MAQAO
Hands-on exercises

ONE View: Performance View Aggregator
lprof: Lightweight Profiler
CQA: Code Quality Analyzer

Setup

Login to INTI with X11 forwarding

```
> ssh -X <login>@inti.ocre.cea.fr
```

Copy handson material

```
> module load datadir/formation
> cp $FORMATION_HOME/maqao/MAQAO_HANDSON_TW26.tgz $HOME
```

Untar the archive at the root of your home folder

```
> tar xf MAQAO_HANDSON_TW26.tgz
> cd MAQAO_HANDSON
```

Load MAQAO

```
> module load maqao
```

MAQAO ONE View
Hands-on exercises

Emmanuel OSERET, Andrés Charif Rubial

Setup

Go to the Oneview handson folder

```
> cd oneview
```

Generate a Oneview config file template

```
> maqao oneview --create-config
```

Modify it accordingly (ex for Haswell + default MPI)

```
> vim config.lua
```

```
...
binary       = "bt-mz.B.4_haswell_openmpi"
...
mpi_command = "OMP_NUM_THREADS=8 ccc_mprun -E\"--enable_perf\" -
p haswell -n 4 -c 8 -x"
...
```

Launching MAQAO ONE View on bt-mz

Launch ONE View

```
> maqao oneview --create-report=one --config=config.lua --format=html
```

Results are located in <exp-dir>/RESULTS/one_html

```
> firefox <exp-dir>/RESULTS/one_html/index.html &
```

You can also use a script to display the last ONE view report

```
> $FORMATION_HOME/maqao/firefox_last_oneview.sh
```

It is also possible to download the results locally

```
> tar -zcf one_view.tar.gz <exp-dir>/RESULTS/one_html
```

Then download the archive locally, inflate it and view the one_html/index.html file through your browser.

MAQAO lprof
Hands-on exercises

Emmanuel OSERET, Andrés Charif Rubial

Advanced profiling collect/display with lprof

Per-thread visualization

```
> maqao lprof of=html xp=<oneview-exp-dir>/lprof_one
```

Standalone/custom run: go to lprof handson directory and submit a job

```
> cd ../lprof
> vim jobscripts/bt-mz_ompi_haswell.jobscript
> ccc_msub jobscripts/bt-mz_ompi_haswell.jobscript
```

Display results

```
> cd $SCRATCHDIR/lprof_workspace_ompi_haswell_<JOBID>
> maqao lprof xp=lprof_xpdir of=html    #html
> maqao lprof xp=lprof_xpdir -df        #ascii-art, functions
> maqao lprof xp=lprof_xpdir -dl        #ascii-art, loops
```

MAQAO / CQA
Hands-on exercises

Emmanuel OSERET

Setup for a Haswell node

Remark: CQA can also be executed directly on a login node because it uses static analysis

Log to a Haswell node (interactive session)

```
> ccc_mprun -p haswell -E"--enable_perf" -s  -X first
```

Load MAQAO

```
> module load maqao
```

Load a recent GCC compiler

```
> module load c/gnu/7.1.0
```

Switch to CQA handson folder

```
> cd $HOME/MAQAO_HANDSON/cqa/matmul
```

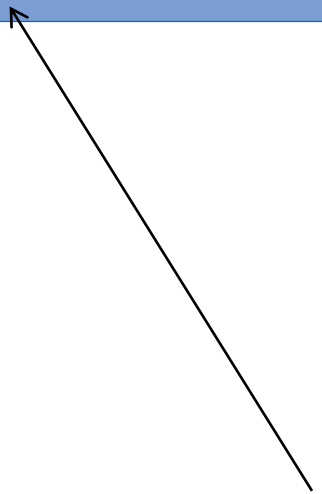
Matrix Multiply code

```
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

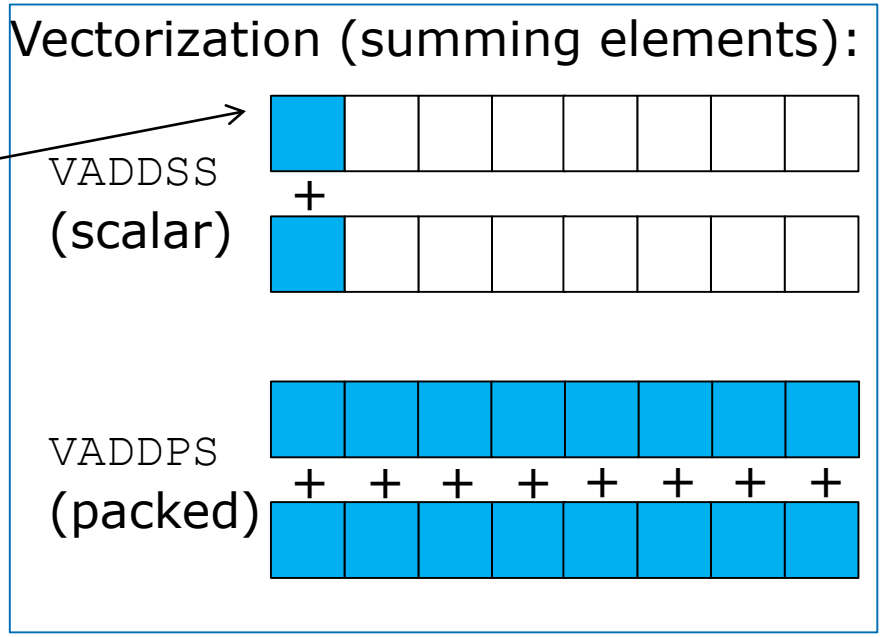
```
> make OPTFLAGS=-O3 KERNEL=0
> ./matmul 100 1000
Cycles per FMA: 2.10
> maqao cqa matmul fct-loops=kernel0 [of=html]
```



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…), or using oneview.
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 in ...NDSON_test/CQA/matmul/kernel.c

It is composed of the loop 2

MAQAO binary loop id: 2

The loop is defined in /home/hpc/a2c06/lu23voj/MAQAO_HANDSON_test/CQA/matmul/kernel.c:9-10
In the binary file, the address of the loop is: 4009f0

8% of peak computational performance is used (0.67 out of 8.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain | Potential gain | Hints | Experts only

Vectorization status

Your loop is not vectorized (all SSE/AVX instructions are used in scalar mode).
Only 25% of vector length is used.

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 0.75 cycles (4.00x 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

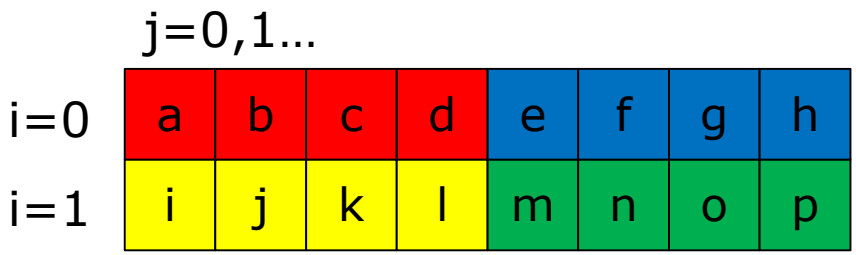
Detected a non usual bottleneck.

Proposed solution(s):

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

Impact of loop permutation on data access

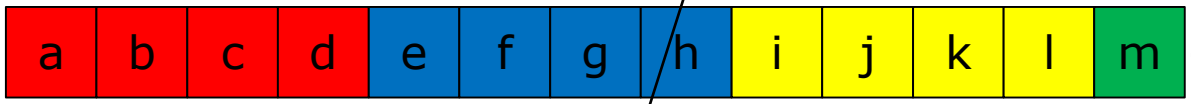
Logical mapping



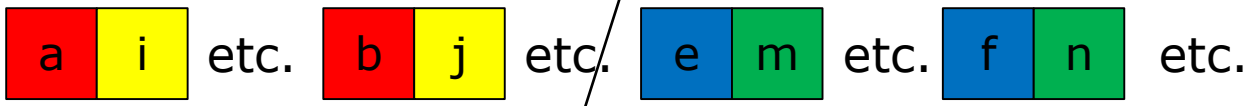
Efficient vectorization + prefetching

Physical mapping

(C stor. order: row-major)




