MAQAO
Hands-on exercises

Profiling bt-mz
Scalability profiling of lulesh
Optimising a code

Setup

Login to the cluster with X11 forwarding

```
> ssh -Y <username>@stampede2.tacc.utexas.edu
```

Copy handson material to your HOME directory

```
> cd $HOME
> tar xvf ~tg856579/tutorial/MAQAO_HANDSON_TW31.tgz
```

Load MAQAO environment

```
> source ~tg856579/tutorial/env_maqao.sh
```

(If not already done) Load compiler + MPI

```
> module load intel/18.0.2 impi/18.0.2
```

Optional: mount working directory remotely on local machine (if using Linux)

```
> mkdir scratch_remote
> sshfs <login>@stampede2.tacc.utexas.edu:/scratch/<group>/<login> \
scratch_remote
```

Compiling Lulesh

Copy Lulesh sources to your working directory

```
> cd $SCRATCH
> tar -xvf ~tg856579/tutorial/lulesh2.0.3.tgz
```

Compile Lulesh

```
> cd $SCRATCH/lulesh
> module load intel/18.0.2 impi/18.0.2
> make
```

(Optional) To execute a sample run of Lulesh:

```
> cd $SCRATCH/lulesh
> less job_lulesh.sbatch
> sbatch job_lulesh.sbatch
```

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/NPB3.3-MZ-MPI/bin
> cp $HOME/MAQAO_HANDSON/bt/config_maqao_bt.lua .
> less config_maqao_bt.lua
```

```
binary = "bt-mz_C.32"
...
batch_script = "bt_maqao.sbatch"
...
batch_command = "sbatch <batch_script>"
...
number_processes = 32
...
number_nodes = 2
...
mpi_command = "ibrun"
...
omp_num_threads = 4
...
```

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/NPB3.3-MZ-MPI/bin
> cp $HOME/MAQAO_HANDSON/bt/bt_maqao.sbatch .
> less bt_maqao.sbatch
```

```
...
#SBATCH -N ~~2~~<number_nodes>
#SBATCH -n ~~32~~<number_processes>
...
export OMP_NUM_THREADS=~~4~~<omp_num_threads>
...
~~ibrun ./bt-mz_C.8~~
<mpi_command> <run_command>
...
```

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

Launch ONE View

```
> cd $SCRATCH/NPB3.3-MZ-MPI/bin
> maqao oneview --create-report=one \
--config=config_maqao_bt.lua -xp=maqao_bt
```

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.

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 maqao_bt/RESULTS/bt-mz_C.32_one_html/index.html
```

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

```
> tar -zcf $HOME/bt_html.tgz maqao_bt/RESULTS/bt-mz_C.32_one_html
```

```
> scp <login>@stampede2.tacc.utexas.edu:bt_html.tgz .
> tar xf bt_html.tgz
> firefox maqao_bt/RESULTS/bt-mz_C.32_one_html/index.html
```

A sample result directory is in **MAQAO_HANDSON/bt/bt-mz_C.32_one_html/**

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

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

Scalability profiling of lulesh with MAQAO

Cédric Valensi

Setup ONE View for scalability analysis

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

```
> cd $SCRATCH/lulesh
> cp $HOME/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.sbatch"
...
batch_command = "sbatch <batch_script>"
...
mpi_command = "ibrun"
...
scalability_params = {
  {nb_processes = 1, nb_threads = 1, binary = nil},
  {nb_processes = 8, nb_threads = 1, binary = nil},
  {nb_processes = 1, nb_threads = 8, binary = nil},
  {nb_processes = 8, nb_threads = 4, binary = nil},
  {nb_processes = 8, nb_threads = 8, binary = nil} }
```

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/lulesh
> cp $HOME/MAQAO_HANDSON/lulesh/job_lulesh_maqao.sbatch .
> less job_lulesh_maqao.sbatch
```

```
...
#SBATCH --ntasks=8<number_processes>
#SBATCH --cpus-per-tasks=4<omp_num_threads>
...
srun ./lulesh2.0 -i 10 -p -s 130
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on lulesh (scalability mode)

Launch ONE View

```
> cd $SCRATCH/lulesh
> maqao oneview --create-report=one --with-scalability=on \
--config=config_maqao_lulesh.lua -xp=maqao_lulesh
```

Execution will be longer!

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

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

or

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

```
> scp <login>@stampede2.tacc.utexas.edu: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/lulesh2.0_one_html**

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 -n 1 \
--reservation VI-HPS_SKX_DAY4 -p skx-normal --pty bash
```

Load MAQAO environment

```
> source ~tg856579/tutorial/env_maqao.sh
```

Load latest GCC compiler

```
> module load gcc
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

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

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

```
> ./matmul_orig 150 10000
cycles per FMA: 2.24
```

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>@stampede2.tacc.utexas.edu:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		33.3
Time in loops (%)		99.51
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
```

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

  Total Time:                          33.34 s
  Time spent in loops:                 99.49 %
  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:      14.86
       Nb Loops to get 80%:    1
```

Viewing results (text)

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

  If Clean:
      Number of loops    | 1      | 2      |
      Cumulated Speedup  | 1.0009 | 1.0009 |
  Top 5 loops:
    matmul_orig - 2:     1.0009
    matmul_orig - 1:     1.0009

  If FP Vectorized:
      Number of loops    | 1      | 2      |
      Cumulated Speedup  | 2.3539 | 2.3539 |
  Top 5 loops:
    matmul_orig - 1:     2.3539
    matmul_orig - 2:     2.3539

  If Fully Vectorized:
      Number of loops    | 1      | 2      |
      Cumulated Speedup  | 14.8630 | 15.6418 |
  Top 5 loops:
    matmul_orig - 1:     14.863
    matmul_orig - 2:     15.6418
```

Loop ID

Viewing CQA output (text)

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

Loop ID

```
      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 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:
  * recompile with fassociative-math (included in Ofast or ffast-math) to extend loop
vectorization to FP reductions.
 - (…)
```

CQA output for the baseline kernel

Vectorization

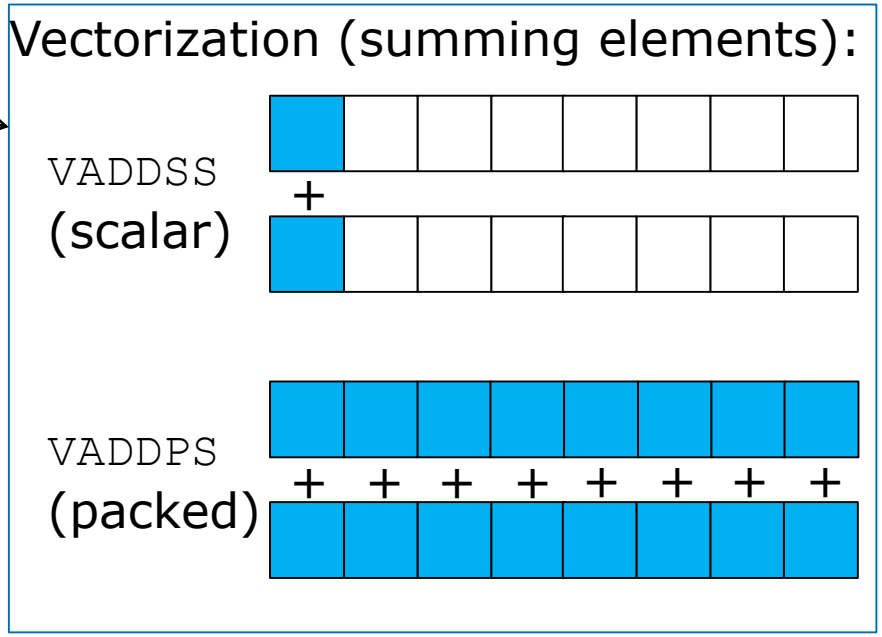
Your loop is not vectorized. 16 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 4.00 to 0.25 cycles (16.00x speedup).

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

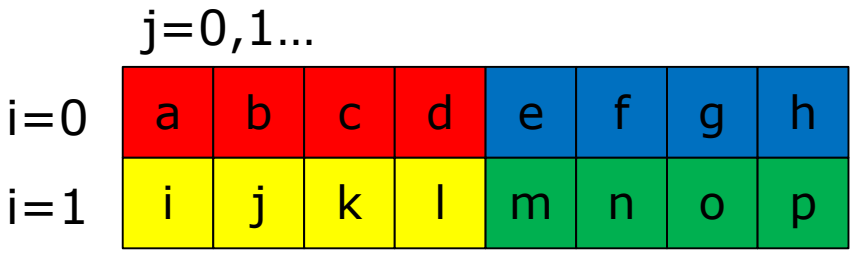
- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check 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)



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

Impact of loop permutation on data access

Logical mapping

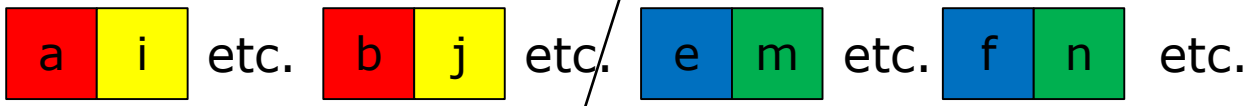


Efficient vectorization + prefetching

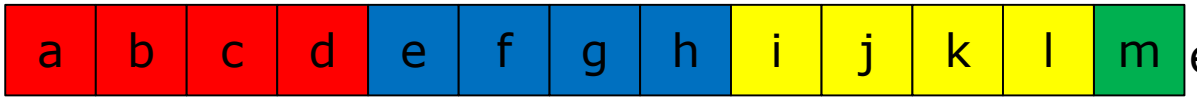
Physical mapping

(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 $HOME/MAQAO_HANDSON/matmul
> make matmul_perm
```

```
> ./matmul_perm 150 10000
cycles per FMA: 0.40
```

