MAQAO
Hands-on exercises

Profiling bt-mz
Optimising a code

Setup

Login to the cluster with X11 forwarding

```
> ssh -Y <login>@romeologin1.univ-reims.fr
```

Load MAQAO environment

```
> module load maqao/2.6.0
```

Copy handson material to your WORK directory

```
> cd /scratch_p/$USER
> tar xvf /home/projects/VIHPS/MAQAO/MAQAO_HANDSON_TW29.tgz
```

(If not already done) Load compiler + MPI

```
> module load intel/2017.7 openmpi/2.0.4.1.1_icc_mt
```

Profiling bt-mz with MAQAO

Cédric Valensi

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 /scratch_p/$USER/NPB3.3-MZ-MPI/bin
> cp /scratch_p/$USER/MAQAO_HANDSON/bt/config_bt_oneview_sbatch.lua .
> less config_bt_oneview_sbatch.lua
```

```
binary = "bt-mz_C.8"
…
batch_script = "maqao_bt.slurm"
…
batch_command = "sbatch <batch_script>"
…
number_processes = 8
…
number_nodes = 2
…
mpi_command = "srun"
…
omp_num_threads = 6
```

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 /scratch_p/$USER/NPB3.3-MZ-MPI/bin
> cp /scratch_p/$USER/MAQAO_HANDSON/bt/maqao_bt.slurm .
> less maqao_bt.slurm
```

```
...
#SBATCH -N 2 <number_nodes>
#SBATCH -n 8 <number_processes>
#SBATCH -c 6 <number_threads>
...
export OMP_NUM_THREADS=6<omp_num_threads>
...
srun ./bt-mz_C.8
<mpi_command> <run_command>
...
```

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

Launch ONE View

```
> cd /scratch_p/$USER/MAQAO_HANDSON/NPB3.3.1-MZ-MPI/bin
> maqao oneview --create-report=one \
-config=config_bt_oneview_sbatch.lua -xp=ov_sbatch
```

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>.

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

(OPTIONAL) Review ONE View for interactive mode

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

```
> cp /scratch_p/$USER/MAQAO_HANDSON/bt/config_bt_oneview_interactive.lua .
> less config_bt_oneview_interactive.lua
```

```
binary = "bt-mz.C.8"
...
number_processes = 8
...
number_nodes = 2
...
mpi_command = "srun -n <number_processes> -c <omp_num_threads> \
-N <number_nodes> --time=0:05:00 -p long \
--reservation=arenard_12"
...
omp_num_threads = 6
```

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

Launch ONE View

```
> 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.

```
> firefox <exp-dir>/RESULTS/bt-mz_C.8_one_html/index.html
```

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

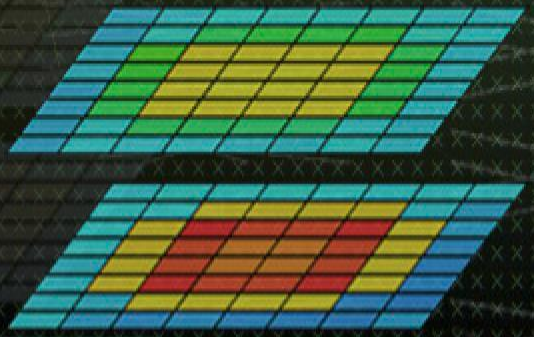
```
> tar -zcf $HOME/ov_html.tgz <exp-dir>/RESULTS/bt-mz_C.8_one_html
```

```
> scp <login>@romeologin1.reims-univ.fr:ov_html.tgz .
> tar xf ov_html.tgz
> firefox <exp-dir>/RESULTS/bt-mz_C.8_one_html/index.html
```

A sample result directory is in **MAQAO_HANDSON/bt/bt-mz_C.8_one_html/**

Results can also be viewed directly on the console:

```
> maqao oneview --create-report=one -xp=<exp-dir> \
--output-format=text | less
```



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

Request interactive session

```
> srun -N 1 --exclusive -t 0:30:00 --reservation=arenard_12 \
-p long --pty bash
```

Load MAQAO environment

```
> module load maqao/2.6.0
```

Load latest GCC compiler

```
> module load gcc
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

```
> cd /sbatch_p/$USER/MAQAO_HANDSON/matmul
> make matmul_orig
```

Analyse matrix multiply with ONE View

```
> maqao oneview --create-report=one \
--binary=./matmul_orig --run-command="<binary> 150 10000" \
-xp=ov_orig
```

OR, using a configuration script:

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

Viewing results (HTML)

```
> tar -zcf $HOME/ov_orig.tgz ov_orig/RESULTS/matmul_orig_one_html
```

```
> scp <login>@romeologin1.univ-reims.fr:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		33.42
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		66.67
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	2.35
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	14.90
	Nb Loops to get 80%	1

Viewing results (text)

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

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

  Total Time:                     33.42 s
  Compilation Options:            binary:  -funroll-loops is missing.
  Flow Complexity:                1.00
  Array Access Efficiency:        66.67 %
  If Clean:
      Potential Speedup:          1.00
      Nb Loops to get 80%:        1
  If FP Vectorized:
      Potential Speedup:          2.35
      Nb Loops to get 80%:        1
  If Fully Vectorized:
      Potential Speedup:          14.90
      Nb Loops to get 80%:        1
```

Viewing results (text)

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

  If Clean:
      Number of loops    | 1       |
      Cumulated Speedup | 1.0000 |
  Top 5 loops:
    matmul_orig - 1:     1

  If FP Vectorized:
      Number of loops    | 1       |
      Cumulated Speedup | 2.3600 |
  Top 5 loops:
    matmul_orig - 1:     2.36

  If Fully Vectorized:
      Number of loops    | 1       |
      Cumulated Speedup | 15.267 |
  Top 5 loops:
    matmul_orig - 1:     15.2672
```

Loop ID

Viewing CQA output (text)

```
> maqao oneview --create-report=one -xp=ov_orig \
  --format=text --text-cqa=1
```

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 0.38 cycles
(8.00x speedup).
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)
```

CQA output for the baseline kernel

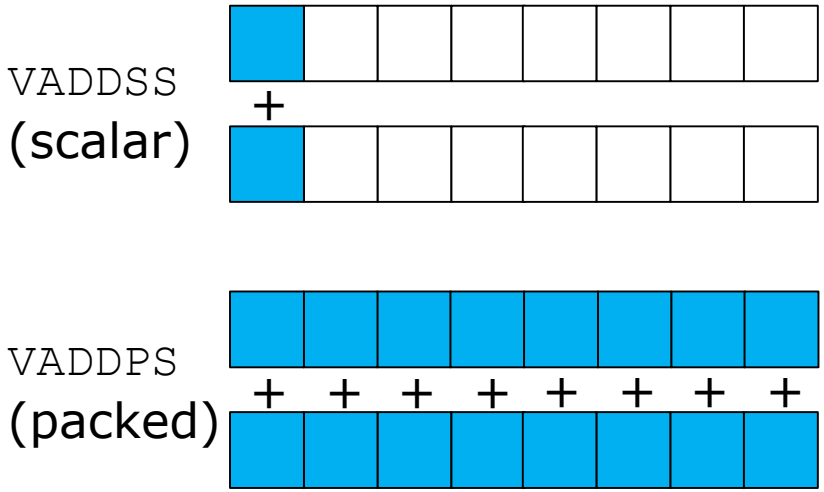
Vectorization

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 0.38 cycles (8.00x speedup).

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)

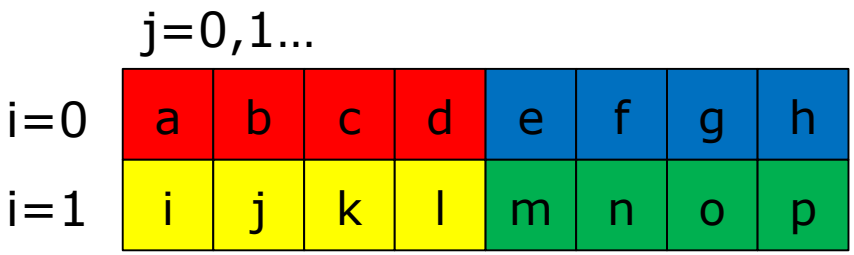
Vectorization (summing elements):



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

Impact of loop permutation on data access

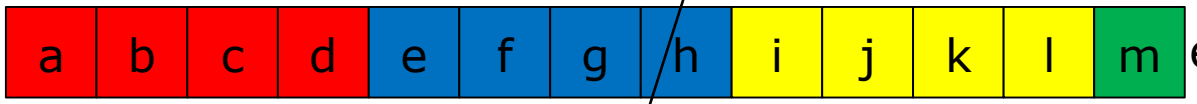
Logical mapping



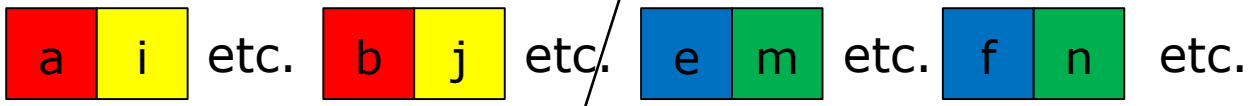
Efficient vectorization + prefetching

Physical mapping

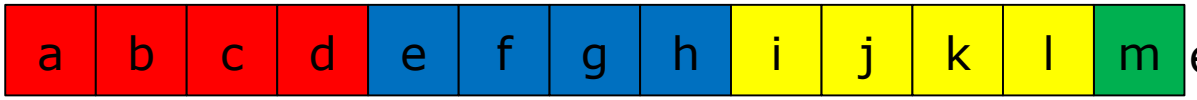
(C stor. order: row-major)




```
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 /scratch_p/$USER/MAQAO_HANDSON/matmul
> make matmul_perm
```

