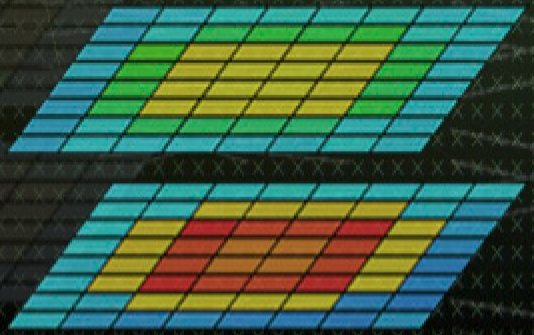


MAQAO
Hands-on exercises

Profiling bt-mz (incl. scalability)
Optimising a code

Setup (reminder)

Login to the cluster

```
> ssh <username>@goethe.hhlr-gu.de
```

Copy handson material to your workspace directory

```
> export TW37=/home/vihps/software # optional: should already be in ~/.bash_profile
> export WORK=/scratch/vihps/$USER # optional: should already be in ~/.bash_profile
> cd $WORK
> tar xvf $TW37/maqao/MAQAO_HANDSON.tgz
```

Load MAQAO environment

```
> module use $TW37/modulefiles
> module load maqao
```

Setup (bt-mz compilation with Intel compiler and MPI & debug symbols)

Go to the NPB directory provided with MAQAO handsons

```
> cd $WORK/NPB3.4-MZ-MPI
```

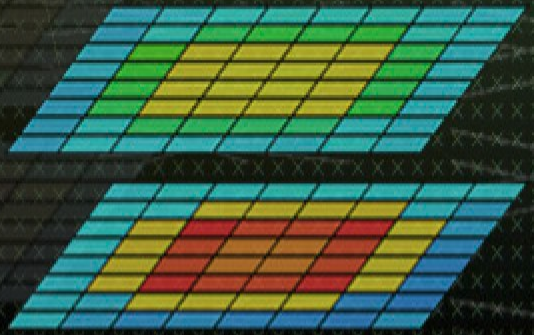
Load Intel compiler and environment

```
> module load comp/intel/2019.5 mpi/intel/2019.5
```

Compile and run

```
> make bt-mz CLASS=C
> cd bin
> cp $WORK/MAQAO_HANDSON/bt/bt.sbatch .
> sbatch bt.sbatch
```

Remark: with version 3.4 the generated executable supports any number of ranks (no need to generate one executable for 6 ranks, another for 8 etc.)



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/NPB3.4-MZ-MPI/bin
> cp $WORK/MAQAO_HANDSON/bt/config_bt_oneview_sbatch.lua .
> less config_bt_oneview_sbatch.lua
```

```
binary = "bt-mz.C.x"
…
batch_script = "bt_maqao.sbatch"
batch_command = "sbatch <batch_script>“
…
number_processes = 4
number_nodes = 2
…
omp_num_threads = 20
…
mpi_command = "mpirun -n <number_processes>"
```

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/NPB3.4-MZ-MPI/bin #if current directory has changed
> cp $WORK/MAQAO_HANDSON/bt/bt_maqao.sbatch .
> less bt_maqao.sbatch
```

```
...
#SBATCH --nodes=2<number_nodes>
#SBATCH --ntasks=4<number_processes>
#SBATCH --cpus-per-task=20<omp_num_threads>
...
export OMP_NUM_THREADS=20<omp_num_threads>
...
mpirun -n ... $EXE
<mpi_command> <run_command>
...
```

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

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin #if current directory has changed
> maqao oneview -R1 --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>.

WARNINGS:

- 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/NPB3.4-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.x"
...
number_processes = 4
number_nodes = 2
...
omp_num_threads = 20
...
mpi_command = "mpirun -n <number_processes>"
```

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

Request interactive session

```
> srun -p general1 --reservation=VIHPS -N 2 --pty bash
```

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin
> maqao oneview -R1 --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 goethe_work
> sshfs <user>@goethe.hhlr-gu.de:/scratch/vihps/<user> \
goethe_work
> firefox goethe_work/NPB3.4-MZ-MPI/bin/ov_sbatch/RESULTS/bt-
mz.C.x_one_html/index.html
```

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

```
> tar czf $HOME/bt_html.tgz ov_sbatch/RESULTS/bt-mz.C.x_one_html
```

```
> scp <login>@goethe.hhlr-gu.de:bt_html.tgz .
> tar xf bt_html.tgz
> firefox ov_sbatch/RESULTS/bt-mz.C.x_one_html/index.html
```

sshfs & scp hints

- To install sshfs on Debian-based Linux distributions (like Ubuntu)

```
> sudo apt install sshfs
```

- Recommended to close a sshfs directory after use

```
> fusermount -u /path/to/sshfs/directory
```

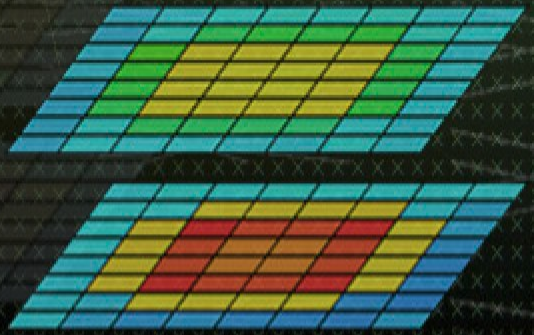
- scp is slow to copy directories (especially when containing many small files), copy a .tgz archive of the directory

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 -R1 -xp=ov_sbatch --output-format=text
```



Scalability profiling of bt-mz with MAQAO

Salah Ibnamar

Setup ONE View for scalability analysis

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

```
> cd $WORK/NPB3.4-MZ-MPI/bin #if cur. dir. has changed
> cp $WORK/MAQAO_HANDSON/bt/config_bt_scalability.lua .
> less config_bt_scalability.lua
```

```
binary = "./bt-mz.C.x"
…
run_command = "<binary>"
…
batch_script = "bt_maqao.sbatch"
…
batch_command = “sbatch <batch_script>"
…
number_processes = 4
…
number_nodes = 1
…
omp_num_threads = 1
…
mpi_command = “mpirun –n <number_processes>"
…
multiruns_params = {
 {nb_processes = 1, nb_threads = 20, number_nodes = 1},
 {nb_processes = 4, nb_threads =  1, number_nodes = 2},
 {nb_processes = 4, nb_threads = 20, number_nodes = 2},
}
scalability_reference = “lowest-threads”
```

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

Launch ONE View (execution will be longer!)

```
> maqao oneview -R1 --with-scalability=on \
--config=config_bt_scalability.lua -xp=ov_scal
```

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

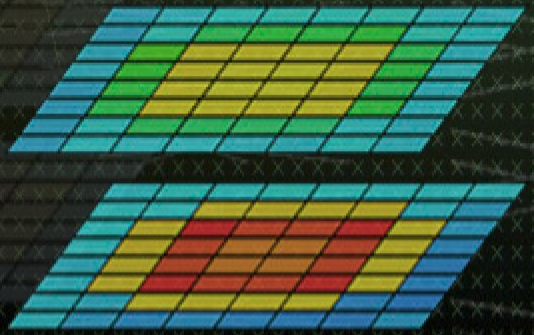
```
> firefox goethe_work/NPB3.4-MZ-MPI/bin/ov_scal/RESULTS/bt-
mz.C.x_one_html/index.html
```

OR

```
> tar czf $HOME/bt_scal.tgz \
ov_scal/RESULTS/bt-mz.C.x_one_html
```

```
> scp <login>@goethe.hhlr-gu.de:ov_scal.tgz .
> tar xf ov_scal.tgz
> firefox ov_scal/RESULTS/bt-mz.C.x_one_html/index.html
```

A sample result directory is in **MAQAO_HANDSON/bt/bt_scal_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

```
> srun -p general1 --reservation=VIHPS --pty bash
> module load comp/gcc/8.2.0
> module use $TW37/modulefiles
> module load maqao
```

Define variable for workspace directory (if not already done)

```
> export WORK=/scratch/vihps/$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>

```
> ./matmul_orig 400 300
cycles per FMA: 2.92
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 --binary=./matmul_orig \
--run-command="<binary> 400 300" -xp=ov_orig
```

OR, using a configuration script:

```
> maqao oneview -R1 -c=ov_orig.lua -xp=ov_orig
```

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox goethe_work/MAQAO_HANDSON/matmul/ov_orig/RESULTS/
matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		20.51
Time in loops (%)		99.94
Time in innermost loops (%)		99.90
Time in user code (%)		99.95
Compilation Options		**binary**: -march=(target) is missing. -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.33
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	2.37
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	15.86
	Nb Loops to get 80%	1