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**)

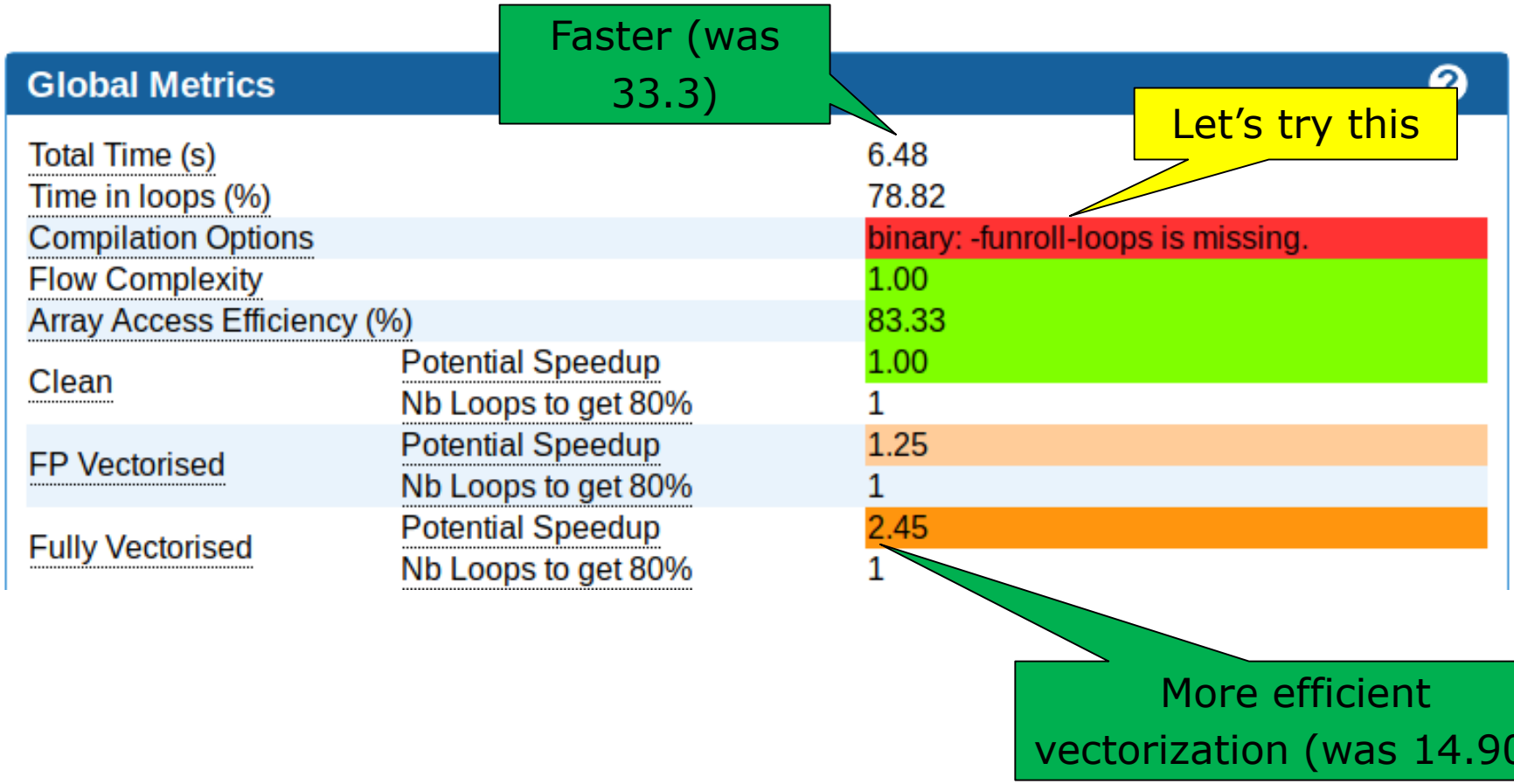
Loop permutation results

Faster (was 33.3)

Global Metrics		
Total Time (s)		6.48
Time in loops (%)		78.82
Compilation Options		binary: -funroll-loops is missing.
Flow Complexity		1.00
Array Access Efficiency (%)		83.33
Clean	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.25
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	2.45
	Nb Loops to get 80%	1

More efficient vectorization (was 14.90)

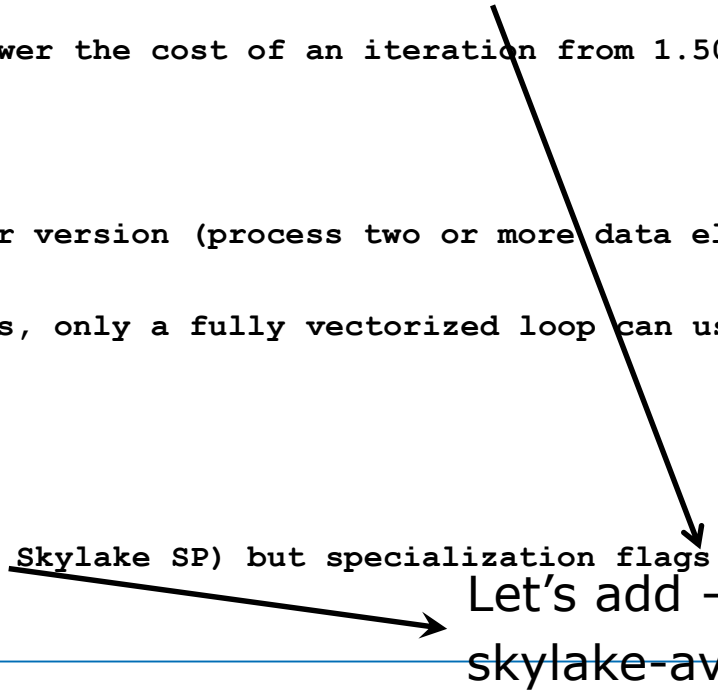
Loop permutation results



CQA output after loop permutation

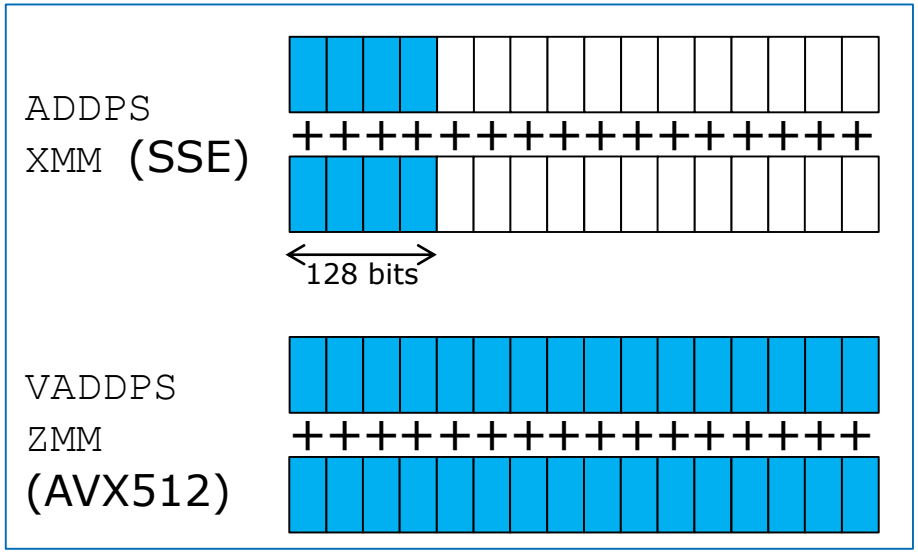
```
> maqao oneview --create-report=one -xp=ov_perm \
  --output-format=text --text-cqa=4 | less
```

```
      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 1.50 to 0.38
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
 - (…)
```