Analyse matrix multiply with ONE View

```
> maqao oneview --create-report=one \
--binary=./matmul_perm --run-command="<binary> 150 10000" \
-xp=ov_perm
```

OR using configuration script:

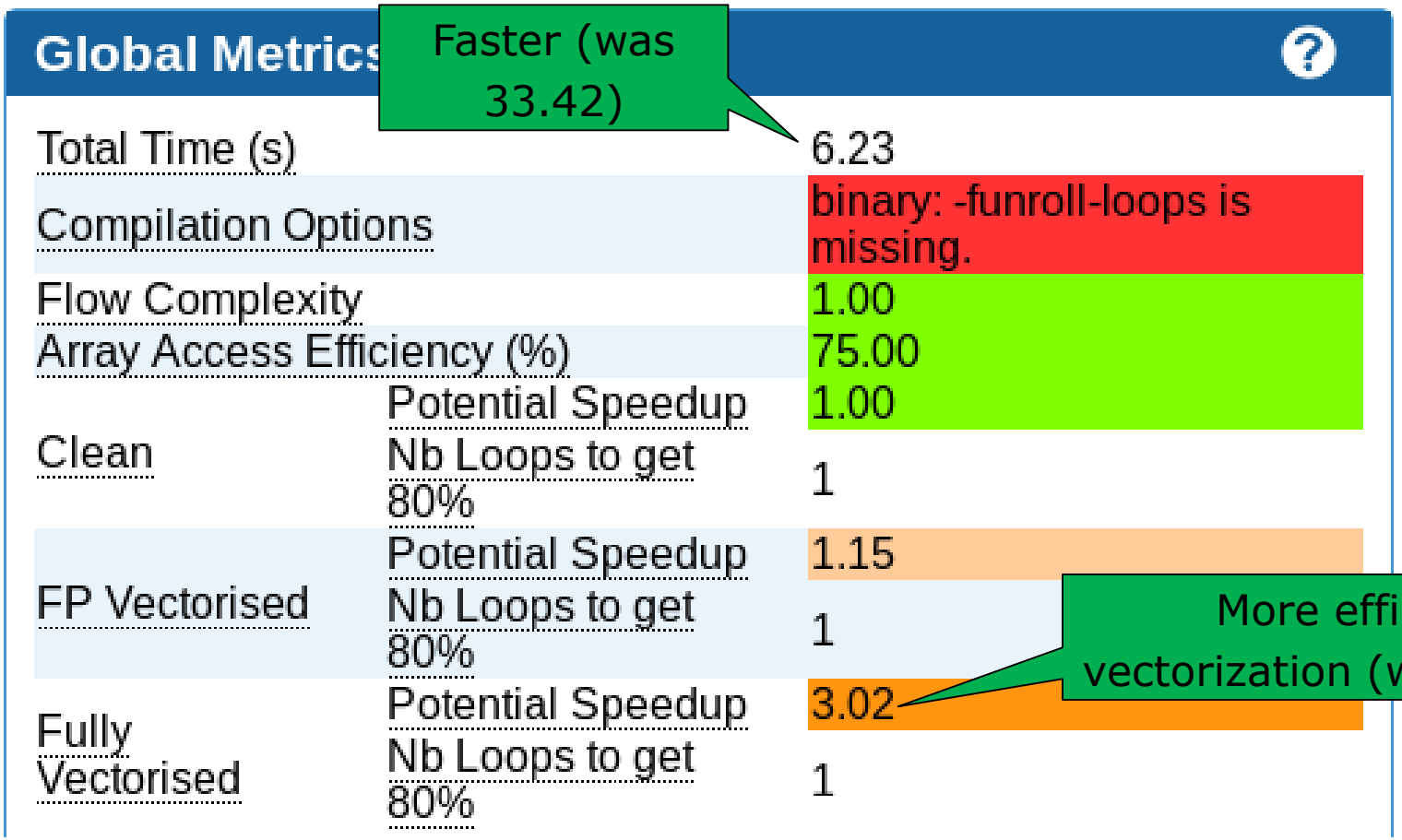
```
> maqao oneview --create-report=one c=ov_perm.lua xp=ov_perm
```

Viewing new results

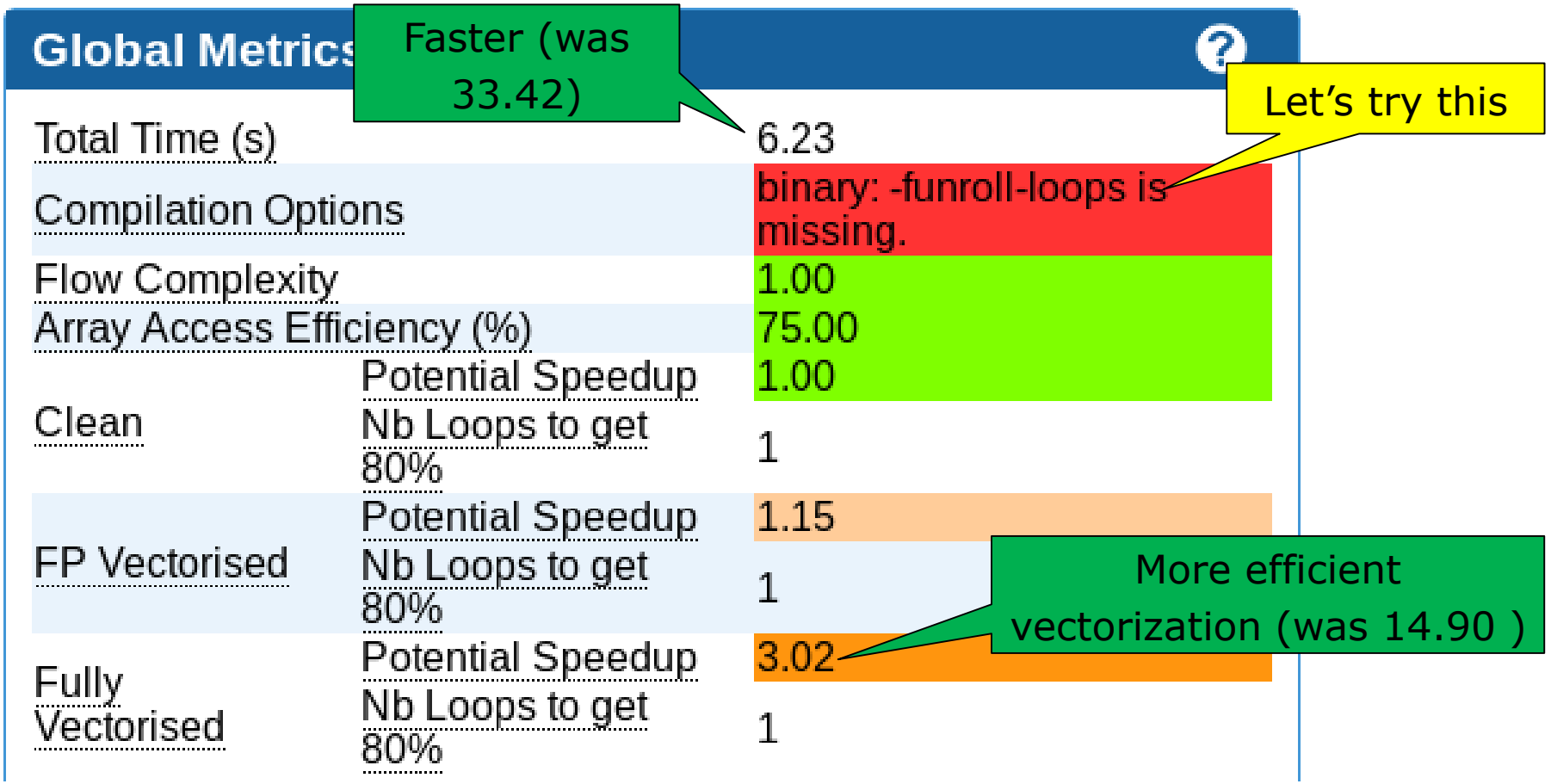
```
> maqao oneview --create-report=one -xp=ov_perm \
  --output-format=text --text-global | less
```

(Or download the **ov_perm/RESULTS/matmul_perm_one_html** folder locally and open **ov_perm/RESULTS/matmul_perm_one_html/index.html**)

