MAQAO
Hands-on exercises

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

Setup

Login to the cluster with X11 forwarding

```
> ssh -Y <login>@lxlogin8.lrz.de
```

Load MAQAO environment

```
> source /home/hpc/a2c06/lu23vof/load_maqao.sh
```

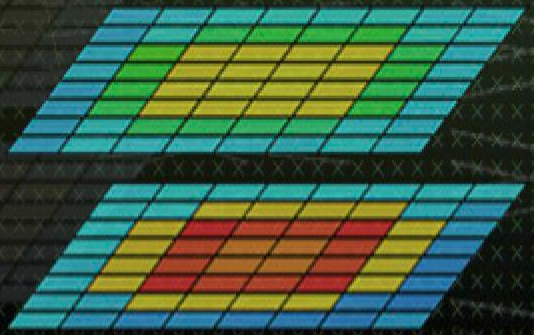
Copy handson material to your SCRATCH_LEGACY directory

```
> cp $MAQAO_DIR/MAQAO_HANDSON_TW27.tgz $SCRATCH_LEGACY
> cd $SCRATCH_LEGACY
> tar xf MAQAO_HANDSON_TW27.tgz
```

Ensure that the NAS are compiled with debug information (update make.def)

```
FFLAGS  = -O3 $(COMPFLAGS) -g
```

```
> cp MAQAO_HANDSON/make.def NPB3.3-MZ-MPI/config
> cd $SCRATCH_LEGACY/NPB3.3-MZ-MPI
> make bt-mz CLASS=C NPROCS=32
```



MAQAO ONE View
Hands-on exercises

Cédric Valensi

Setup: Generating ONE View config file

Generate a Oneview config file template

```
> cd $SCRATCH_LEGACY
> maqao oneview --create-config
```

Modify it to specify the execution parameters.

```
> vim config.lua
```

/!\ Environment variables are **not** accepted, but variables can be used

```
SCRATCH_LEGACY="/naslx/..."
binary = SCRATCH_LEGACY.."/NPB3.3-MZ-MPI/bin/bt-mz_C.32"
```

Sample config files are located in the MAQAO_HANDSON/oneview directory.

Setup ONE View for batch mode

Modify configuration file (sample file ***config_bt_oneview_sbatch.lua***)

```
binary = "NPB3.3-MZ-MPI/bin/bt-mz_C.32"
...
sbatch_script = "bt_oneview.sbatch"
...
mpi_command = "mpiexec"
...
omp_num_threads = 4
```

Modify batch script to replace execution command

```
> cp NPB3.3-MZ-MPI/jobscript/coolmuc3/reference.sbatch bt_oneview.sbatch
> vim bt_oneview.sbatch
```

```
...
mpiexec ./NPB3.3-MZ-MPI/bin/bt-mz_C.32
<run_command>
...
```

A sample batch script is located in the MAQAO_HANDSON/oneview directory.

Setup ONE View for interactive mode

Modify configuration file (sample file ***config_bt_oneview_interactive.lua***)

```
> vim config.lua
```

```
binary = "NPB3.3-MZ-MPI/bin/bt-mz_C.32"
...
mpi_command = "mpiexec -n 32"
...
omp_num_threads = 4
```

Request interactive session

```
> salloc --nodes=2 --tasks-per-node=32
```

Launching MAQAO ONE View on bt-mz

Launch ONE View

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

By default, the experiment directory <exp_dir> is named maqao_<timestamp>
Directory name can be specified using –xp=<exp_dir> (if <exp_dir> already exists, ONE View will reuse its content but not overwrite it).

HTML files are located in <exp-dir>/RESULTS/bt-mz_one_html

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

It is also possible to download the results locally

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

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

MAQAO LProf
Hands-on exercises

Cédric Valensi

LProf standalone execution

Modify batch script to replace execution command

```
> cd $SCRATCH_LEGACY
> cp NPB3.3-MZ-MPI/jobscript/coolmuc3/reference.sbatch bt_lprof.sbatch
> vim bt_lprof.sbatch
```

```
...
source /home/hpc/a2c06/lu23vof/load_maqao.sh
...
EXE="maqao lprof -- ./NPB3.3-MZ-MPI/bin/bt-mz_$CLASS.$PROCS"
...
```

A sample batch script is located in the MAQAO_HANDSON/lprof directory.

Submit the modified batch script

```
> sbatch bt_lprof.sbatch
```

By default, the experiment directory <exp_dir> is named maqao_lprof_<timestamp>

Directory name can be specified using –xp=<exp_dir> (if <exp_dir> already exists, LProf will refuse to overwrite it).

Advanced profiling display with LProf

Display the results from an experiment stored in <exp_dir> in the console

```
> maqao lprof xp=<exp_dir> -df        #functions
> maqao lprof xp=<exp_dir> -dl        #loops
> maqao lprof xp=<exp_dir> -df -dt    #functions, thread breakdown
> maqao lprof xp=<exp_dir> -df -cv=full #categorization details
```

Generate and display the results in HTML format

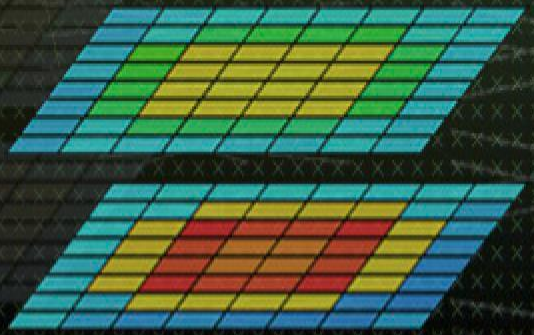
```
> maqao lprof xp=<exp_dir> of=html    #html
```

HTML files are located in <exp-dir>/html

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

It is also possible to display the profiling results from a ONE View experiment:

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



MAQAO / CQA
Hands-on exercises

Emmanuel OSERET

Setup for a Broadwell node

Remark: CQA can also be executed directly on a login node because it uses static analysis. Yet it is possible to cross-analyze to KNL (explained later in this presentation)

Load MAQAO (if not already done)

```
> source /home/hpc/a2c06/lu23vof/load_maqao.sh
```

Load a recent GCC compiler

```
> module load gcc/7
```

Switch to CQA handson folder

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

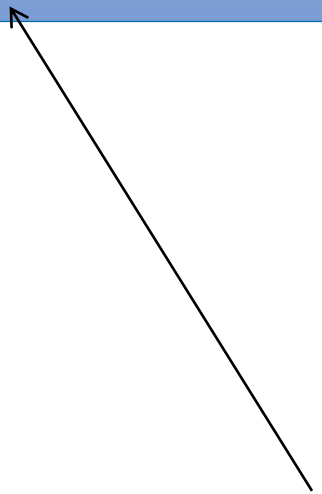
Matrix Multiply code

```
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

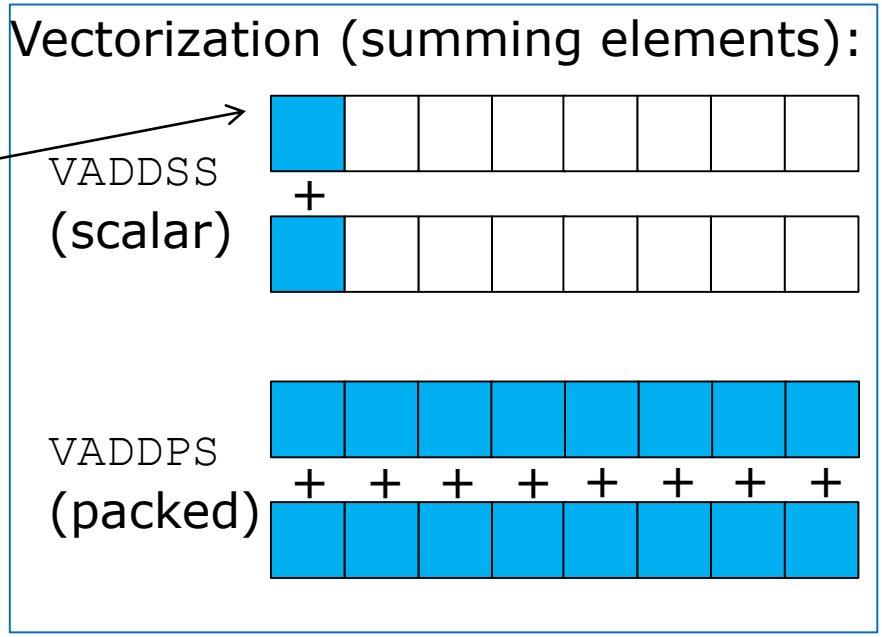
```
> make OPTFLAGS=-O3 KERNEL=0
> ./matmul 100 1000
Cycles per FMA: 1.89
> maqao cqa matmul fct-loops=kernel0 [of=html]
```



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
```

