MAQAO hands-on exercises

Perf: generic profiler
Perf/MPI: Lightweight MPI oriented profiler
CQA: code quality analyzer

Setup

Recompile NPB-MZ with -dynamic if using cray compiler

```
#---------------------------------------------------------------------------
# The fortran compiler used for hybrid MPI programs
#---------------------------------------------------------------------------
MPIF77 = ftn -dynamic
```

Copy handson material

```
> cp ~xhpar/MAQAO/handson/material.tar.bz2 $HOME
```

Untar archive

```
> tar xf material.tar.bz2
> cd MAQAO_handson
```

MAQAO Perf hands-on exercises

Andres S. CHARIF-RUBIAL

Setup

Reserve 2 nodes

```
> qsub -I -l nodes=2:ppn=24 -l walltime=01:00:00 -q R_vihps02
```

Load MAQAO environment

```
> module use ~xhpar/MAQAO/module/
> module load maqao
```

Using MAQAO Perf

Collect data

```
OMP_NUM_THREADS=6 aprun -n 8 -d 6 maqao perf t=TX xp=tx ./bt-mz_C.8.dyn
```

Analyze collected data

- **Text output**

```
> maqao perf xp=tx d=SFX
```

- **HTML GUI**

```
> maqao perf xp=tx d=SFX of=html
```

Copy tx/html folder to your local station and open html/index.html with you favorite browser.

Using MAQAO Perf

A copy of the output is located in Perf folder in case you experience a problem

Now follow live demo/comments

MAQAO PerfMPI hands-on exercises

Andres S. CHARIF-RUBIAL

Setup

Reserve 2 nodes

```
> qsub -I -l nodes=2:ppn=24 -l walltime=01:00:00 -q R_vihps02
```

Load MAQAO environment

```
> module use ~xhpar/MAQAO/module/
> module load maqao
```

Using MAQAO PerfMPI

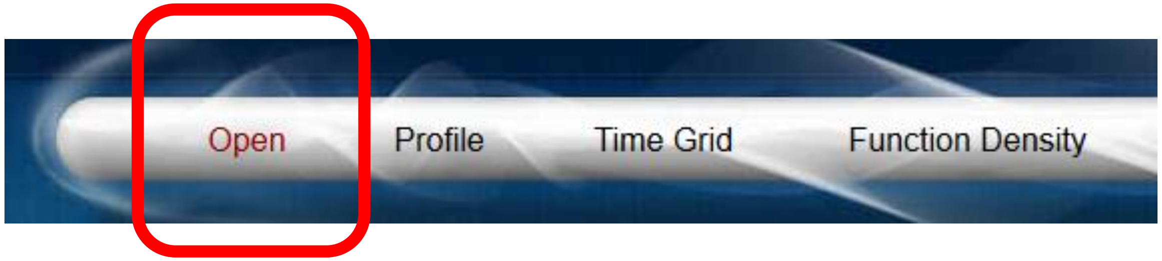
Collect data

```
LD_PRELOAD=$MAQAOMPI OMP_NUM_THREADS=6 aprun -n 8 -d 6 ./bt-mz_C.8.dyn
```

Analyze collected data

Copy the generated MPI_Profile.js to your local station

Open PerfMPI/MPI_GUI/MPI.html with your favorite browser
Then click on the « Open » button and select the MPI_Profile.js file



Using MAQAO PerfMPI

A copy of the output is located in the PerfMPI/ folder in case you experience a problem.
Other examples are also available in the PerfMPI/MPI_examples folder from the handson material

Now follow live demo/comments

MAQAO / CQA hands-on exercises

Emmanuel OSERET

Setup

Reserve 1 node

```
> qsub -I -l nodes=1:ppn=24 -l walltime=01:00:00 -q R_vihps02
```

Load MAQAO environment

```
> module use ~xhpar/MAQAO/module/
> module load maqao
```

Setup

Load gcc (*)

```
> module load gcc
```

Untar the CQA handson tarball

```
> tar xf ~/MAQAO_handson/maqao_cqa_matmul.tgz
> cd matmul
```

* We explicitely load gcc because use module swap PregEnv-gnu has an impact on the default flags of the compiler