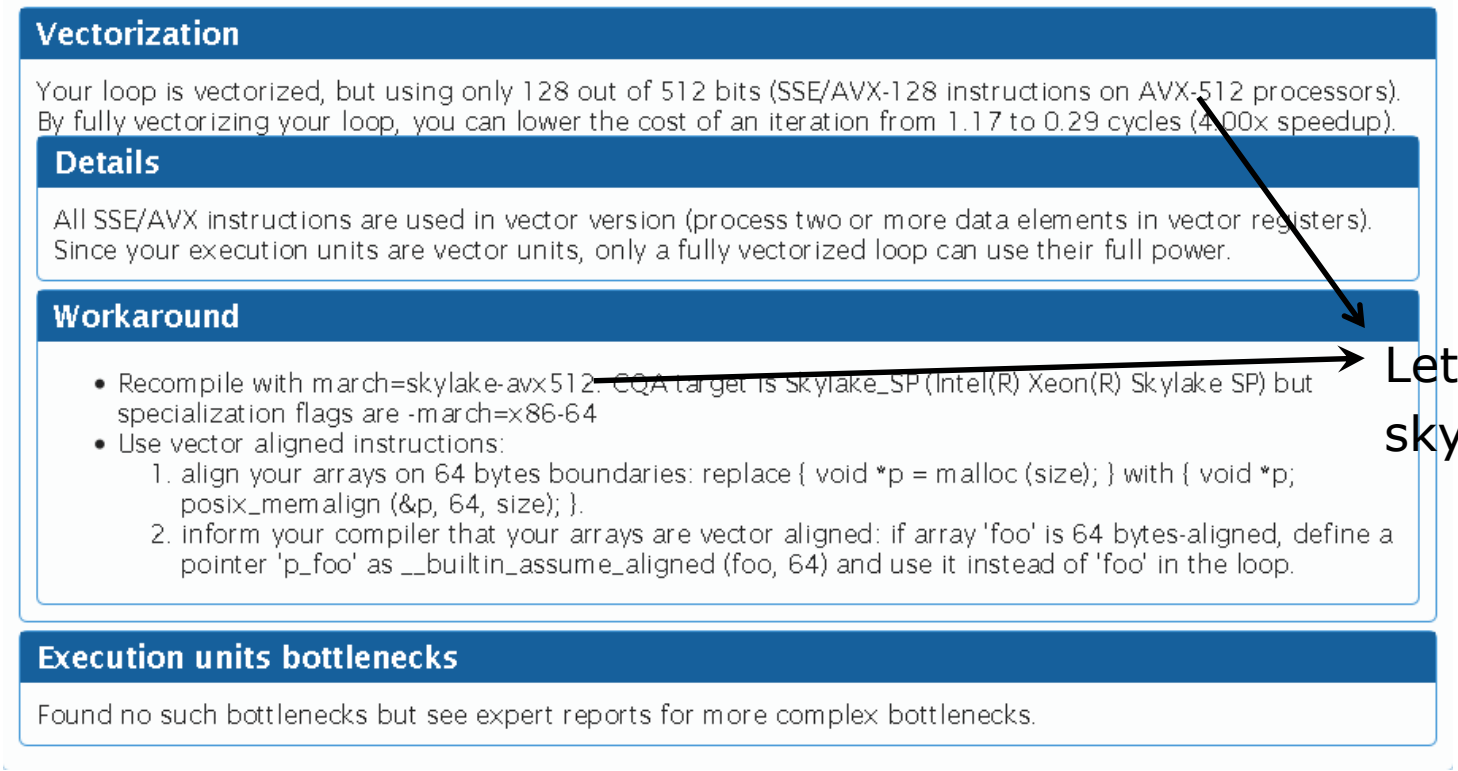
Loop permutation results



Loop permutation results



CQA output after loop permutation



Vectorization

Your loop is vectorized, but using only 128 out of 512 bits (SSE/AVX-128 instructions on AVX-512 processors).
By fully vectorizing your loop, you can lower the cost of an iteration from 1.17 to 0.29 cycles (4.00x 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=skylake-avx512. CQA target is Skylake_SP (Intel(R) Xeon(R) Skylake SP) but specialization flags are -march=x86-64
- Use vector aligned instructions:
  1. align your arrays on 64 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 64, size); }.
  2. inform your compiler that your arrays are vector aligned: if array 'foo' is 64 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 64) and use it instead of 'foo' in the loop.

Execution units bottlenecks

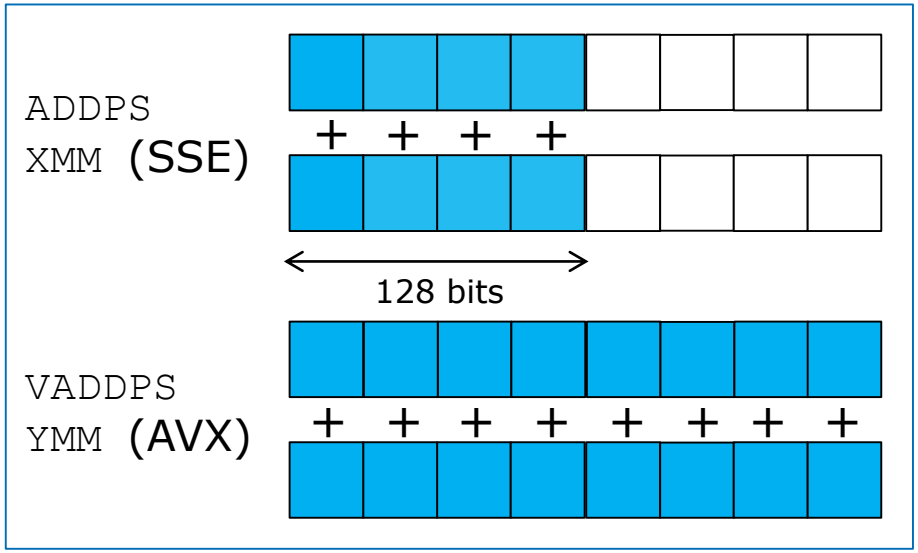
Found no such bottlenecks but see expert reports for more complex bottlenecks.

Let's add –march= skylake-avx512

```
> maqao oneview --create-report=one -xp=ov_perm \
  --format=text --text-cqa=4 | less
```

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 /scratch_p/$USER/MAQAO_HANDSON/matmul
> make matmul_perm_opt
```

Analyse matrix multiply with ONE View

```
> maqao oneview --create-report=one \
--binary=./matmul_perm_opt --run-command="<binary> 150 10000" \
-xp=ov_perm_opt
```

OR using configuration script:

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

Viewing new results:

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

(or download the **`ov_perm/RESULTS/matmul_perm_opt_one_html`** folder locally and open **`index.html`** in your browser)

Loop permutation + (-march=skylake-avx512 -funroll-loops)

Global Metrics

Total Time (s)		4.34
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		50.00
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.40
	Nb Loops to get 80%	1

Faster (was 6.23)

Now OK (-funroll-loops prev. missing)

Better vectorization (was 1.88)

CQA output with (-march=skylake-avx512 -funroll-loops)

Workaround

Use vector aligned instructions:

1. align your arrays on 64 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 64, size); }.
2. inform your compiler that your arrays are vector aligned: if array 'foo' is 64 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 64) and use it instead of 'foo' in the loop.

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

```
> maqao oneview --create-report=one -xp=ov_perm_opt \
  --output-format=text --text-cqa=4 | less
```

Using aligned arrays in matrix multiply

Compile aligned array version of matrix multiply

```
> cd /scratch_p/$USER/MAQAO_HANDSON/matmul
> make matmul_align
```

Checking aligned version:

```
> ./matmul_align 150 10000
Cannot call kernel on matrices with size%8 != 0 (data not
aligned on 32B boundaries)
Aborted
```

=> Alignment imposes restrictions on input parameters.

```
> ./matmul_align 152 10000
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.18
```

Analysing matrix multiply with aligned arrays

Analyse matrix multiply with ONE View