```
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: 0.46
> maqao cqa matmul fct-loops=kernel1
```

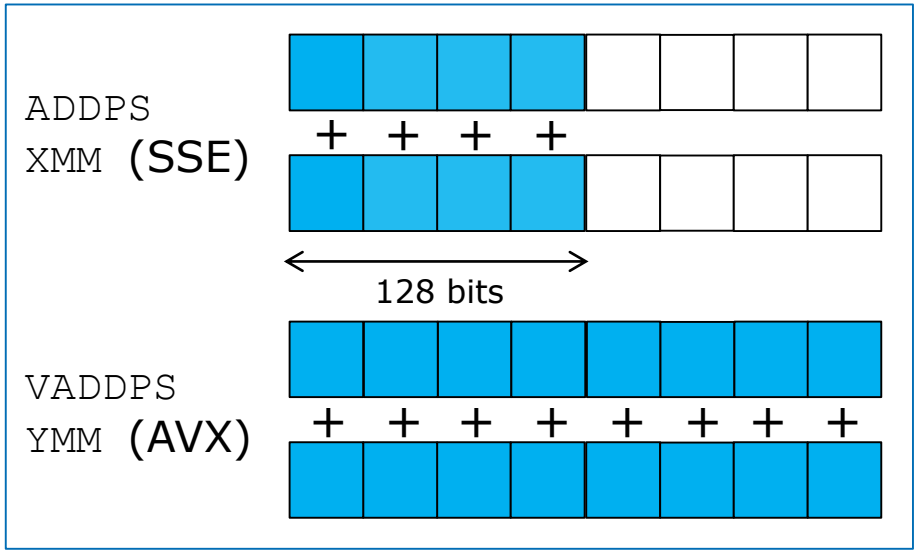
CQA output for kernel1

```
Vectorization status
--------------------
Your loop is vectorized (…) but using only
128 out of 256 bits (SSE/AVX-128
instructions on AVX/AVX2 processors).

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




```
# 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: 0.36
> maqao cqa matmul fct-loops=kernel1 --confidence-
levels=gain,hint
```

CQA output for kernel1 (using "gain" and "hint" conf. levels)

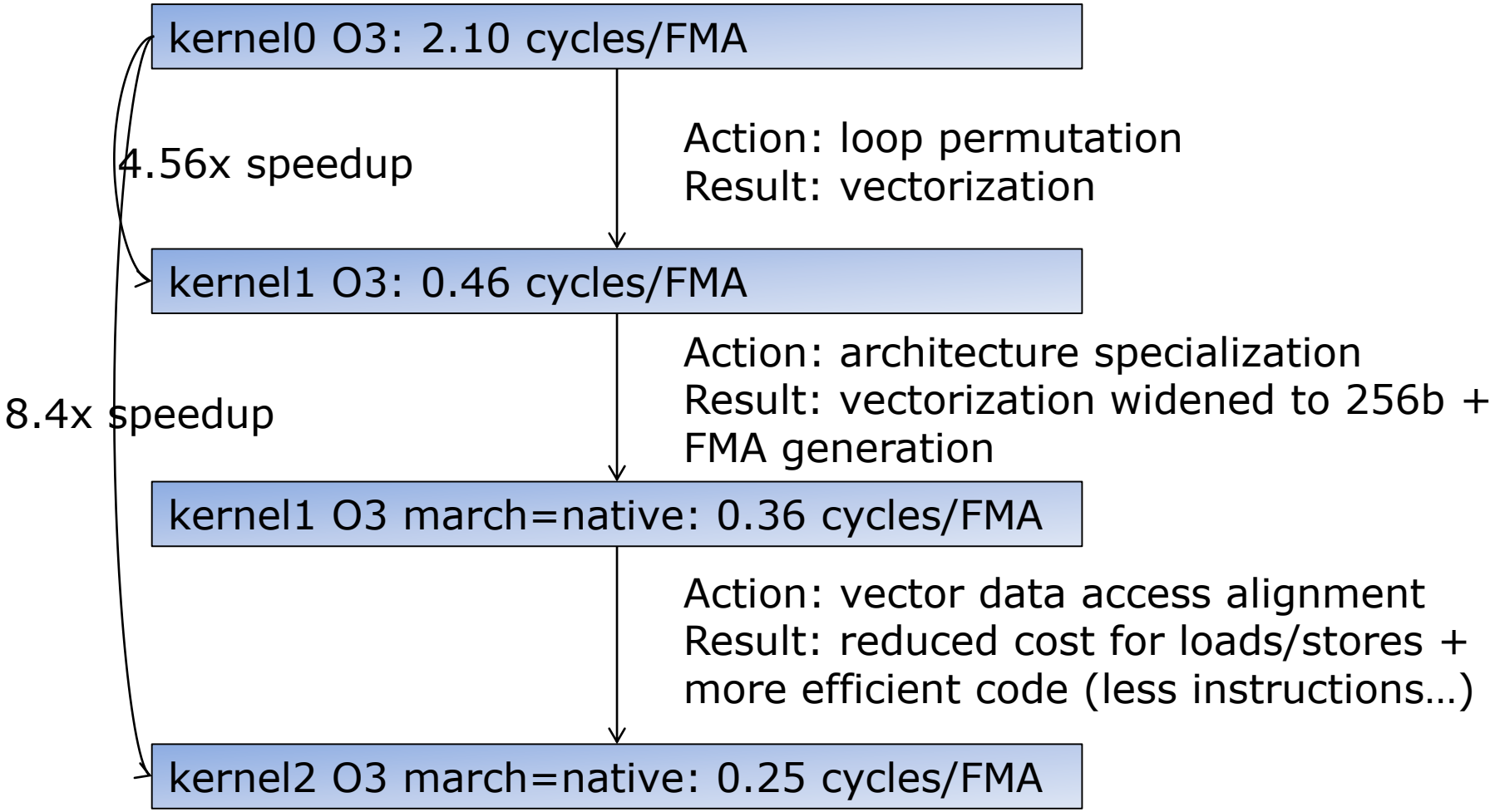
```
Vectorization status
--------------------
Your loop is fully vectorized (…)

Vector unaligned load/store…
----------------------------
 - 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

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: 0.25
```

