MAQAO
Hands-on exercises

Profiling bt-mz
Scalability profiling of lulesh
Optimising a code

Setup (reminder)

Login to the cluster

```
> ssh <username>@login.archer.ac.uk
```

Load MAQAO environment

```
> source /work/ta002/shared/maqao/load_maqao.sh
```

Copy handson material to your WORK directory

```
> export WORK=/work/ta002/ta002/$USER
> cd $WORK
> tar xvf $MAQAO_DIR/MAQAO_HANDSON.tgz
```

Setup (optional: bt-mz compilation with debug symbols)

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

```
> grep FFLAGS $WORK/NPB3.3-MZ-MPI/config/make.def
```

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

Recompile bt-mz with debug information if necessary

```
> cd $WORK/NPB3.3-MZ-MPI
> module switch PrgEnv-cray PrgEnv-gnu
> make bt-mz CLASS=C NPROCS=8
```

Profiling bt-mz with MAQAO

Salah Ibnamar

Setup ONE View for batch mode

The ONE View configuration file must contain all variables for executing the application.
Retrieve the configuration file prepared for bt-mz in batch mode from the MAQAO_HANDSON directory

```
> cd $WORK/NPB3.3-MZ-MPI/bin
> cp $WORK/MAQAO_HANDSON/bt/config_bt_oneview_pbs.lua .
> less config_bt_oneview_pbs.lua
```

```
binary = "bt-mz_C.8"
...
batch_script = "bt_maqao.pbs"
...
batch_command = "qsub <batch_script>"
...
number_processes = 8
...
omp_num_threads = 6
...
mpi_command = "aprun -n <number_processes> -d <omp_num_threads>"
...
```

Review jobscript for use with ONE View

All variables in the jobscript defined in the configuration file must be replaced with their name from it.
Retrieve jobscript modified for ONE View from the MAQAO_HANDSON directory.

```
> cd $WORK/NPB3.3-MZ-MPI/bin #if current directory has changed
> cp $WORK/MAQAO_HANDSON/bt/bt_maqao.pbs .
> less bt_maqao.pbs
```

```
...
#PBS -l select=2<number_nodes>
...
export OMP_NUM_THREADS=6<omp_num_threads>
...
aprun -n ... $EXE
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on bt-mz (batch mode)

Launch ONE View

```
> cd $WORK/NPB3.3-MZ-MPI/bin #if current directory has changed
> maqao oneview --create-report=one \
config=config_bt_oneview_pbs.lua xp=ov_pbs
```

The -xp parameter allows to set the path to the experiment directory, where ONE View stores the analysis results and where the reports will be generated.
If -xp is omitted, the experiment directory will be named maqao_<timestamp>.

WARNINGS:

- Only directories based in /work are reachable from Archer compute nodes.

- If the directory specified with -xp already exists, ONE View will reuse its content but not overwrite it.

(OPTIONAL) Setup ONE View for interactive mode

Retrieve the configuration file prepared for bt-mz in interactive mode from the MAQAO_HANDSON directory

```
> cd $WORK/NPB3.3-MZ-MPI/bin #if current directory has changed
> cp $WORK/MAQAO_HANDSON/bt/config_bt_oneview_interactive.lua .
> less config_bt_oneview_interactive.lua
```

```
binary = "bt-mz_C.8"
...
number_processes = 8
...
omp_num_threads = 6
...
mpi_command = "aprun -n <number_processes> -d <omp_num_threads>"
```

(OPTIONAL) Launch MAQAO ONE View on bt-mz (interactive mode)

Request interactive session

```
> qsub -I -l select=2,walltime=00:10:00 -q R7133971 -A ta002
```

Load MAQAO environment

```
> source /work/ta002/shared/maqao/load_maqao.sh
```

Launch ONE View

```
> export WORK=/work/ta002/ta002/$USER
> cd $WORK/NPB3.3-MZ-MPI/bin
> maqao oneview --create-report=one \
config=config_bt_oneview_interactive.lua \
xp=ov_interactive
```

Display MAQAO ONE View results

The HTML files are located in **<exp-dir>/RESULTS/<binary>_one_html**, where *<exp-dir>* is the path of he experiment directory (set with -xp) and *<binary>* the name of the executable.

Mount $WORK locally:

```
> mkdir archer_work
> sshfs <username>@login.archer.ac.uk:/work/ta002/ta002/<username> \
archer_work
> firefox archer_work/NPB3.3-MZ-MPI/bin/ov_pbs/RESULTS/bt-
mz_C.8_one_html/index.html
```

It is also possible to compress and download the results to display them:

```
> tar czf $HOME/bt_html.tgz ov_pbs/RESULTS/bt-mz_C.8_one_html
```

```
> scp <login>@login.archer.ac.uk:bt_html.tgz .
> tar xf bt_html.tgz
> firefox ov_pbs/RESULTS/bt-mz_C.8_one_html/index.html
```

Display MAQAO ONE View results (optional)

A sample result directory is in **`MAQAO_HANDSON/bt/bt_html_example.tgz`**

Results can also be viewed directly on the console in text mode:

```
> maqao oneview create-report=one xp=ov_pbs output-format=text
```

Scalability profiling of lulesh with MAQAO

Salah Ibnamar

Compiling Lulesh

Copy Lulesh sources to your working directory

```
> cd $WORK
> tar xvf $MAQAO_DIR/examples/lulesh2.0.3.tgz
```

Compile Lulesh

```
> cd lulesh
> module switch PrgEnv-cray PrgEnv-intel
> make
```

(Optional) To execute a sample run of Lulesh:

```
> less job_lulesh.pbs
> qsub job_lulesh.pbs
```

Setup ONE View for scalability analysis

Retrieve the configuration file prepared for lulesh in batch mode from the MAQAO_HANDSON directory

```
> cd $WORK/lulesh #if current directory has changed
> cp $WORK/MAQAO_HANDSON/lulesh/config_maqao_lulesh.lua .
> less config_maqao_lulesh.lua
```

```
binary = "./lulesh2.0"
…
run_command = "<binary> -i 10 -p -s 130"
…
batch_script = "job_lulesh_maqao.pbs"
…
batch_command = "qsub <batch_script>"
…
number_processes = 1
…
number_nodes = 1
…
mpi_command = "aprun -n <number_processes> -N <number_tasks_nodes> -d <omp_num_threads>"
…
omp_num_threads = 1
…
multiruns_params = {
 {nb_processes = 1, nb_threads = 6, number_nodes = 1, …, run_command = nil, …},
 {nb_processes = 8, nb_threads = 1, number_nodes = 1, …, run_command = "<binary> -i 10 -p -s 65"},
 {nb_processes = 8, nb_threads = 1, number_nodes = 2, …, run_command = "<binary> -i 10 -p -s 65"},
 {nb_processes = 8, nb_threads = 6, number_nodes = 2, …, run_command = "<binary> -i 10 -p -s 65"},
}
```

Review jobscript for use with ONE View

All variables in the jobscript defined in the configuration file must be replaced with their name from it.
Retrieve jobscript modified for ONE View from the MAQAO_HANDSON directory.

```
> cd $WORK/lulesh #if current directory has changed
> cp $WORK/MAQAO_HANDSON/lulesh/job_lulesh_maqao.pbs .
> less job_lulesh_maqao.pbs
```

```
...
#PBS -l select=2<number_nodes>
...
export OMP_NUM_THREADS=6<omp_num_threads>
...
aprun -n ... $EXE
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on lulesh (scalability mode)

Launch ONE View (execution will be longer!)

```
> maqao oneview --create-report=one --with-scalability=on \
config=config_maqao_lulesh.lua xp=maqao_lulesh
```

The results can then be accessed similarly to the analysis report.

```
> firefox
archer_work/lulesh/maqao_lulesh/RESULTS/lulesh2.0_one_html/index.html
```

OR

```
> tar czf $HOME/lulesh_html.tgz \
maqao_lulesh/RESULTS/lulesh2.0_one_html
```

```
> scp <login>@login.archer.ac.uk:lulesh_html.tgz .
> tar xf lulesh_html.tgz
> firefox maqao_lulesh/RESULTS/lulesh2.0_one_html/index.html
```

A sample result directory is in **`MAQAO_HANDSON/lulesh/lulesh_html_example.tgz`**

Optimising a code with MAQAO

Emmanuel OSERET

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

Preparing interactive session with GNU compiler

Request interactive session and load MAQAO environment

```
> qsub -I -l select=1,walltime=00:20:00 -q R7133971 -A ta002
> source /work/ta002/shared/maqao/load_maqao.sh
```