```
> maqao oneview --create-report=one \
--binary=./matmul_align --run-command="<binary> 152 10000" \
-xp=ov_align
```

OR using configuration script:

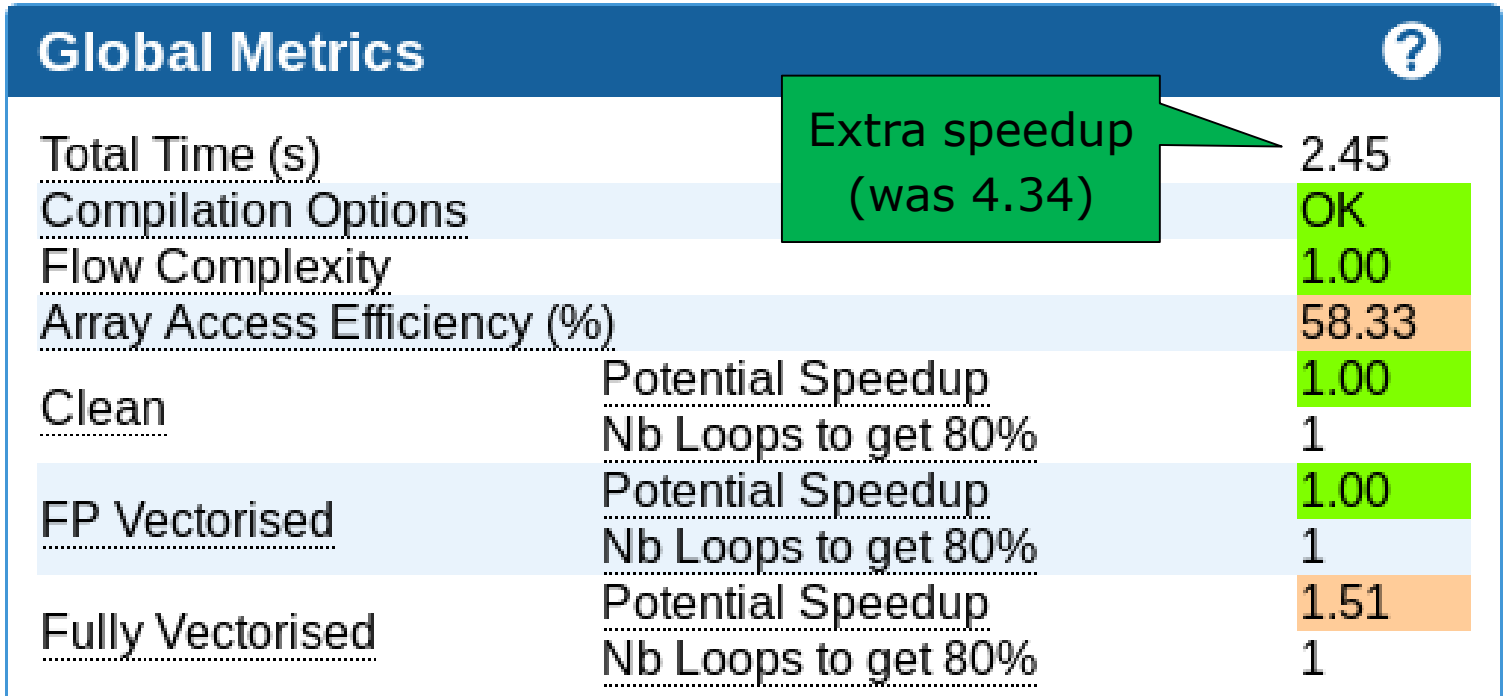
```
> maqao oneview --create-report=one c=ov_align.lua \
 xp=ov_align
```

Viewing new results

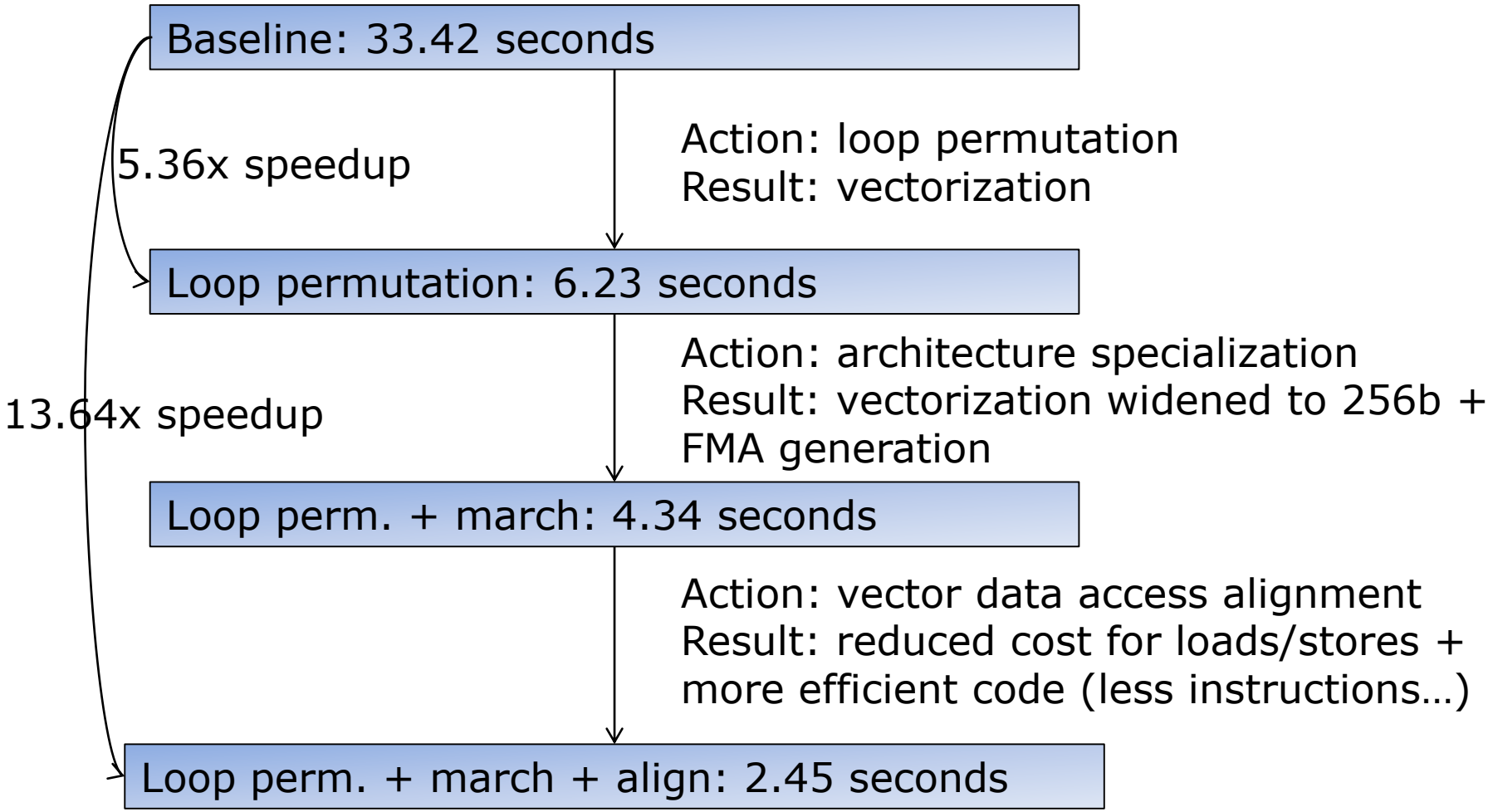
```
> maqao oneview --create-report=one -xp=ov_align \
  --format=text --text-global | less
```

(Or download the **ov_align/RESULTS/matmul_align_one_html** folder locally and open **ov_align/RESULTS/matmul_align_one_html/index.html** in your browser)

Vector-aligning array accesses



Summary of optimizations and gains



Hydro example

Exit interactive session to allow Intel compilation

```
> exit
```

Switch to the hydro handson folder

```
> cd /scratch_p/$USER/MAQAO_HANDSON/hydro
```

Load Intel compiler environment

```
> module load intel
```

Compile

```
> make
```

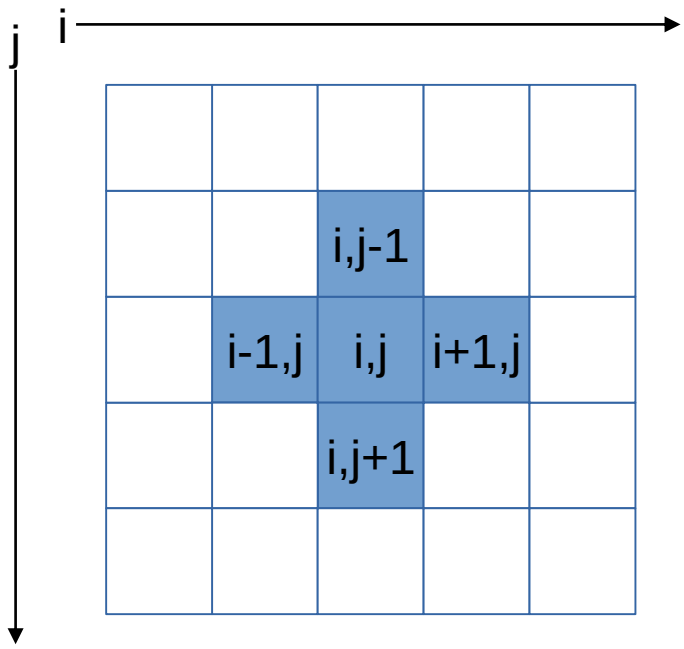
Hydro code

```
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 interactive session

Request interactive session and load MAQAO environment

```
> srun -N 1 --exclusive -t 0:30:00 --reservation=arenard_12 \
-p long --pty bash
> module load maqao/2.6.0
```

Running and analyzing kernel0 (icc -O3 -xHost)