Summary of optimizations and gains



Hydro example

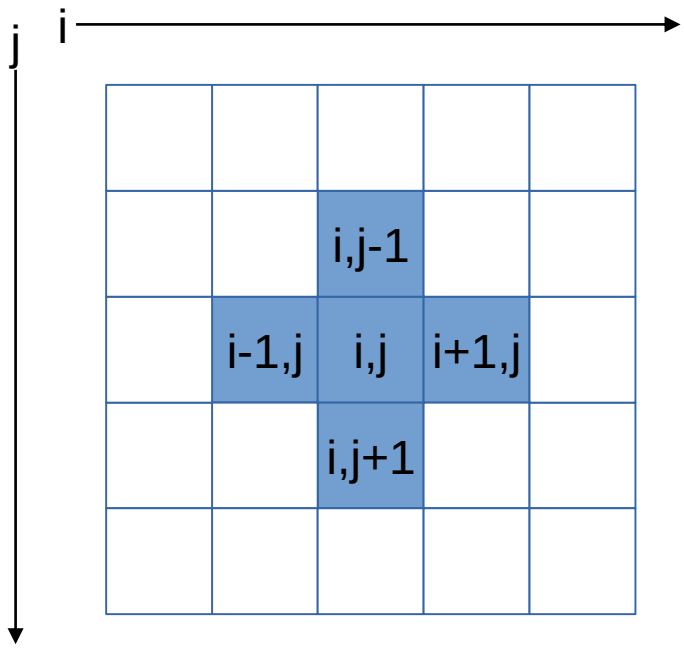
Switch to the other CQA handson folder

```
> cd $HOME/MAQAO_HANDSON/cqa/hydro
```

Hydro code

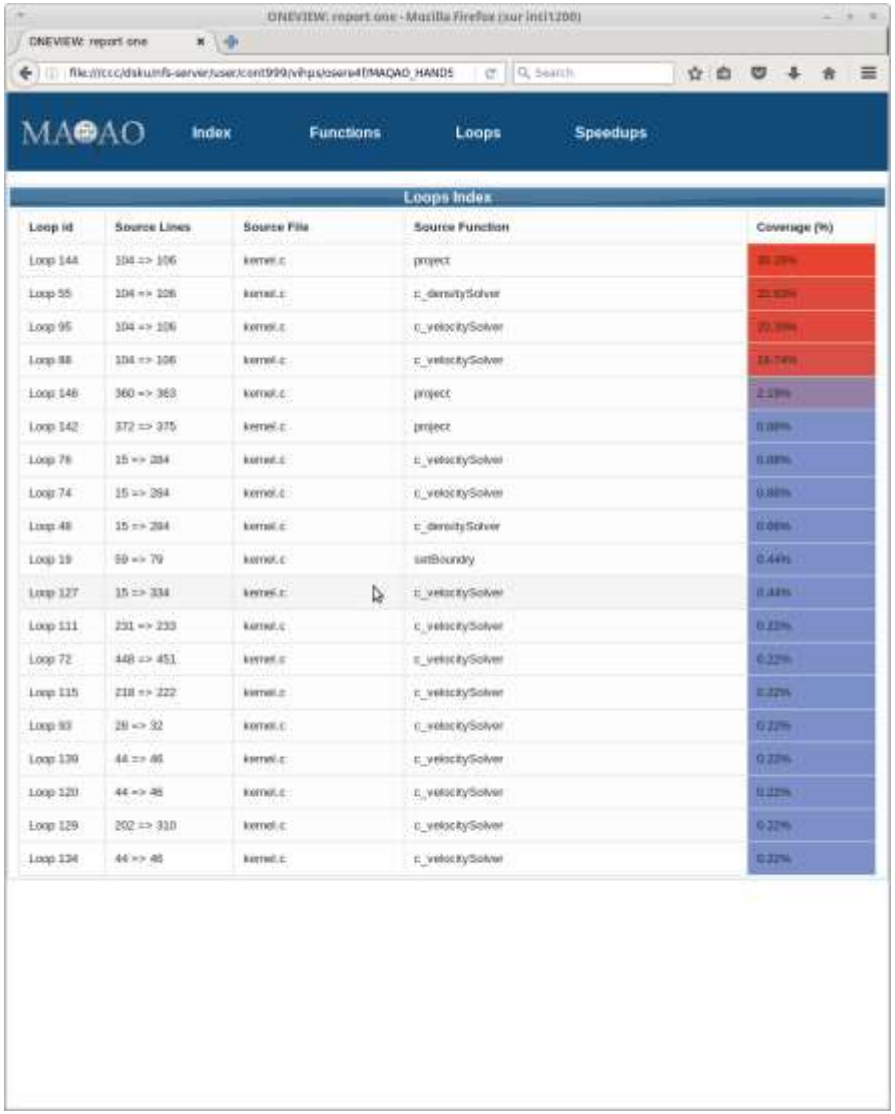
```
int build_index (int i, int j, int grid_size)
{
  return (i + (grid_size + 2) * j);
}

void linearSolver0 (...) {
  int i, j, k;

  for (k=0; k<20; k++)
    for (i=1; i<=grid_size; i++)
      for (j=1; j<=grid_size; j++)
        x[build_index(i, j, grid_size)] =
  (a * ( x[build_index(i-1, j, grid_size)] +
         x[build_index(i+1, j, grid_size)] +
         x[build_index(i, j-1, grid_size)] +
         x[build_index(i, j+1, grid_size)]
       ) + x0[build_index(i, j, grid_size)]
  ) / c;
}
```

Iterative linear system solver using the Gauss-Siedel relaxation technique.
« Stencil » code



Compiling, running and analyzing kernel0 (icc -O3 -xHost)

```
> make KERNEL=0
> ./hydro 250 10 # 1st param: grid_size and 2nd param: repet nb
Cycles per element for solvers: 1661.36
> maqao oneview --create-report=one --config=config_oneview.lua
 --format=html
> $FORMATION_HOME/maqao/firefox_last_oneview.sh
```



