td>1.17</td>
</tr>
<tr>
<td>Haswell</td>
<td>Medium</td>
<td>1</td>
<td>1.05</td>
<td>1.06</td>
<td>1.12</td>
</tr>
<tr>
<td>Skylake</td>
<td>Small</td>
<td>1</td>
<td>1.09</td>
<td>1.10</td>
<td>1.33</td>
</tr>
<tr>
<td>Skylake</td>
<td>Medium</td>
<td>1</td>
<td>1.07</td>
<td>1.08</td>
<td>1.29</td>
</tr>
</tbody>
</table>
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.
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 ?