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

> maqao oneview --create-report=one xp=ov_k0 \
--output-format=text --text-global | less
> ...
> Total time: 6.56s
```

Loops Index

☑ Coverage (%) ☐ Time (s) ☐ Vectorization Ratio (%) ☐ Speedup If Clean ☐ Speedup If FP Vectorized ☐ Speedup If Fully Vectorized

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 123	104-110	hydro_k0:kernel.c	project	31.13
Loop 50	104-110	hydro_k0:kernel.c	c_densitySolver	20.75
Loop 81	104-110	hydro_k0:kernel.c	c_velocitySolver	20.75
Loop 88	104-110	hydro_k0:kernel.c	c_velocitySolver	20.6
Loop 69	15-292	hydro_k0:kernel.c	c_velocitySolver	0.92
Loop 42	15-292	hydro_k0:kernel.c	c_densitySolver	0.92
Loop 67	15-292	hydro_k0:kernel.c	c_velocitySolver	0.92
Loop 114	210-342	hydro_k0:kernel.c	c_velocitySolver	0.76
Loop 125	380-383	hydro_k0:kernel.c	project	0.76
Loop 101	210-318	hydro_k0:kernel.c	c_velocitySolver	0.46
Loop 17	59-79	hydro_k0:kernel.c	setBoundry	0.15
Loop 100	239-241	hydro_k0:kernel.c	c_velocitySolver	0.15
Loop 96	44-46	hydro_k0:kernel.c	c_velocitySolver	0
Loop 65	456-459	hydro_k0:kernel.c	c_velocitySolver	0
Loop 79	28-32	hydro_k0:kernel.c	c_velocitySolver	0
Loop 74	28-32	hydro_k0:kernel.c	c_velocitySolver	0
Loop 48	28-32	hydro_k0:kernel.c	c_densitySolver	0
Loop 58	44-46	hydro_k0:kernel.c	c_densitySolver	0
Loop 117	44-46	hydro_k0:kernel.c	c_velocitySolver	0
Loop 93	28-32	hydro_k0:kernel.c	c_velocitySolver	0
Loop 86	28-32	hydro_k0:kernel.c	c_velocitySolver	0
Loop 112	44-46	hydro_k0:kernel.c	c_velocitySolver	0
Loop 59	44-46	hydro_k0:kernel.c	c_densitySolver	0
Loop 109	44-46	hydro_k0:kernel.c	c_velocitySolver	0
Loop 15	59-79	hydro_k0:kernel.c	setBoundry	0
Loop 120	44-46	hydro_k0:kernel.c	c_velocitySolver	0
Loop 131	59-74	hydro_k0:kernel.c	setBoundry	0

Source **Assembly**

```
/home/emoseret/MAQAO_HANDSON/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 **Advanced**

← Path 1 / 1 OK →

Coverage	31.13 %
Function	project
Source file and lines	kernel.c:104-110
Module	hydro_k0

The loop is defined in /home/emoseret/MAQAO_HANDSON/hydro/kernel.c:104-110.

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

gain | potential | hint | expert

Vectorization

Your loop is not vectorized. Only 6% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 4.00 to 0.25 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.
- 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 =

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

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

gain | potential | **hint** | expert

Type of elements and instruction set

5 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 5 FP arithmetical operations:

- 4: addition or subtraction
- 1: multiply

The binary loop is loading 20 bytes (5 single precision FP elements). The binary loop is storing 4 bytes (1 single precision FP elements).

Arithmetic intensity

Arithmetic intensity is 0.21 FP operations per loaded or stored byte.

Unroll opportunity

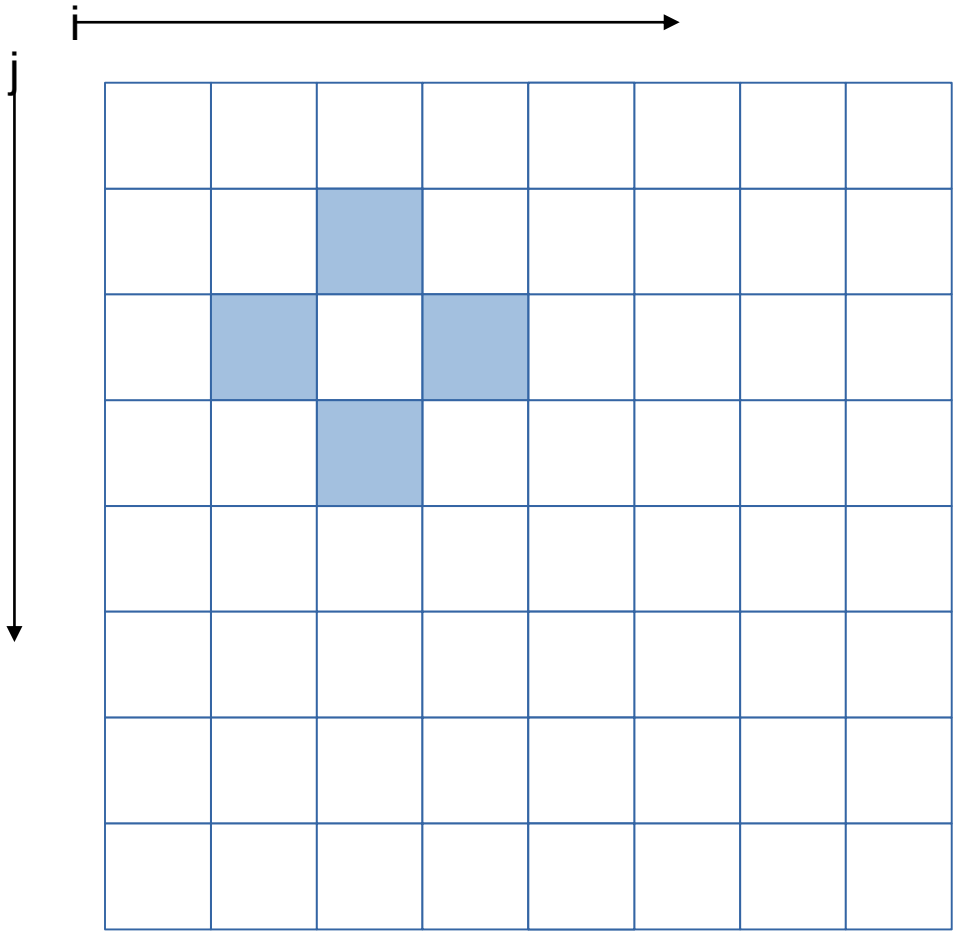
Loop is potentially data access bound.

Workaround

Unroll your loop if trip count is significantly higher than target unroll factor and if some data references are common to consecutive iterations. This can be done manually. Or by combining O2/O3 with the UNROLL (resp. UNROLL_AND_JAM) directive on top of the inner (resp. surrounding) loop. You can enforce an unroll factor: e.g. UNROLL(4).

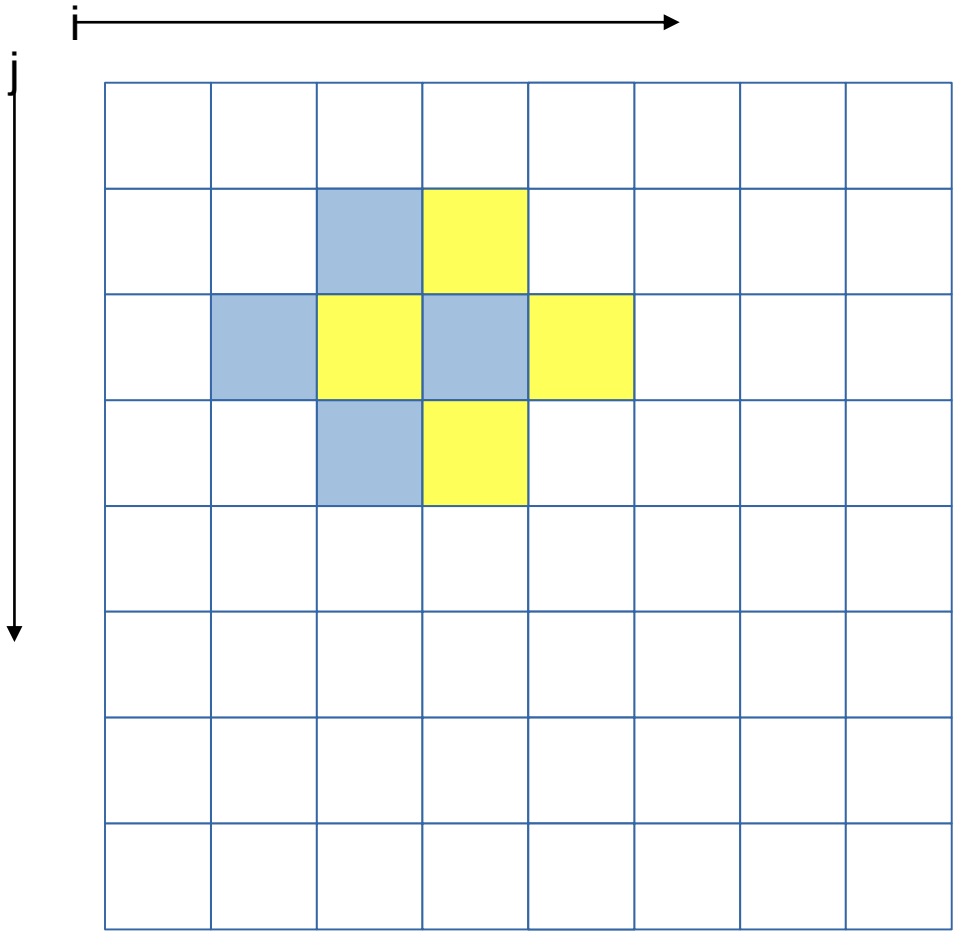
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