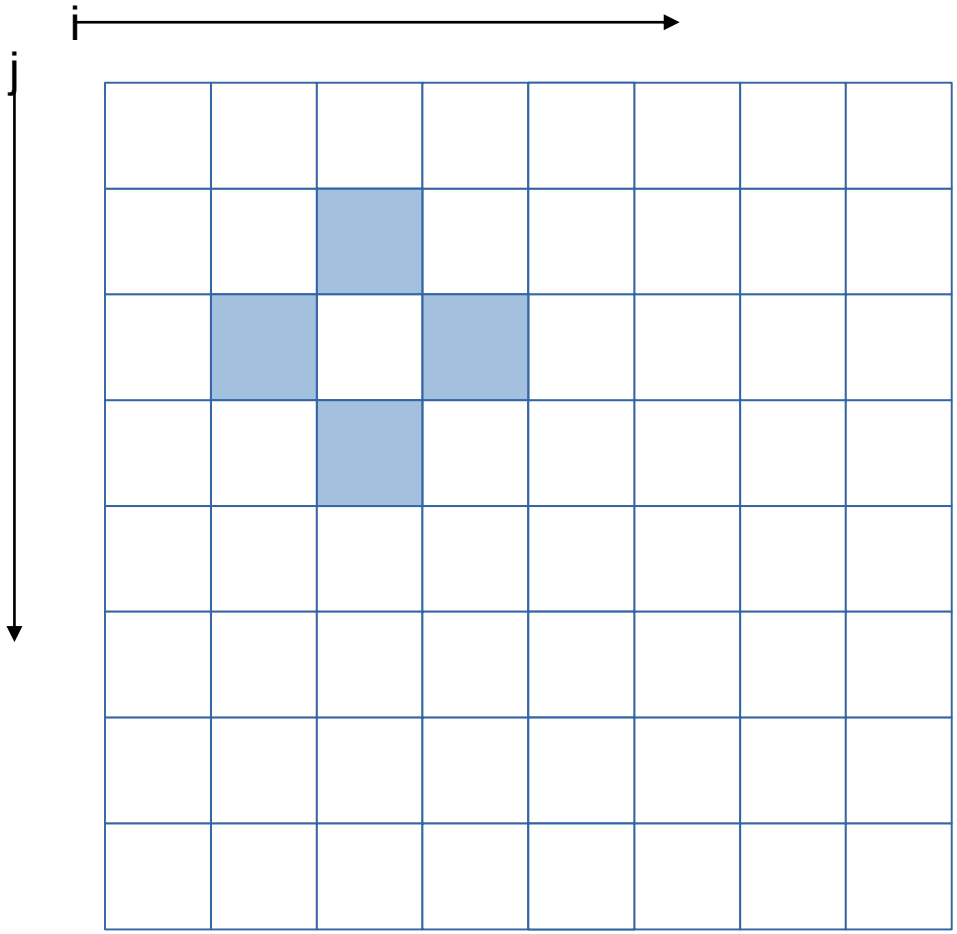
The kernel routine, linearSolver, were inlined in caller functions. Moreover, there is direct mapping between source and binary loop. Consequently the 4 hot loops are identical and only one need analysis.

CQA output for kernel0

```
Composition and unrolling
-------------------------
It is composed of the loop 142
and is not unrolled or unrolled with
no peel/tail loop.
The analysis will be displayed for
the requested loops: 142


Unroll opportunity
------------------
Loop is potentially data access
bound.
Unroll your loop if trip count is
significantly higher than target
unroll factor and if some data
references are common to consecutive
iterations…
```

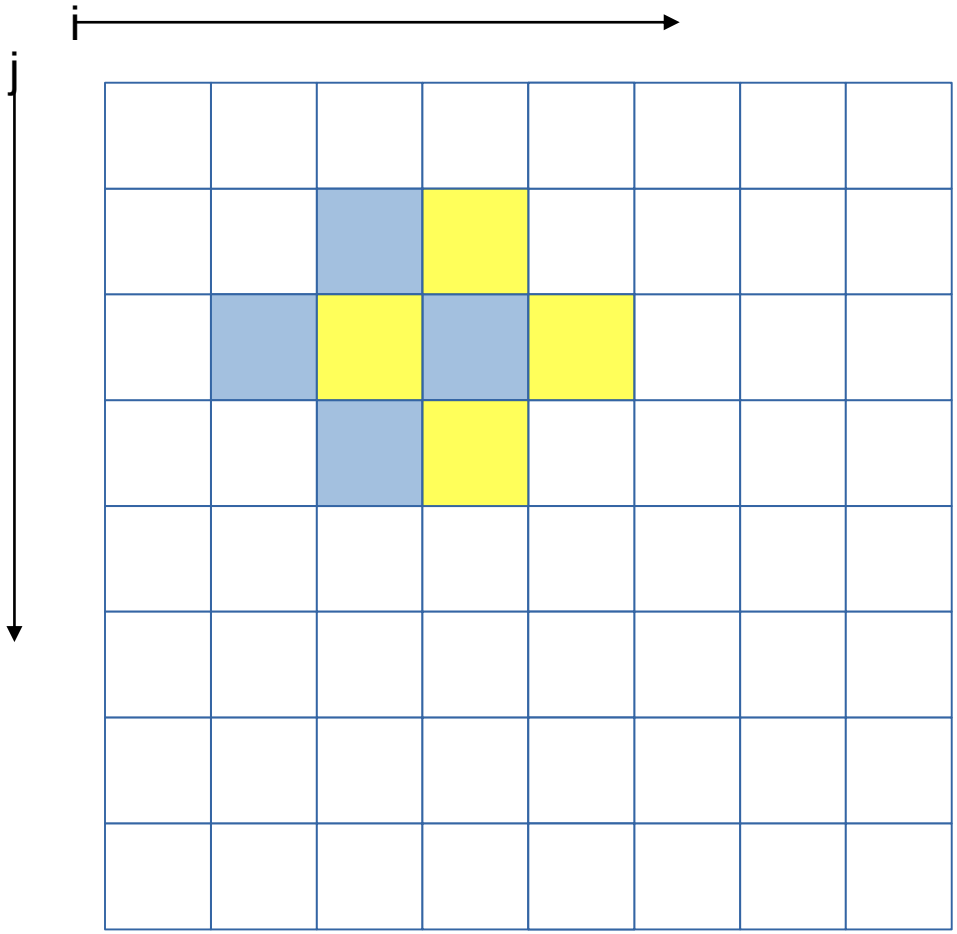
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



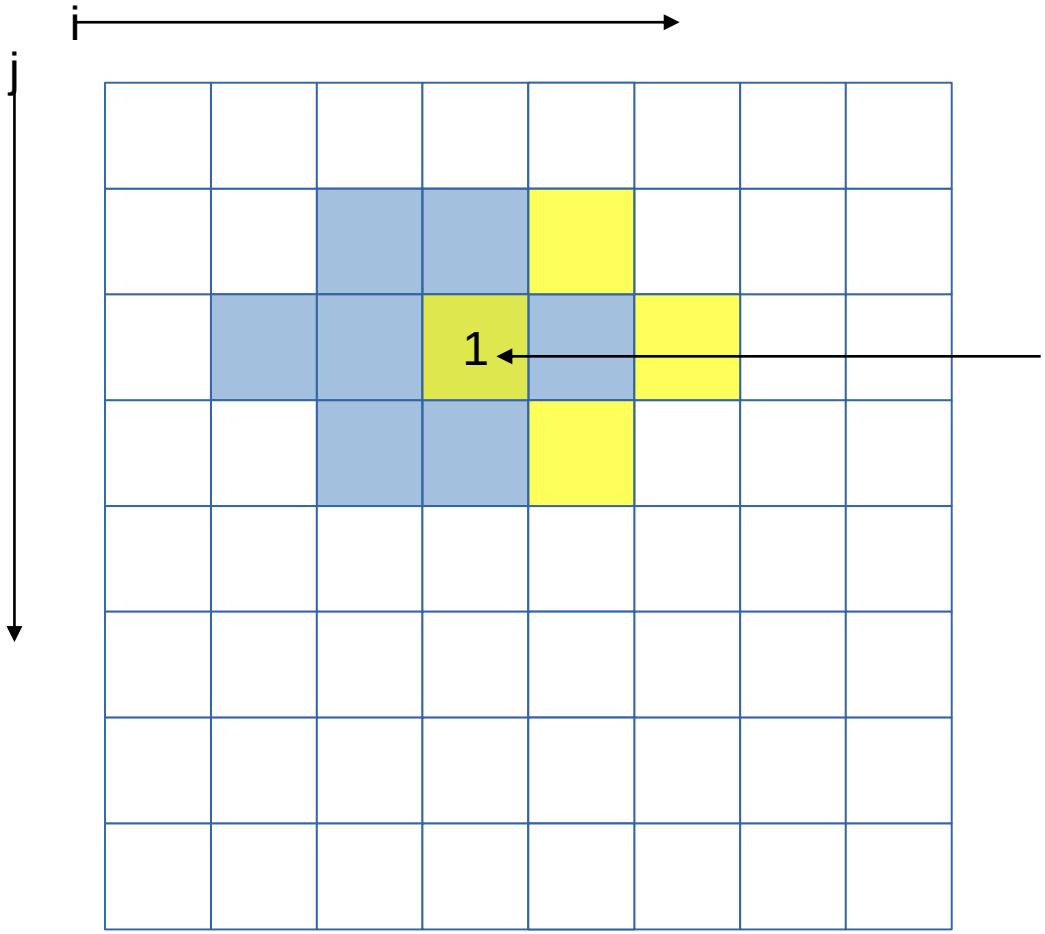
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)