CQA output for the baseline kernel

Vectorization

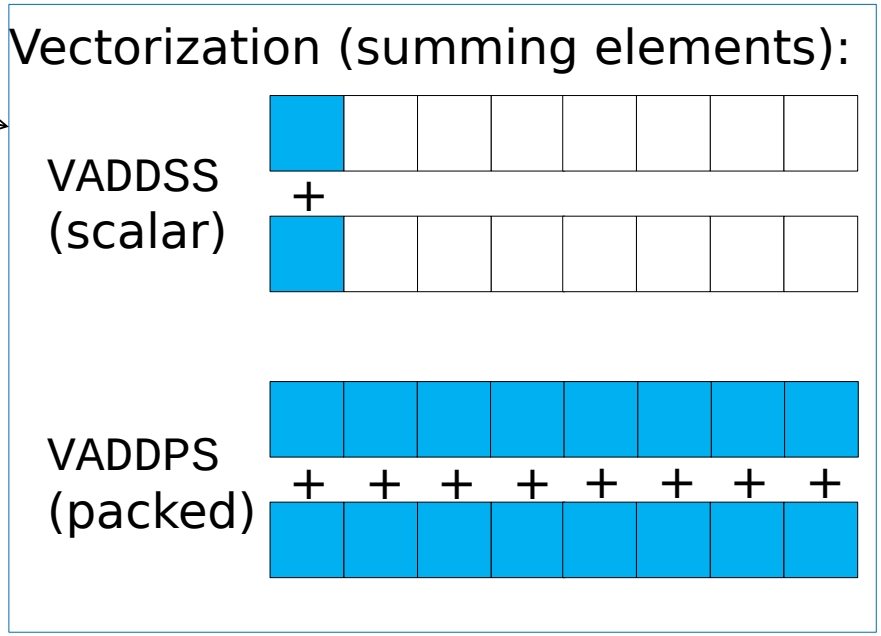
Your loop is not vectorized. 16 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 4.00 to 0.25 cycles (16.00x speedup).

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

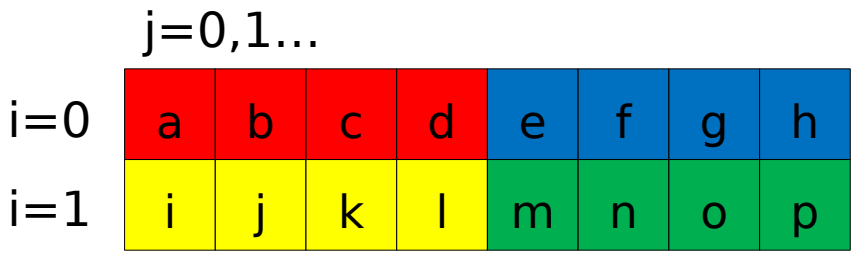
- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check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

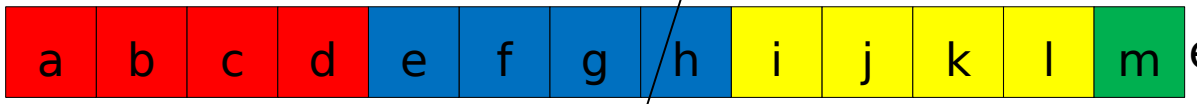
Logical mapping



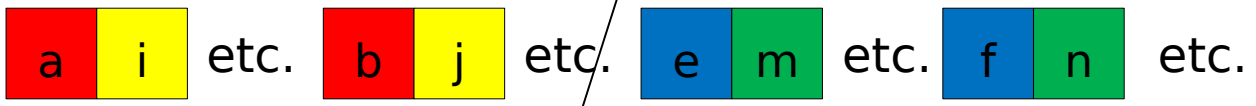
Efficient vectorization + prefetching

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

```
> ./matmul_perm 400 300
cycles per FMA: 0.44
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 --binary=./matmul_perm \
--run-command="<binary> 400 300" -xp=ov_perm
```

OR, using a configuration script:

```
> maqao oneview -R1 -c=ov_perm.lua -xp=ov_perm
```

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox goethe_work/MAQAO_HANDSON/matmul/ov_perm/RESULTS/
matmul_perm_one_html/index.html &
```

Global Metrics

Total Time (s)		2.92
Time in loops (%)		99.60
Time in innermost loops (%)		97.06
Time in user code (%)		99.63
Compilation Options		**binary**: -march=(target) is missing. -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		100.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.01
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.74
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.03
	Nb Loops to get 80%	1

Faster (was 20.51)

More efficient vectorization (was 15.86)

CQA output after loop permutation

gain | potential | hint | expert

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 1.75 to 0.44 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.

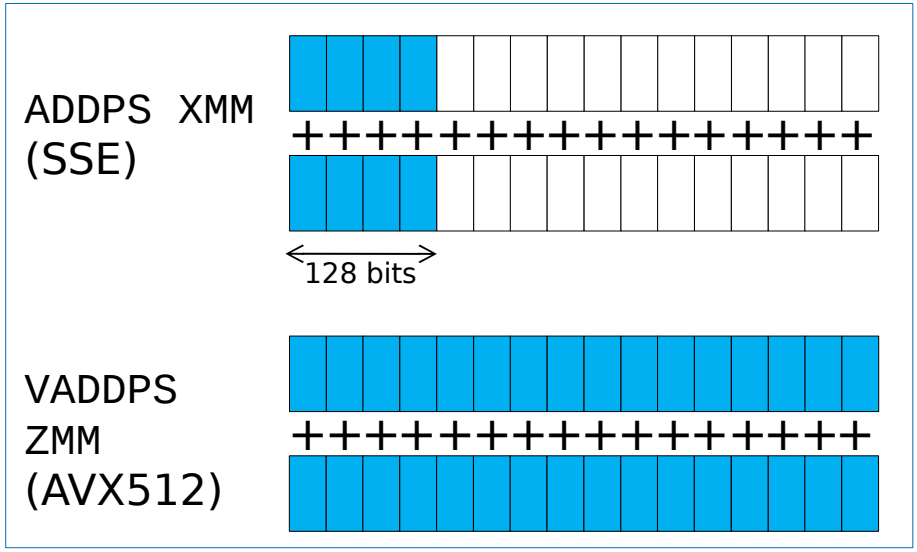
Workaround

- Recompile with march=skylake-avx512. CQA target is Skylake_SP (Intel(R) Xeon(R) Skylake SP) b
  -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.

Let’s try this

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/KNL 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 microarchitecture-specialization and array alignment

Compile array-aligned version of matrix multiply

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

```
> ./matmul_align 400 300 # remark: size%16 has to equal 0
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.21
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 --binary=./matmul_align \
--run-command="<binary> 400 300" -xp=ov_align
```

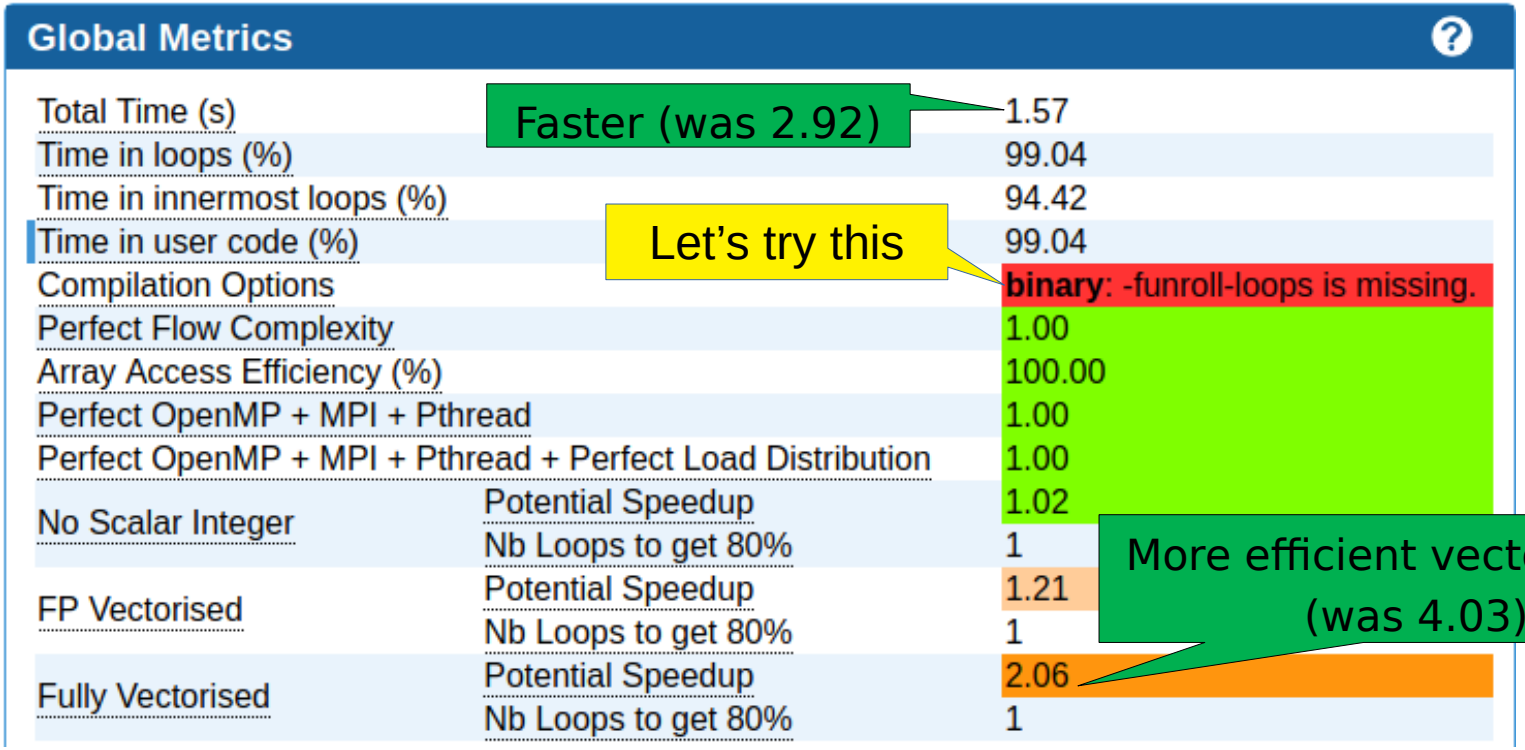
OR using configuration script:

```
> maqao oneview -R1 -c=ov_align.lua -xp=ov_align
```

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox goethe_work/MAQAO_HANDSON/matmul/ov_align/RESULTS/
matmul_align_one_html/index.html &
```



CQA output after microarchitecture-specialization and array alignment

gain | potential | hint | expert

Vectorization

Your loop is vectorized, but using only 256 out of 512 bits (AVX/AVX2 instructions on AVX-512 processors).