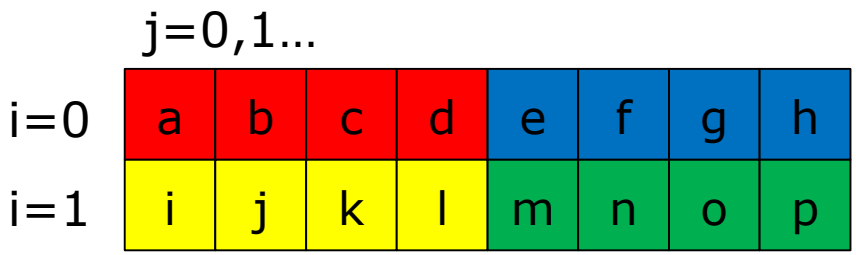


- Accesses are not contiguous => let's permute k and j loops
- No structures here…

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

Impact of loop permutation on data access

Logical mapping

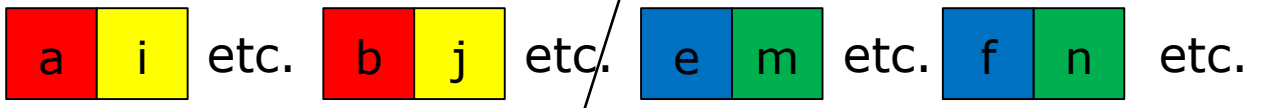


Efficient vectorization + prefetching

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: 0.41
> maqao cqa matmul fct-loops=kernel1
```

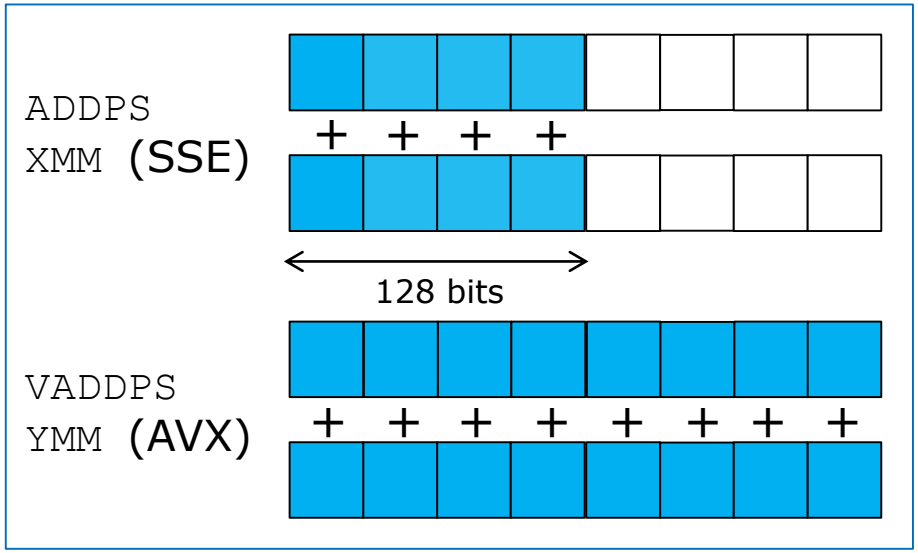
CQA output for kernel1

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

Workaround(s):
 - Recompile with march=broadwell. CQA
target is Core_i7X_Xeon_E5_v4… but
specialization flags are -march=x86_64
- Use vector aligned instructions…

FMA
---
Presence of both ADD/SUB and MUL
operations.
 - Recompile with march=broadwell.
 - Try to change order in which…
```

- Let's add –march=broadwell 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=broadwell" KERNEL=1
> ./matmul 100 1000
Cycles per FMA: 0.32
> maqao cqa matmul fct-loops=kernel1 --confidence-
levels=gain,hint
```

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

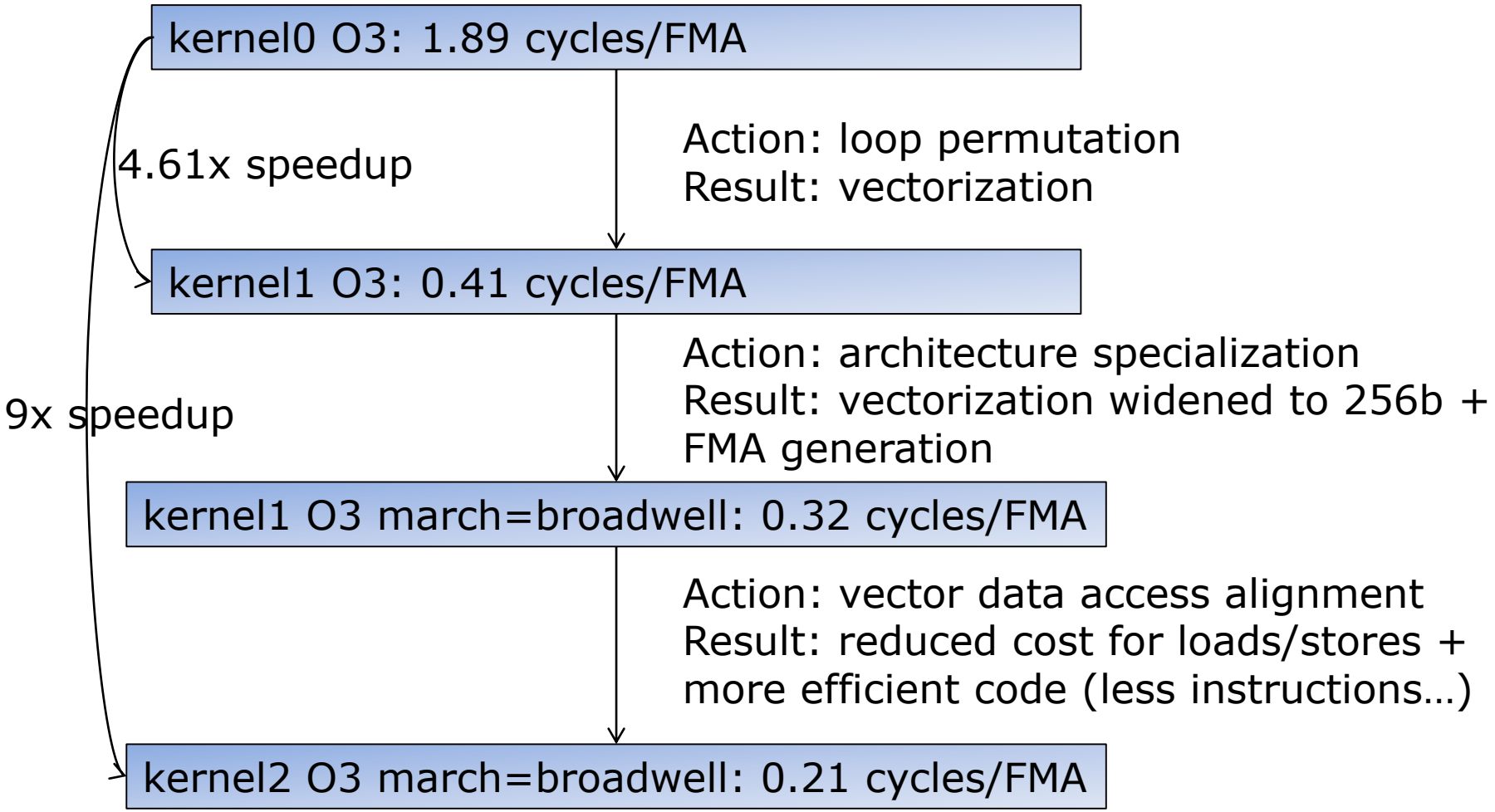
```
Vectorization status
--------------------
Your loop is fully vectorized, using
full register length

Vector unaligned load/store…
----------------------------
 - Use vector aligned instructions:
  1) align your arrays on 32 bytes
boundaries: replace { void *p =
malloc… }
  2) inform your compiler that your
arrays are vector aligned: if array
'foo' is 32 bytes-aligned, define a
pointer 'p_foo' as
__builtin_assume_aligned (foo,32)…
```

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

kernel2: assuming aligned vector accesses