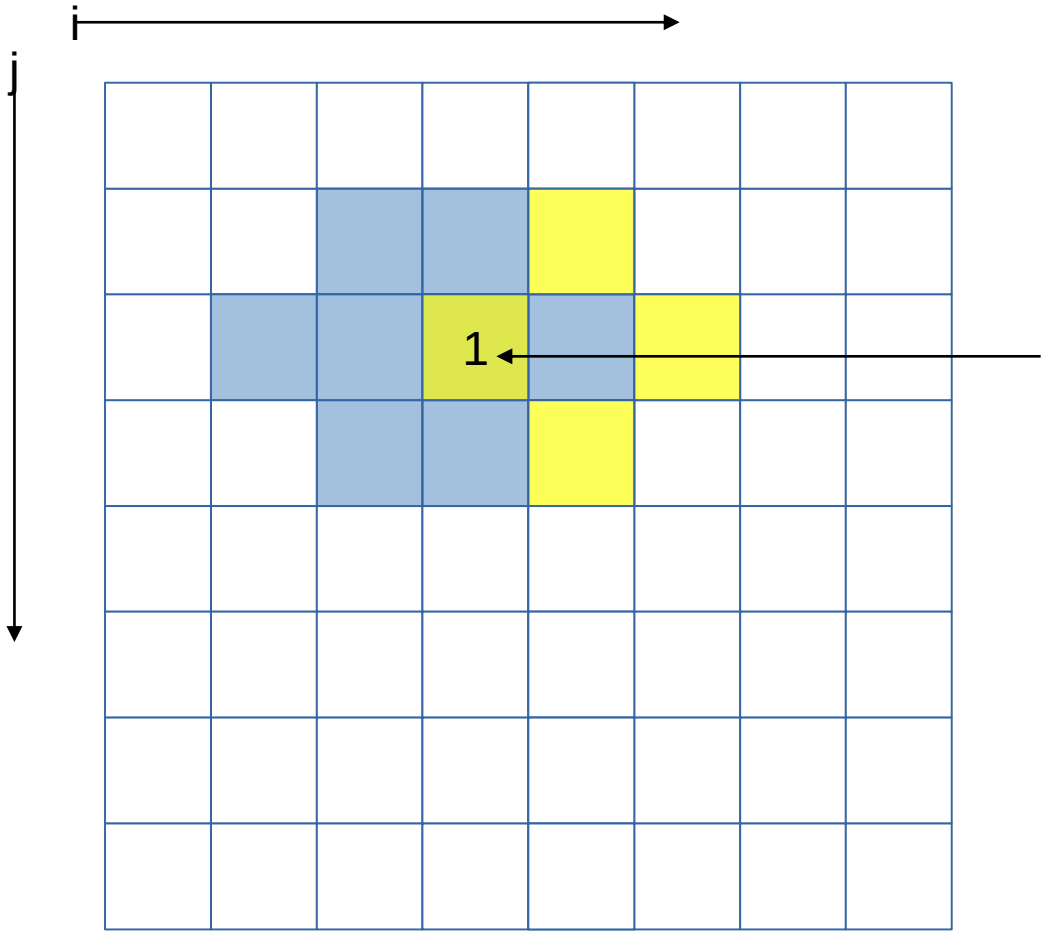

```
LINEAR_SOLVER(i+0,j+0)
```

Memory references reuse : 4x4 unroll footprint on loads




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

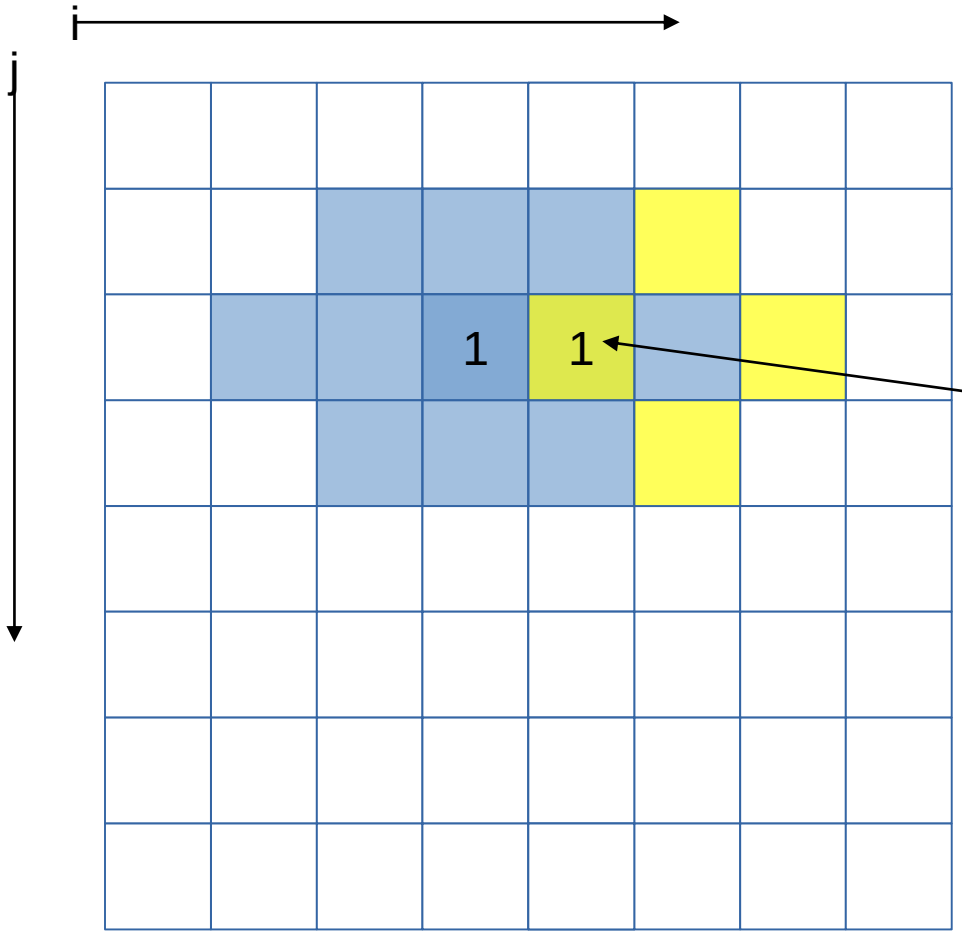
Memory references reuse : 4x4 unroll footprint on loads




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

1 reuse

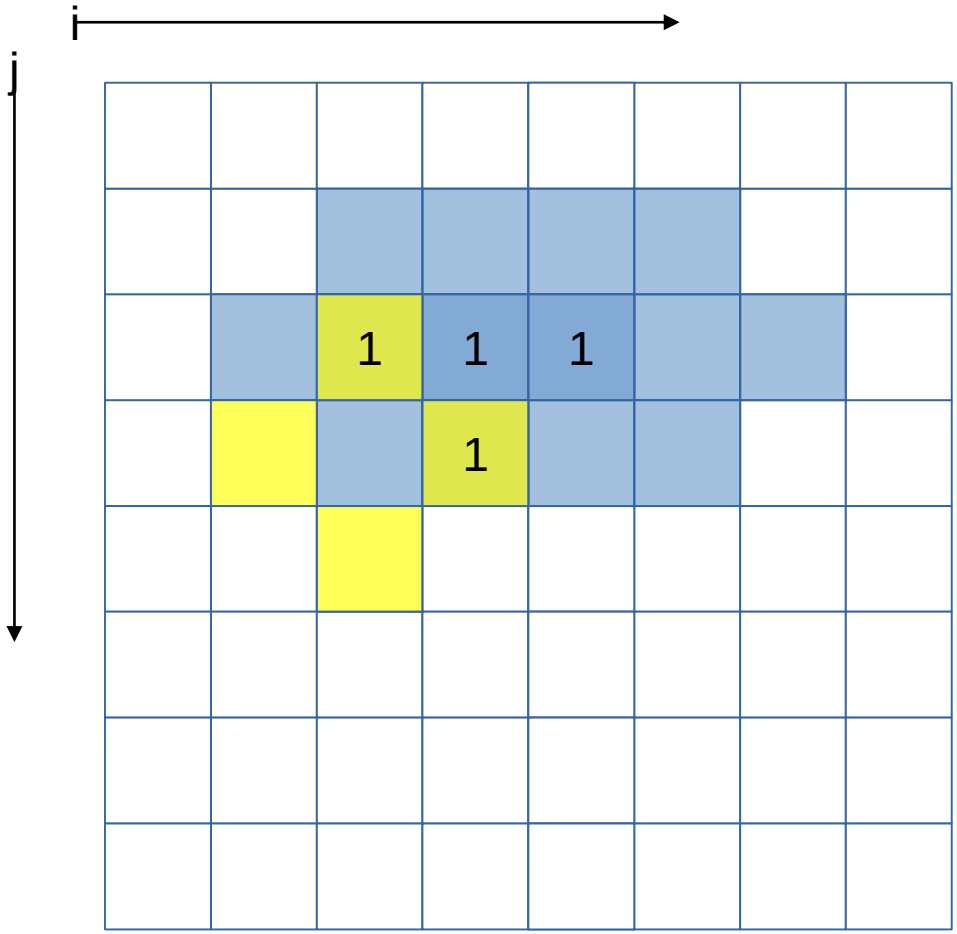
Memory references reuse : 4x4 unroll footprint on loads




```
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

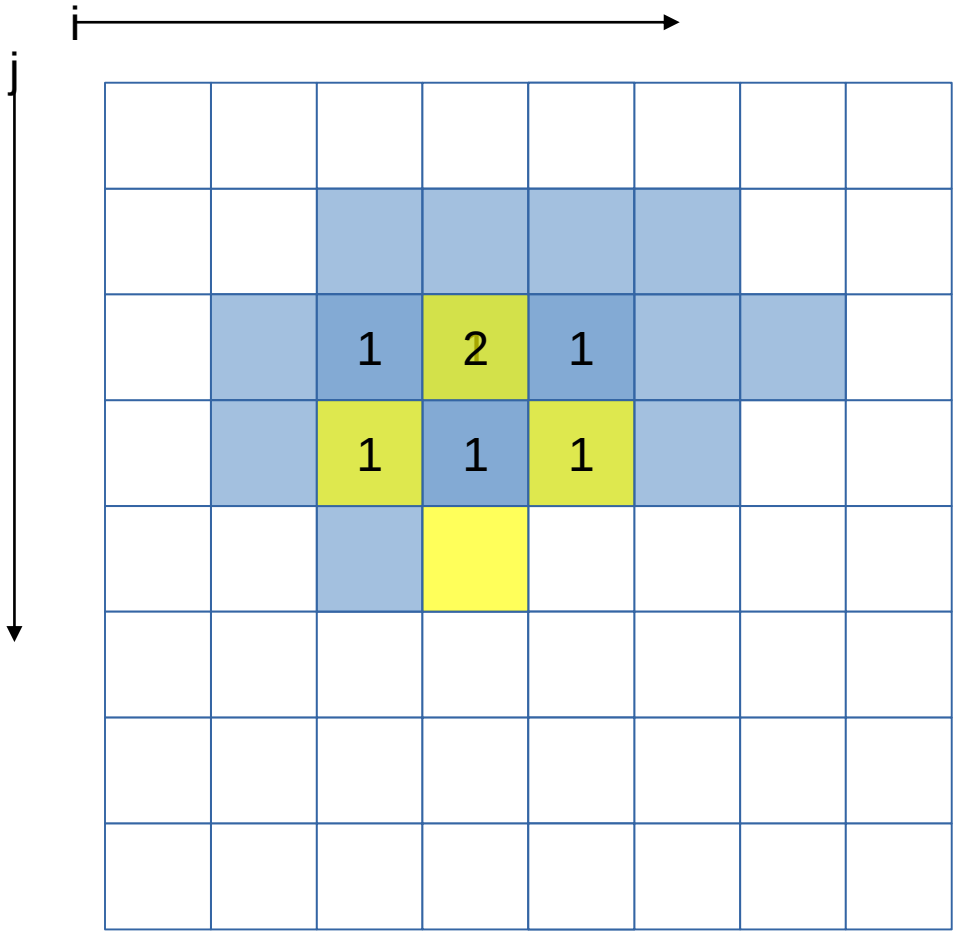