Details

All SSE/AVX instructions are used in vector version (process two or more data elements in vector registers).

Workaround

Read the "512-bits vectorization on Skylake SP" report at "Potential" confidence level.

gain | potential | hint | expert

512-bits vectorization on Skylake SP

On Gold 5122, 6xxx and Platinum processors, performance can be improved by using 512-bits vectorization if the number of vectorized loops is high and with high trip count.

Workaround

Recompile with -mprefer-vector-width=512

Let’s try this

Analyse matrix multiply with loop unrolling and 512 bits vectorization

Compile unrolled+512b version of matrix multiply

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

```
> ./matmul_unroll 400 300
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.12
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 --binary=./matmul_unroll \
--run-command="<binary> 400 300" -xp=ov_unroll
```

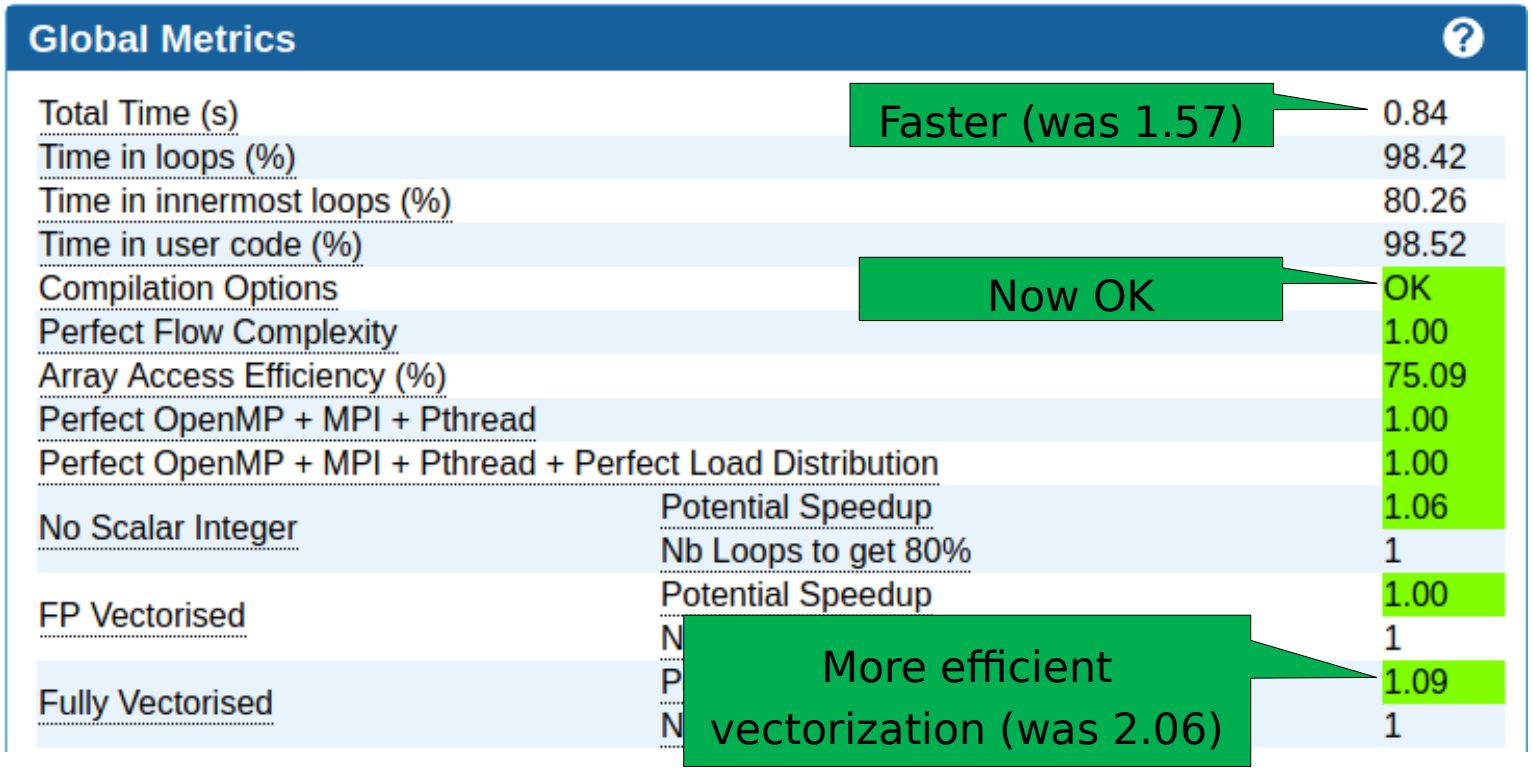
OR using configuration script:

```
> maqao oneview -R1 -c=ov_unroll.lua -xp=ov_unroll
```