Impacts of architecture specialization: vectorization and FMA

- Vectorization
  - SSE instructions (SIMD 128 bits) used on a processor supporting AVX512 ones (SIMD 512 bits)
  - => 75% 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 $HOME/MAQAO_HANDSON/matmul
> make matmul_perm_opt
```

```
> ./matmul_perm_opt 150 10000
cycles per FMA: 0.26
```

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:

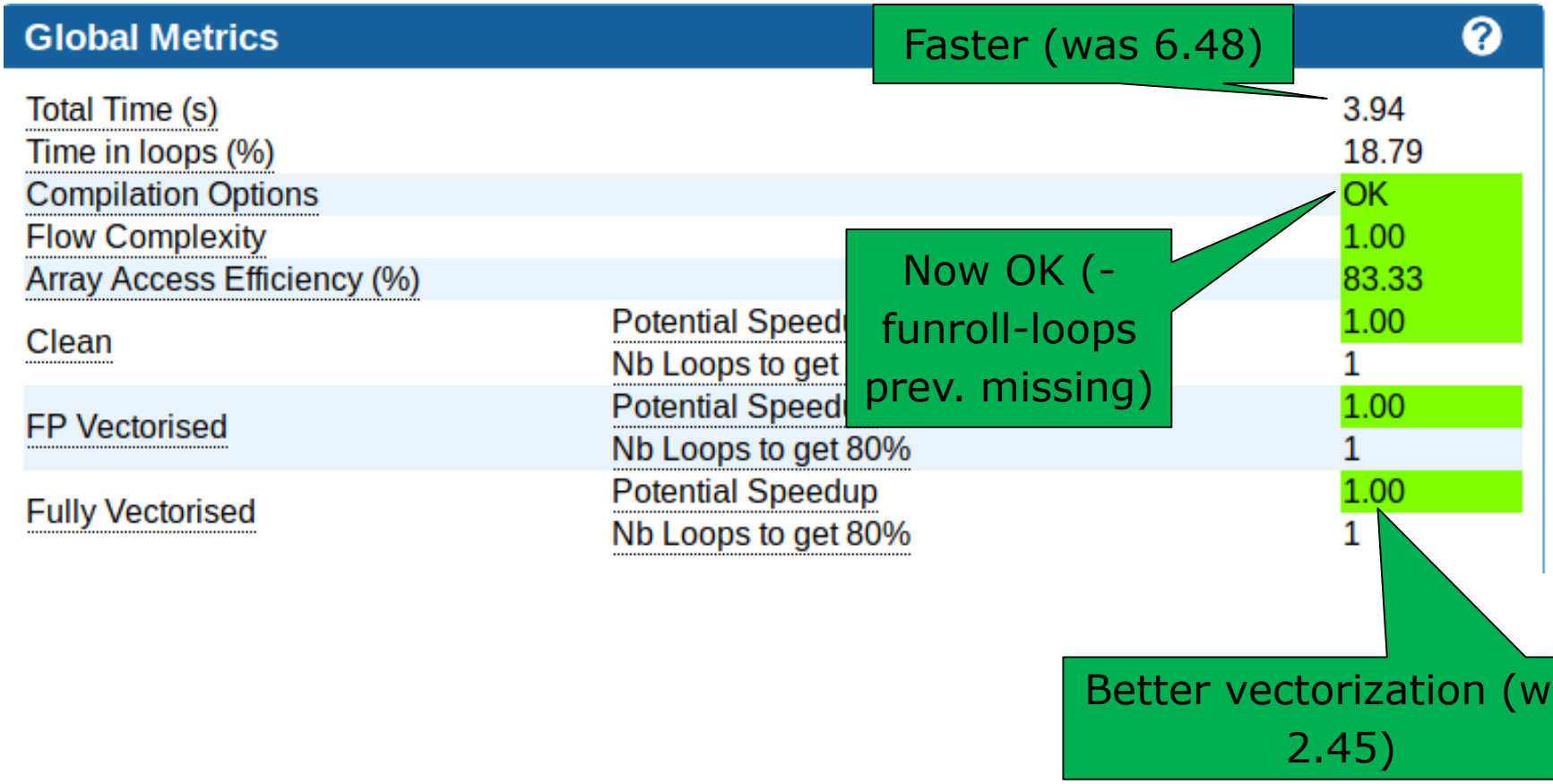
```
> maqao oneview --create-report=one c=ov_perm_opt.lua -xp=ov_perm_opt
```

Viewing new results:

```
> maqao oneview --create-report=one -xp=ov_perm_opt \
  --output-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)



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=5 | less
```

Using aligned arrays in matrix multiply

Compile aligned array version of matrix multiply

```
> cd $HOME/MAQAO_HANDSON/matmul
> make matmul_align
```

Checking aligned version:

```
> ./matmul_align 150 10000
Cannot call kernel on matrices with size%16 != 0 (data not
aligned on 64B boundaries)
Aborted
```

=> Alignment imposes restrictions on input parameters.

```
> ./matmul_align 160 10000
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.13
```

Analysing matrix multiply with aligned arrays

Analyse matrix multiply with ONE View

```
> maqao oneview --create-report=one \
--binary=./matmul_align --run-command="<binary> 160 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 \
  --output-format=text --text-global | less
```

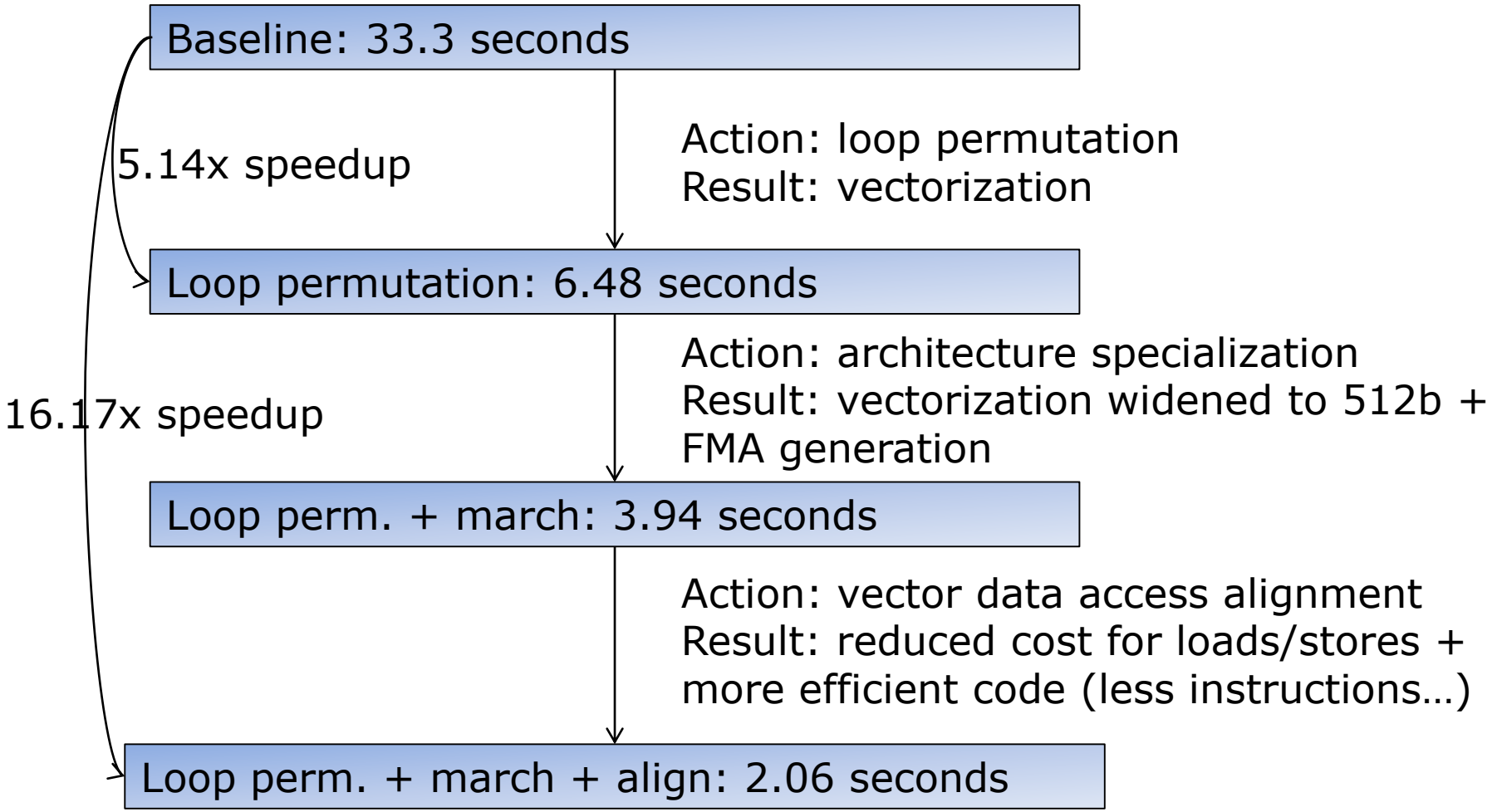
(Or download the **ov_align/RESULTS/matmul_align_one_html** folder locally and open **ov_align/RESULTS/matmul_align_one_html/index.html** in your browser)

Vector-aligning array accesses

Global Metrics		
Total Time (s)		2.06
Time in loops (%)		52.43
Compilation Options		OK
Flow Complexity		1.00
Array Access Efficiency (%)		75.00
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	1.00
	Nb Loops to get 80%	1

Extra speedup (was 3.94)

Summary of optimizations and gains



Hydro example

(If necessary) Request interactive session and load MAQAO environment