Load GCC compiler

```
> module switch PrgEnv-cray PrgEnv-gnu
```

Define variable for work directory

```
> export WORK=/work/ta002/ta002/$USER
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul
> make matmul_orig
```

Parameters are: <size of matrix> <number of repetitions>

```
> aprun -n 1 ./matmul_orig 390 300
cycles per FMA: 3.66
```

Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one mpi_command="aprun -n 1" \
binary=./matmul_orig run-command="<binary> 390 300" \
xp=ov_orig
```

OR, using a configuration script:

```
> maqao oneview create-report=one c=ov_orig.lua xp=ov_orig
```

Viewing results (HTML)

```
> tar czf $HOME/ov_orig.tgz ov_orig/RESULTS/matmul_orig_one_html
```

```
> scp <login>@login.archer.ac.uk:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		22.28
Time in loops (%)		99.94
Time in innermost loops (%)		99.75
Time in user code (%)		99.94
Compilation Options		**binary**: -march=(target) is missing. -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		83.32
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	2.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.97
	Nb Loops to get 80%	1

Viewing results (text)

```
> maqao oneview create-report=one xp=ov_orig \
  output-format=text --text-global | less
```

```
+-------------------------------------------------------------------------------+
+                                1.2  -  Global Metrics                         +
+-------------------------------------------------------------------------------+

  Total Time:                          22.28 s
  Time spent in loops:                 99.94 %
  Compilation Options:                 binary:  -march=(target) is missing. -funroll-
loops is missing.
  Flow Complexity:           1.00
  Array Access Efficiency:   83.32 %
  If Clean:
      Potential Speedup:     1.00
      Nb Loops to get 80%:   1
  If FP Vectorized:
      Potential Speedup:     2.00
      Nb Loops to get 80%:   1
  If Fully Vectorized:
      Potential Speedup:     7.97
      Nb Loops to get 80%:   1
```

Viewing results (text)

```
+-------------------------------------------------------------------------------+
+                              1.3  -  Potential Speedups                       +
+-------------------------------------------------------------------------------+

  If Clean:
      Number of loops    | 1       | 2       |
      Cumulated Speedup  | 1.0010  | 1.0010  |
  Top 5 loops:
    matmul_orig - 3:     1.001
    matmul_orig - 2:     1.001

  If FP Vectorized:
      Number of loops    | 1       | 2       |
      Cumulated Speedup  | 1.9950  | 1.9950  |
  Top 5 loops:
    matmul_orig - 2:     1.995
    matmul_orig - 3:     1.995

  If Fully Vectorized:
      Number of loops    | 1       | 2       |
      Cumulated Speedup  | 7.8624  | 7.9665  |
  Top 5 loops:
    matmul_orig - 2:     7.8624
    matmul_orig - 3:     7.9665
```

Loop ID

Viewing CQA output (text)

```
> maqao oneview create-report=one xp=ov_orig \
  output-format=text text-cqa=2 | less
```

Loop ID

```
      Vectorization
  --------------------
Your loop is not vectorized.
8 data elements could be processed at once in vector registers.
By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.37 cycles
(8.00x speedup).

Details
All SSE/AVX instructions are used in scalar version (process only one data element in
vector registers).
Since your execution units are vector units, only a vectorized loop can use their full
power.

Workaround
 - Try another compiler or update/tune your current one:
  * recompile with fassociative-math (included in Ofast or ffast-math) to extend loop
vectorization to FP reductions.
 - (…)
```

CQA output for the baseline kernel

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.37 cycles (8.00x speedup).

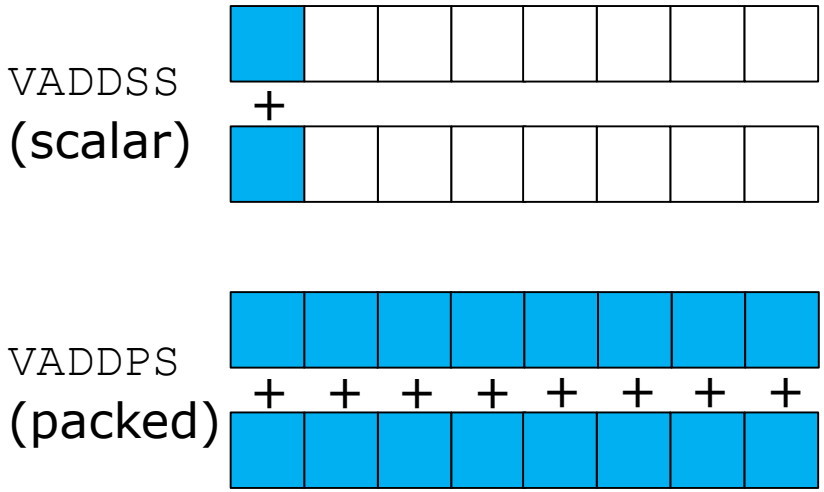
Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- 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)

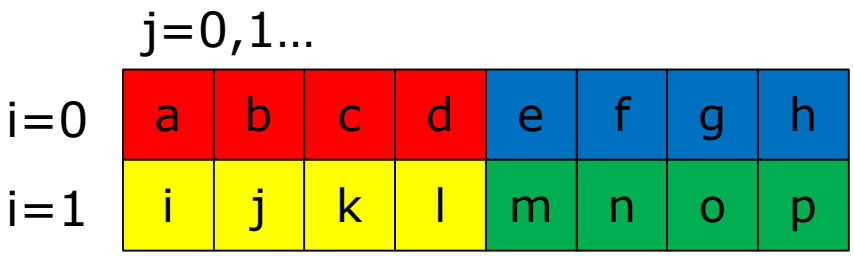
Vectorization (summing elements):



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

Impact of loop permutation on data access

Logical mapping

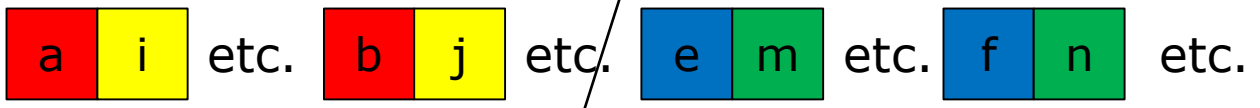


Efficient vectorization + prefetching

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];
  }
}
```

Analyse matrix multiply with permuted loops

Compile permuted loops version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> make matmul_perm
```

```
> aprun -n 1 ./matmul_perm 390 300
cycles per FMA: 0.78
```

Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one mpi_command="aprun -n 1" \
binary=./matmul_perm run-command="<binary> 390 300" \
xp=ov_perm
```

OR, using a configuration script:

```
> maqao oneview create-report=one c=ov_perm.lua xp=ov_perm
```

Viewing results (HTML)

```
> tar czf $HOME/ov_perm.tgz ov_perm/RESULTS/matmul_perm_one_html
```

```
> scp <login>@login.archer.ac.uk:ov_perm.tgz .
> tar xf ov_perm.tgz
> firefox ov_perm/RESULTS/matmul_perm_one_html/index.html &
```

CQA output after loop permutation

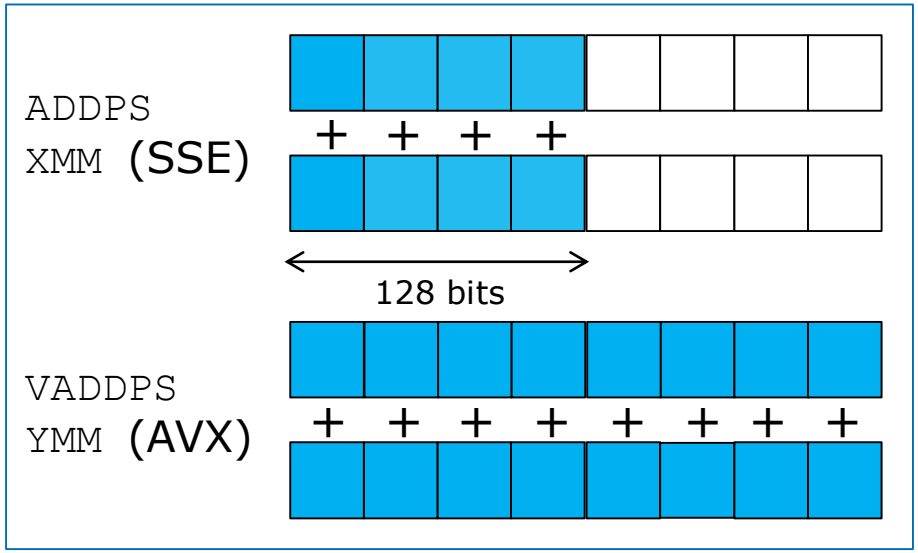
```
> maqao oneview create-report=one xp=ov_perm \
  output-format=text text-cqa=5 | less
```

```
      Vectorization
  --------------------
Your loop is vectorized, but using only 128 out of 256 bits (SSE/AVX-128 instructions on
AVX/AVX2 processors).
By fully vectorizing your loop, you can lower the cost of an iteration from 2.00 to 1.25
cycles (1.60x speedup).