```
> make clean
> make OPTFLAGS="-O3 -march=broadwell" 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: 0.21
```

Summary of optimizations and gains



Hydro example

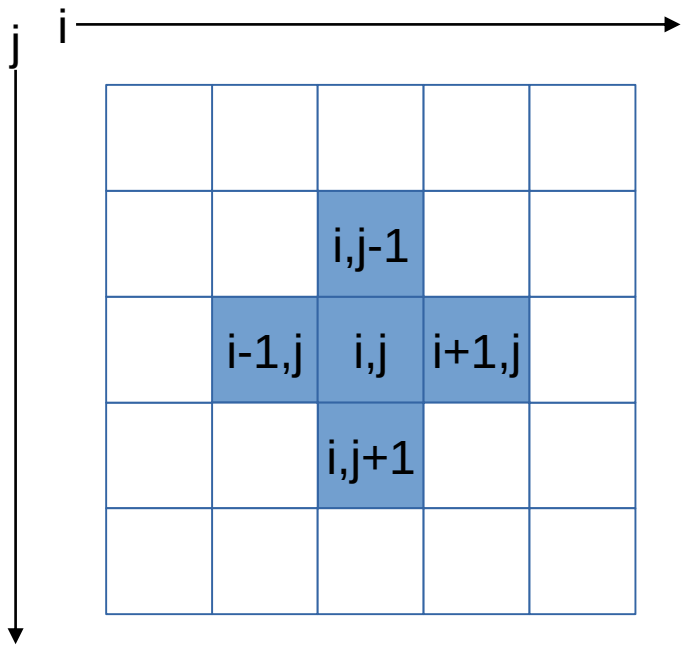
Switch to the other CQA handson folder

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

Hydro code

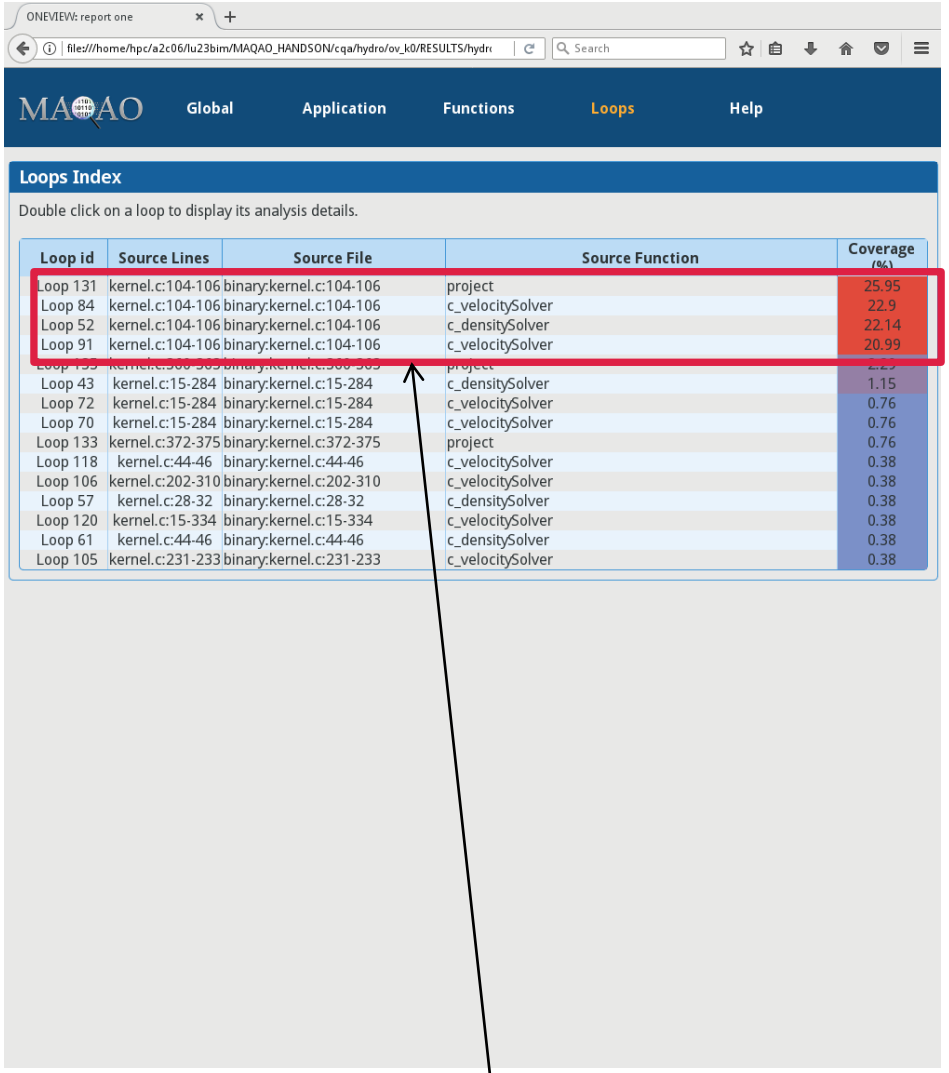
```
int build_index (int i, int j, int grid_size)
{
  return (i + (grid_size + 2) * j);
}