```
> srun -N 1 --exclusive -t 0:30:00 -n 1 \
--reservation VI-HPS_SKX_DAY4 -p skx-normal --pty bash
> source ~tg856579/tutorial/env_maqao.sh
```

Switch to the hydro handson folder

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

Make sure Intel compiler environment is loaded

```
> module load intel
```

Compile

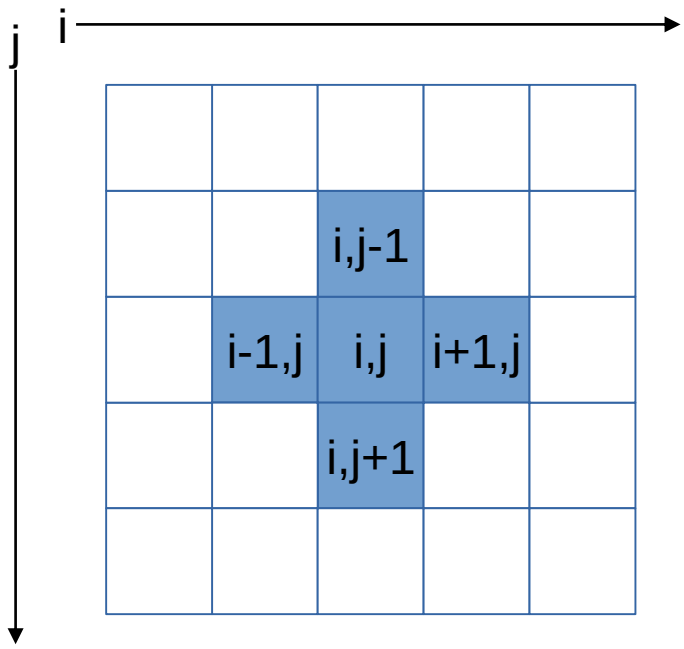
```
> make
```

```
> ./hydro_k0 300 100
Cycles per element for solvers: 1497.87
```

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



Running and analyzing kernel0 (icc -O3 -xHost)

- 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:                          7.4 s
  Time spent in loops:                 99.46 %
  Compilation Options:                 OK
  Flow Complexity:                     1.09
  Array Access Efficiency:             25.24 %
```

MAQAO Global Application Functions Loops Topology

Loops Index

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

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 129	104-110	hydro_k0:kernel.c	project	31.23
Loop 51	104-110	hydro_k0:kernel.c	c_densitySolver	21.2
Loop 83	104-110	hydro_k0:kernel.c	c_velocitySolver	20.76
Loop 90	104-110	hydro_k0:kernel.c	c_velocitySolver	20.76
Loop 133	368-371	hydro_k0:kernel.c	project	0.96
Loop 42	15-292	hydro_k0:kernel.c	c_densitySolver	0.87
Loop 131	380-383	hydro_k0:kernel.c	project	0.79
Loop 69	15-292	hydro_k0:kernel.c	c_velocitySolver	0.79
Loop 71	15-292	hydro_k0:kernel.c	c_velocitySolver	0.79
Loop 118	15-342	hydro_k0:kernel.c	c_velocitySolver	0.61
Loop 104	210-318	hydro_k0:kernel.c	c_velocitySolver	0.44
Loop 103	239-241	hydro_k0:kernel.c	c_velocitySolver	0.17
Loop 18	59-79	hydro_k0:kernel.c	setBoundry	0.09

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.

Loop Id: 129 | **Module: hydro_k0** | **Source: kernel.c:104-110** | **Coverage: 31.23%**