Details
All SSE/AVX instructions are used in vector version (process two or more data elements in
vector registers).Since your execution units are vector units, only a fully vectorized
loop can use their full power.


Workaround
 - Recompile with march=core-avx-i.
CQA target is Core_i7X_Xeon_E5E7_v2 (…) but specialization flags are -march=x86-64
 - …
```

Let's add –march=core-avx-i

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>
```

Analyse matrix multiply with architecture specialisation

Compile architecture specialisation version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> make matmul_perm_opt
```

```
> aprun -n 1 ./matmul_perm_opt 390 300
cycles per FMA: 0.56
```

Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one mpi_command="aprun -n 1" \
binary=./matmul_perm_opt run-command="<binary> 390 300" \
xp=ov_perm_opt
```

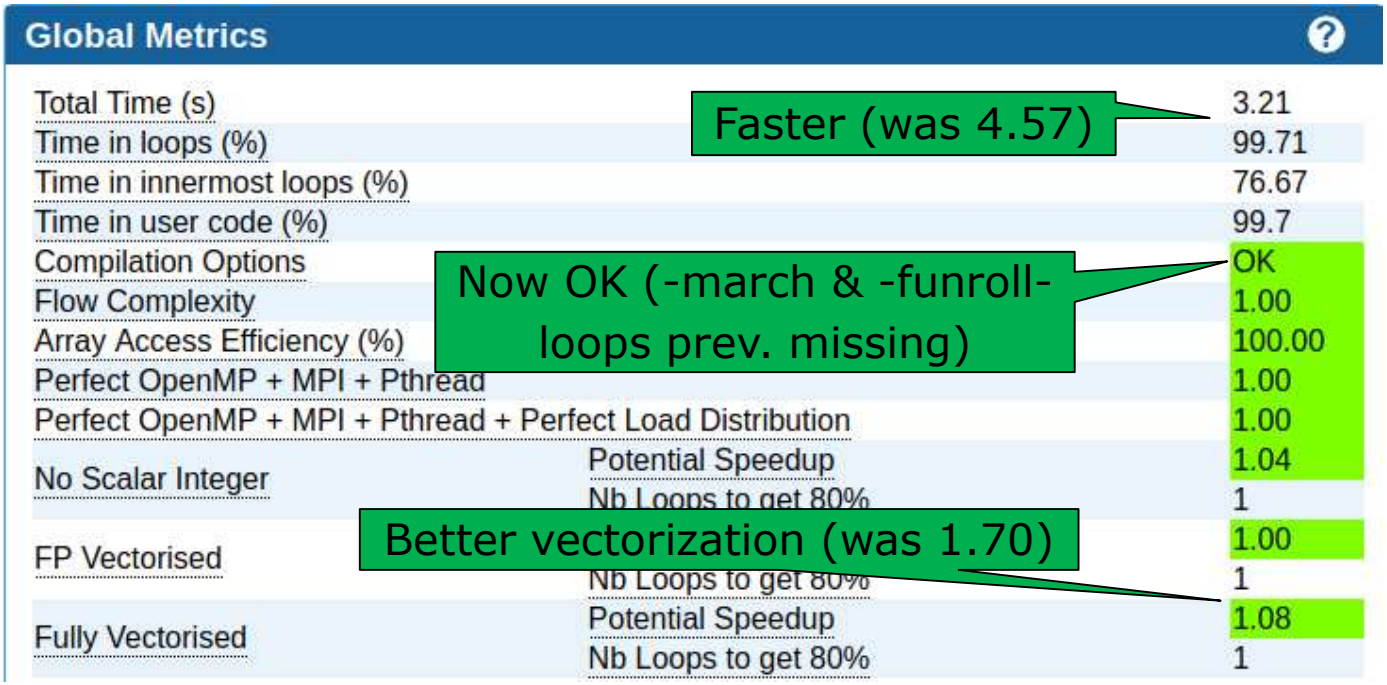
OR using configuration script:

```
> maqao oneview create-report=one c=ov_perm_opt.lua xp=ov_perm_opt
```

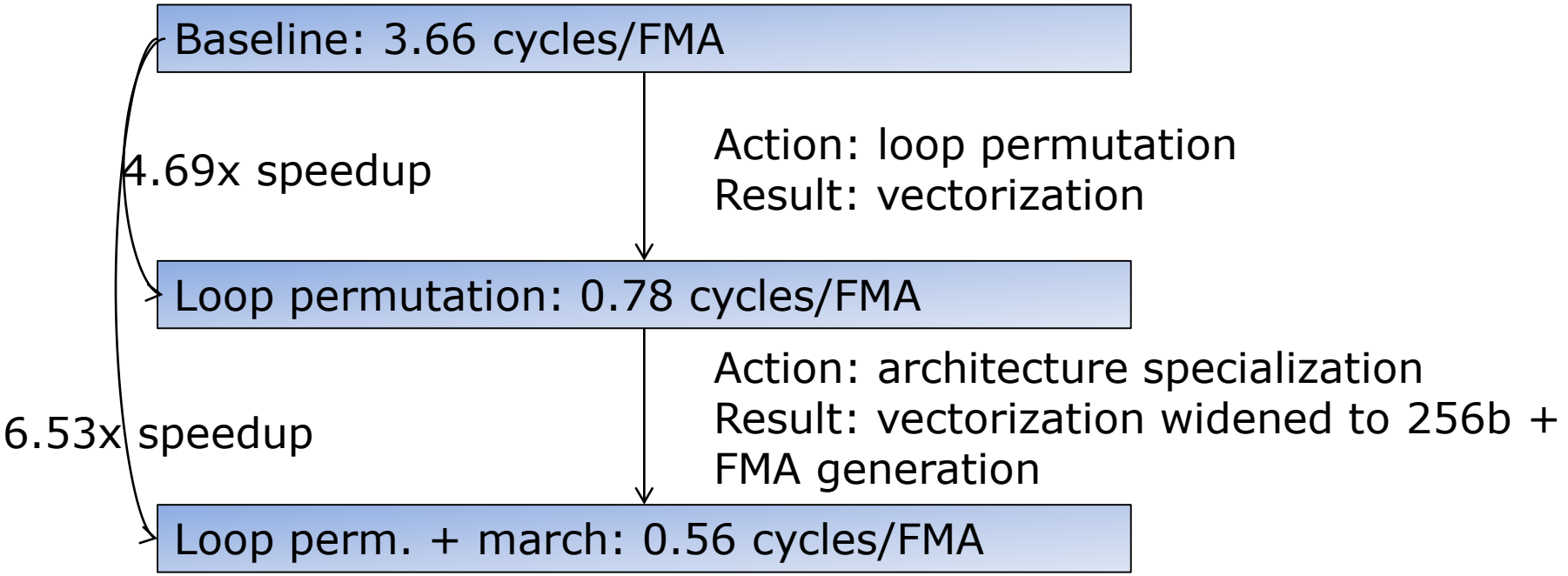
Viewing results (HTML)

```
> tar czf $HOME/ov_opt.tgz ov_perm_opt/RESULTS/matmul_perm_opt_one_html
```

```
> scp <login>@login.archer.ac.uk:ov_opt.tgz .
> tar xf ov_opt.tgz
> firefox ov_perm_opt/RESULTS/matmul_perm_opt_one_html/index.html &
```