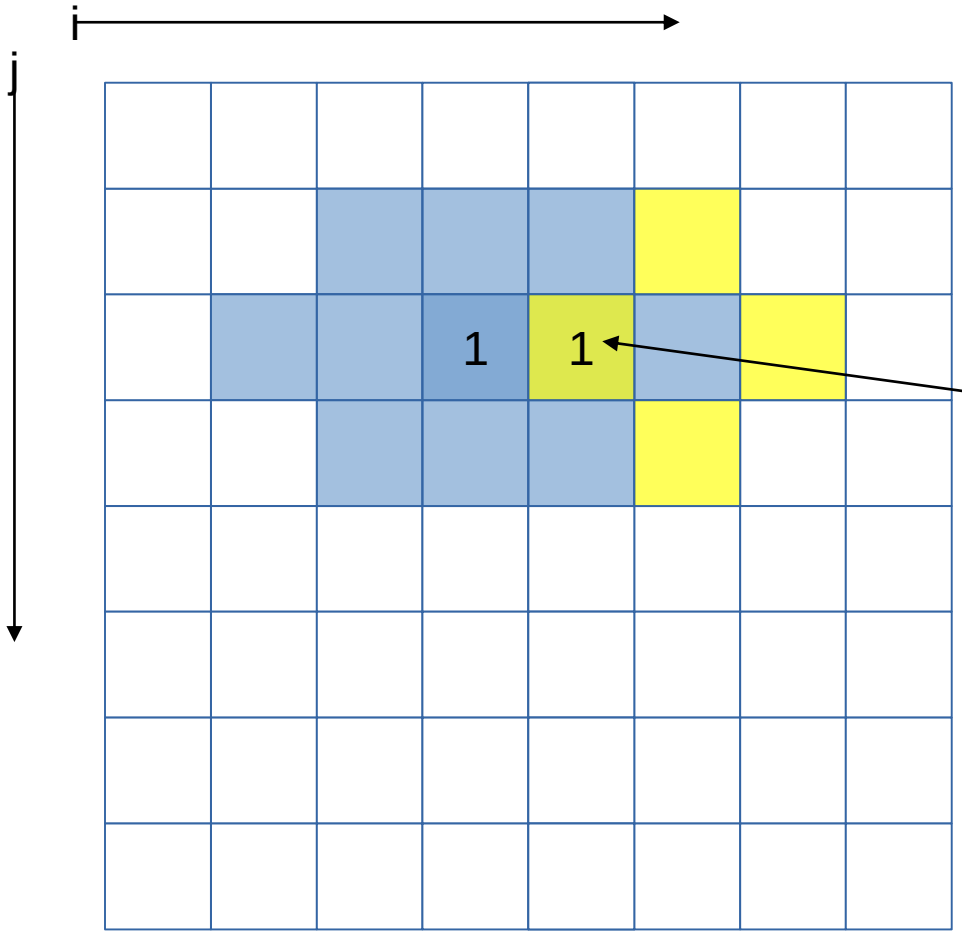
Memory references reuse : 4x4 unroll footprint on loads



LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)

1 reuse

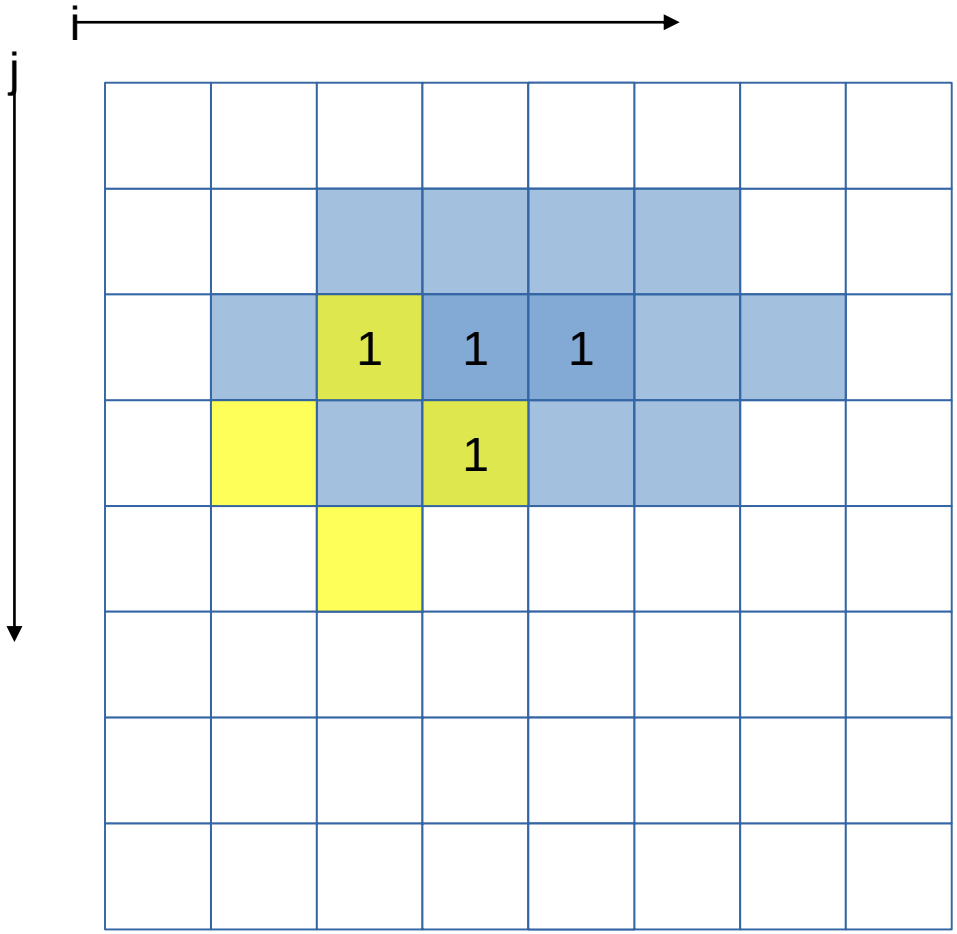
Memory references reuse : 4x4 unroll footprint on loads



LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

2 reuses

Memory references reuse : 4x4 unroll footprint on loads

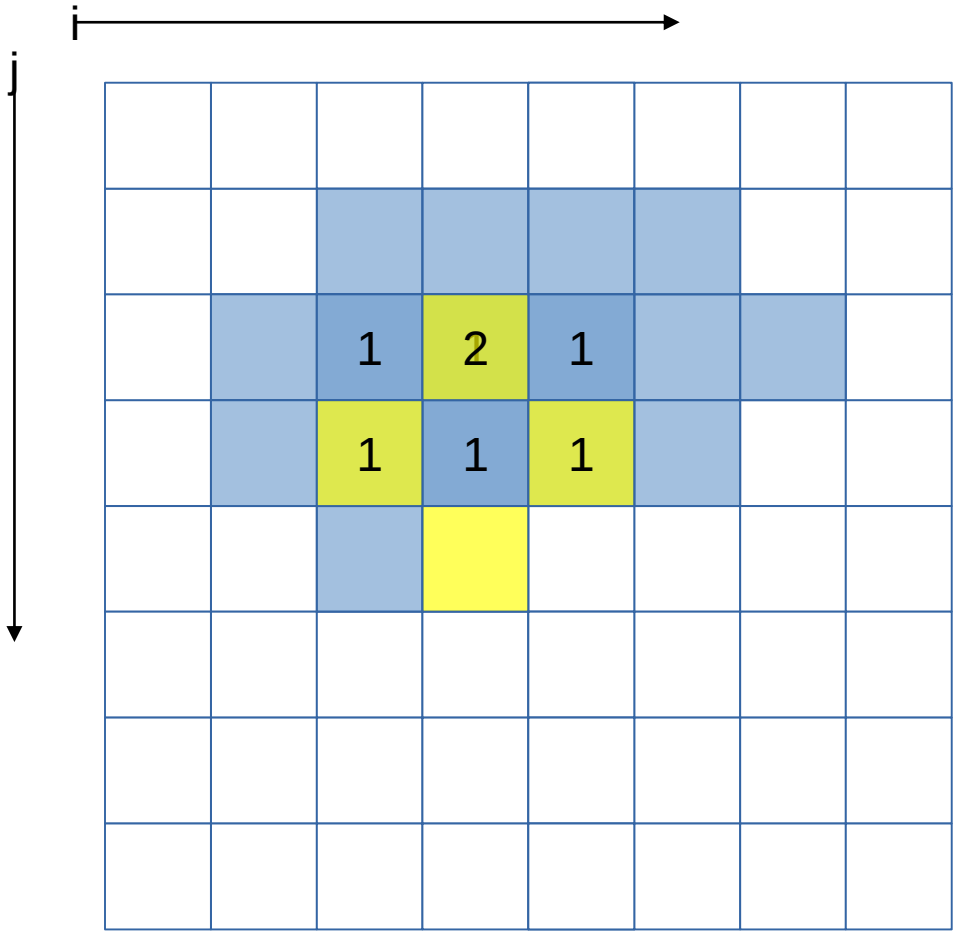


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

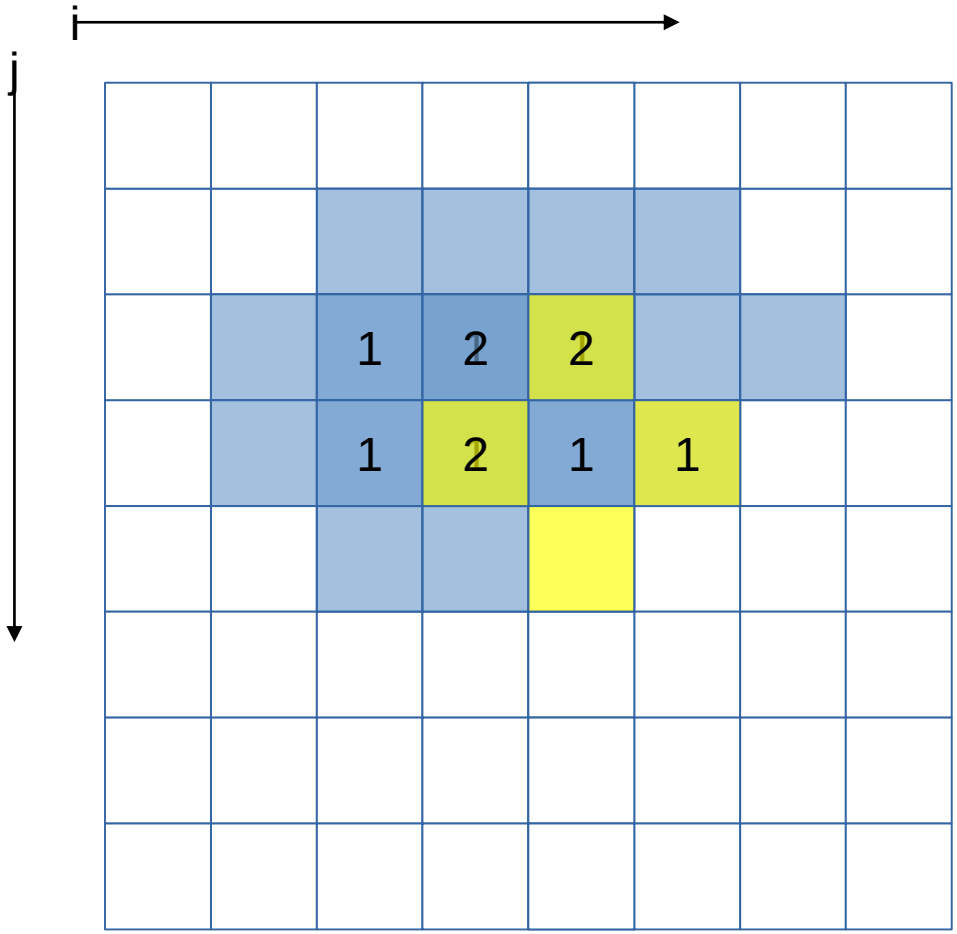


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
LINEAR_SOLVER(i+1,j+1)