```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

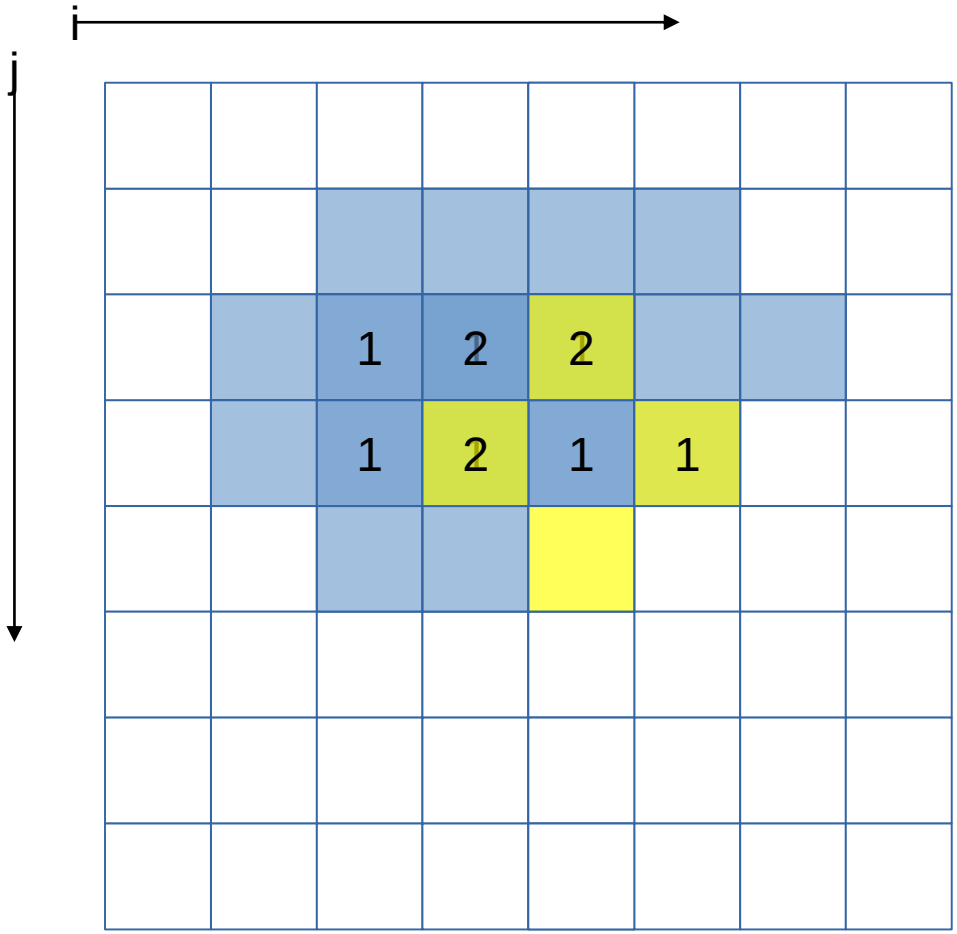



```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

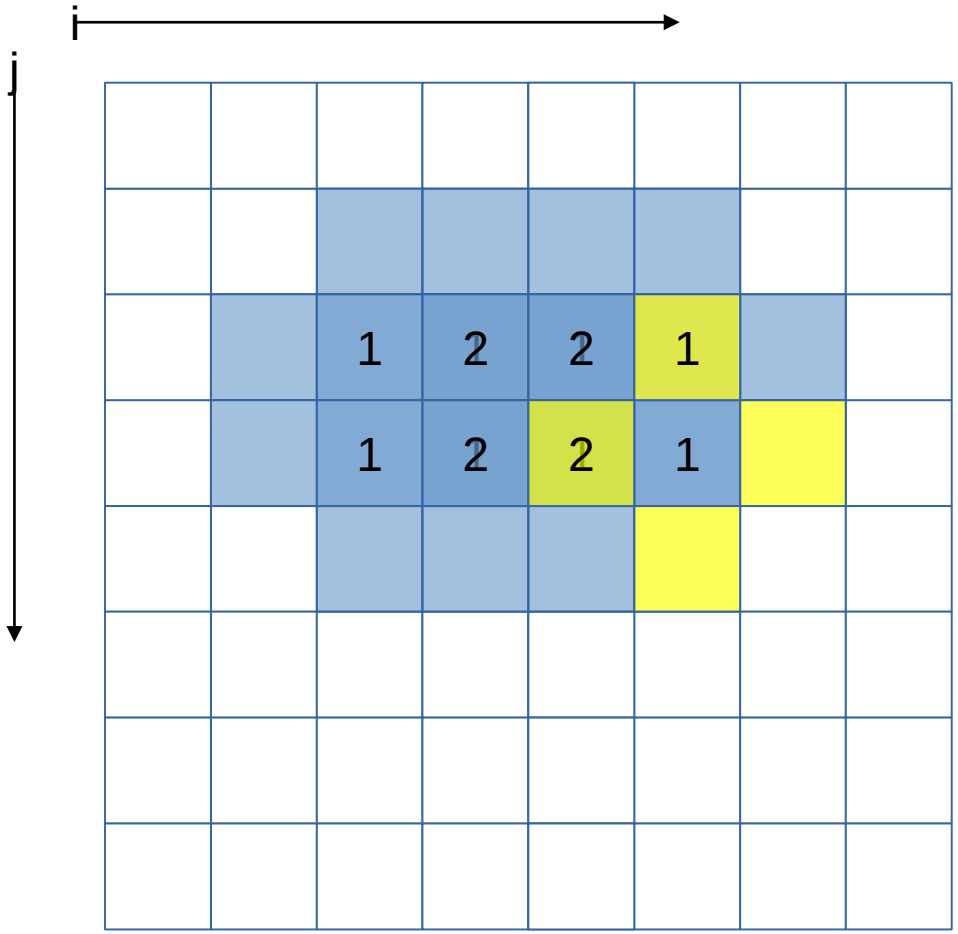