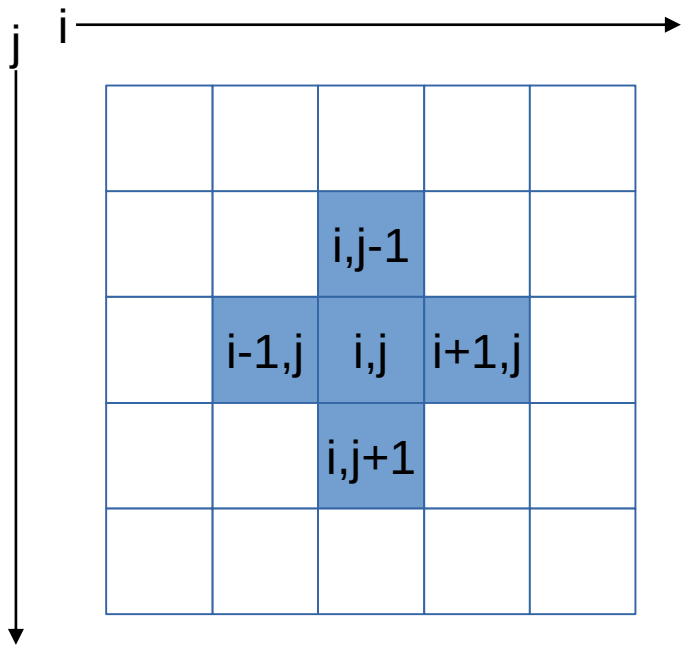
Summary of optimizations and gains



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



Preparing (new) interactive session with Intel compiler

If necessary, exit any active interactive session

```
<username>@momX:~> exit
logout
```

Request interactive session and load MAQAO environment

```
> qsub -I -l select=1,walltime=00:20:00 -q R7133971 -A ta002
> source /work/ta002/shared/maqao/load_maqao.sh
```

Load Intel compiler

```
> module switch PrgEnv-cray PrgEnv-intel
```

Define variable for work directory

```
> export WORK=/work/ta002/ta002/$USER
```

Hydro example

Switch to the hydro handson folder

```
> cd $WORK/MAQAO_HANDSON/hydro
```

Compile

```
> make
```

Running and analyzing kernel0 (icc -O3 -xHost)

```
> aprun -n 1 ./hydro_k0 300 50
Cycles per element for solvers: 2179.48
```

- Profile with MAQAO

```
> maqao oneview create-report=one xp=ov_k0 c=ov_k0.lua
```

- Display results

```
> maqao oneview create-report=one xp=ov_k0 \
output-format=text --text-global | less
```

```
+-------------------------------------------------------------+
+                      1.2  -  Global Metrics                  +
+-------------------------------------------------------------+

  Total Time:                          3.81 s
  Time spent in loops:                 99.46 %
  Compilation Options:                 OK
  Flow Complexity:                     1.04
  Array Access Efficiency:             50.34 %
```

Loop id	Source Location	Source Function	Coverage (%)
141	hydro_k0 - kernel.c:104-110	project	27.96
55	hydro_k0 - kernel.c:104-110	c_densitySolver	20.86
88	hydro_k0 - kernel.c:104-110	c_velocitySolver	20.84
95	hydro_k0 - kernel.c:104-110	c_velocitySolver	20.77
46	hydro_k0 - kernel.c:15-292	c_densitySolver	1.71
76	hydro_k0 - kernel.c:15-292	c_velocitySolver	1.38
74	hydro_k0 - kernel.c:15-292	c_velocitySolver	1.28
145	hydro_k0 - kernel.c:368-371	project	1.26
143	hydro_k0 - kernel.c:380-383	project	1.13

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

Source Code

/fs4/ta002/ta002/eoseret/MAQAO_HANDSON_TW34/hydro//kernel.c: 104 - 110

```
104:       for (j = 1; j <= grid_size; j++)
105:         {
106:         x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] +
107:                                             x[build_index(i+1, j, grid_size)] +
108:                                             x[build_index(i, j-1, grid_size)] +
109:                                             x[build_index(i, j+1, grid_size)]) +
110:                                           x0[build_index(i, j, grid_size)]) / c;
111:         }
112:       }
```

CQA

Path 0 / 1 OK

Average path: Display a virtual path defined by average values of all real paths

Coverage 27.96 %
Function project
Source file and lines kernel.c:104-110
Module hydro_k0
The loop is defined in /fs4/ta002/ta002/eoseret/MAQAO_HANDSON_TW34/hydro/kernel.c:104-110.

The related source loop is not unrolled or unrolled with no peel/tail loop.

gain | potential | hint | expert

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 5.00 to 4.00 cycles (1.25x speedup).

Workaround

- Try to reorganize arrays of structures to structures of arrays
- Consider to permute loops (see vectorization gain report)

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 5.00 to 0.62 cycles (8.00x speedup).

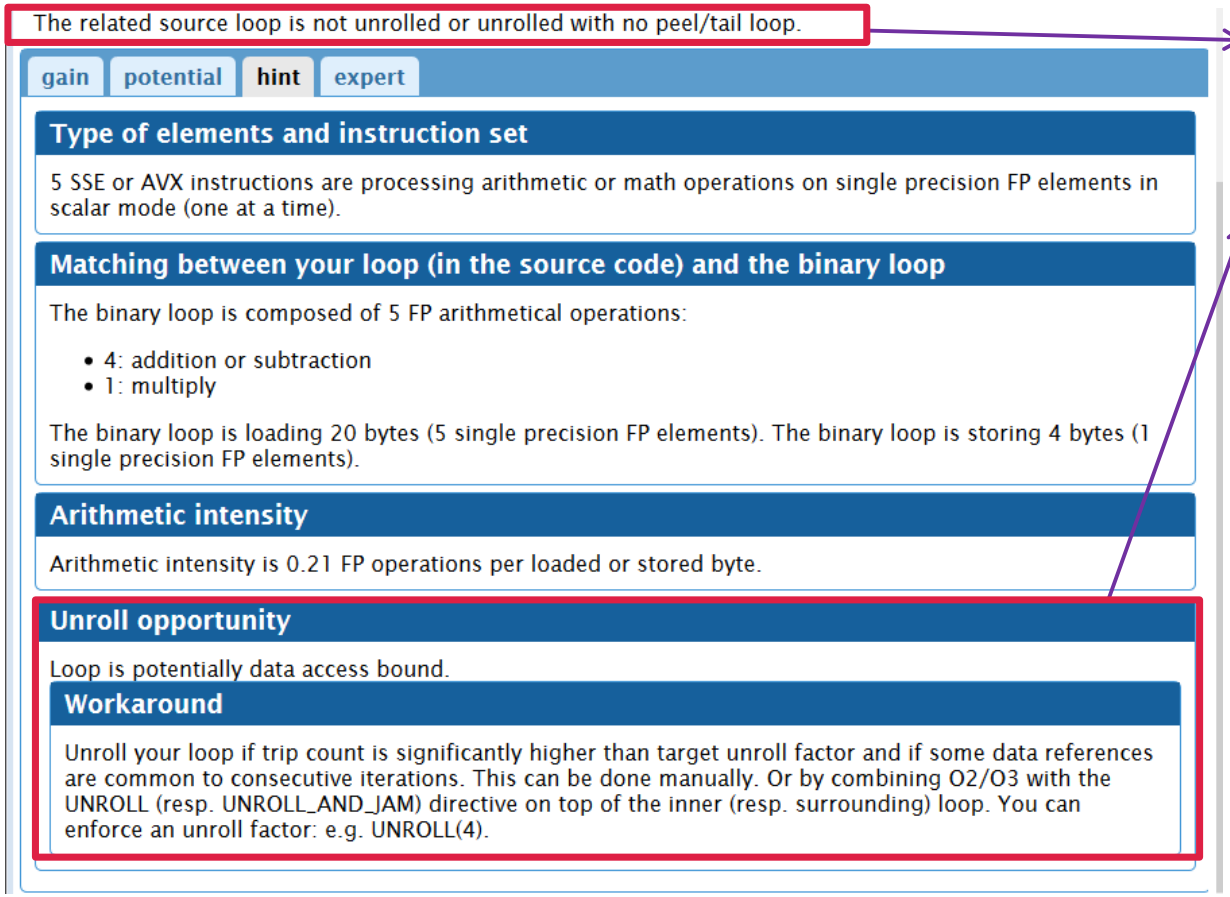
Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- 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]