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox goethe_work/MAQAO_HANDSON/matmul/ov_unroll/RESULTS/
matmul_unroll_one_html/index.html &
```



Using comparison mode (iso-source)

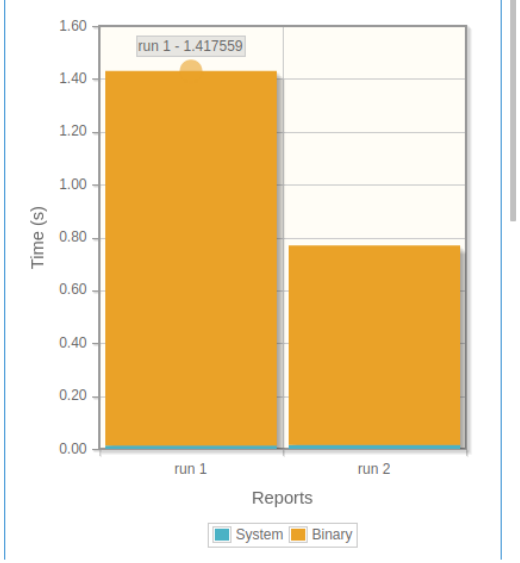
```
> maqao oneview --compare-reports --inputs=ov_align,ov_unroll \
-xp=ov_align_vs_unroll
```

Global Metrics

Metric		run 1	run 2
Total Time (s)		1.43	0.77
Profil Time (s)		1.43	0.77
Time in loops (%)		99.13	97.96
Time in innermost loops (%)		94.53	79.34
Time in user code (%)		99.13	98.06
Compilation Options		**binary**: -funroll-loops is missing.	OK
Perfect Flow Complexity		1.00	1.00
Array Access Efficiency (%)		100.00	75.07
Perfect OpenMP + MPI + Pthread		1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00
No Scalar Integer	Potential Speedup	1.02	1.06
	Nb Loops to get 80%	1	1
FP Vectorised	Potential Speedup	1.21	1.00
	Nb Loops to get 80%	1	1
Fully Vectorised	Potential Speedup	2.07	1.10
	Nb Loops to get 80%	1	1

Application Categorization

Time



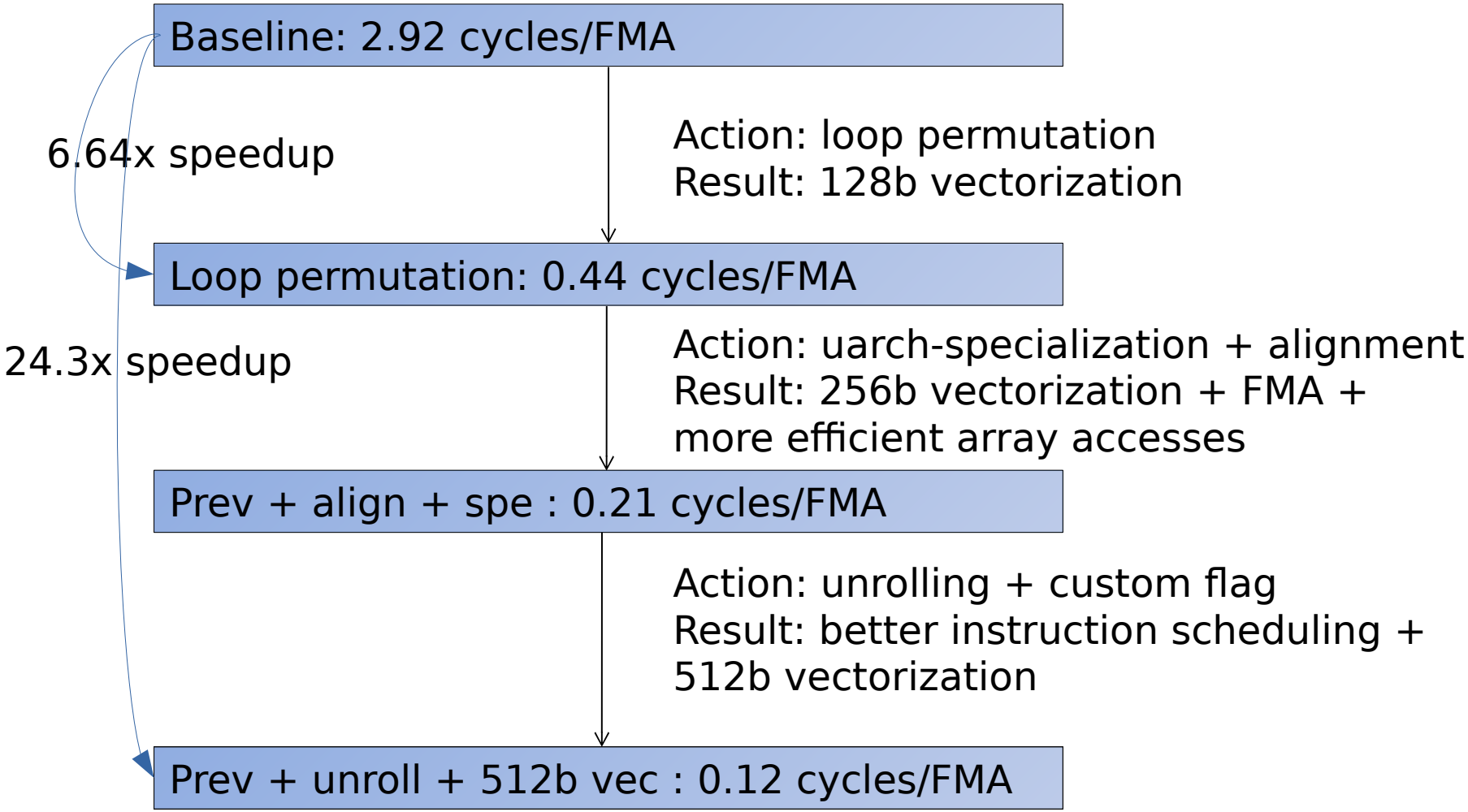
Experiment Summaries

	run 1	run 2
Application	./matmul_align	./matmul_unroll
Timestamp		
Experiment Type	Sequential	Sequential
Machine	node46-021.cm.cluster	node46-021.cm.cluster
Architecture	x86_64	x86_64
Micro Architecture	SKYLAKE	SKYLAKE
Model Name	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
Cache Size	28160 KB	28160 KB
Number of Cores	20	20
Maximal Frequency	3.7 GHz	3.7 GHz
OS Version	Linux 3.10.0-1127.13.1.el7.x86_64 #1 SMP Tue Jun 23 10:32:27 CDT 2020	Linux 3.10.0-1127.13.1.el7.x86_64 #1 SMP Tue Jun 23 10:32:27 CDT 2020
Architecture used during static analysis	x86_64	x86_64
Micro Architecture used during static analysis	SKYLAKE	SKYLAKE
Compilation Options	**binary**: GNU 8.2.0 -march=skylake-avx512 -g -O3 -fno-omit-frame-pointer	**binary**: GNU 8.2.0 -march=skylake-avx512 -mprefer-vector-width=512 -g -O3 -funroll-loops -fno-omit-frame-pointer

Functions

Name	Module	Coverage (%)		Time (s)		Nb Threads		Deviation (coverage)	
		run 1	run 2	run 1	run 2	run 1	run 2	run 1	run 2
kernel	binary	99.01	97.74	1.42	0.76	1	1	0.00	0.00
__GI_memset	libc-2.17.so	0.41	1.18	0.01	0.01	1	1	0.00	0.00
__random_r	libc-2.17.so	0.29	0.32	0	0	1	1	0.00	0.00
init_mat	binary	0.12	0.22	0	0	1	1	0.00	0.00
rand	libc-2.17.so	0.06	0.22	0	0	1	1	0.00	0.00
__random	libc-2.17.so	0.12	0.11	0	0	1	1	0.00	0.00
Unknown function	binary	0	0.11		0		1		0.00
unknown kernel region	kernel	0	0.11		0		1		0.00

Summary of optimizations and gains



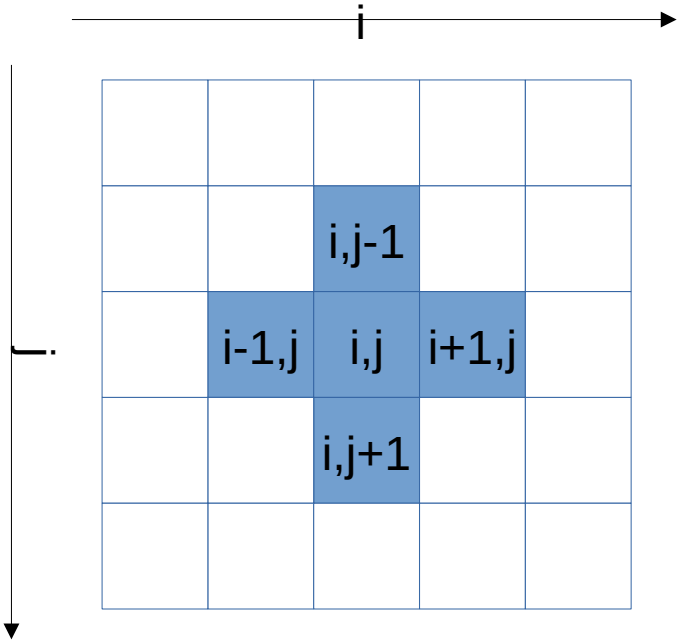
Hydro code

```
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



Compile with Intel compiler on login node (not possible on goethe compute nodes)

Go back to login node (if necessary)

```
> exit
```

Define variable for workspace directory (if not already done)

```
> export WORK=/scratch/vihps/$USER
```

Switch to the hydro handson folder

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

Load Intel 19 compiler

```
> module purge
> module load comp/intel/2019.5
```

Compile

```
> make
```

Hydro example: interactive session

Request interactive session and load maqao

```
> srun -p general1 --reservation=VIHPS --pty bash
> module use $TW37/modulefiles
> module load maqao
```

Running and analyzing kernel0

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

```
> ./hydro_k0 300 50
Cycles per element for solvers: 1739.38
```

Profile with MAQAO (200 repetitions)

```
> maqao oneview -R1 -xp=ov_k0 -c=ov_k0.lua
```

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox goethe_work/MAQAO_HANDSON/hydro/ov_k0/RESULTS/
hydro_k0_one_html/index.html &
```

Global Metrics

Total Time (s)		13.01
Time in loops (%)		99.96
Time in innermost loops (%)		99.91
Time in user code (%)		99.97
Compilation Options		OK
Perfect Flow Complexity		1.03
Array Access Efficiency (%)		50.31
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	1.59
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	6.89
	Nb Loops to get 80%	7

Running and analyzing kernel0

Source Code

/scratch/vihps/vihps_inst05/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;
```

CQA

← Path 0 / 1 OK →

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

Coverage	30.93 %
Function	project
Source file and lines	kernel.c:104-110
Module	hydro_k0

The loop is defined in /scratch/vihps/vihps_inst05/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. 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.