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

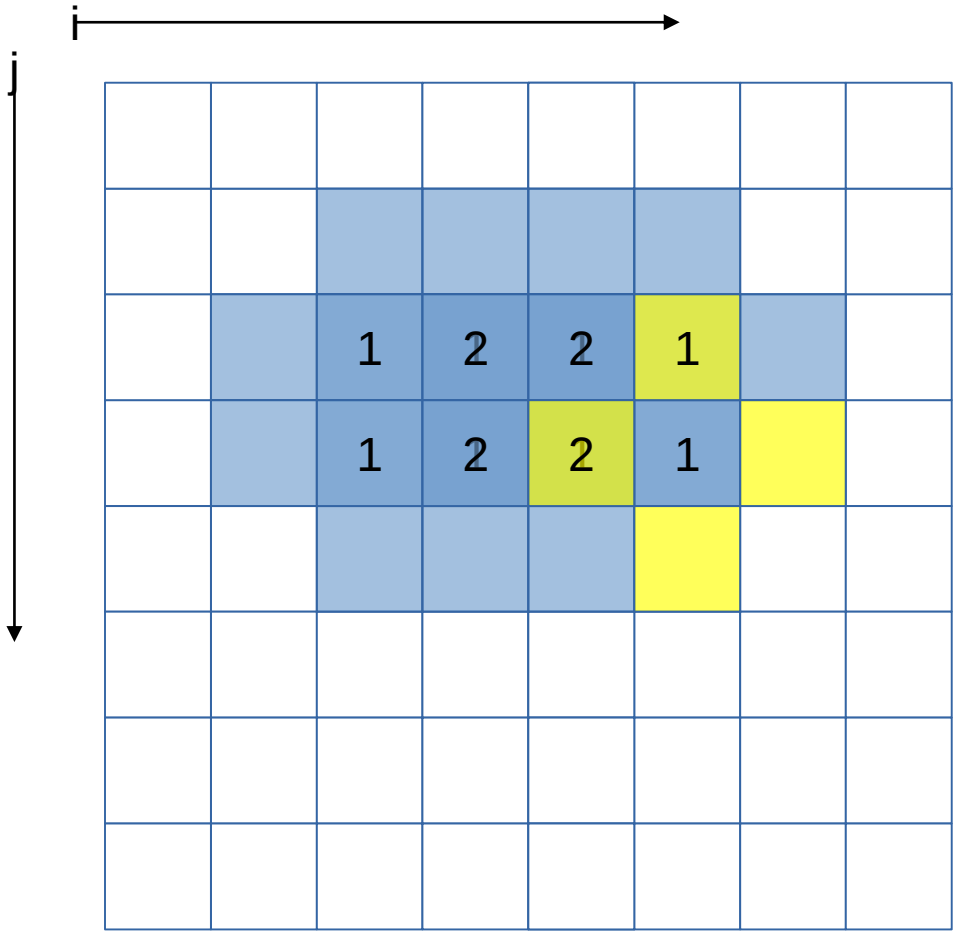


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
LINEAR_SOLVER(i+1,j+1)
LINEAR_SOLVER(i+2,j+1)

10 reuses

Memory references reuse : 4x4 unroll footprint on loads

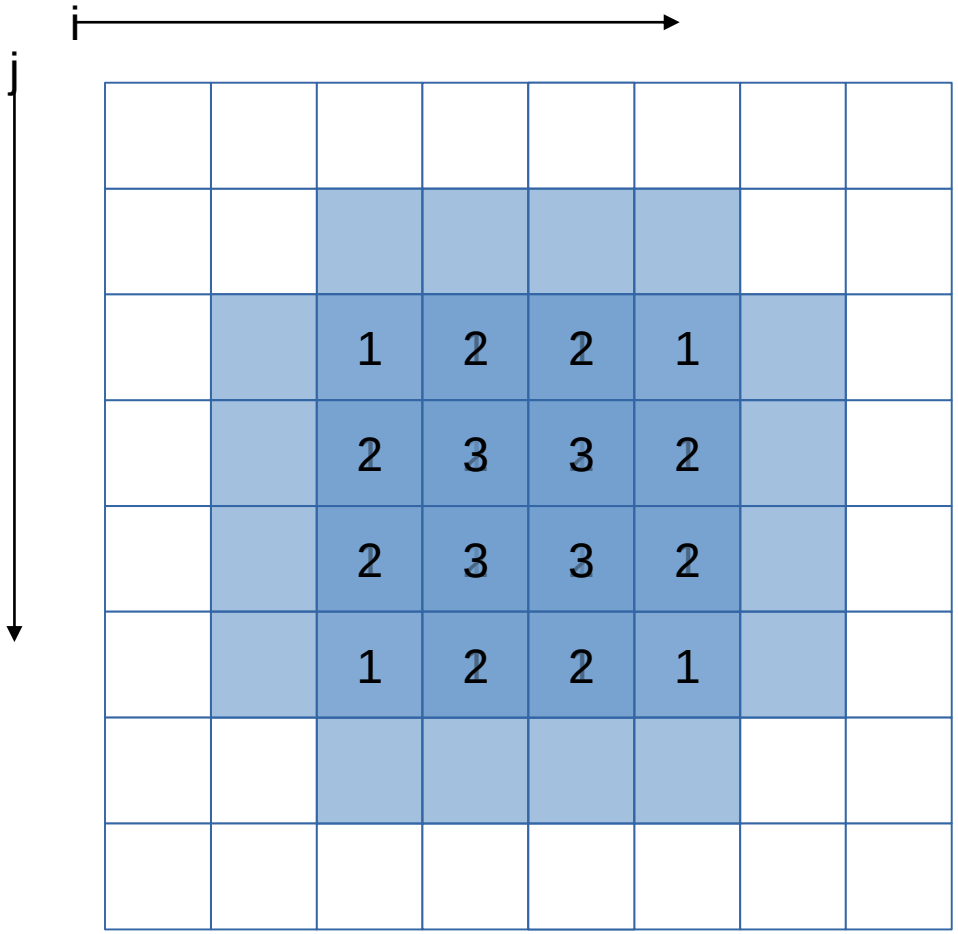


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)
LINEAR_SOLVER(i+1,j+1)
LINEAR_SOLVER(i+2,j+1)
LINEAR_SOLVER(i+3,j+1)

12 reuses

Memory references reuse : 4x4 unroll footprint on loads



LINEAR_SOLVER(i+0-3,j+0)

LINEAR_SOLVER(i+0-3,j+1)

LINEAR_SOLVER(i+0-3,j+2)

LINEAR_SOLVER(i+0-3,j+3)

32 reuses

Impacts of memory reuse

- For the x array, instead of 4x4x4 = 64 loads, now only 32 (32 loads avoided by reuse)
- For the x0 array no reuse possible : 16 loads
- Total loads : 48 instead of 80