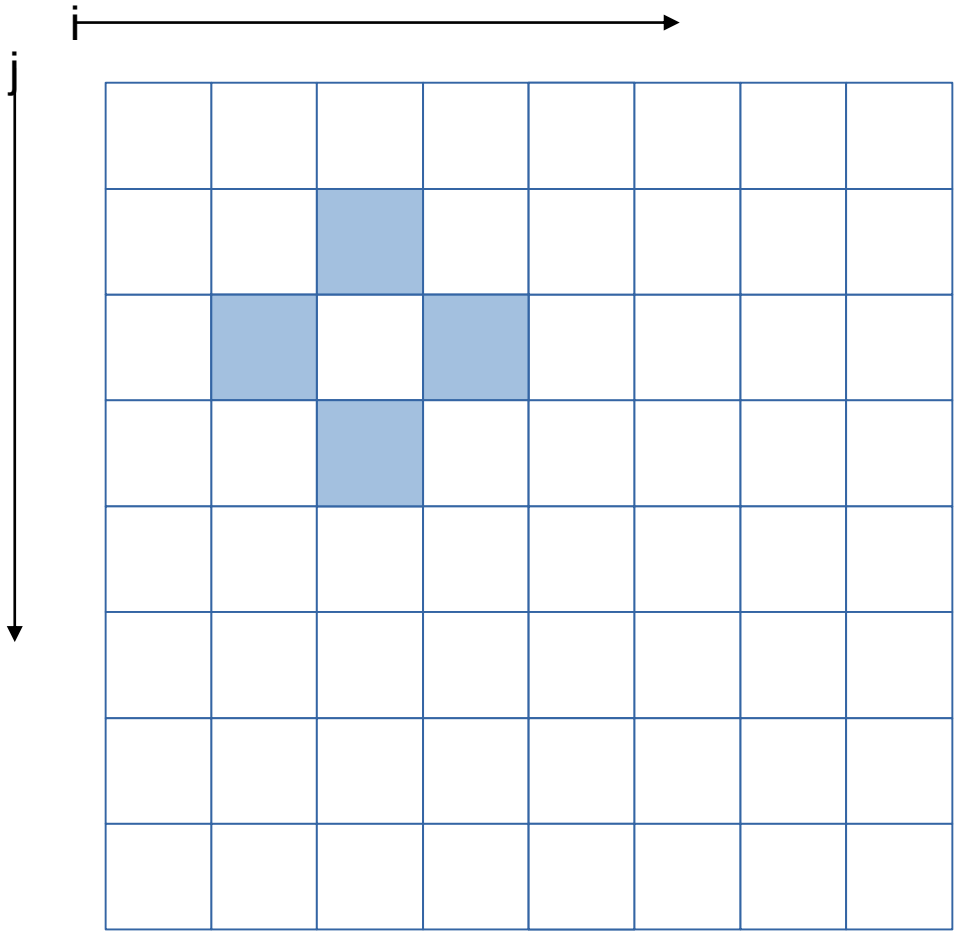
CQA output for kernel0



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

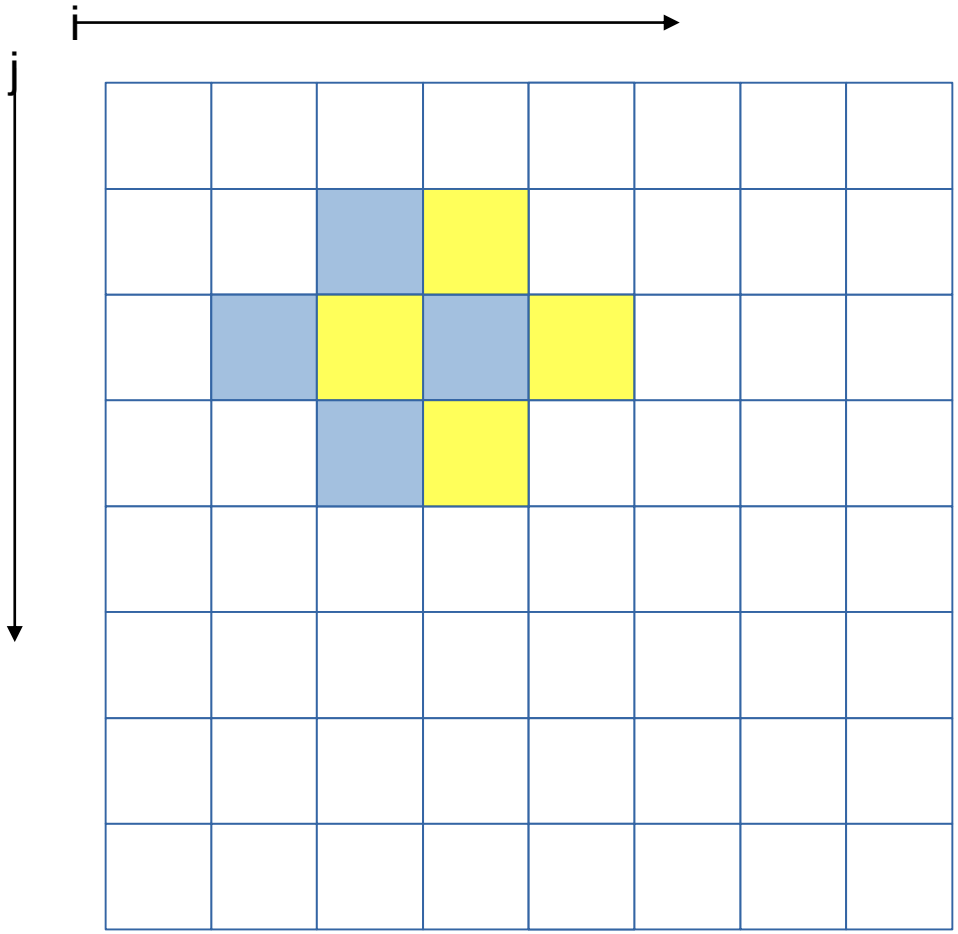
Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



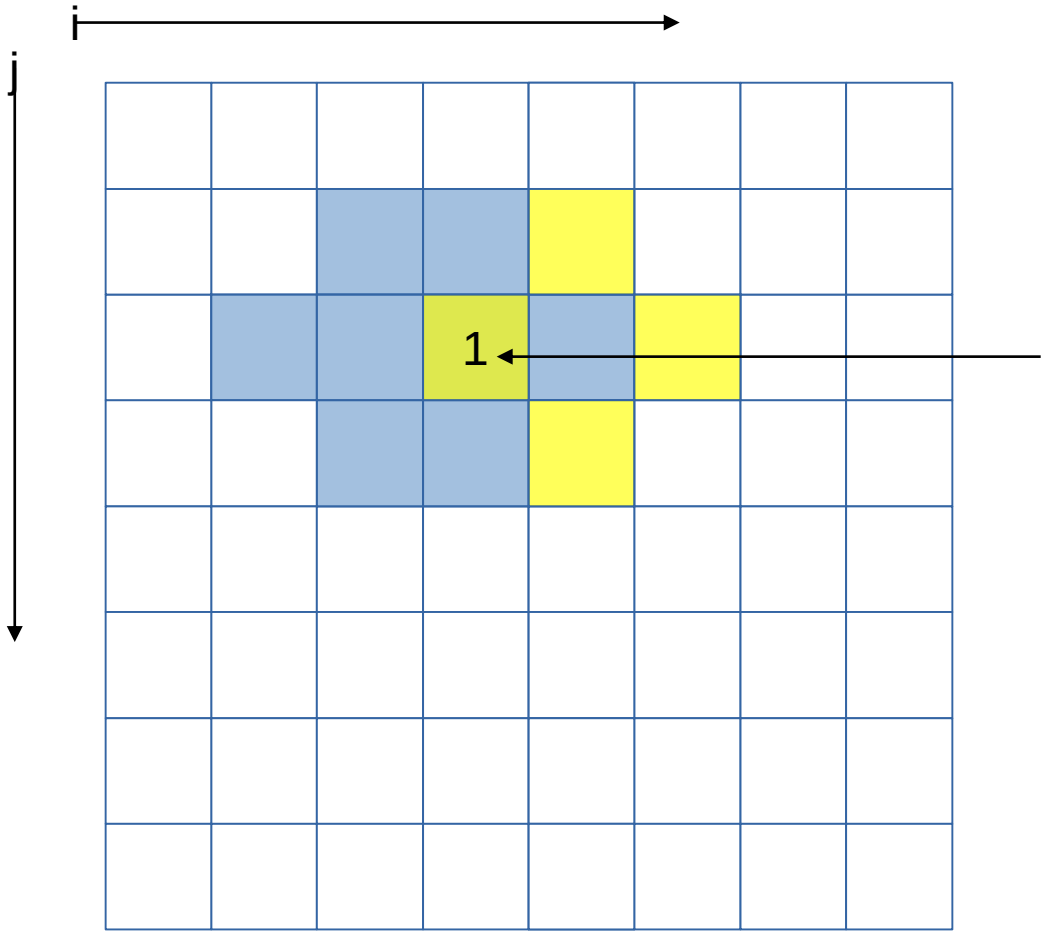
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