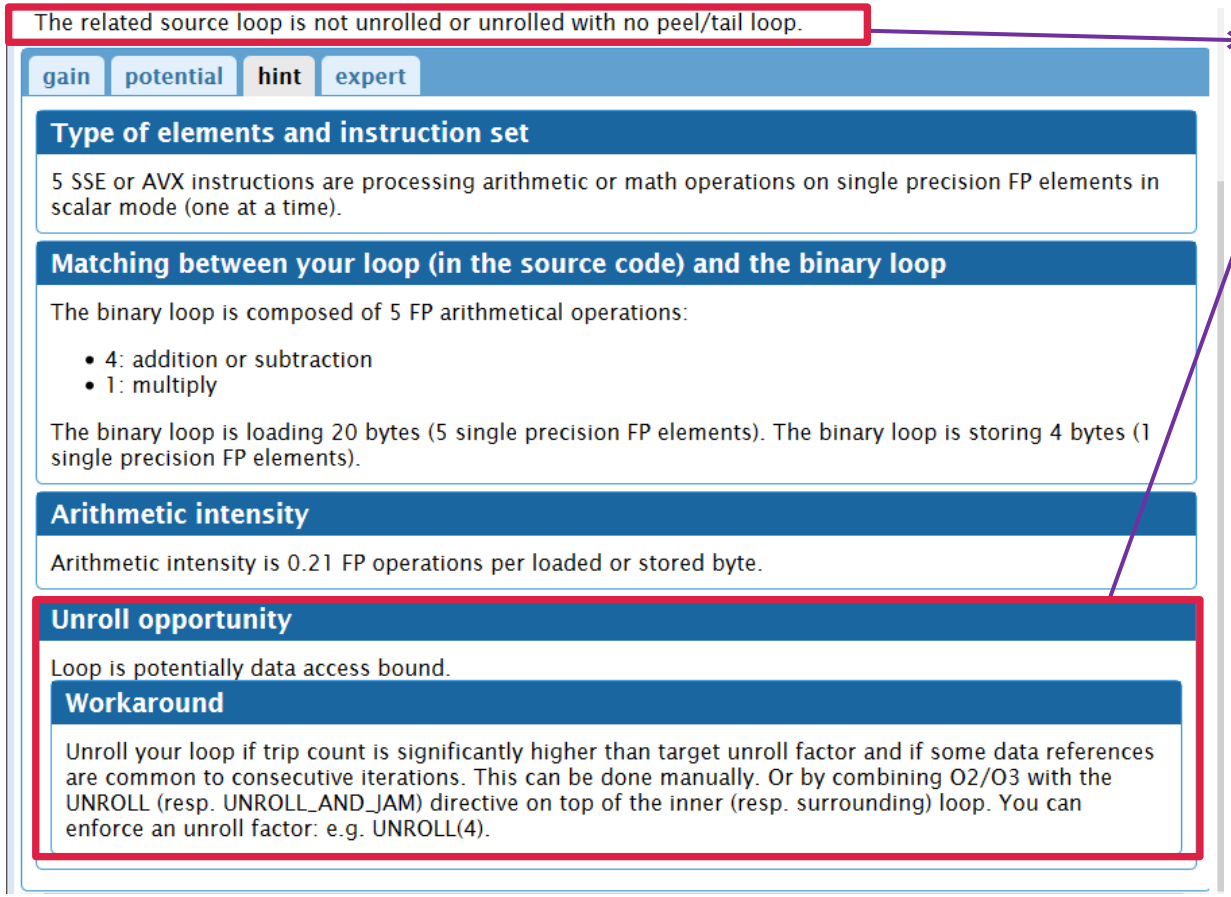
Workaround

- Try another compiler or update/tune your current one:
  - 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] = b.x[i]; (fast, stride 1)

Execution units bottlenecks

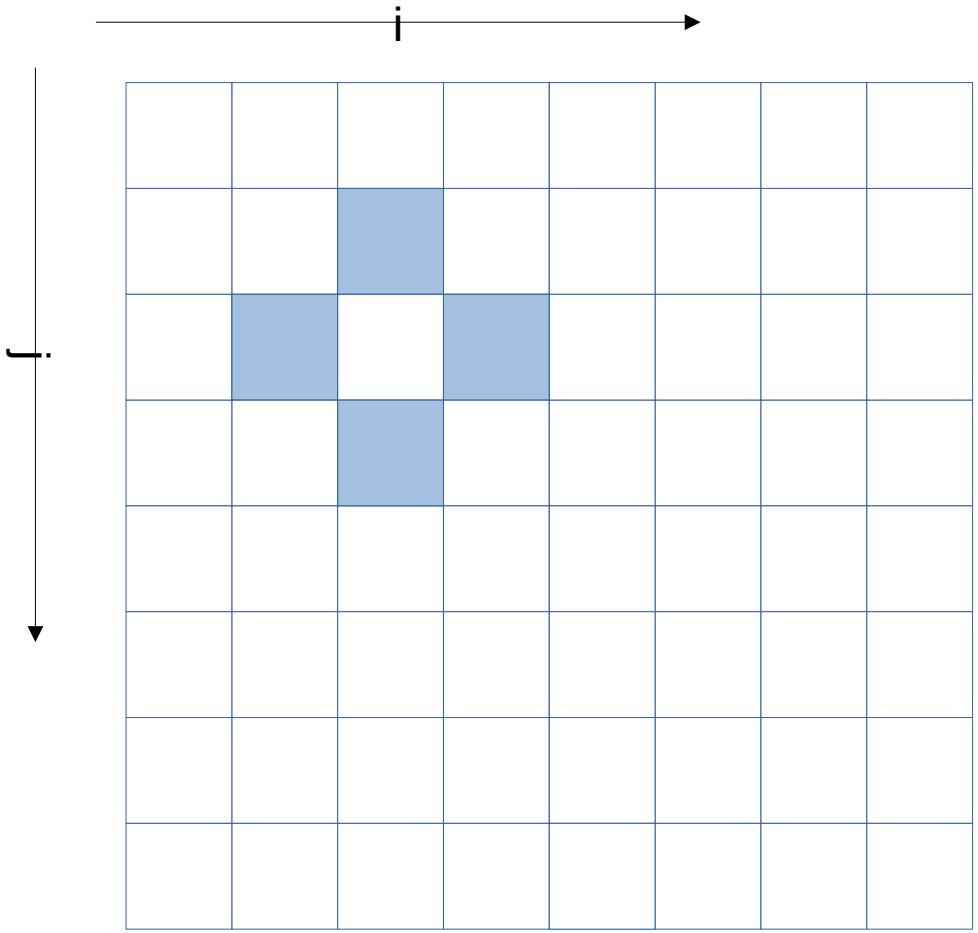
Found no such bottlenecks but see expert reports for more complex bottlenecks.

CQA output for kernel0



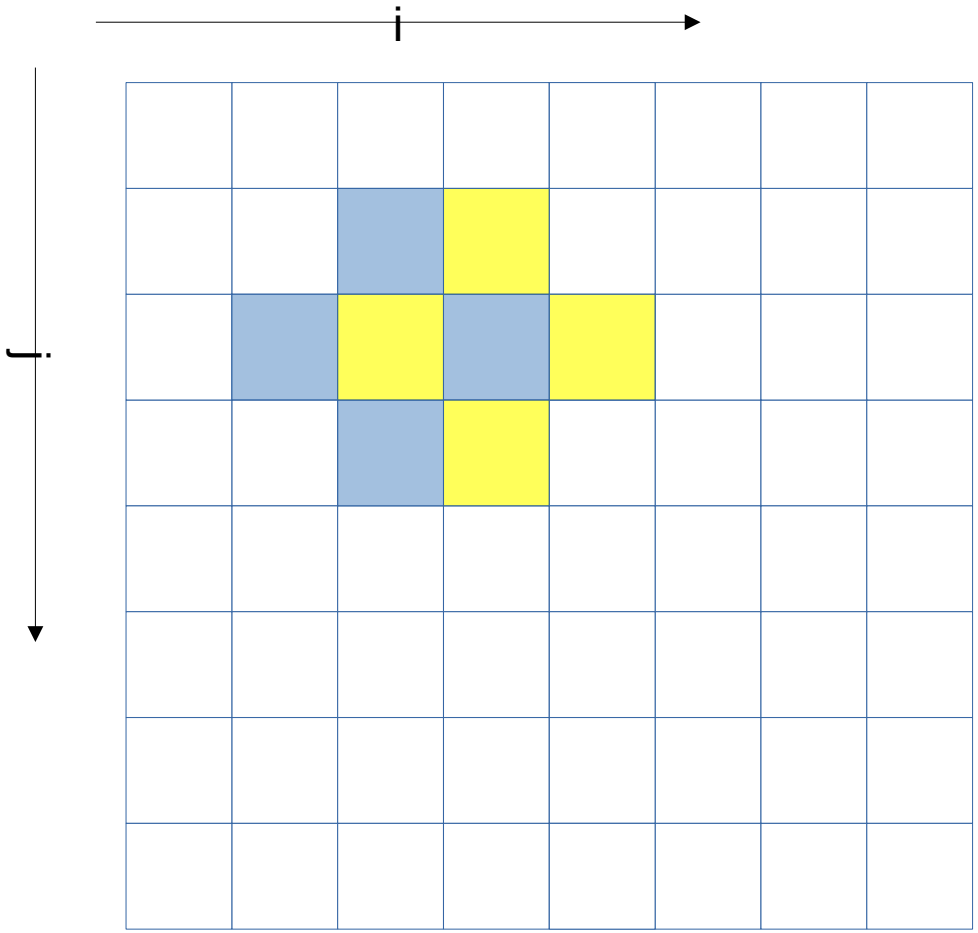
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