Original code

```
void kernel0 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
  int i, j, k;

  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      c[i][j] = 0.0f;
      for (k=0; k<n; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

"Naïve" dense matrix multiply implementation in C

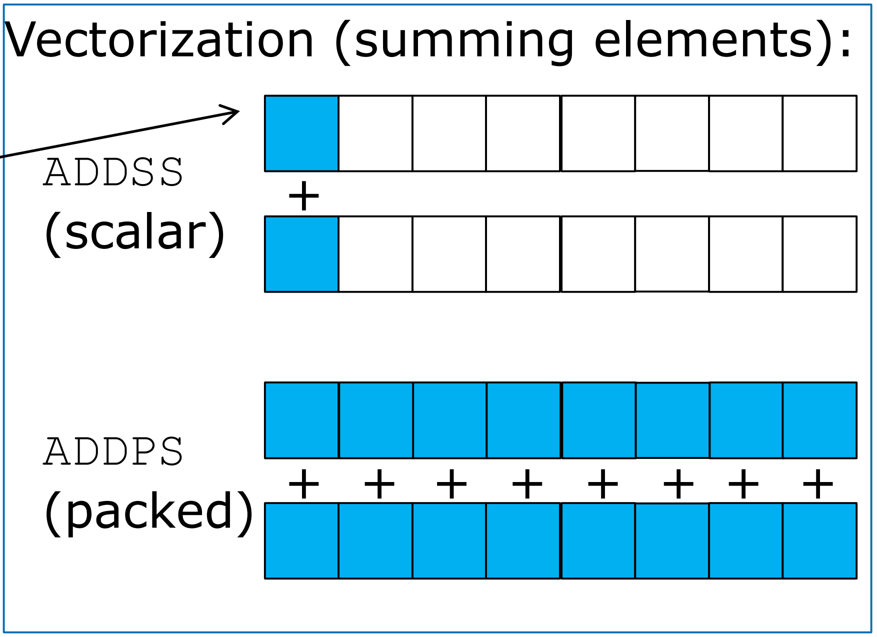
Compiling, running and analyzing kernel0 in –O3

```
> make OPTFLAGS=-O3 KERNEL=0
> ./matmul 100 1000
Cycles per FMA: 3.02
> maqao cqa matmul fct=kernel0 [of=html]
```

NB: the usual way to use CQA consists in finding IDs of hot loops with the MAQAO profiler and forwarding them to CQA (loop=17,42…).
To simplify this hands-on, we will bypass profiling and directly requesting CQA to analyze all innermost loops in functions (max 2-3 loops/function for this hands-on).

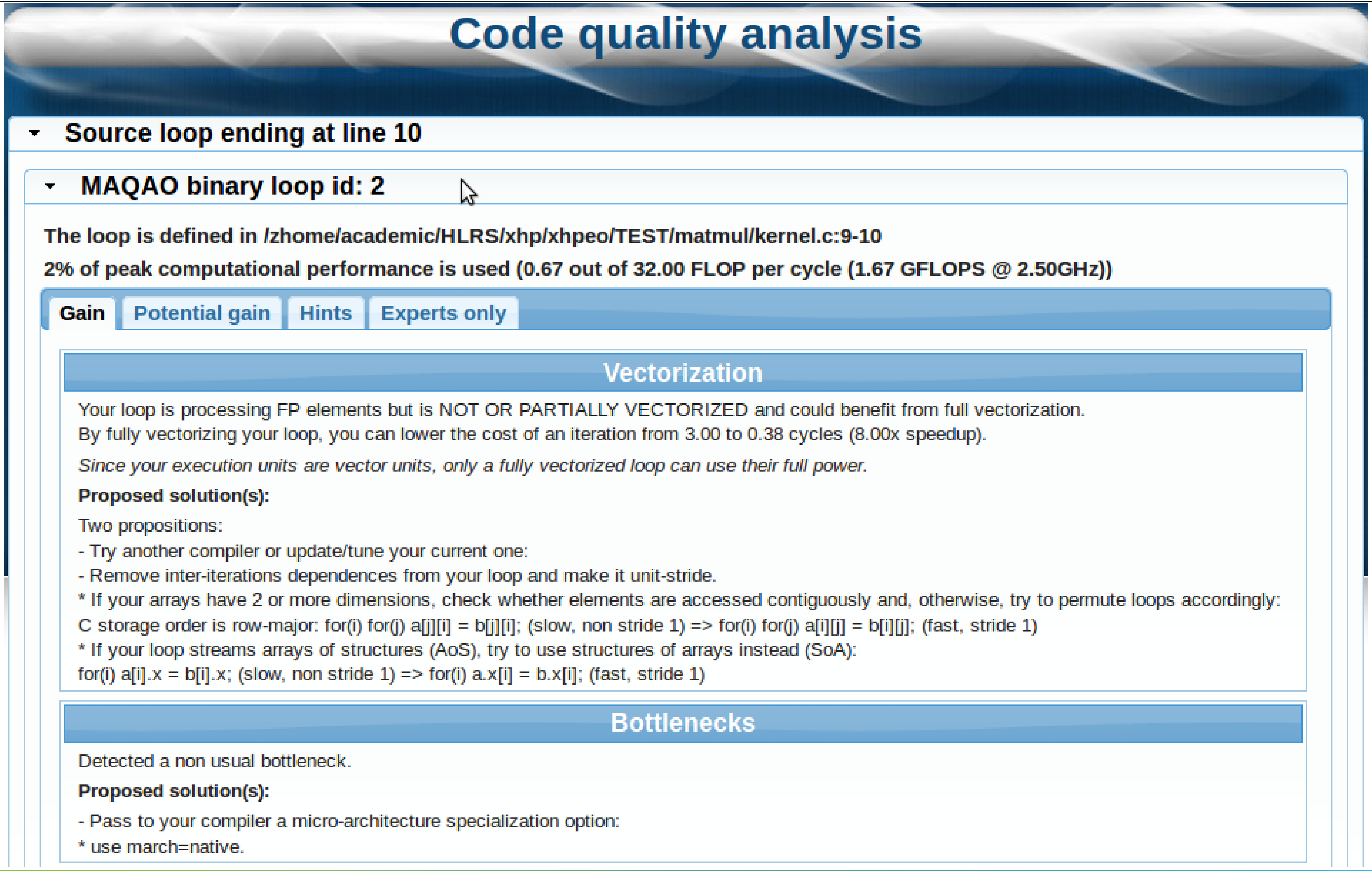
CQA output for kernel0 (from the "gain" confidence level)