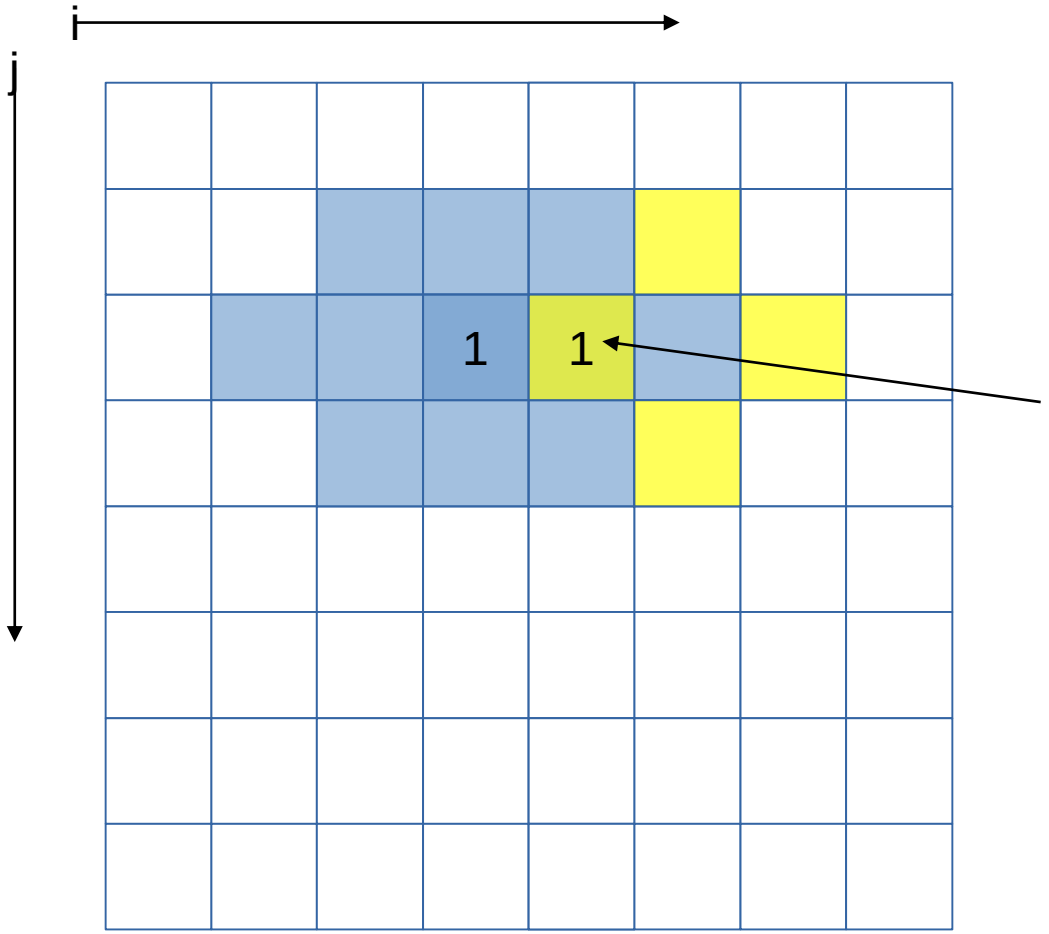
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

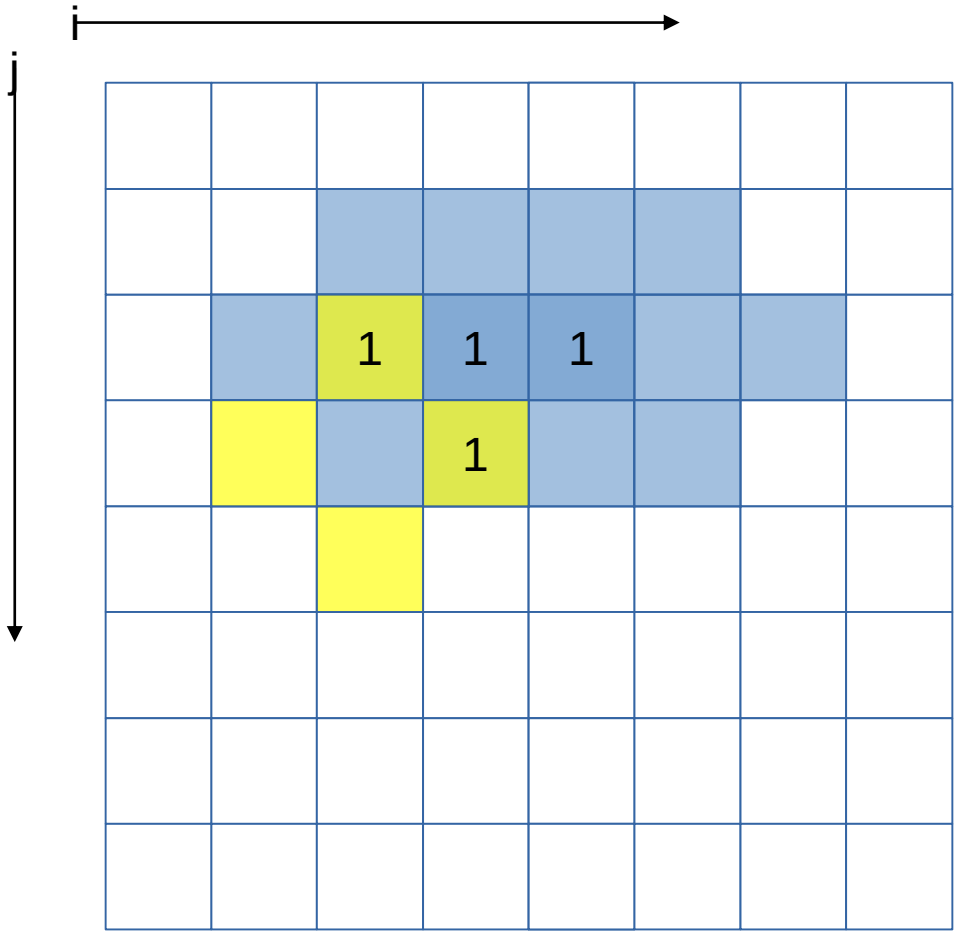
Memory references reuse : 4x4 unroll footprint on loads