Source Code

```
/gpfs/home/nct00/nct00010/TESTS_HANDSON/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

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).

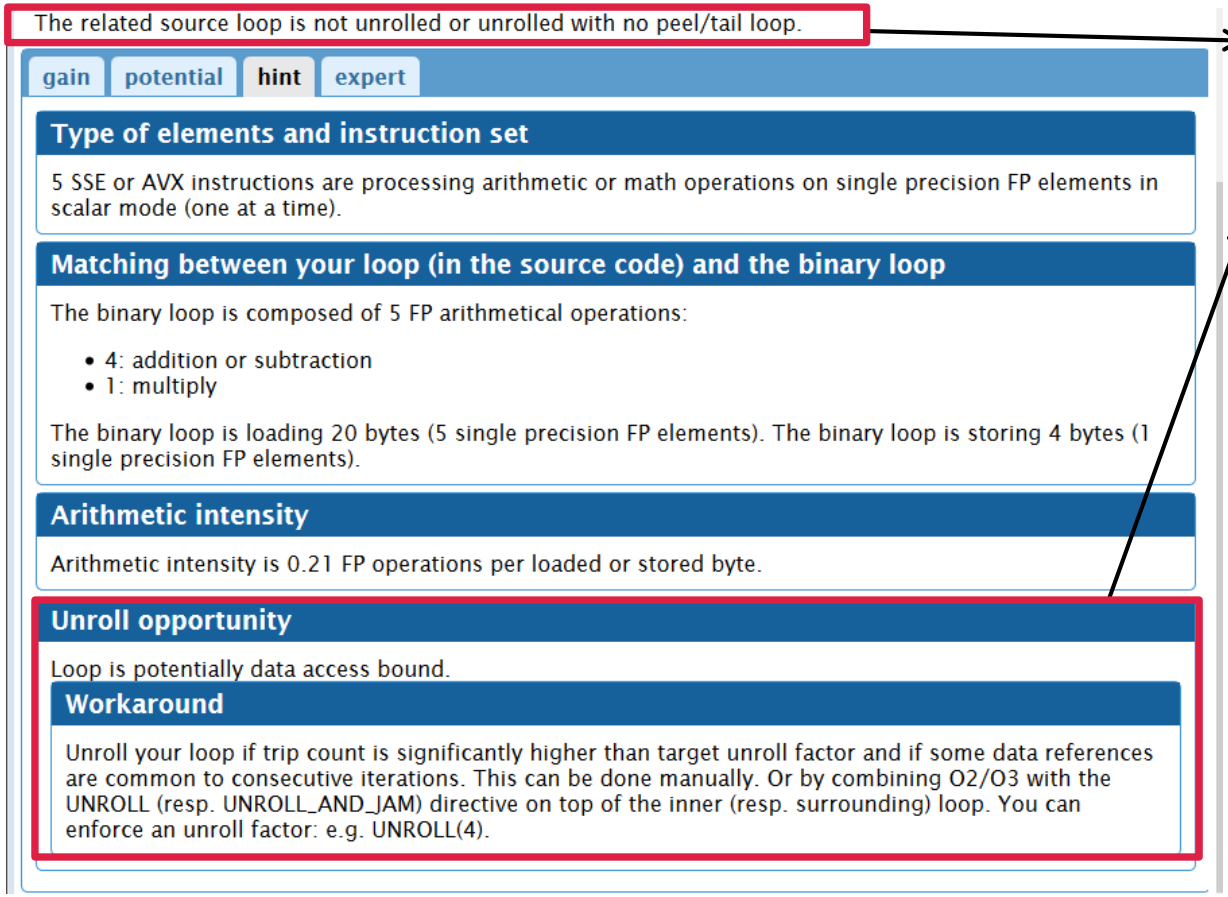
Details

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

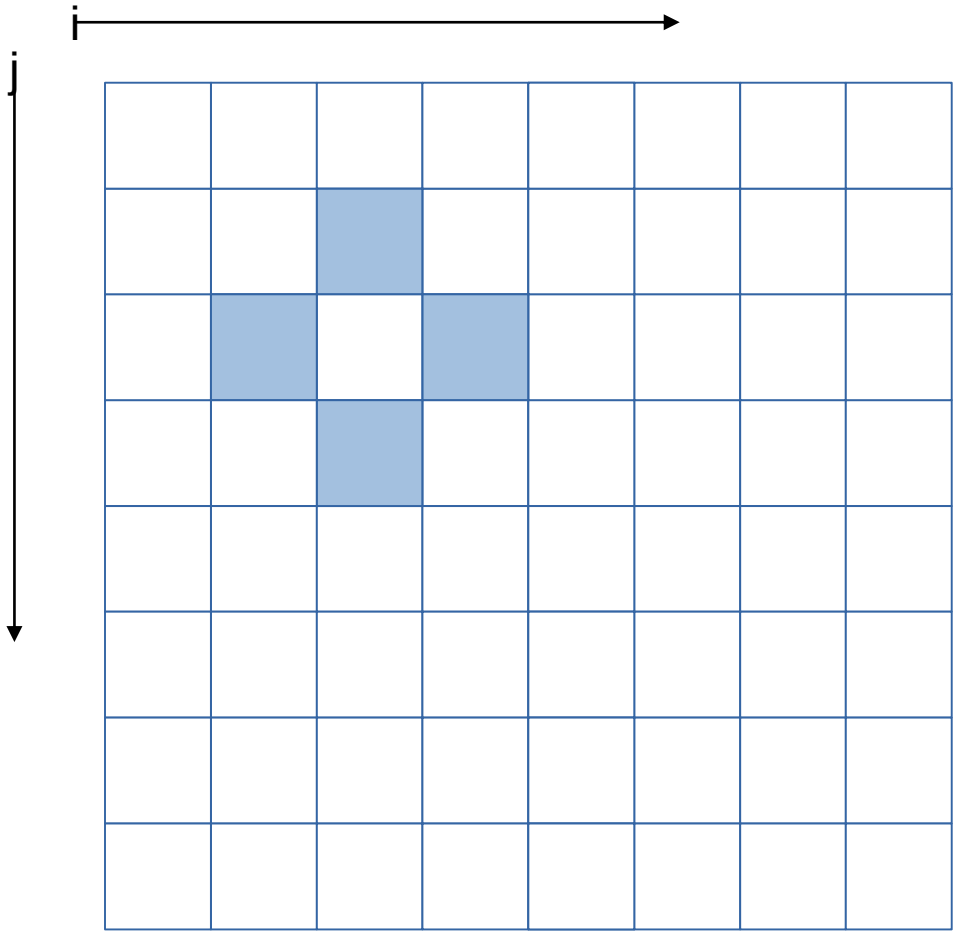
CQA output for kernel0



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