```
Vectorization
-------------
(…) By fully vectorizing your loop,
you can lower the cost of an iteration
from 3.00 to 0.38 cycles (8.00x
speedup). (…)
 - Remove inter-iterations dependences
from your loop and make it unit-
stride.
  * If your arrays have 2 or more
dimensions, check whether elements are
accessed contiguously and, otherwise,
try to permute loops accordingly:
C storage order is row-major: for(i)
a[j][i] = b[j][i]; (slow, non stide 1)
=> for(i) for(j) a[i][j] = b[i][j];
(fast, stride 1)
  * If your loop streams arrays of
structures (AoS), try to use (…) SoA
```



- Accesses are not contiguous => let's permute k and j loops
- No structures here…

CQA output for kernel0 (from the "gain" confidence level)



Code quality analysis

Source loop ending at line 10

MAQAO binary loop id: 2

The loop is defined in /zhome/academic/HLRS/xhp/xhpeo/TEST/matmul/kernel.c:9-10

2% of peak computational performance is used (0.67 out of 32.00 FLOP per cycle (1.67 GFLOPS @ 2.50GHz))

Vectorization

Your loop is processing FP elements but is NOT OR PARTIALLY VECTORIZED and could benefit from full vectorization.
By fully vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.38 cycles (8.00x speedup).

Since your execution units are vector units, only a fully vectorized loop can use their full power.

Proposed solution(s):

Two propositions:
- Try another compiler or update/tune your current one:
- Remove inter-iterations dependences from your loop and make it unit-stride.
* If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:
C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
* If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA):
for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

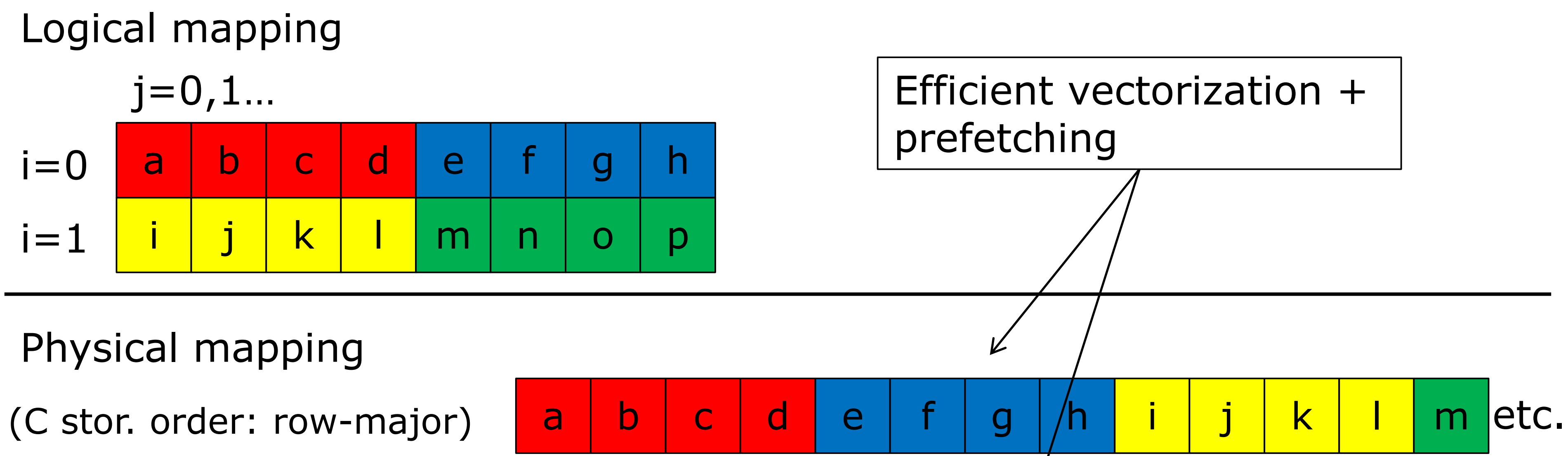
Bottlenecks

Detected a non usual bottleneck.

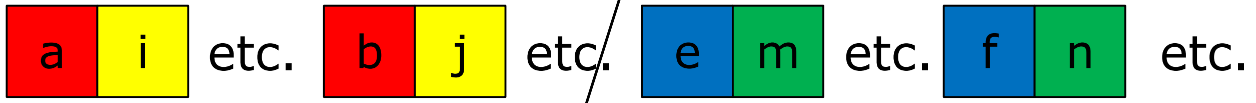
Proposed solution(s):

- Pass to your compiler a micro-architecture specialization option:
* use march=native.

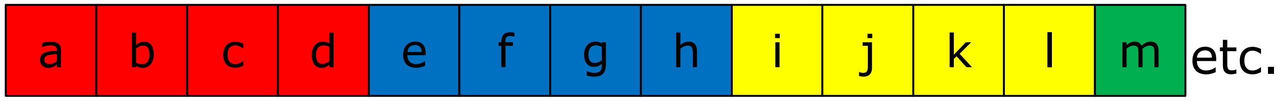
Impact of loop permutation on data access


Logical mapping
j=0,1…
i=0
i=1
a b c d e f g h
i j k l m n o p
Efficient vectorization + prefetching
Physical mapping
(C stor. order: row-major)
a b c d e f g h i j k l m etc.