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;
}
```

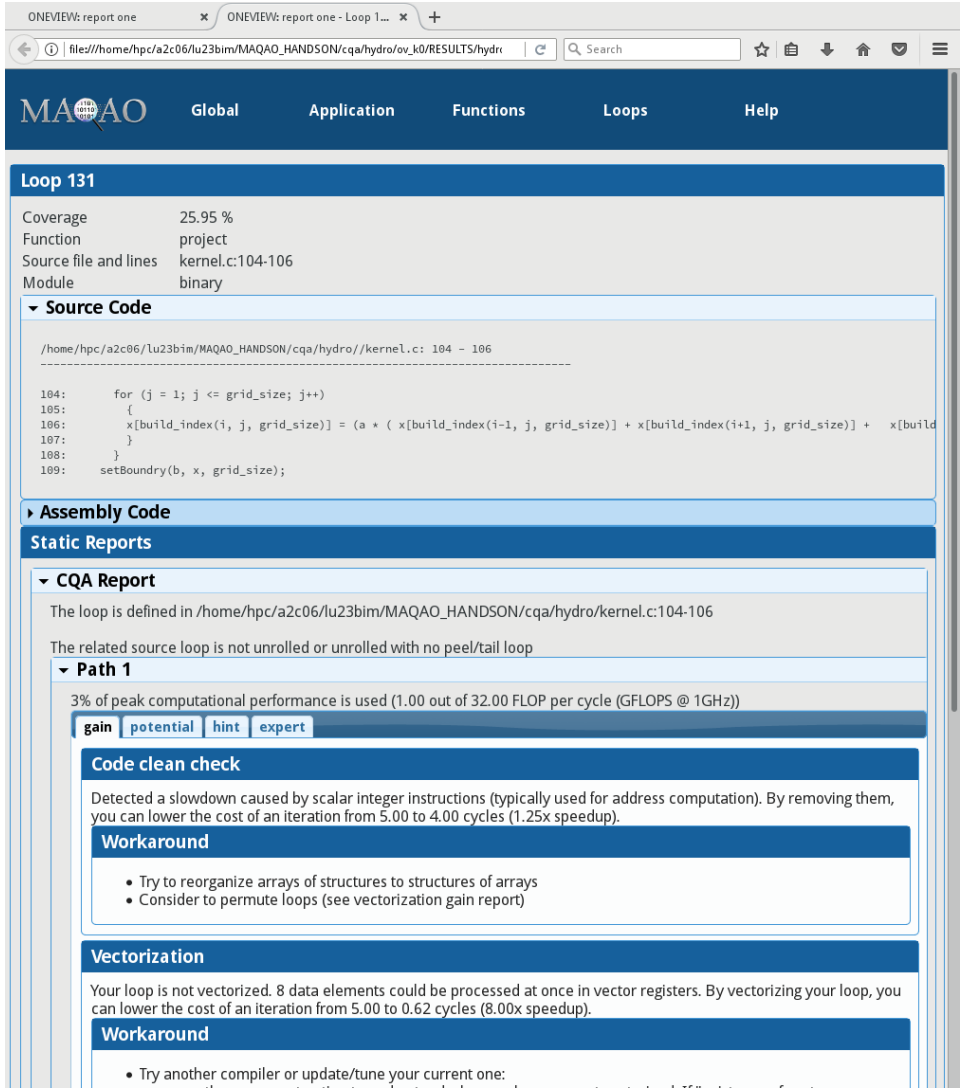
Iterative linear system solver using the Gauss-Siedel relaxation technique.
« Stencil » code



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

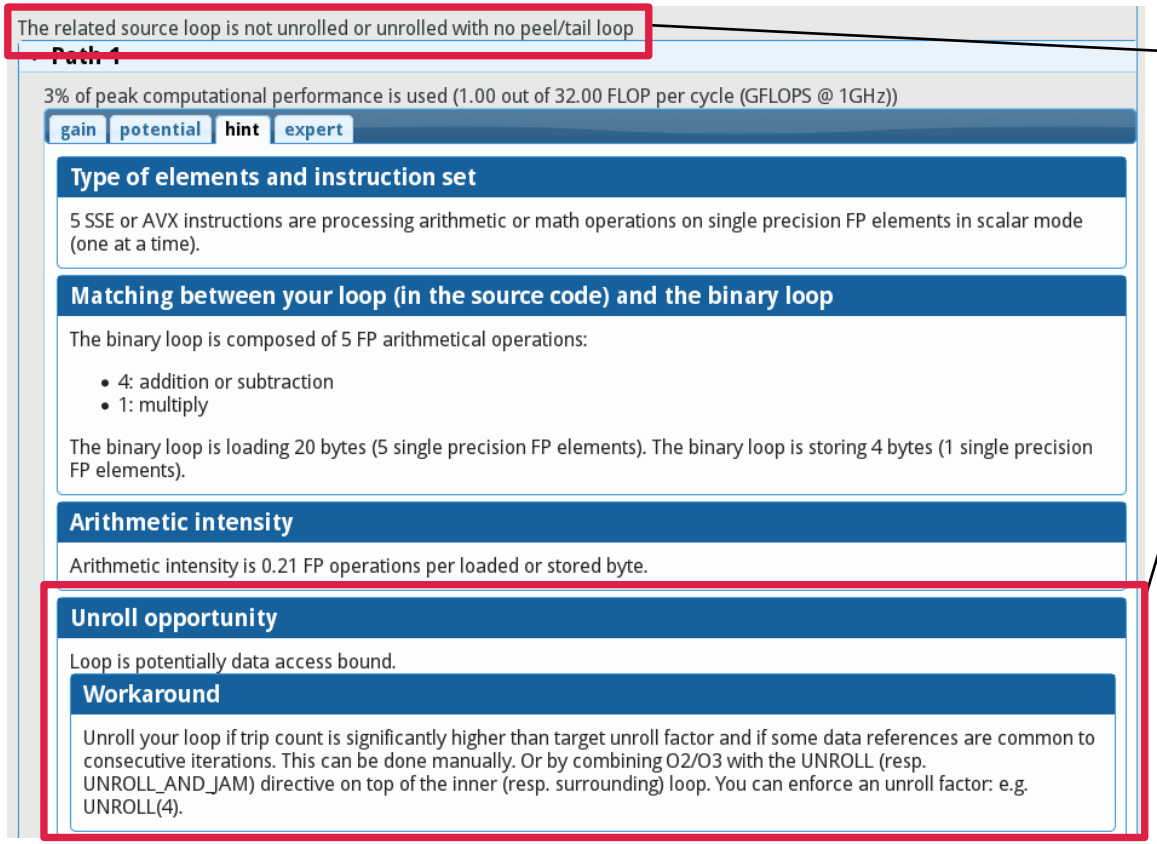
```
> make KERNEL=0
> ./hydro 250 100 # 1st param: grid_size and 2nd param: repet nb
Cycles per element for solvers: 1582.3
> maqao oneview --create-report=one xp=ov_k0 c=ov_cfg.lua
 --format=html
> firefox ov_k0/RESULTS/hydro_one_html/index.html &
```





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.

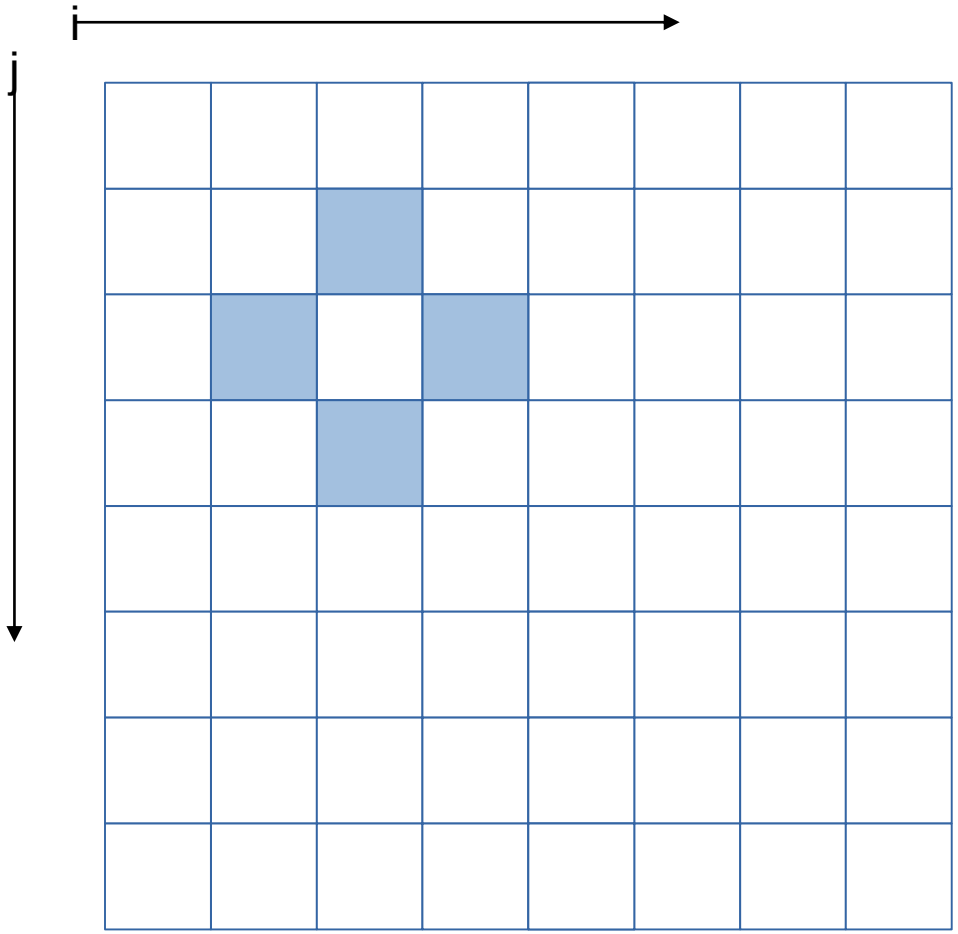
CQA output for kernel0



Unrolling is generally a good deal: fast to apply and often provides gain.

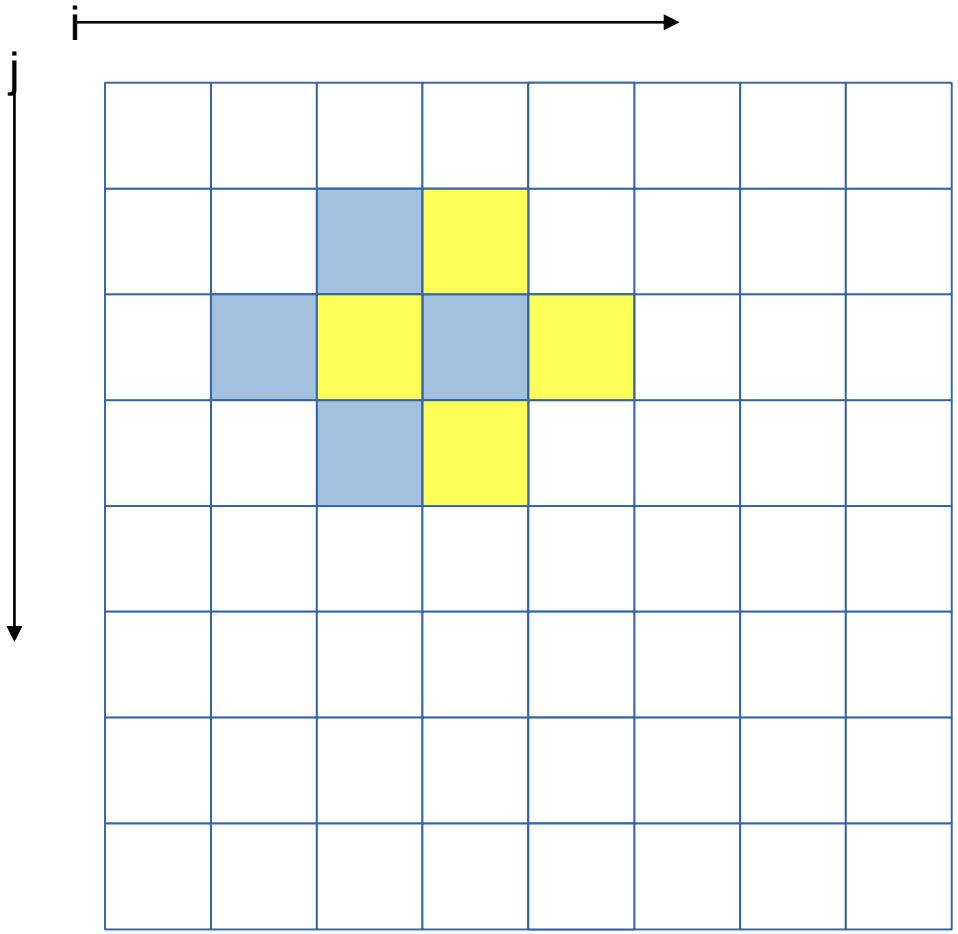
Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



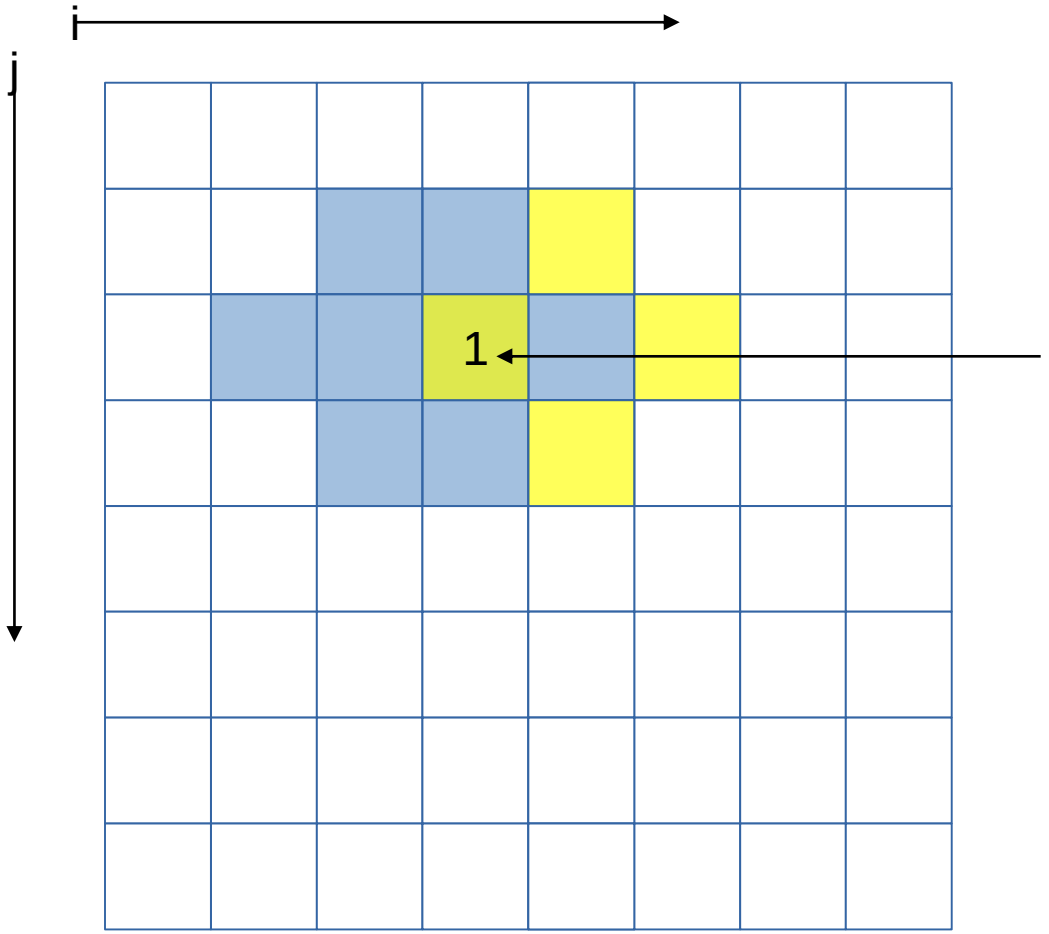