```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

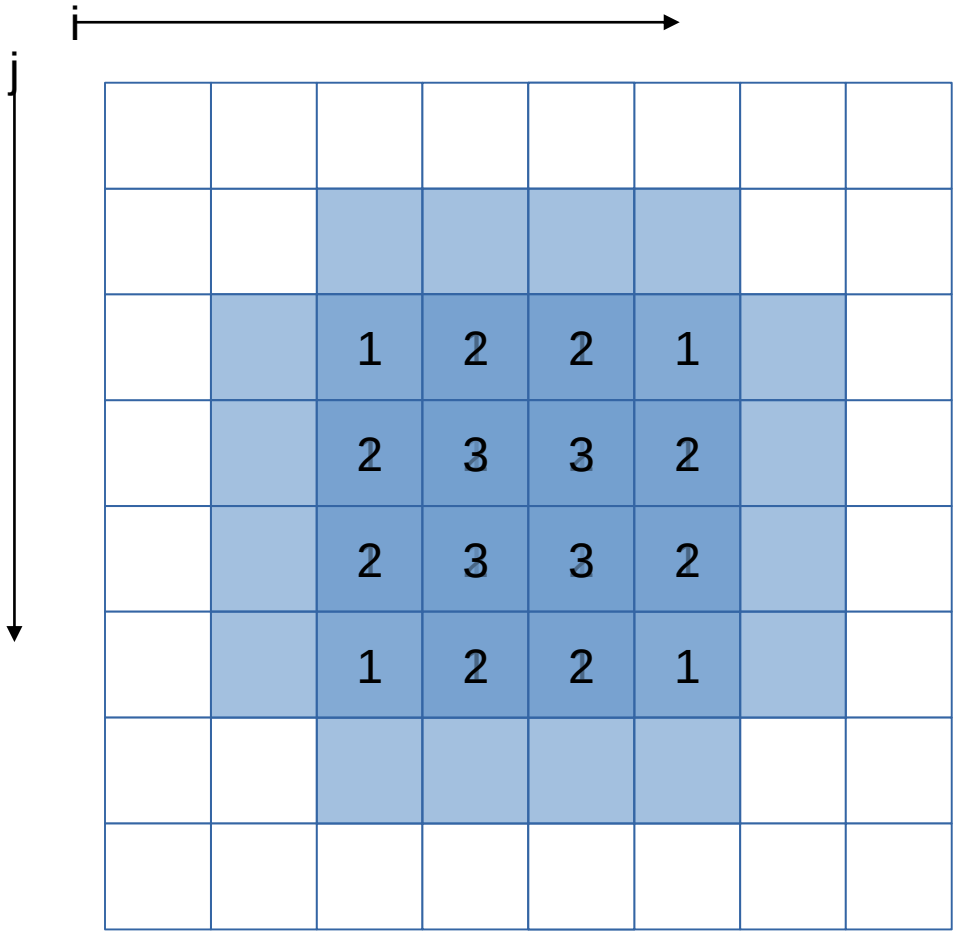



```
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

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

> maqao oneview --create-report=one xp=ov_k1 \
--output-format=text --text-global | less
> ...
> Total time: 1.9s
```

Loops Index

☑ Coverage (%) ☐ Time (s) ☐ Vectorization Ratio (%) ☐ Speedup If Clean ☐ Speedup If FP Vectorized ☐ Speedup If Fully Vectorized

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 129	15-176	hydro_k1:kernel.c	linearSolver1	62.13
Loop 51	15-176	hydro_k1:kernel.c	c_densitySolver	15.27
Loop 43	15-292	hydro_k1:kernel.c	c_densitySolver	3.16
Loop 70	15-292	hydro_k1:kernel.c	c_velocitySolver	3.16
Loop 68	15-292	hydro_k1:kernel.c	c_velocitySolver	3.16
Loop 117	210-342	hydro_k1:kernel.c	c_velocitySolver	2.63
Loop 104	210-318	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 66	380-383	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 72	368-371	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 86	368-371	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 84	380-383	hydro_k1:kernel.c	c_velocitySolver	1.05
Loop 103	239-241	hydro_k1:kernel.c	c_velocitySolver	0.53
Loop 123	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 64	456-459	hydro_k1:kernel.c	c_velocitySolver	0
Loop 77	28-32	hydro_k1:kernel.c	c_velocitySolver	0
Loop 109	226-230	hydro_k1:kernel.c	c_velocitySolver	0
Loop 120	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 110	226-230	hydro_k1:kernel.c	c_velocitySolver	0
Loop 115	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 17	59-79	hydro_k1:kernel.c	setBoundry	0
Loop 91	28-32	hydro_k1:kernel.c	c_velocitySolver	0
Loop 127	59-79	hydro_k1:kernel.c	setBoundry	0
Loop 112	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 82	28-32	hydro_k1:kernel.c	c_velocitySolver	0
Loop 99	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 49	28-32	hydro_k1:kernel.c	c_densitySolver	0
Loop 59	44-46	hydro_k1:kernel.c	c_densitySolver	0
Loop 96	28-32	hydro_k1:kernel.c	c_velocitySolver	0

```
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:     }
```

> CQA Advanced

← Path 1 / 1 OK →

Coverage	62.13 %
Function	linearSolver1
Source file and lines	kernel.c:15-176
Module	hydro_k1

The loop is defined in /home/emoseret/MAQAO_HANDSON/hydro/kernel.c:15-176.

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

gain | potential | hint | expert

Vectorization

Your loop is not vectorized. Only 6% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 41.50 to 2.59 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.
- 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 =

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

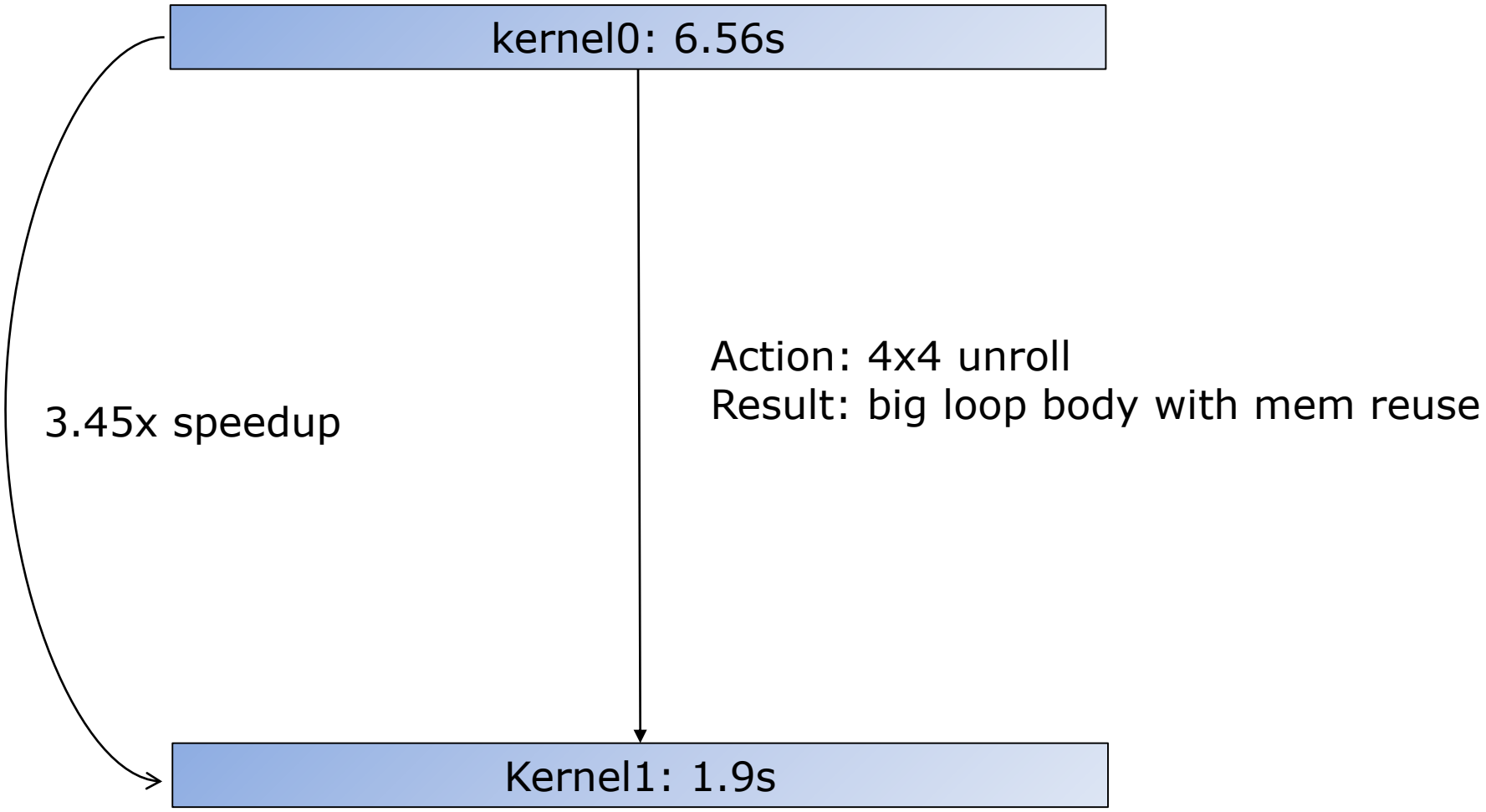
The binary loop is loading 272 bytes (68 single precision FP elements). The binary loop is storing 64 bytes (16 single precision FP elements).

4x4 Unrolling were applied

Expected 48... But still better than 80

```
> maqao oneview --create-report=one xp=ov_k1 \
--output-format=text --text-cqa=127 | less
```

Summary of optimizations and gains



More sample codes

More codes to study with MAQAO in

```
/home/projects/VIHPS/MAQAO/loop_optim_tutorial.tgz
```