```
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    f(a[i][j]);
```



```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    f(a[i][j]);
```



Removing inter-iteration dependences and getting stride 1 by permuting loops on j and k

```
void kernel1 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
  int i, j, k;

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      c[i][j] = 0.0f;

    for (k=0; k<n; k++)
      for (j=0; j<n; j++)
        c[i][j] += a[i][k] * b[k][j];
  }
}
```

kernel1: loop interchange

```
> make clean
> make OPTFLAGS=-O3 KERNEL=1
> ./matmul 100 1000
Cycles per FMA: 0.68
> maqao cqa matmul fct=kernel1 --confidence-levels=gain,potential,hint
```

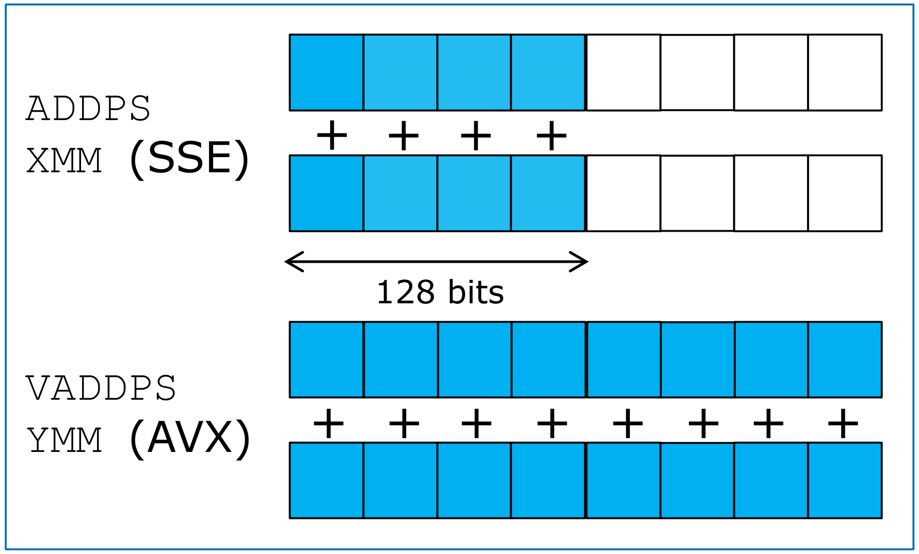
CQA output for kernel1