```
LINEAR_SOLVER(i+0,j+0)
```

Memory references reuse : 4x4 unroll footprint on loads




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

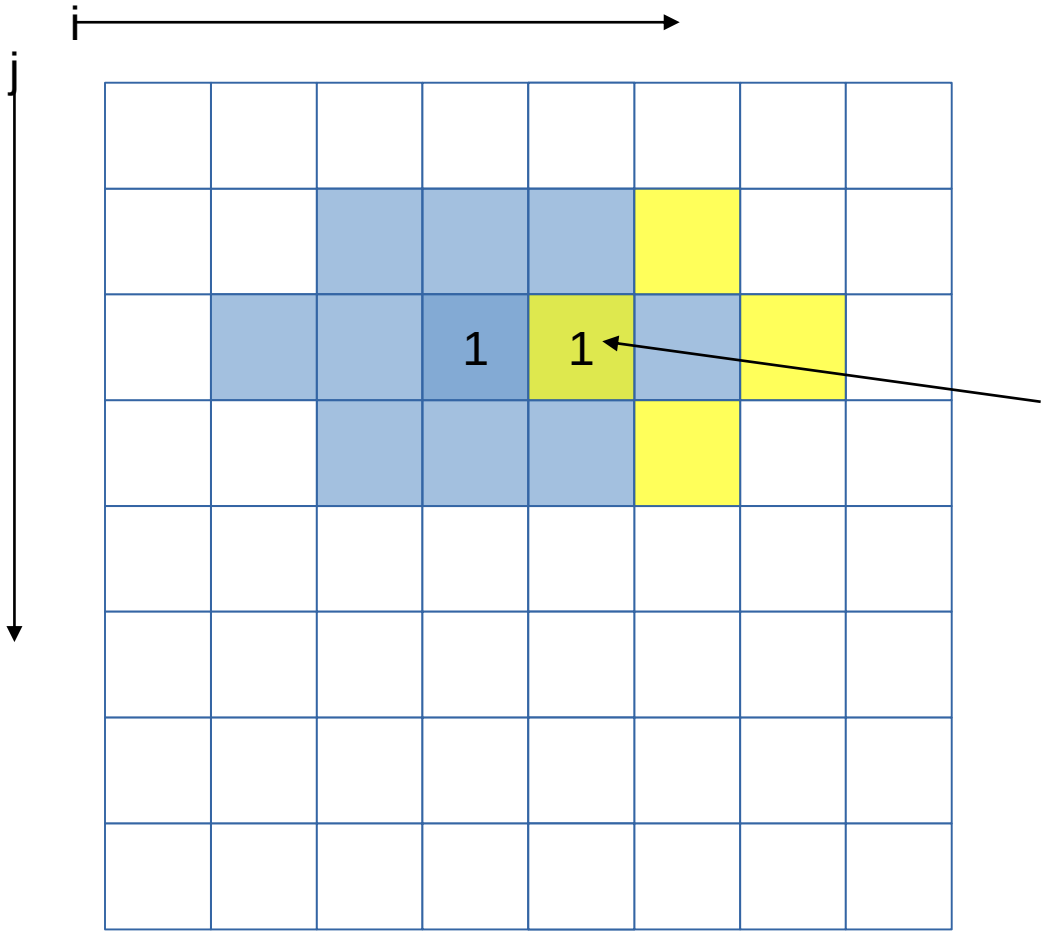
Memory references reuse : 4x4 unroll footprint on loads




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

1 reuse

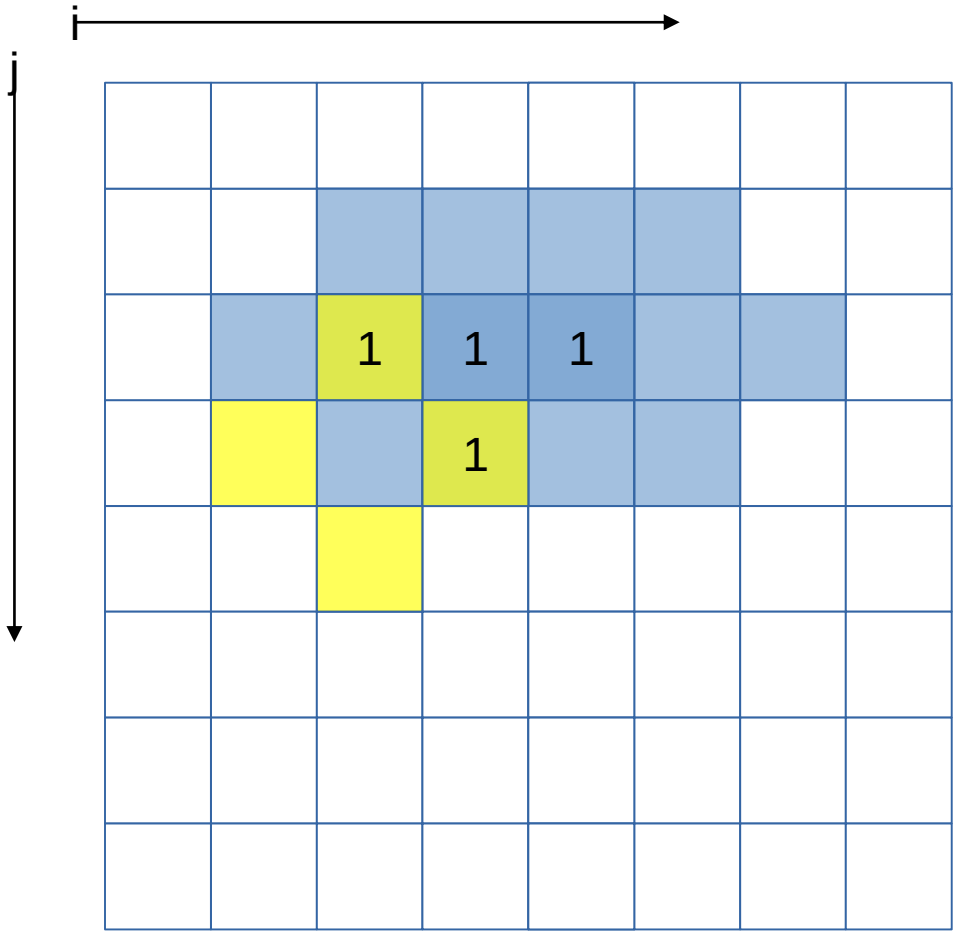
Memory references reuse : 4x4 unroll footprint on loads