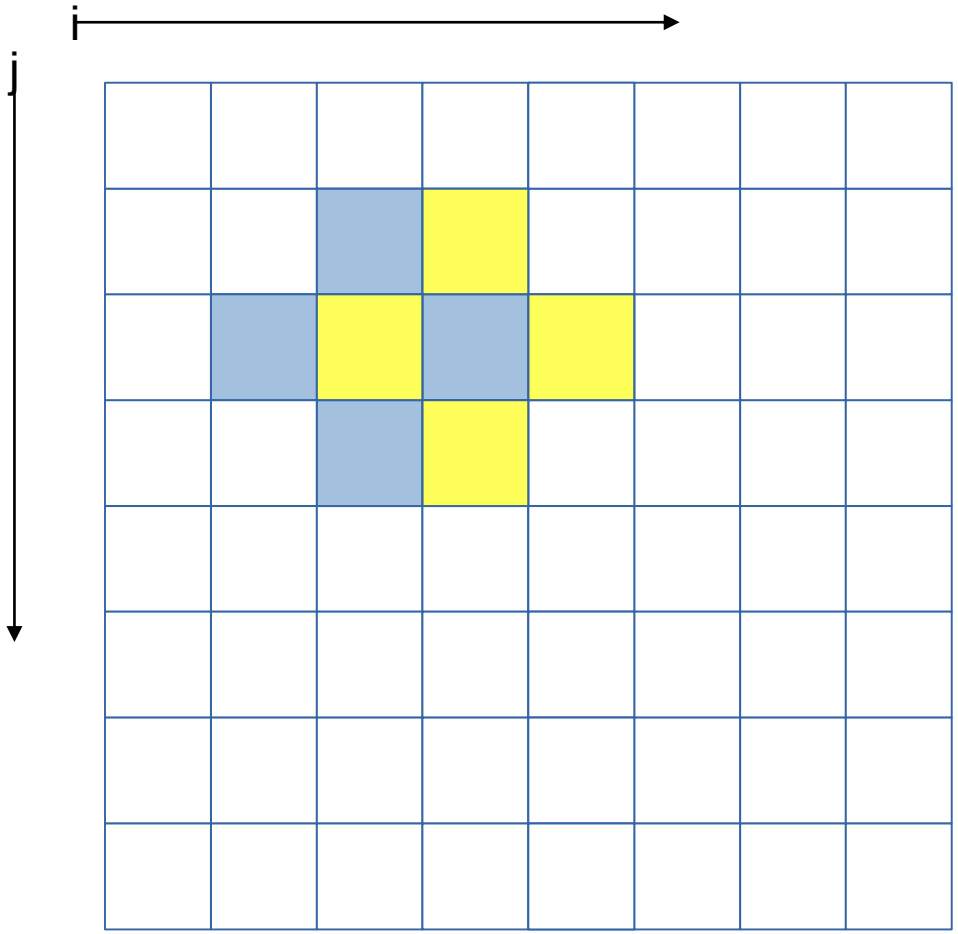
Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



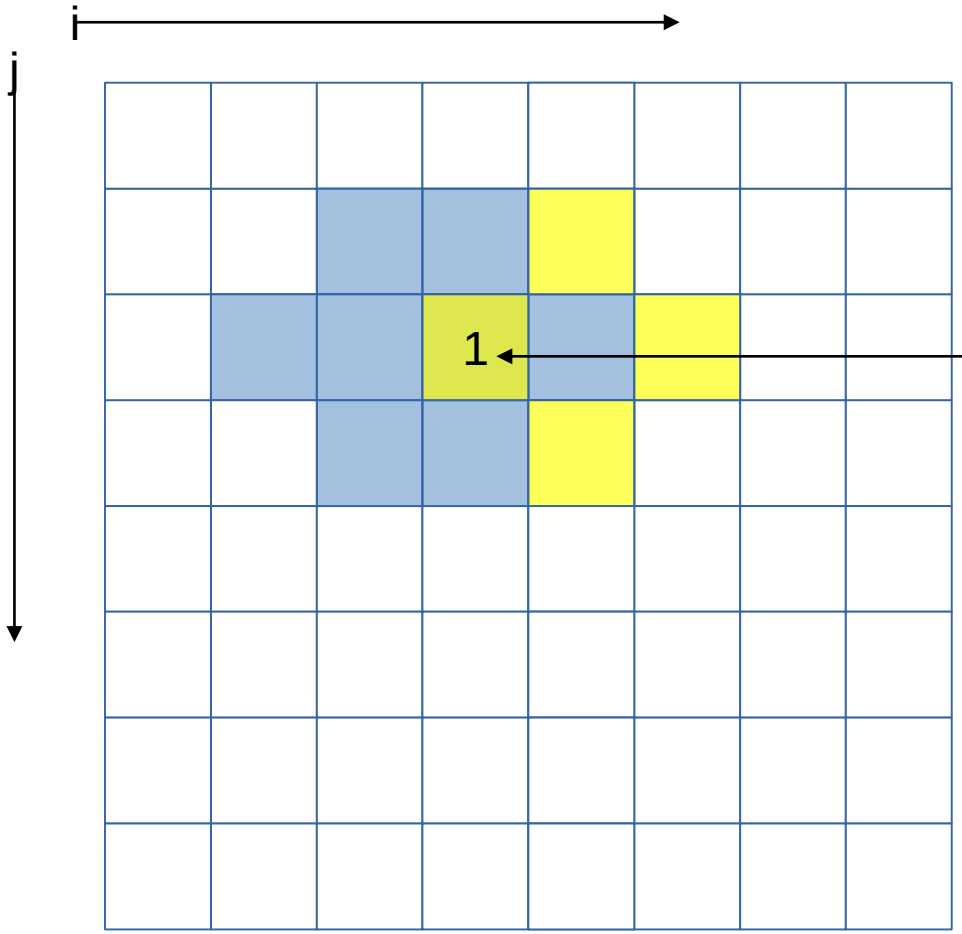
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

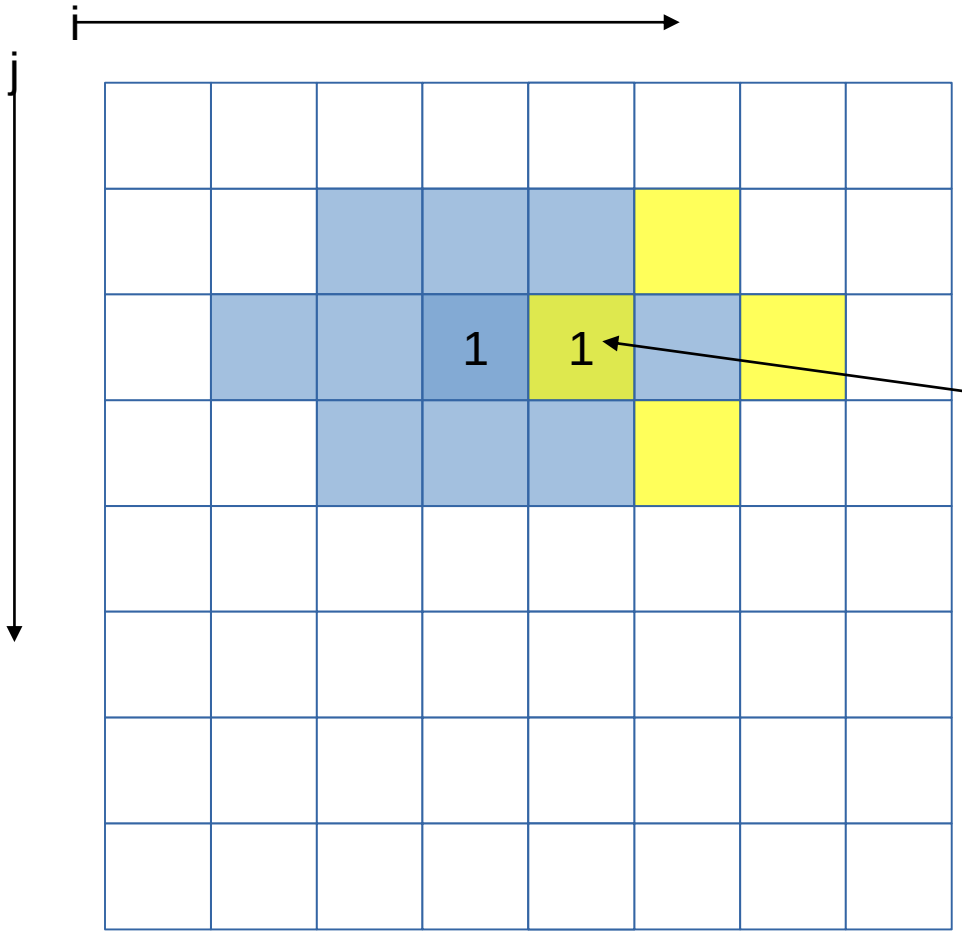
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

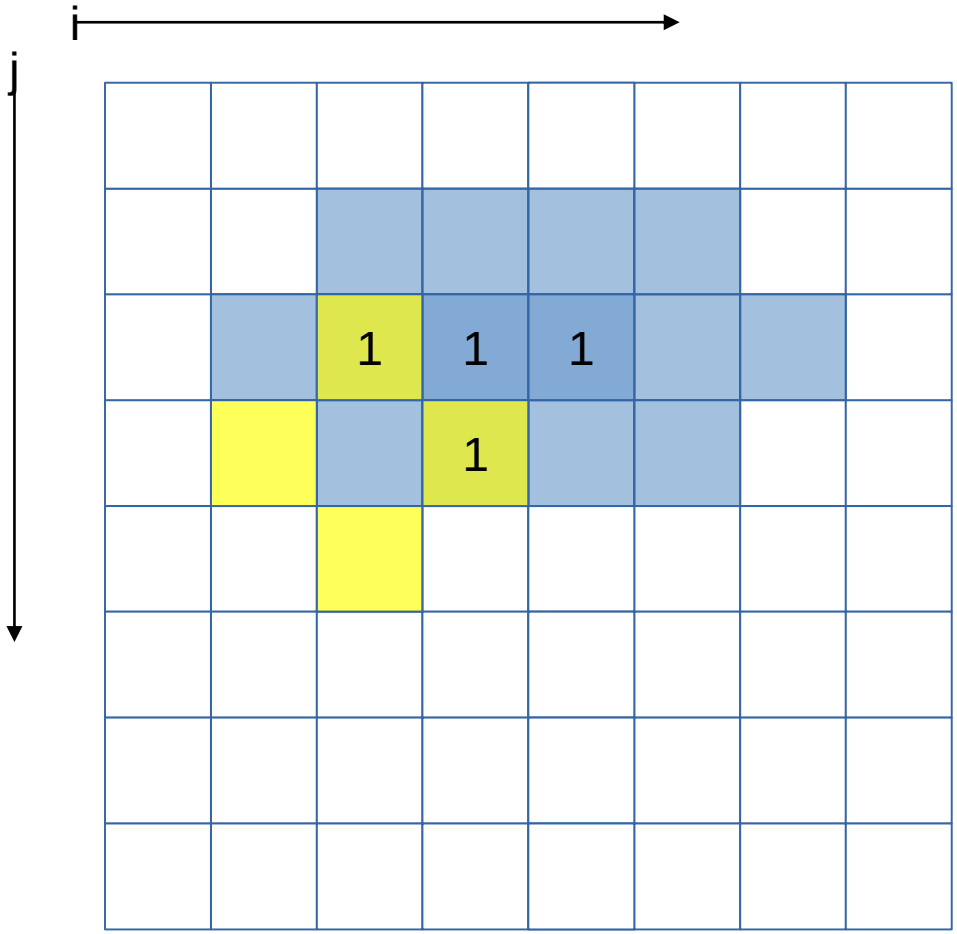
Memory references reuse : 4x4 unroll footprint on loads



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