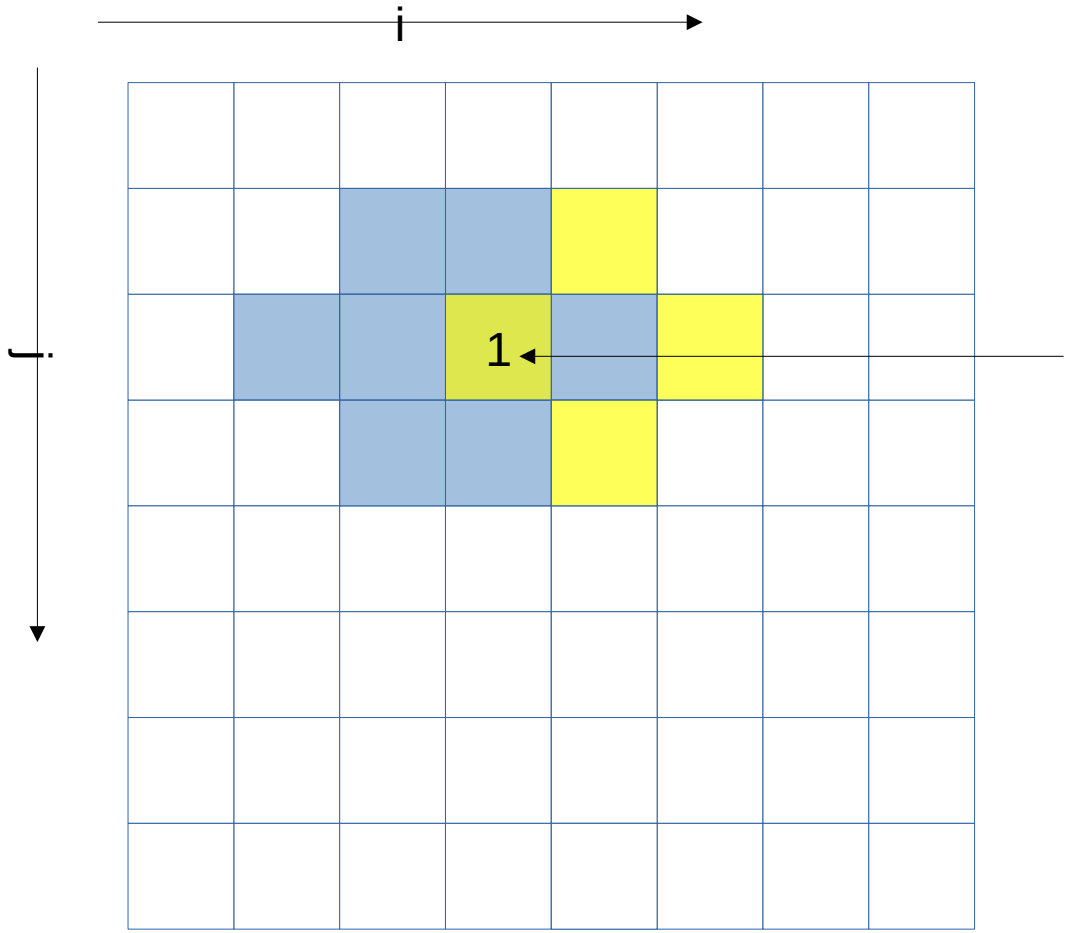
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

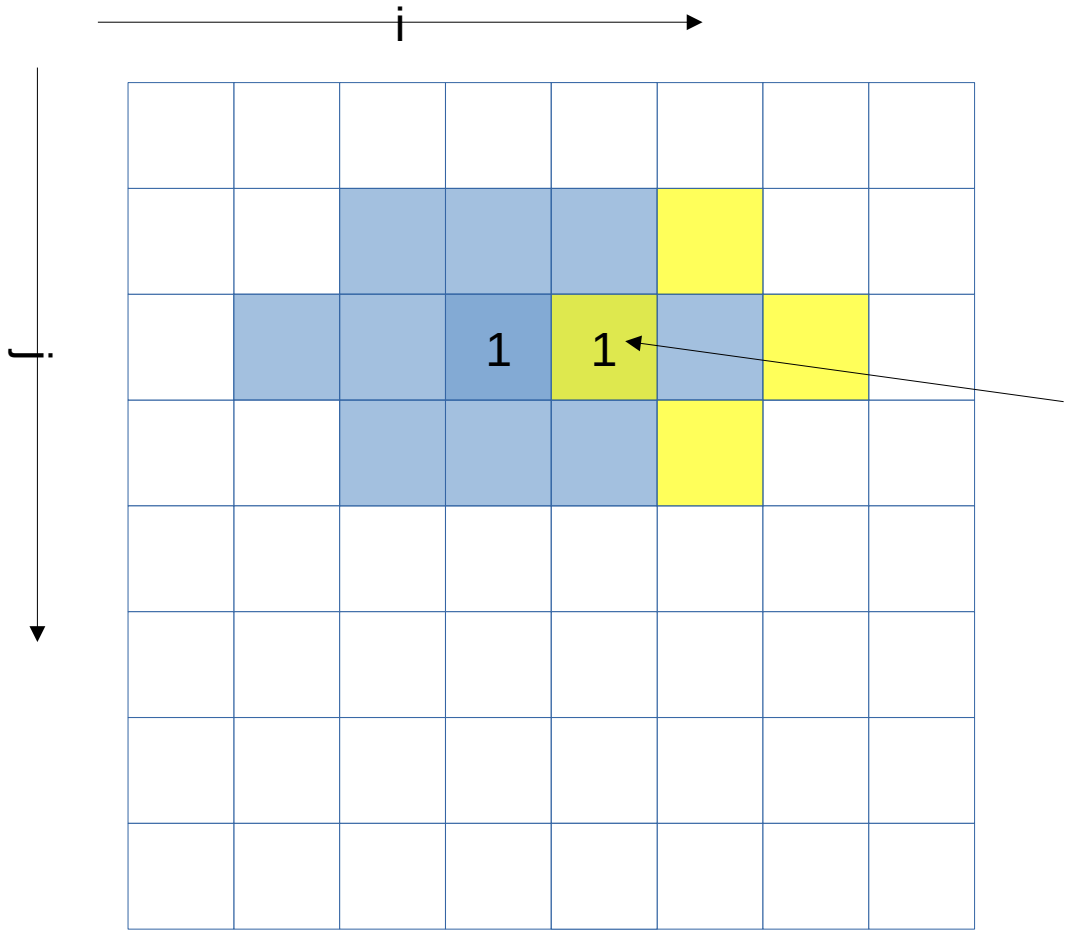
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

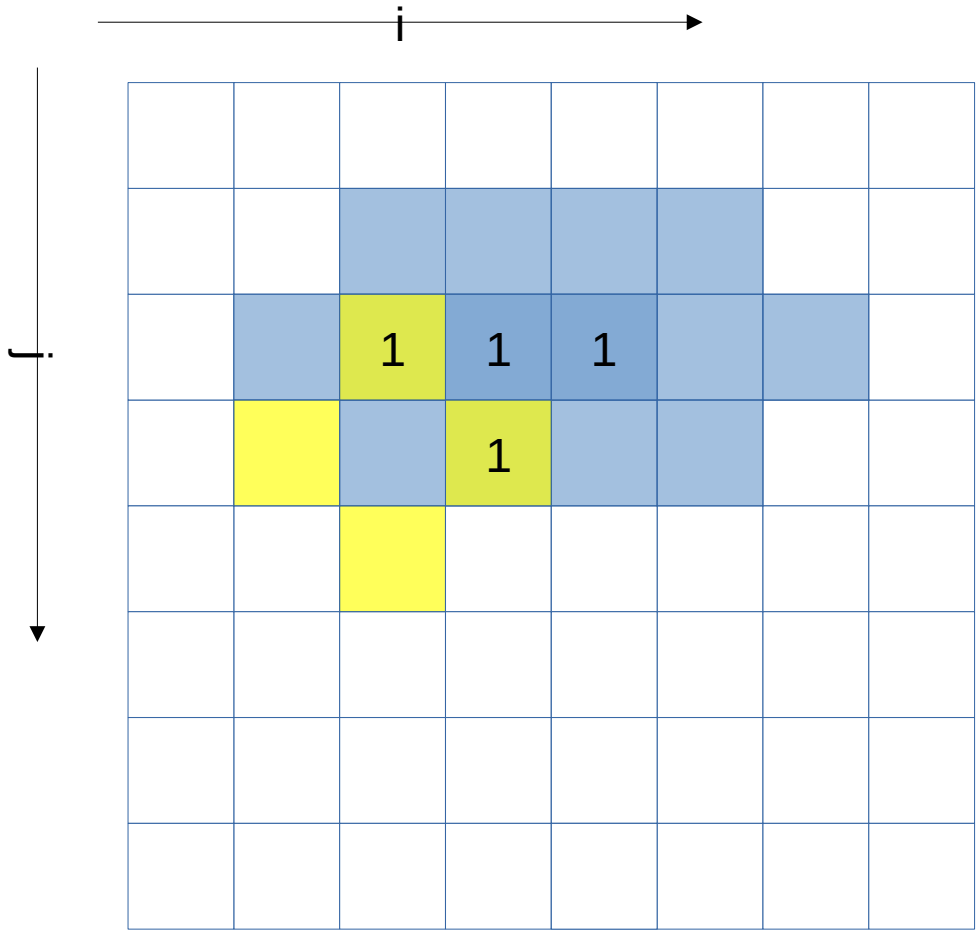
Memory references reuse : 4x4 unroll footprint on loads