```
Vectorization status
--------------------
Your loop is fully vectorized (…)
but on 50% vector length.

Vectorization
-------------
 - Pass to your compiler a micro-
architecture specialization option:
   * use march=native
 - Use vector aligned instructions…

FMA
---
Presence of both ADD/SUB and MUL
operations.
 - Pass to your compiler a micro-
architecture specialization option…
 - Try to change order in which…
```

- Let's add –march=native to OPTFLAGS

Impacts of architecture specialization: vectorization and FMA

- Vectorization
  - SSE instructions (SIMD 128 bits) used on a processor supporting AVX ones (SIMD 256 bits)
  - => 50% efficiency loss



- FMA
  - Fused Multiply-Add (A+BC)
  - Intel architectures: supported on MIC/KNC and Xeon starting from Haswell (hornet)

```
# A = A + BC

VMULPS <B>,<C>,%XMM0
VADDPS <A>,%XMM0,<A>
# can be replaced with
something like:
VFMADD312PS <B>,<C>,<A>
```

Kernel1 + -march=native

```
> make clean
> make OPTFLAGS="-O3 -march=native" KERNEL=1
> ./matmul 100 1000
Cycles per FMA: 0.58
> maqao cqa matmul fct=kernel1 --confidence-levels=gain,hint
```

CQA output for kernel1 (using gain and hint levels)

```
Vectorization status
--------------------
Your loop is fully vectorized (…) but
on 75% vector length.

Vectorization
-------------
 - Pass to your compiler a micro-
architecture specialization option:
   * use march=native
 - Use vector aligned instructions:
   1) align your arrays on 32 bytes
boundaries,
   2) inform your compiler that your
arrays are vector aligned:
     * use the
__builtin_assume_aligned built-in
```

- Let’s switch to the next proposal: vector aligned instructions

Aligning vector accesses in driver + assuming them in kernel

```
int main (…) {
   (…)
#if KERNEL==2
   puts (« driver.c: Using
posix_memalign instead of malloc »);
   posix_memalign ((void **) &a, 32,
size_in_bytes);
   posix_memalign ((void **) &b, 32,
size_in_bytes);
   posix_memalign ((void **) &c, 32,
size_in_bytes);
#else
   a = malloc (size_in_bytes);
   b = malloc (size_in_bytes);
   c = malloc (size_in_bytes);
#endif
   (…)
}
```

```
void kernel2 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
  int i, j, k;

  for (i=0; i<n; i++) {
    float *ci =
__builtin_assume_aligned (c[i], 32);
    for (j=0; j<n; j++)
      ci[j] = 0.0f;
    for (k=0; k<n; k++) {
      float *bk =
__builtin_assume_aligned (b[k], 32);
      for (j=0; j<n; j++)
        ci[j] += a[i][k] * bk[j];
  }
}
```

kernel2: assuming aligned vector accesses