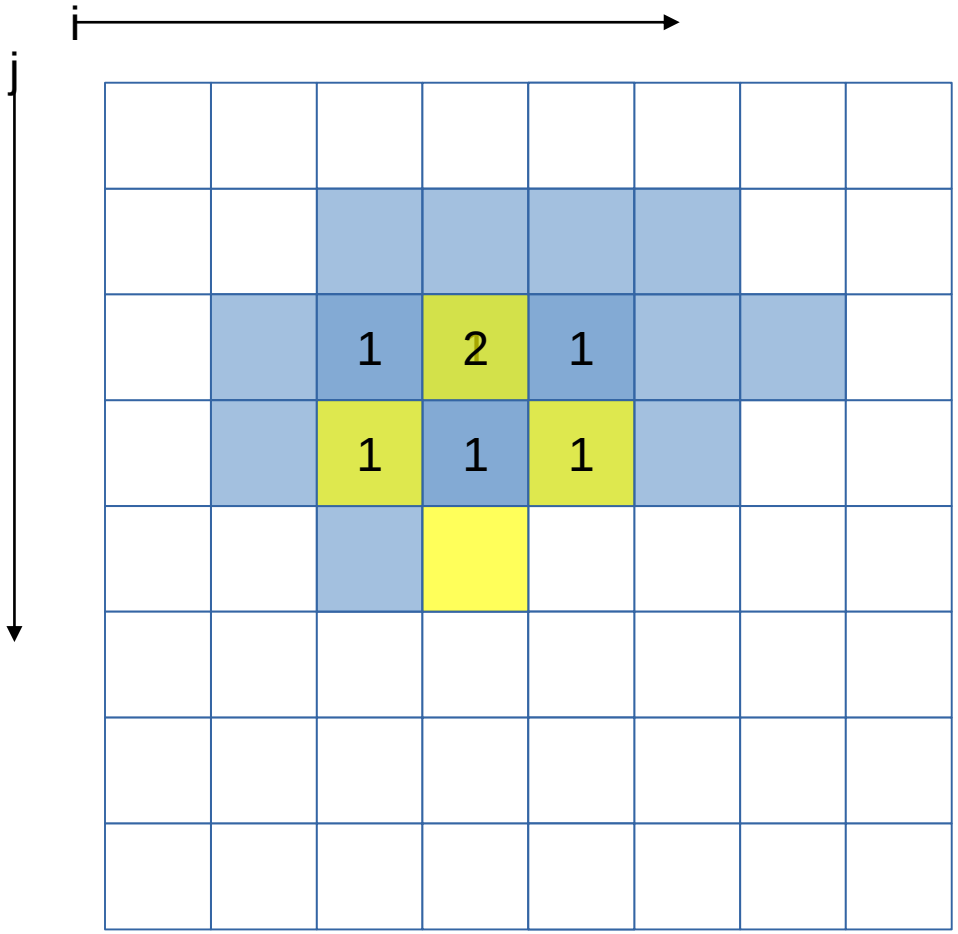


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

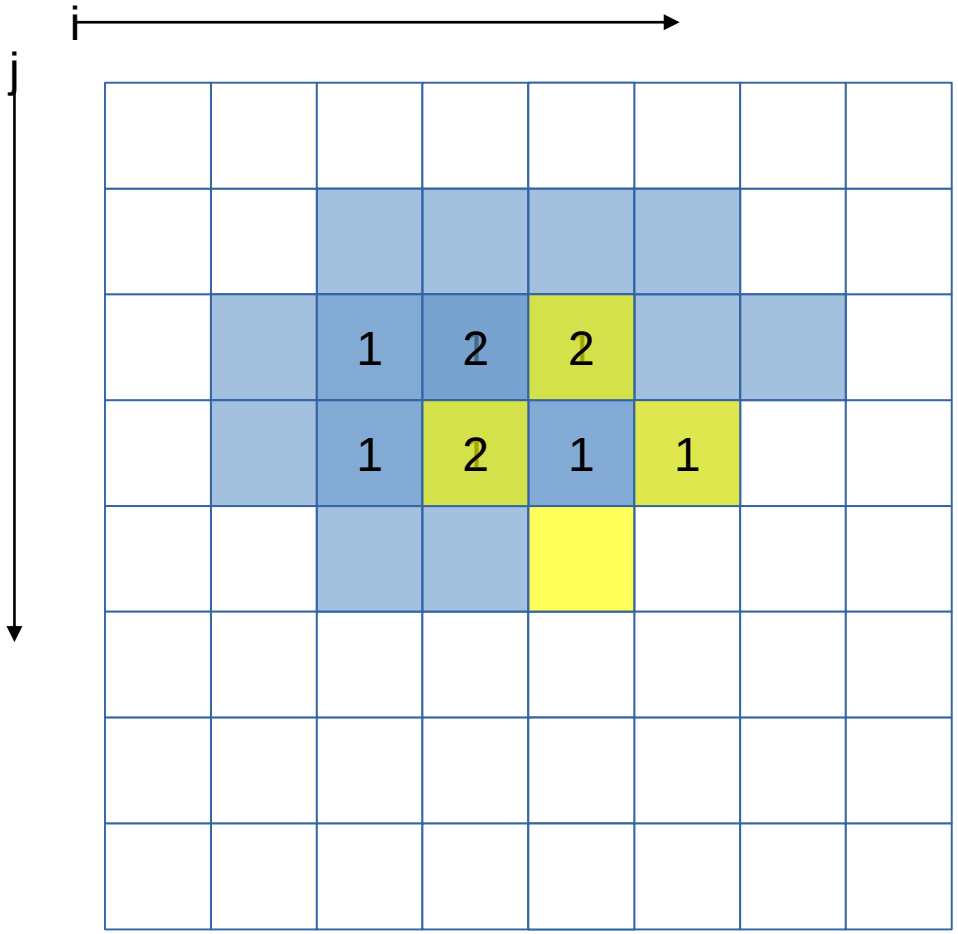


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

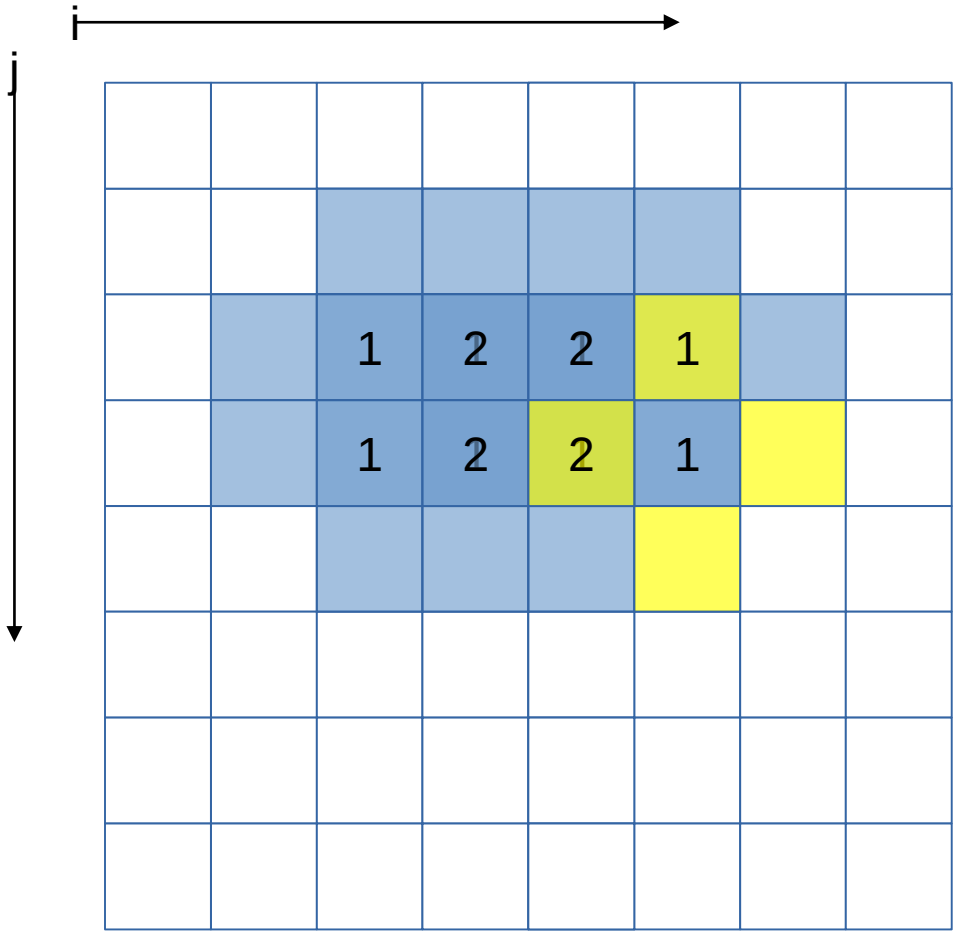


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

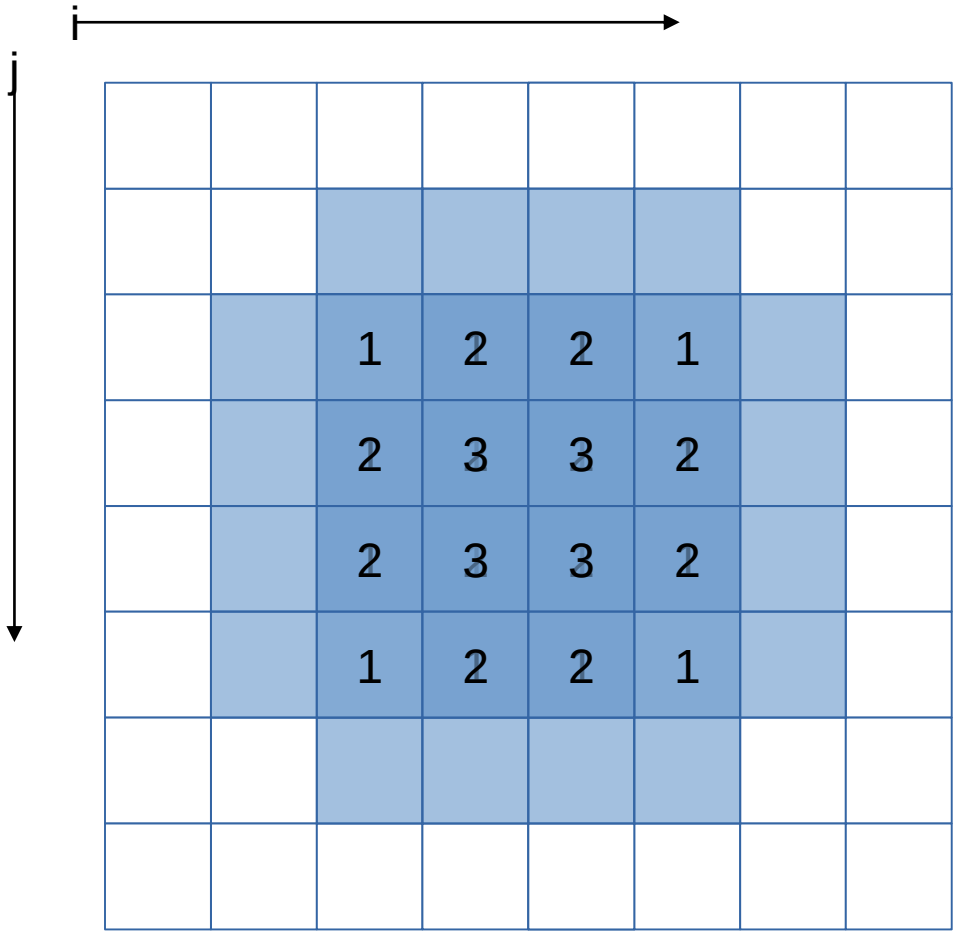


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