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

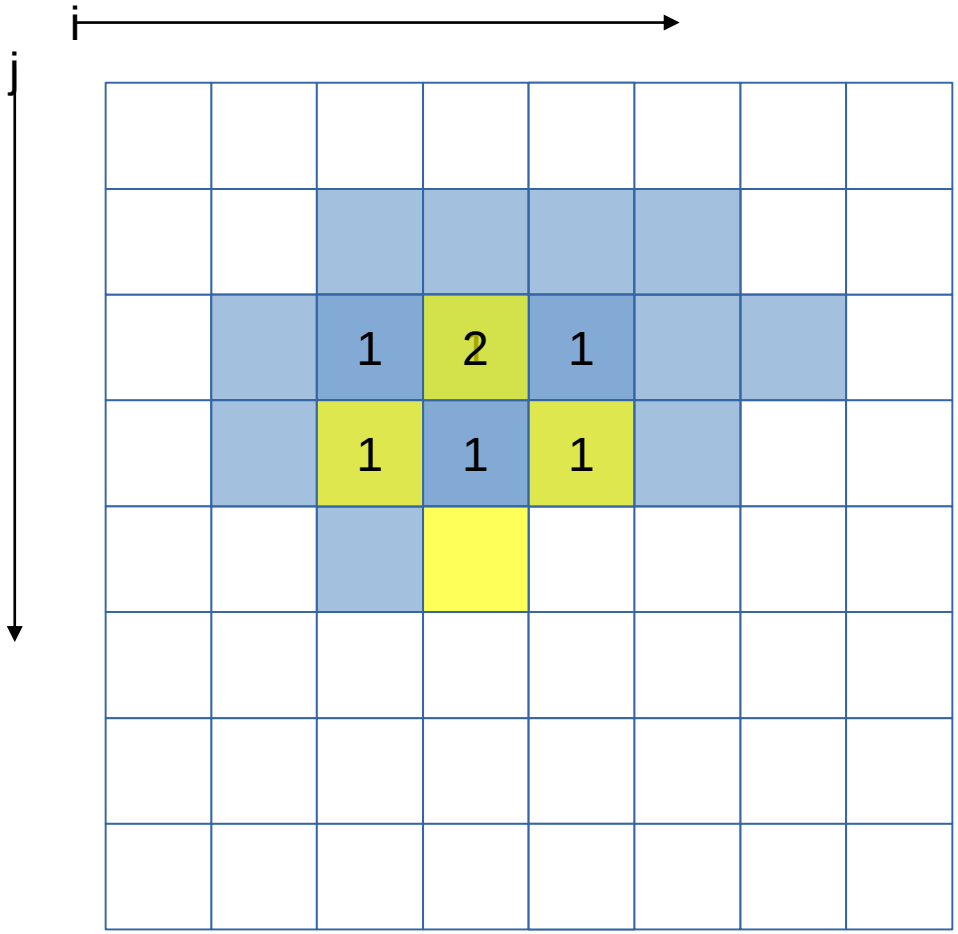


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

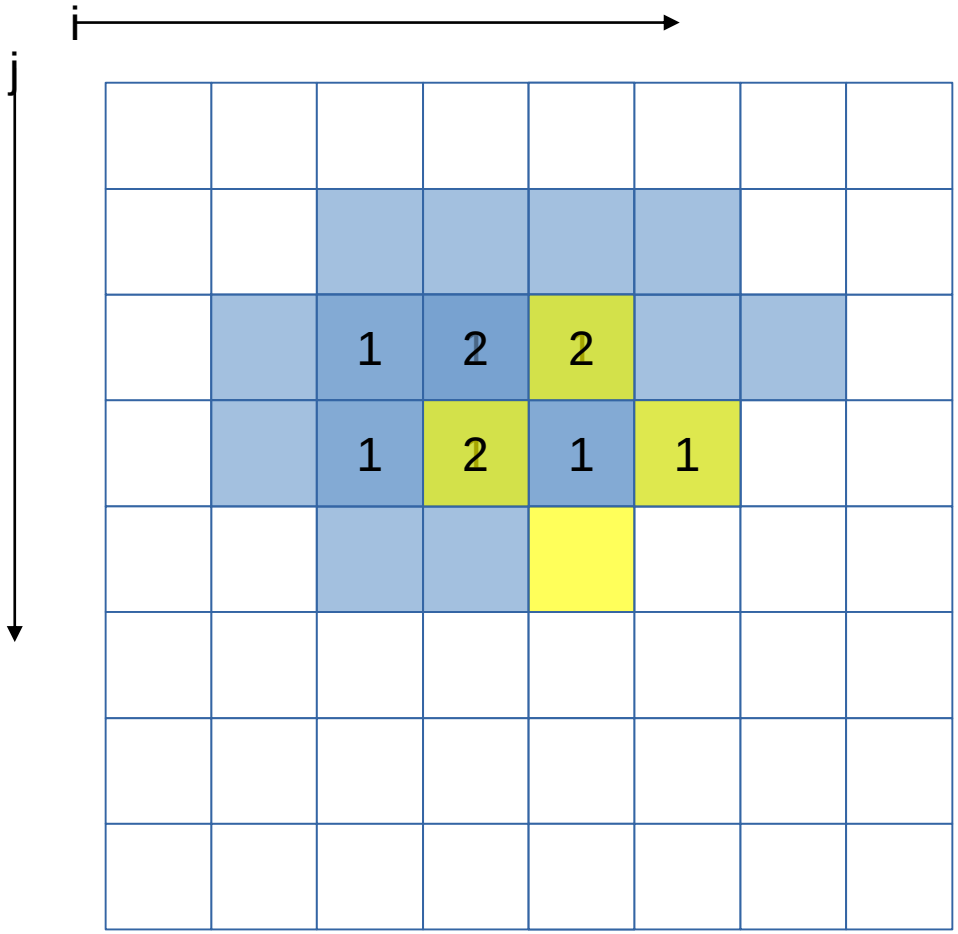


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

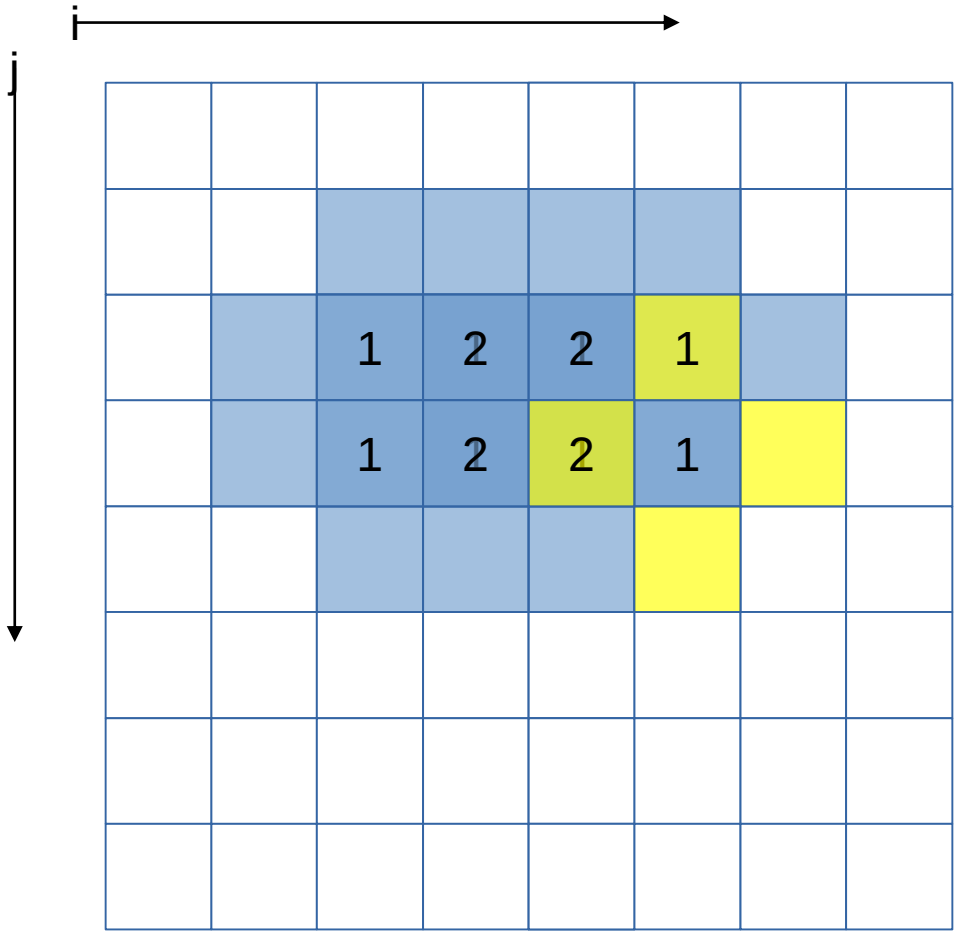


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

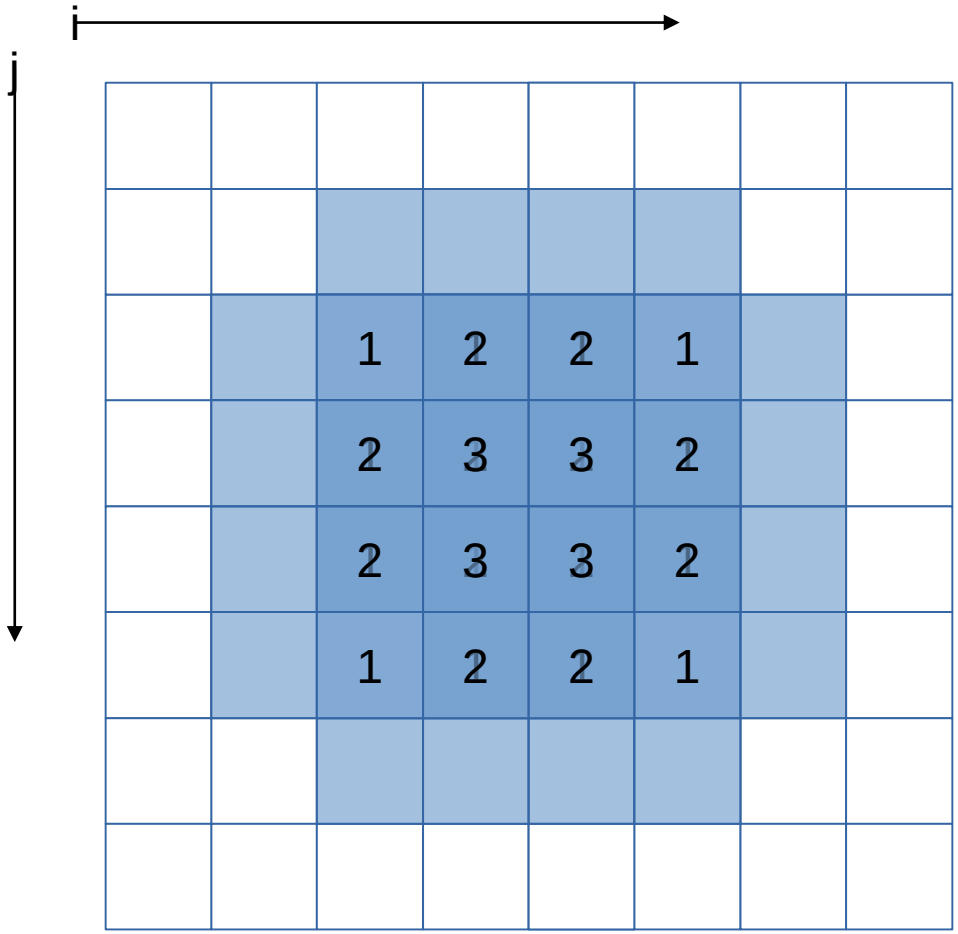


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

Running and analyzing kernel1

```
> aprun -n 1 ./hydro_k1 300 50
Cycles per element for solvers: 911.19
```