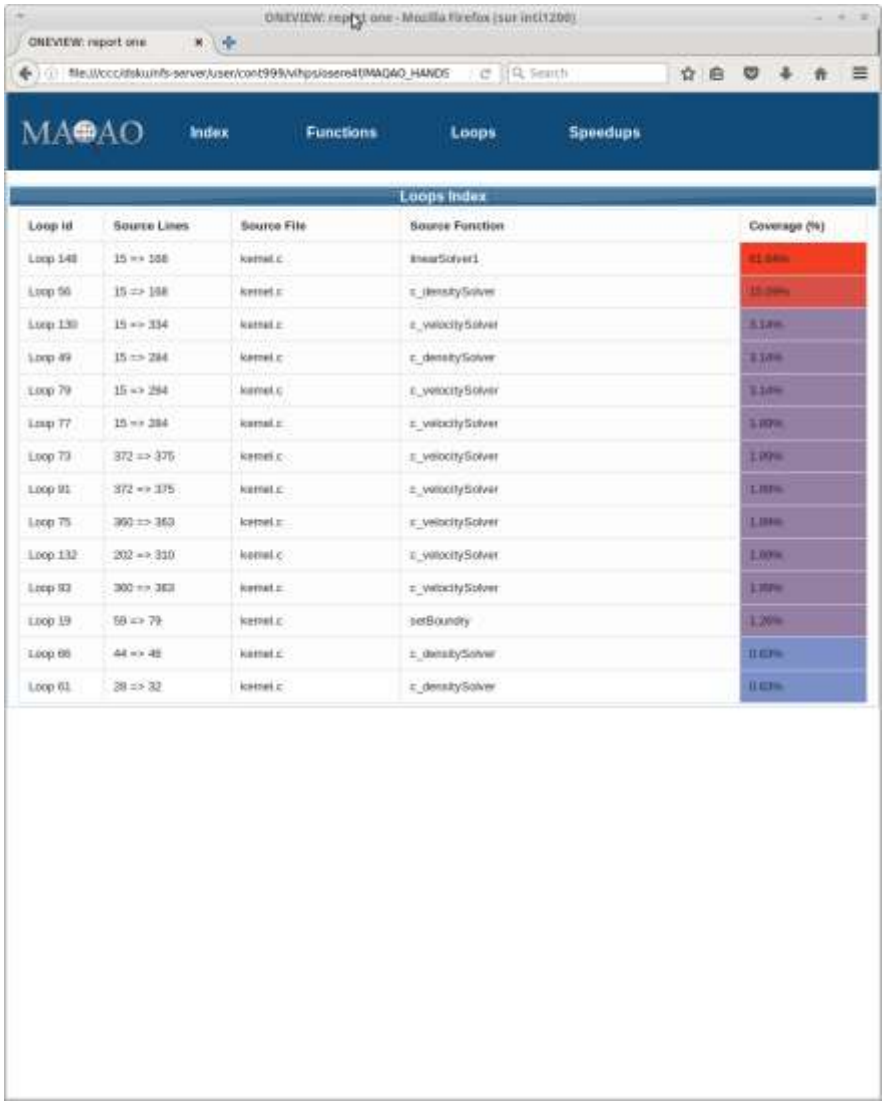
4x4 unroll

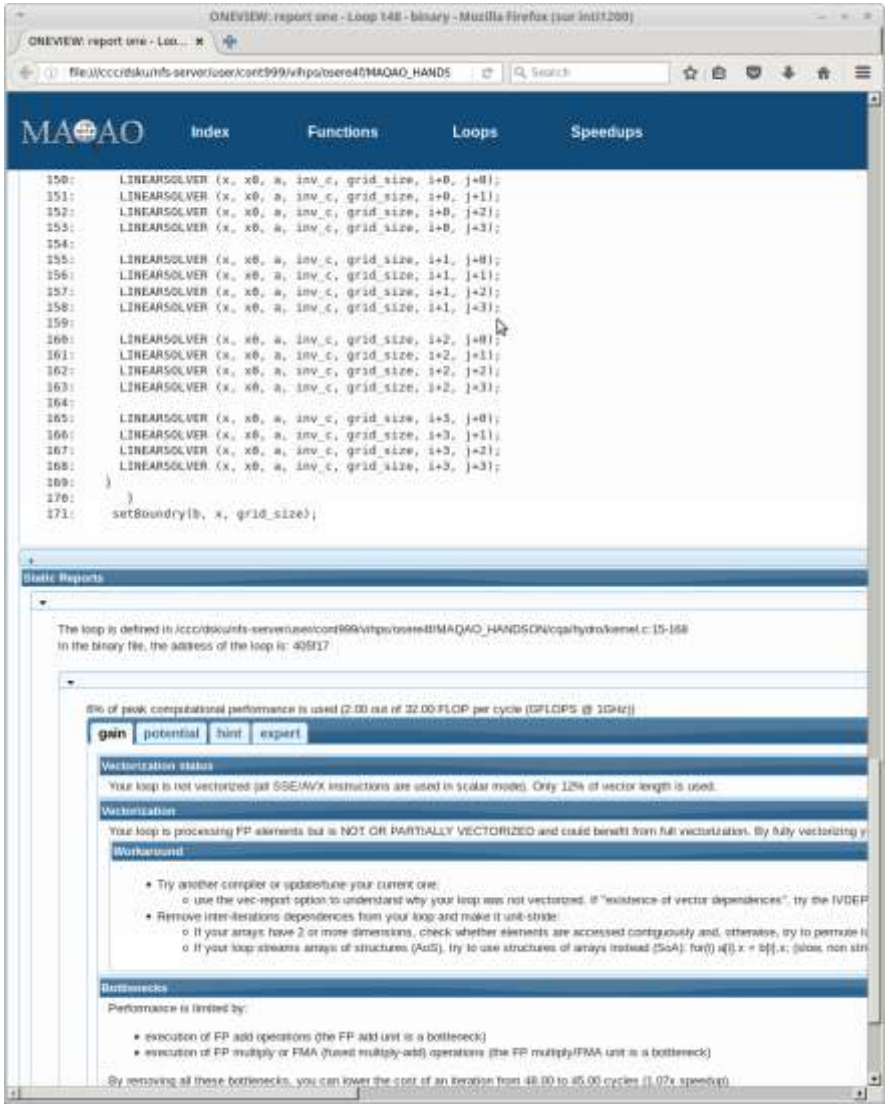
```
#define LINEARSOLVER(...) x[build_index(i, j, grid_size)] = …

void linearSolver2 (...) {
  (...)

  for (k=0; k<20; k++)
    for (i=1; i<=grid_size-3; i+=4)
      for (j=1; j<=grid_size-3; j+=4) {
        LINEARSOLVER (…, i+0, j+0);
        LINEARSOLVER (…, i+0, j+1);
        LINEARSOLVER (…, i+0, j+2);
        LINEARSOLVER (…, i+0, j+3);

        LINEARSOLVER (…, i+1, j+0);
        LINEARSOLVER (…, i+1, j+1);
        LINEARSOLVER (…, i+1, j+2);
        LINEARSOLVER (…, i+1, j+3);

        LINEARSOLVER (…, i+2, j+0);
        LINEARSOLVER (…, i+2, j+1);
        LINEARSOLVER (…, i+2, j+2);
        LINEARSOLVER (…, i+2, j+3);

        LINEARSOLVER (…, i+3, j+0);
        LINEARSOLVER (…, i+3, j+1);
        LINEARSOLVER (…, i+3, j+2);
        LINEARSOLVER (…, i+3, j+3);
      }
}
```

grid_size must now be multiple of 4. Or loop control must be adapted (much less readable) to handle leftover iterations

Kernel1

```
> make clean
> make KERNEL=1
> ./hydro 250 10
Cycles per element for solvers: 583.74
> maqao oneview --create-report=one --config=config_oneview.lua
  --output=html
> $FORMATION_HOME/maqao/firefox_last_oneview.sh
```





Remark: less calls were unrolled since linearSolver is now much more bigger

CQA output for kernel1

```
Matching between your loop ...
------------------------------
The binary loop is composed of 96 FP
arithmetical operations:
 - 64: addition or subtraction
 - 16: multiply
The binary loop is loading 272 bytes
(68 single precision FP elements).
The binary loop is storing 64 bytes
(16 single precision FP elements).
```

4x4 Unrolling were applied

Expected 48... But still better than 80

Summary of optimizations and gains