```
> ./hydro_k1 300 100
Cycles per element for solvers: 457.15
```

- 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:                          2.11 s
  Time spent in loops:                 97.97 %
  Compilation Options:                 OK
  Flow Complexity:                     1.29
  Array Access Efficiency:             26.23 %
```

MAQAO Global Application Functions Loops Topology

Loops Index

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

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 135	15-176	hydro_k1:kernel.c	linearSolver1	63.14
Loop 43	15-292	hydro_k1:kernel.c	c_densitySolver	3.11
Loop 72	15-292	hydro_k1:kernel.c	c_velocitySolver	2.8
Loop 70	15-292	hydro_k1:kernel.c	c_velocitySolver	2.8
Loop 121	15-342	hydro_k1:kernel.c	c_velocitySolver	2.49
Loop 74	368-371	hydro_k1:kernel.c	c_velocitySolver	1.87
Loop 88	368-371	hydro_k1:kernel.c	c_velocitySolver	1.56
Loop 107	210-318	hydro_k1:kernel.c	c_velocitySolver	1.24
Loop 86	380-383	hydro_k1:kernel.c	c_velocitySolver	1.24
Loop 68	380-383	hydro_k1:kernel.c	c_velocitySolver	1.24
Loop 106	239-241	hydro_k1:kernel.c	c_velocitySolver	0.62
Loop 18	59-79	hydro_k1:kernel.c	setBoundry	0.31
Loop 115	44-46	hydro_k1:kernel.c	c_velocitySolver	0

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

Loop Id: 135 **Module: hydro_k1** **Source: kernel.c:15-176** **Coverage: 63.14%**

Source Code

```
146:
147: void linearSolver1(int b, float* x, float* x0, float a, float c, float d
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

Path 1 / 1 OK

Coverage 63.14 %
Function linearSolver1
Source file and lines kernel.c:15-176
Module hydro_k1
The loop is defined in /gpfs/home/nct00/nct00010/TESTS_HANDSON/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

CQA output for kernel1

```
> maqao oneview --create-report=one xp=ov_k1 \
--output-format=text --text-cqa=129 | less
```

gain | potential | **hint** | expert

Type of elements and instruction set

80 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:

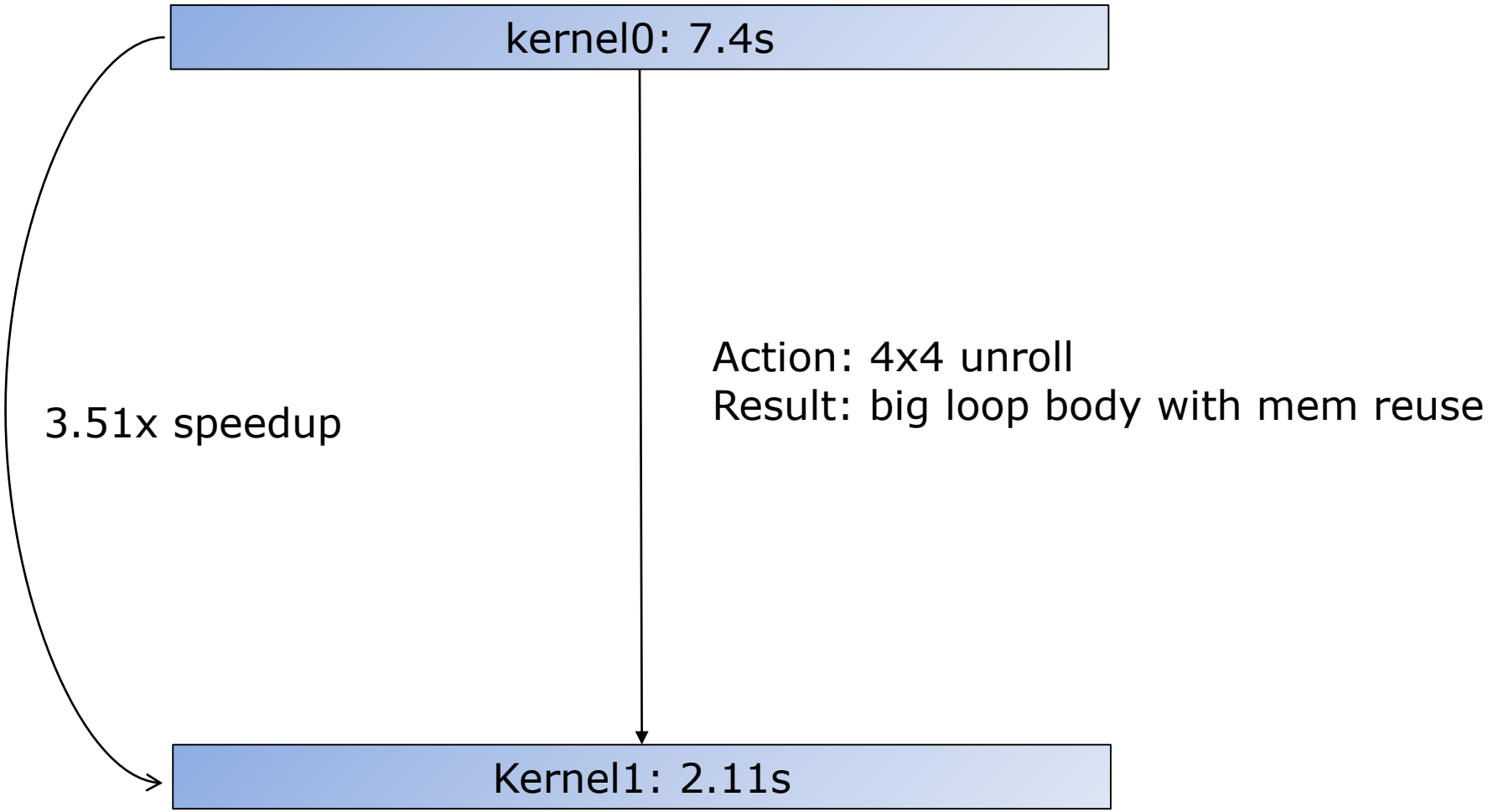
- 64: addition or subtraction
- 32: multiply

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

```
~tg856579/tutorial/loop_optim_tutorial.tgz
```