- Profile with MAQAO

```
> maqao oneview create-report=one xp=ov_k1 c=ov_k1.lua
```

- Display results

```
> maqao oneview create-report=one xp=ov_k1 \
output-format=text --text-global | less
```

```
+-------------------------------------------------------------+
+                      1.2  -  Global Metrics                  +
+-------------------------------------------------------------+

  Total Time:                          1.54 s
  Time spent in loops:                 99.43 %
  Compilation Options:                 OK
  Flow Complexity:                     1.09
  Array Access Efficiency:             50.20 %
```

Loop id	Source Location	Source Function	Coverage (%)
147	hydro_k1 - kernel.c:15-176	linearSolver1	61.33
56	hydro_k1 - kernel.c:15-176	c_densitySolver	15.48
47	hydro_k1 - kernel.c:15-292	c_densitySolver	4.19
75	hydro_k1 - kernel.c:15-292	c_velocitySolver	3.13
77	hydro_k1 - kernel.c:15-292	c_velocitySolver	2.75
131	hydro_k1 - kernel.c:15-342	c_velocitySolver	2.22
79	hydro_k1 - kernel.c:368-371	c_velocitySolver	1.69
93	hydro_k1 - kernel.c:368-371	c_velocitySolver	1.64
73	hydro_k1 - kernel.c:380-383	c_velocitySolver	1.4
91	hydro_k1 - kernel.c:380-383	c_velocitySolver	1.35

Remark: less calls are unrolled because linearSolver is much bigger

Loop Id: 158 | Module: hydro_k1 | Source: kernel.c:15-176 | Coverage: 58.44%

Source Code

```
143:     (a * ( x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] + \
144:       x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) + \
145:      x0[build_index(i, j, grid_size)]) * inv_c;
146:
147: void linearSolver1(int b, float* x, float* x0, float a, float c, float dt, int grid_s
148:   {
149:   int i,j,k;
150:   const float inv_c = 1.0f / c;
151:
152:   for (k = 0; k < 20; k++)
153:     {
154:     for (i = 1; i <= grid_size-3; i+=4)
155:       {
156:       for (j = 1; j <= grid_size-3; j+=4)
157:     {
158:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+0);
159:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+1);
160:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+2);
161:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+3);
162:
163:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+0);
164:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+1);
165:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+2);
166:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+3);
167:
168:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+0);
169:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+1);
170:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+2);
171:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+3);
172:
173:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+0);
174:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+1);
175:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+2);
176:       LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+3);
177:     }
178:       }
```

CQA

The related source loop is not unrolled or unrolled with no peel/tail loop.

gain | potential | hint | expert

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 67.00 to 63.00 cycles (1.06x speedup).

Workaround

- Try to reorganize arrays of structures to structures of arrays
- Consider to permute loops (see vectorization gain report)

Vectorization

Your loop is not vectorized. 16 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 67.00 to 4.19 cycles (16.00x speedup).

Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.

CQA output for kernel1

```
> maqao oneview create-report=one xp=ov_k1 \
output-format=text text-cqa=147 | less
```

Type of elements and instruction set

96 SSE or AVX instructions are processing arithmetic or math operations on single precision FP elements in scalar mode (one at a time).

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

The binary loop is composed of 96 FP arithmetical operations:

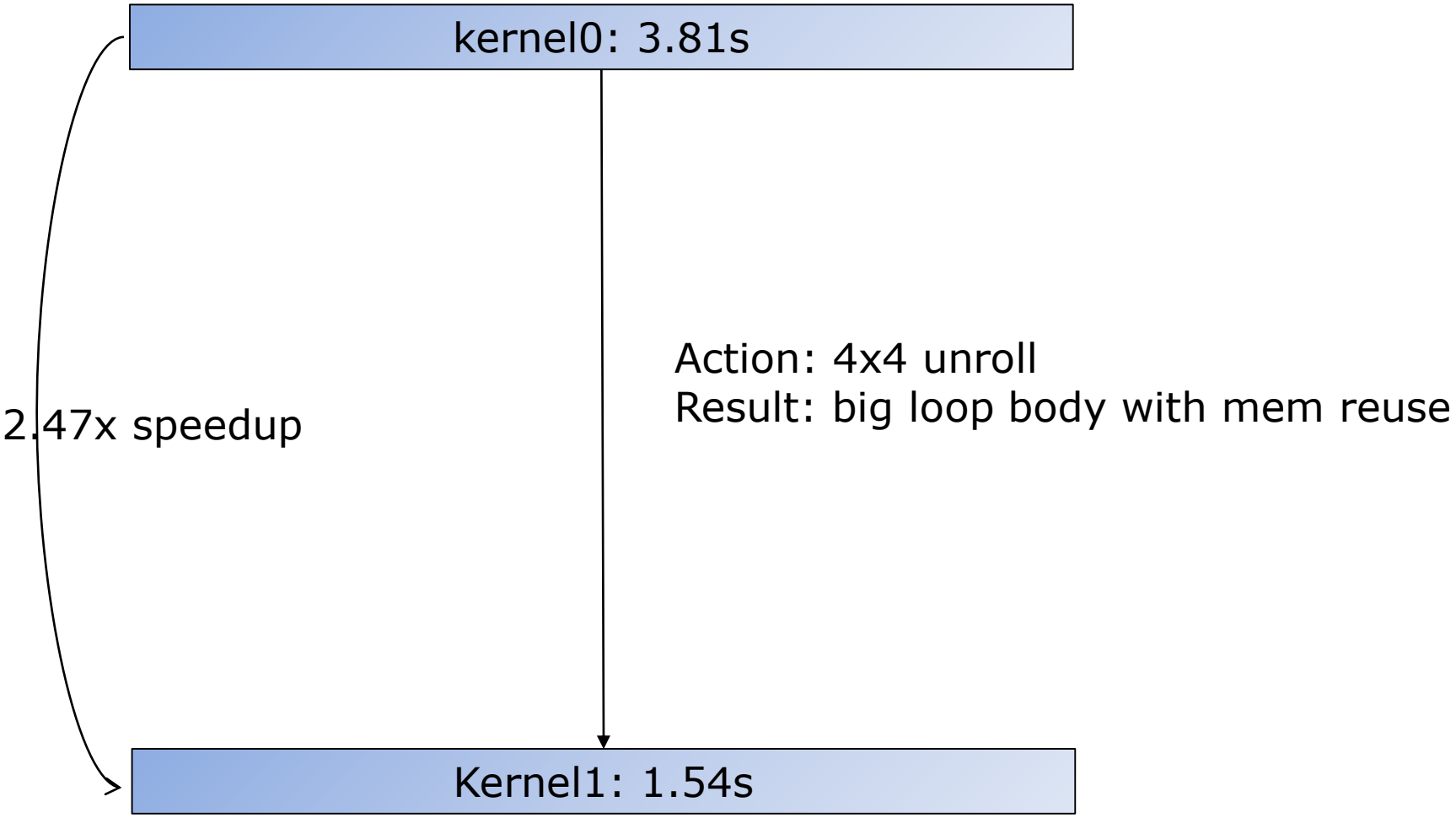
- 64: addition or subtraction
- 32: multiply

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



More sample codes

More codes to study with MAQAO in

```
$WORK/MAQAO_HANDSON/loop_optim_tutorial.tgz
```