```
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

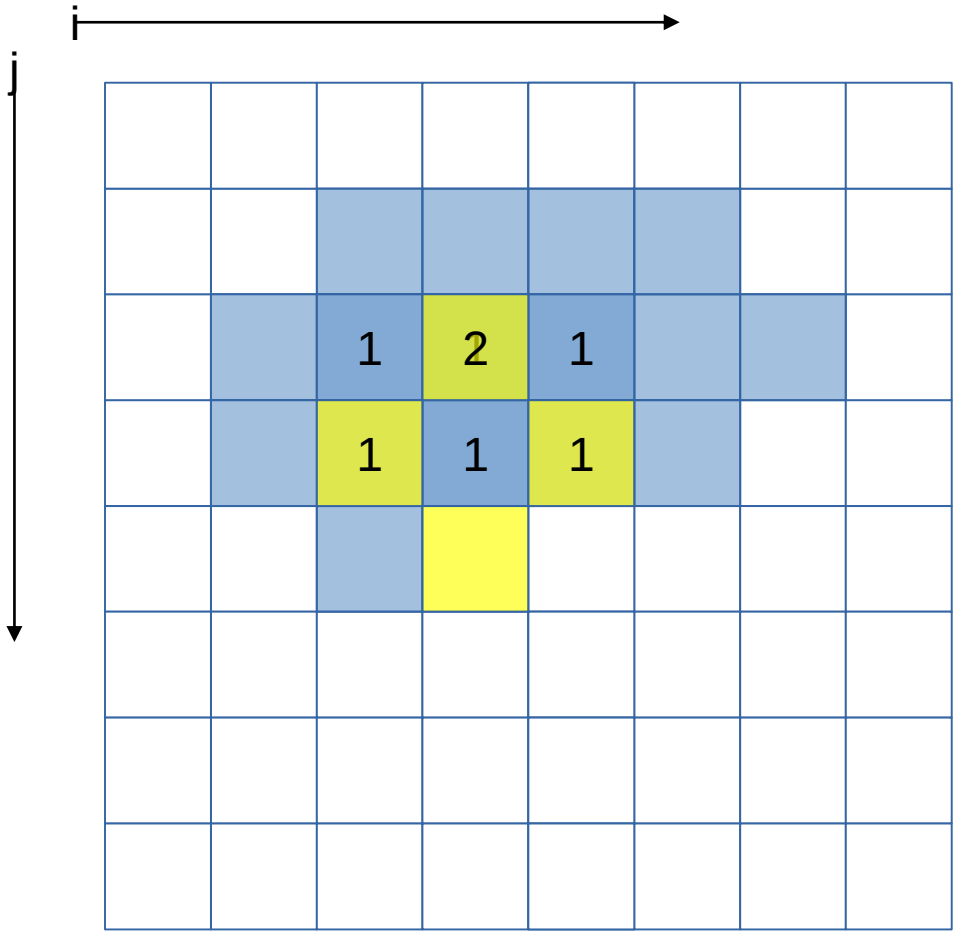
Memory references reuse : 4x4 unroll footprint on loads




```
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

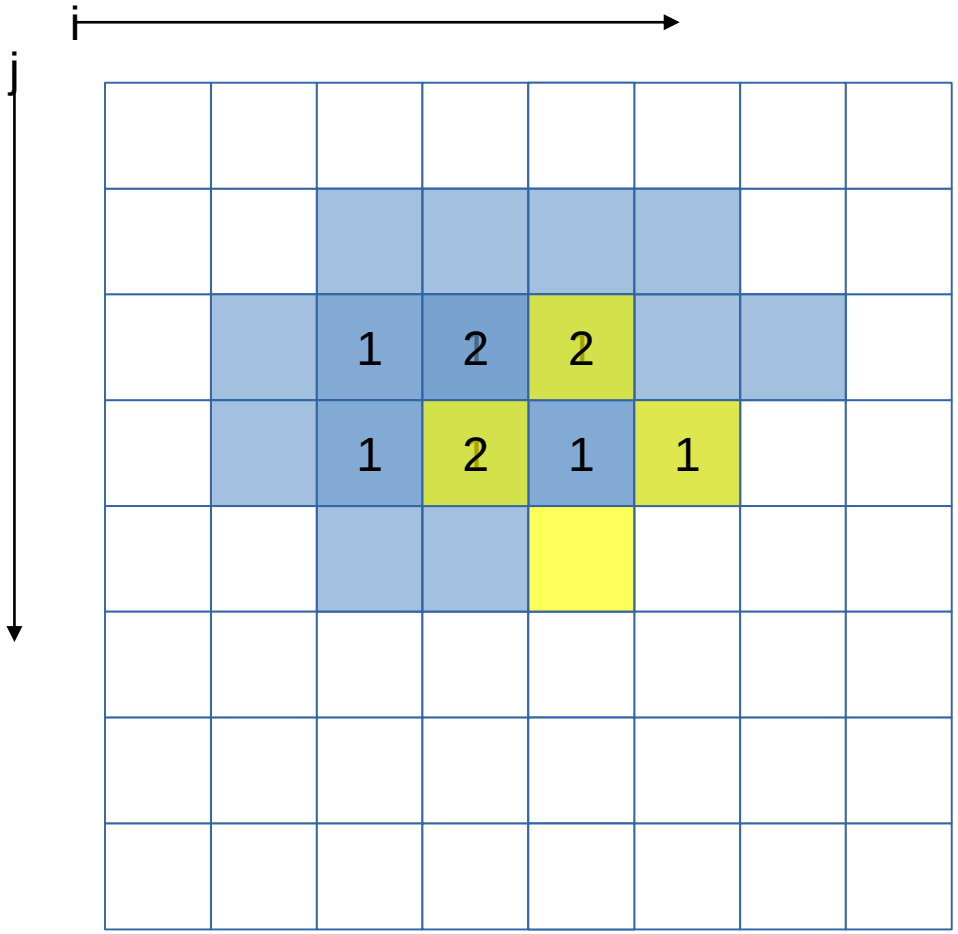
Memory references reuse : 4x4 unroll footprint on loads