```
> make clean
> make OPTFLAGS="-O3 -march=native" KERNEL=2
> ./matmul 100 1000
Cannot call kernel2 on matrices with size%8 != 0 (data non
aligned on 32B boundaries)
Aborted
> ./matmul 104 1000
Cycles per FMA: 0.41
> maqao cqa matmul fct=kernel2 --confidence-levels=gain,hint
```

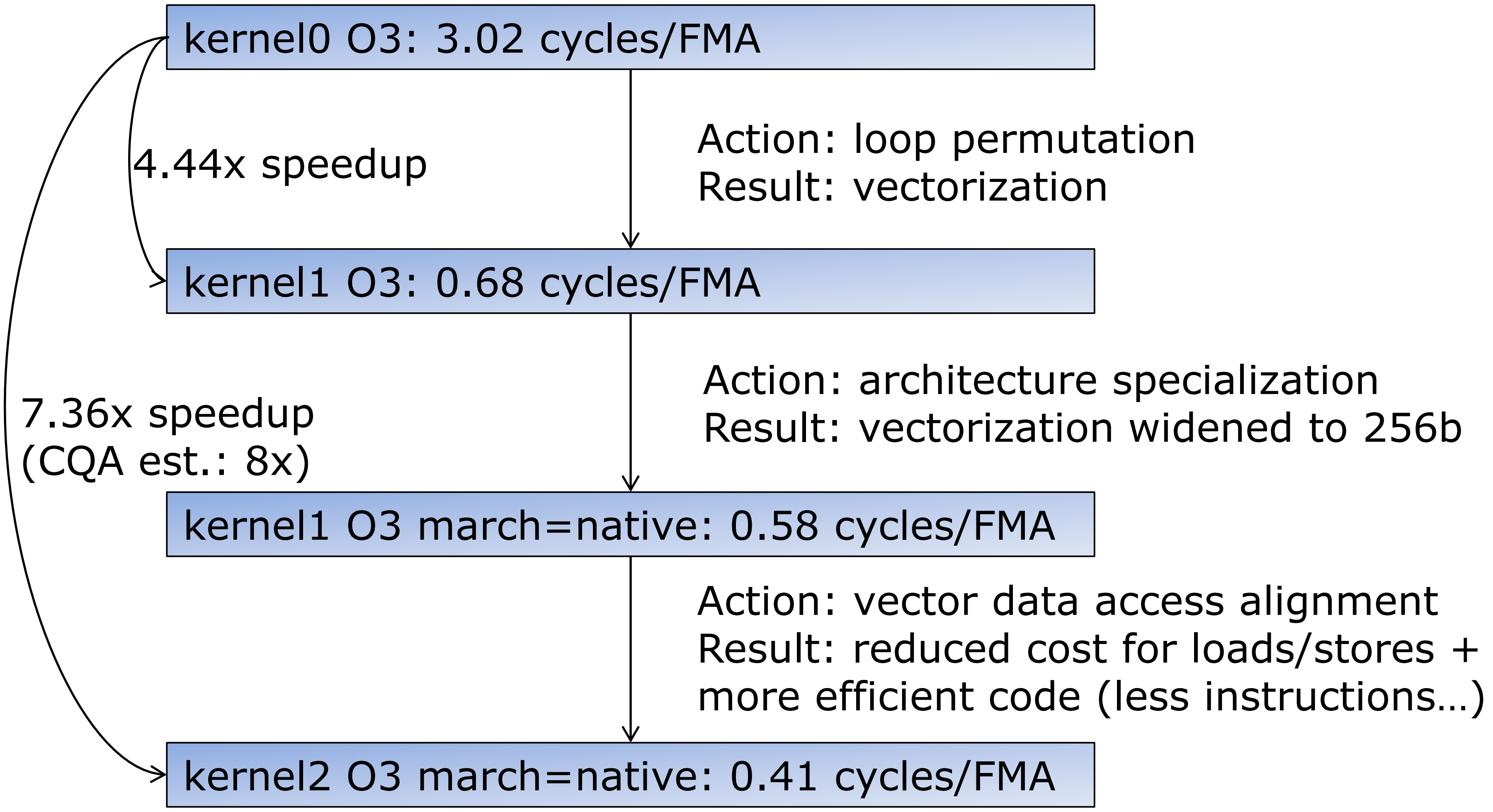
CQA output: diff kernel1 kernel2

```
Vectorization status
--------------------
Your loop is ful
on 75% vector le
```

Better vectorization: now all instructions are vectorized

```
Vectorization status
--------------------
                   vectorized (…) on
```

Summary of optimizations and gains


kernel0 O3: 3.02 cycles/FMA
4.44x speedup
Action: loop permutation
Result: vectorization
kernel1 O3: 0.68 cycles/FMA
7.36x speedup
(CQA est.: 8x)
Action: architecture specialization
Result: vectorization widened to 256b
kernel1 O3 march=native: 0.58 cycles/FMA
Action: vector data access alignment
Result: reduced cost for loads/stores + more efficient code (less instructions…)
kernel2 O3 march=native: 0.41 cycles/FMA