```
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

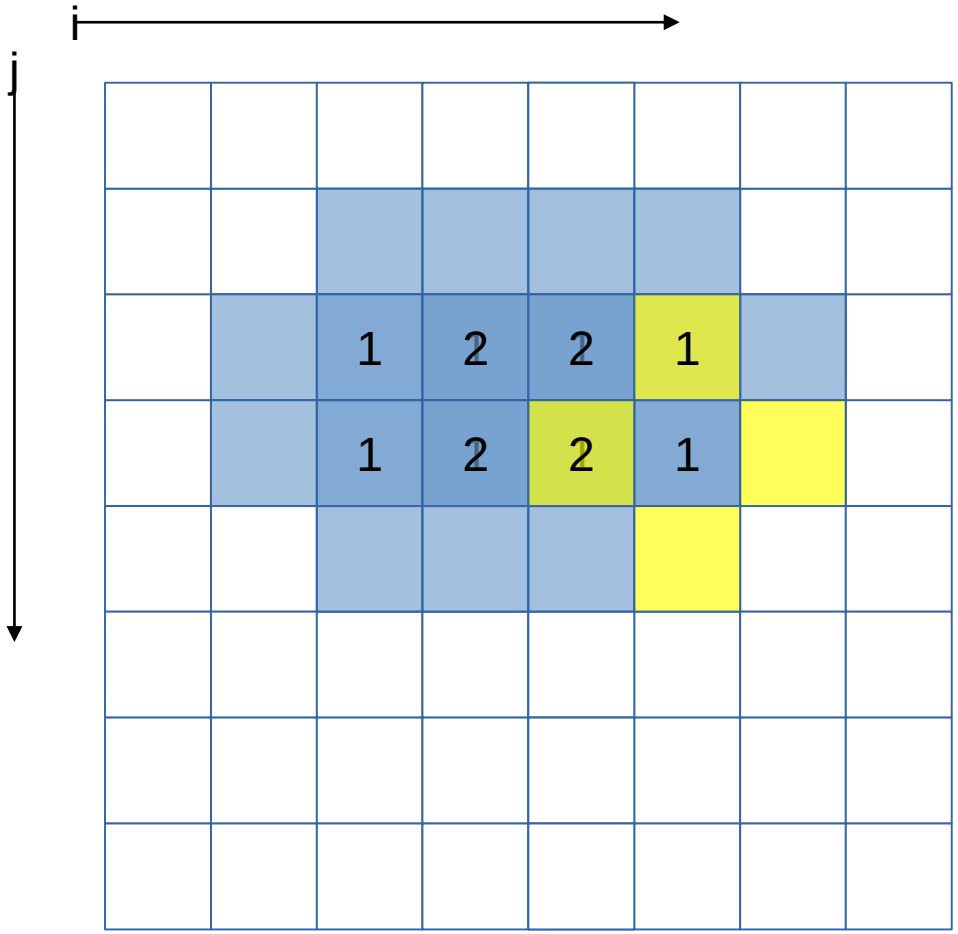
Memory references reuse : 4x4 unroll footprint on loads




```
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

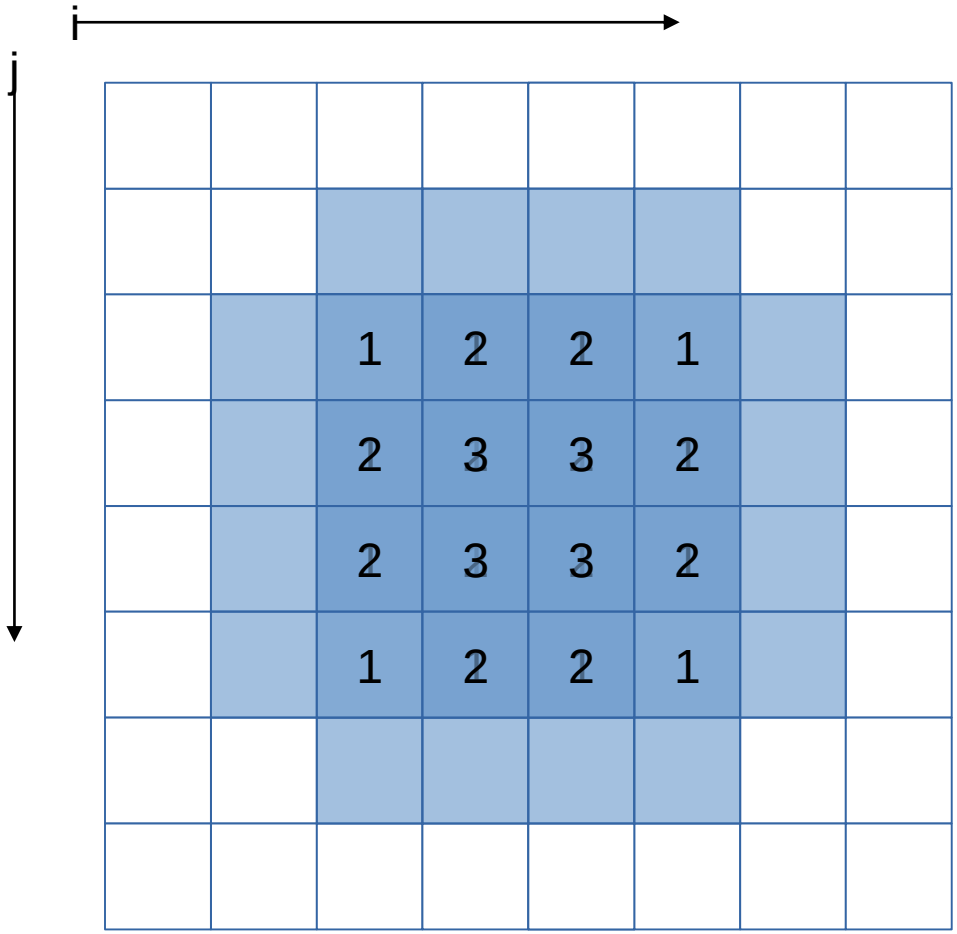
Memory references reuse : 4x4 unroll footprint on loads




```
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

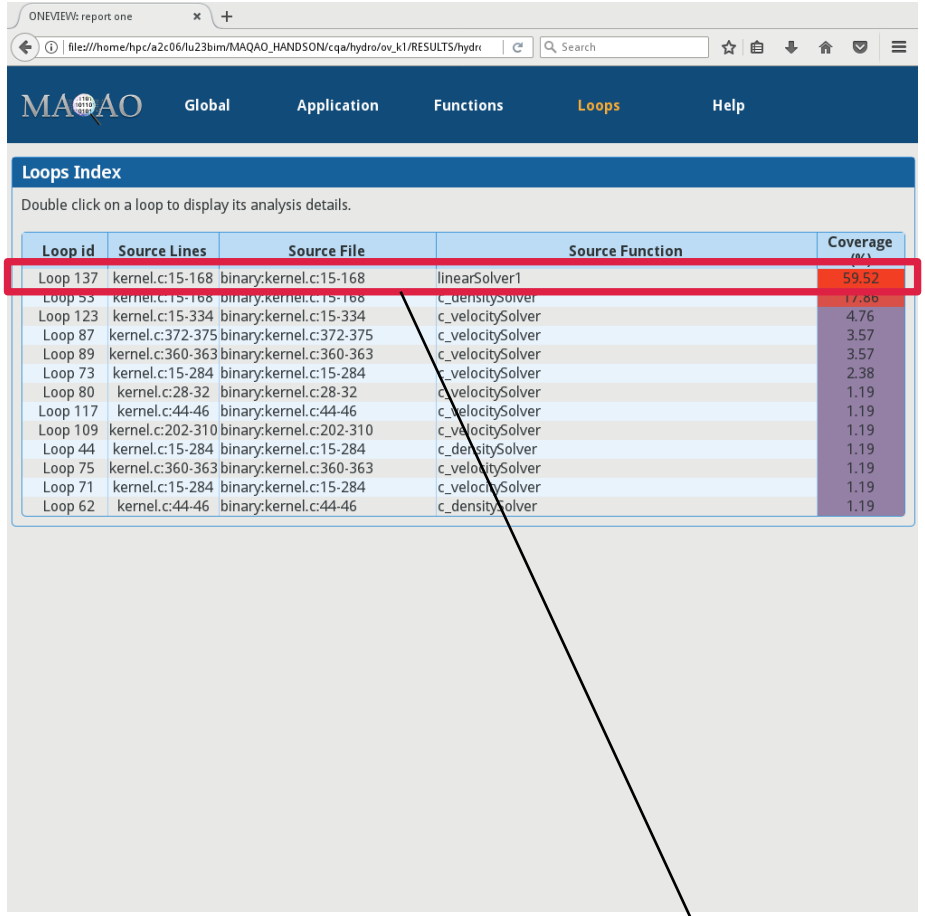
4x4 unroll

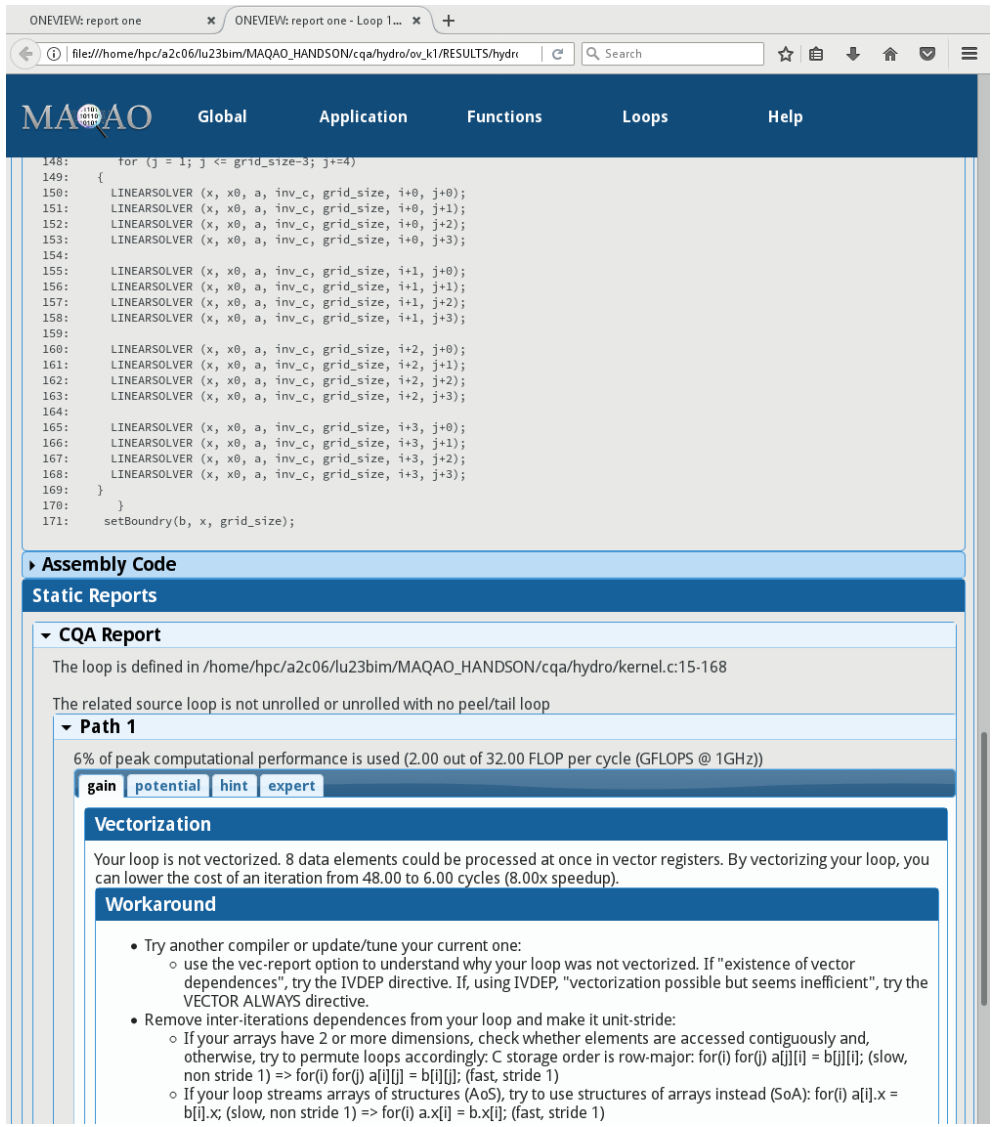
```
#define LINEARSOLVER(...) x[build_index(i, j, grid_size)] = …

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 250 100
Cycles per element for solvers: 571.65
> maqao oneview --create-report=one xp=ov_k1 c=ov_cfg.lua
  --format=html
> firefox ov_k1/RESULTS/hydro_one_html/index.html
```





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

CQA output for kernel1

Matching between your loop (in the source code) and the binary loop

The binary loop is composed of 96 FP arithmetical operations:

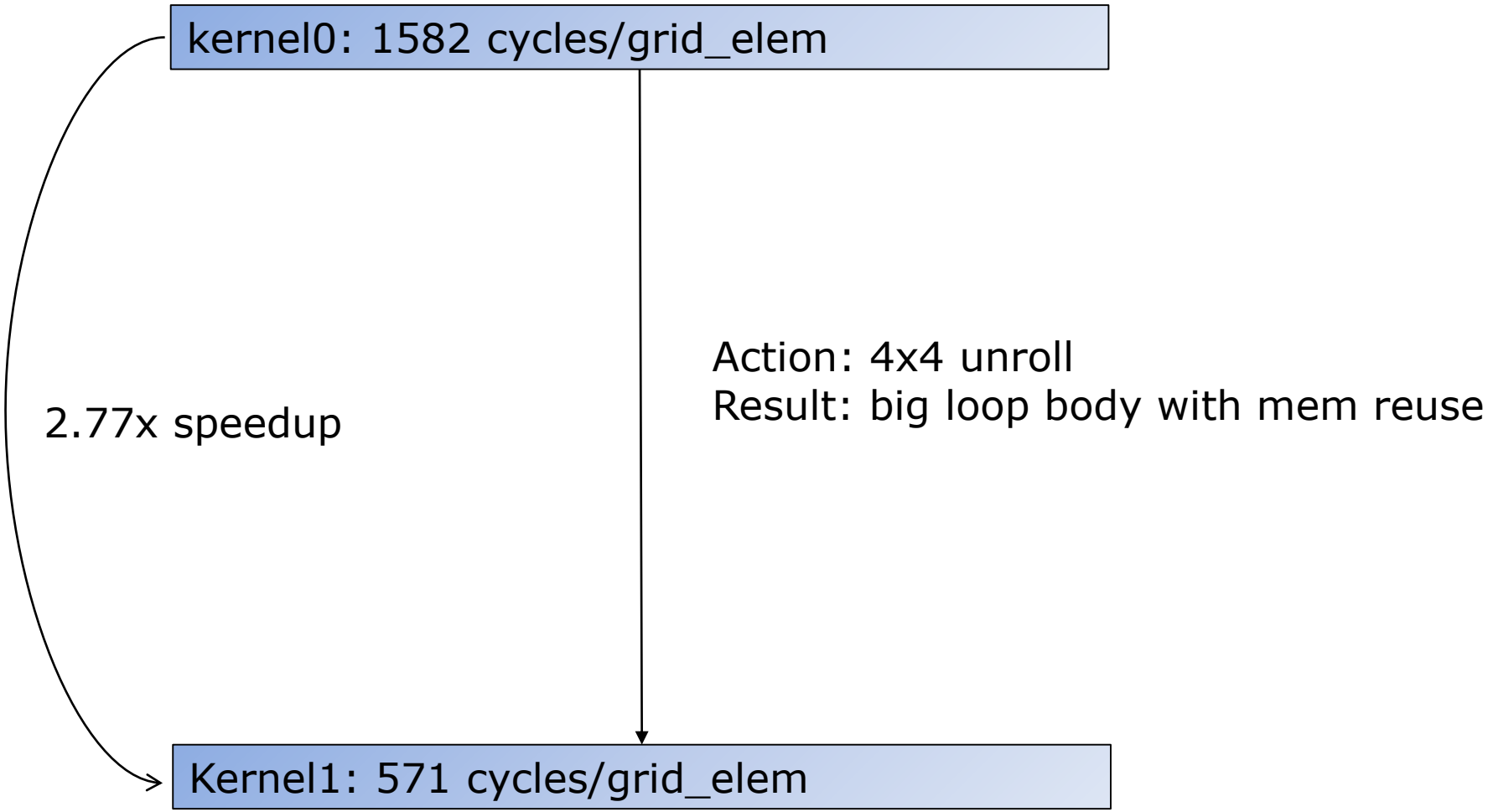
- 64: addition or subtraction
- 32: multiply

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



Kernel1: KNL

```
> salloc --nodes=1 --tasks-per-node=1
> mpiexec -n 1 ./hydro 250 100
Cycles per element for solvers: 1249.20
> maqao oneview --create-report=one xp=ov_KNL c=ov_cfg_KNL.lua
  --format=html
> exit
> firefox ov_KNL/RESULTS/hydro_one_html/index.html
```

More sample codes

More codes to study with MAQAO in

```
MAQAO_DIR/MAQAO_sample_loops.tgz
```