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

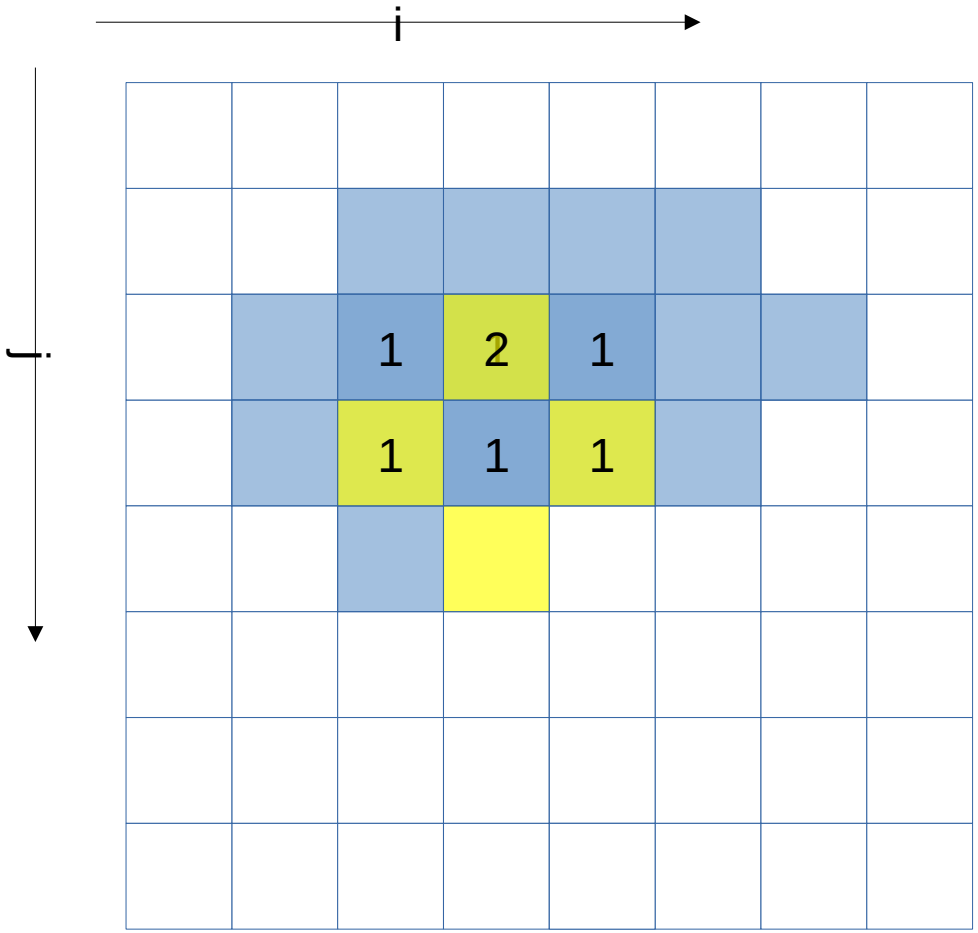


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

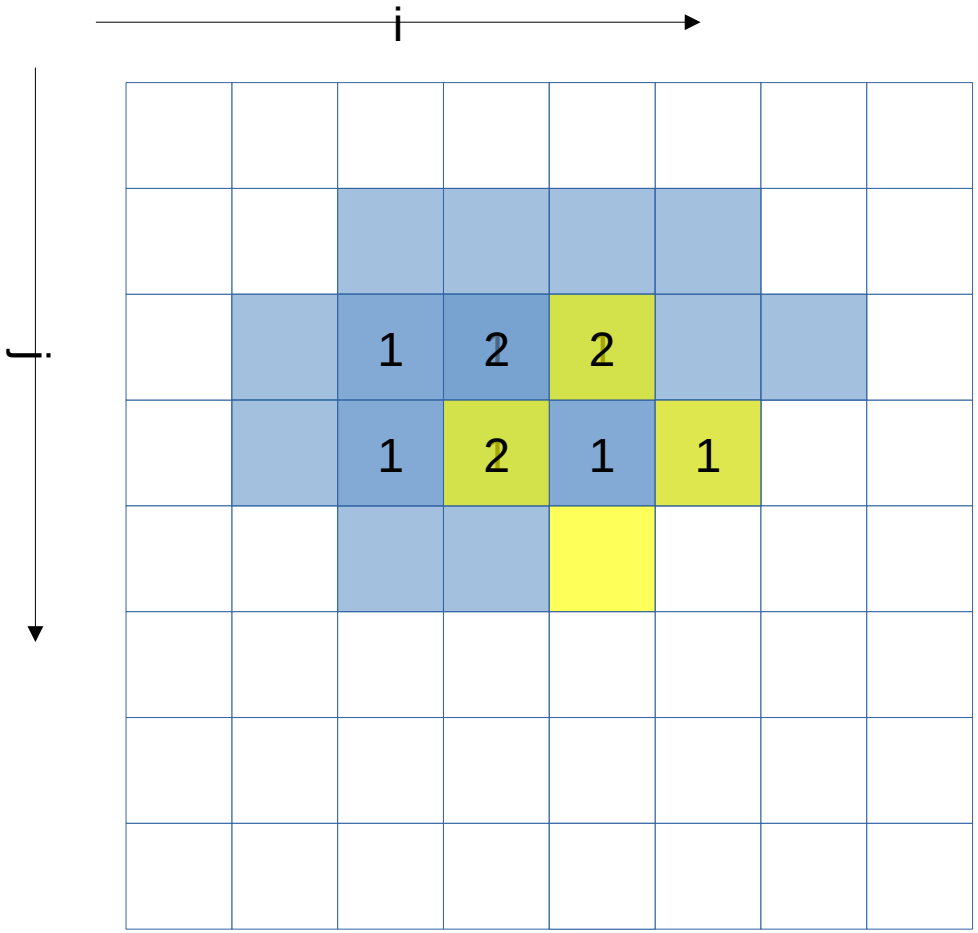


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

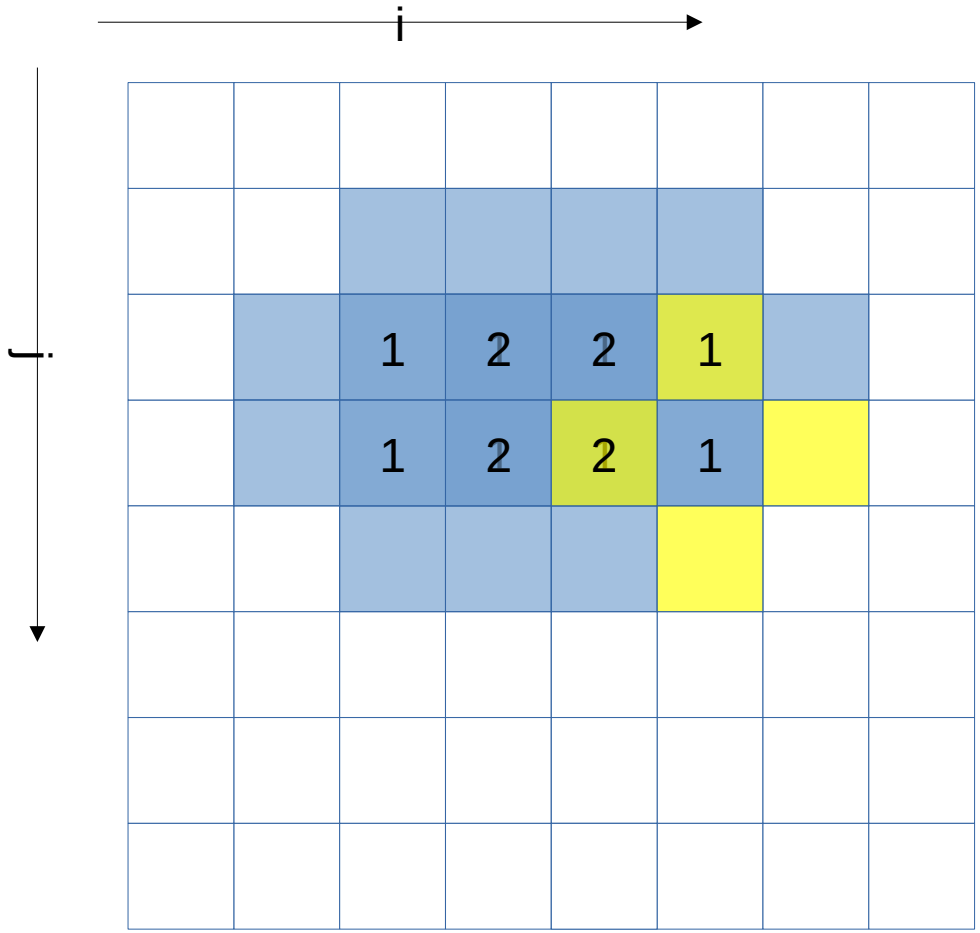


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

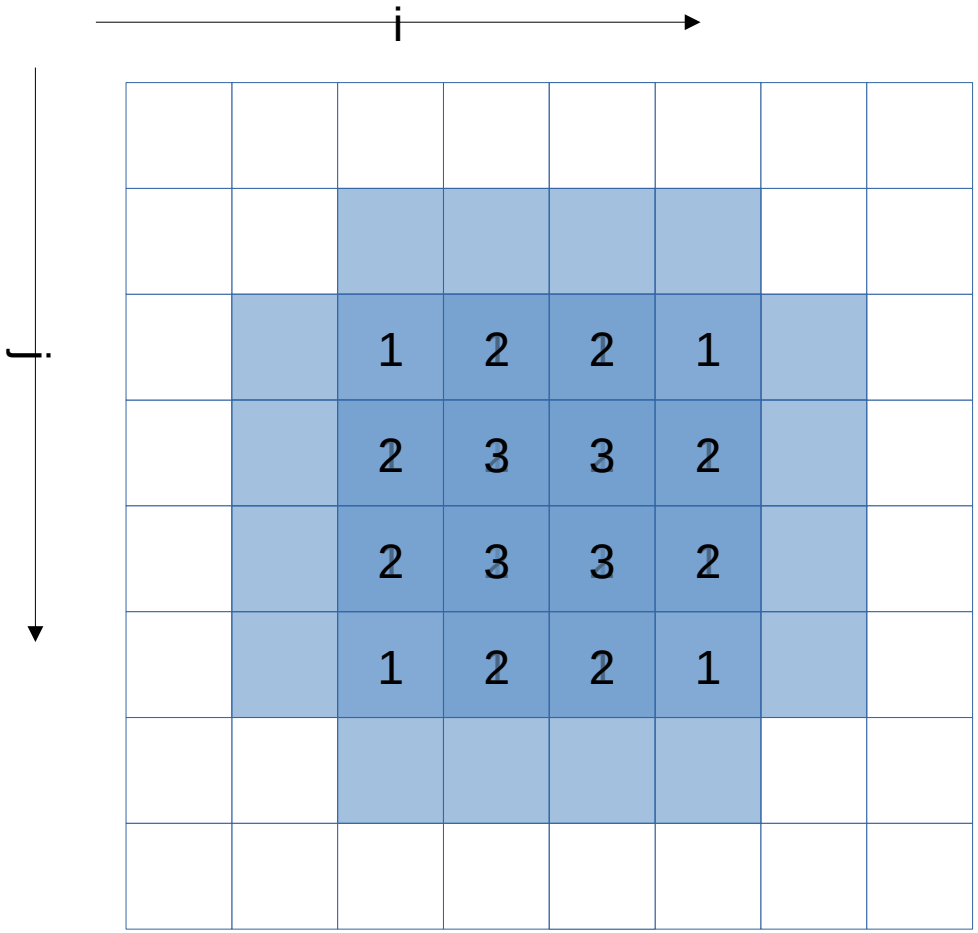


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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 50
Cycles per element for solvers: 521.09
```

Profile with MAQAO (200 repetitions)

```
> maqao oneview -R1 -xp=ov_k1 -c=ov_k1.lua
```

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox goethe_work/MAQAO_HANDSON/hydro/ov_k1/RESULTS/
hydro_k1_one_html/index.html &
```

Global Metrics

Total Time (s)		3.78
Time in loops (%)		99.87
Time in innermost loops (%)		99.79
Time in user code (%)		99.91
Compilation Options		OK
Perfect Flow Complexity		1.10
Array Access Efficiency (%)		47.59
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.07
	Nb Loops to get 80%	4
FP Vectorised	Potential Speedup	1.36
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	12.69
	Nb Loops to get 80%	11

Running and analyzing kernel1

Source Code

/scratch/vihps/vihps_inst05/MAQAO_HANDSON/hydro/kernel.c: 15 - 176

```
15:   return (i + (grid_size + 2) * j);
[...]
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);
```

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 45.75 to 2.86 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 = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

CQA output for kernel1

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:

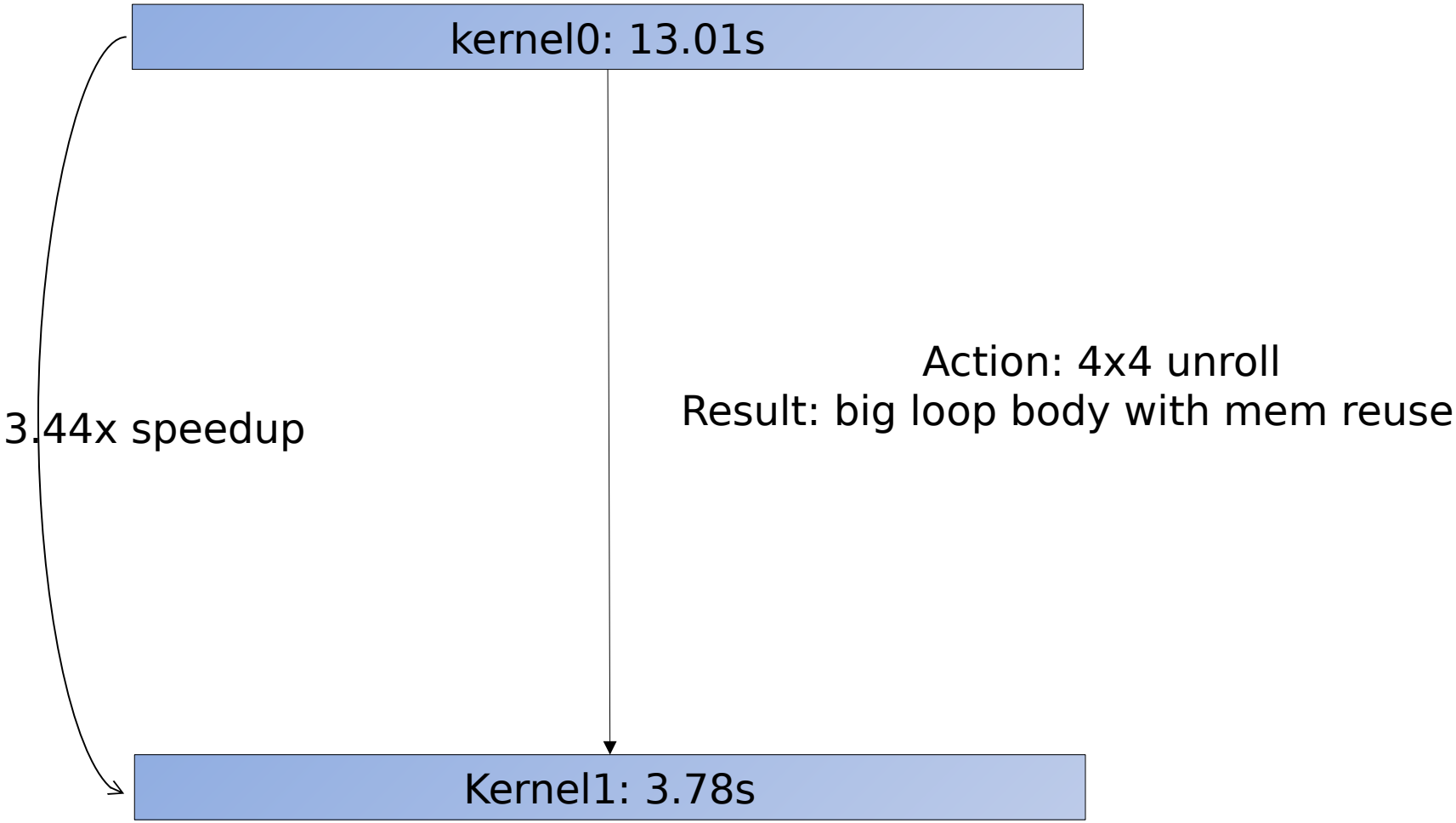
- 64: addition or subtraction
- 32: multiply

The binary loop is loading 276 bytes (69 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

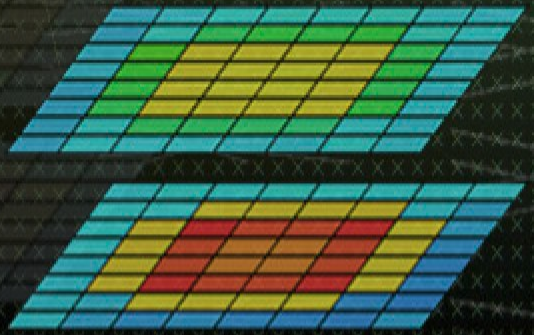
Summary of optimizations and gains



More sample codes

More codes to study with MAQAO in

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



Scalability profiling of lulesh with MAQAO

Salah Ibnamar

Compiling Lulesh (on login node)

Copy Lulesh sources to your working directory

```
> cd $WORK
> tar xvf $TW37/maqao/examples/lulesh2.0.3.tgz
```

Compile Lulesh

```
> cd lulesh
> module purge
> module load comp/intel/2019.5 mpi/intel/2019.5
> make
```

(Optional) To execute a sample run of Lulesh:

```
> less job_lulesh.sbatch
> sbatch job_lulesh.sbatch
```

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.sbatch"
…
batch_command = “sbatch <batch_script>"
…
number_processes = 1
…
number_nodes = 1
…
mpi_command = “mpirun –n <number_processes>"
…
omp_num_threads = 1
…
multiruns_params = {
 {nb_processes = 1, nb_threads = 10, number_nodes = 1},
 {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 = 10, 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.sbatch .
> less job_lulesh_maqao.sbatch
```

```
...
#SBATCH --nodes=2<number_nodes>
...
export OMP_NUM_THREADS=10<omp_num_threads>
...
mpirun -n ... $EXE
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on lulesh (scalability mode)

Launch ONE View (execution will be longer!)

```
> module use $TW37/modulefiles
> module load maqao
> maqao oneview -R1 --with-scalability=on \
-c=config_maqao_lulesh.lua -xp=maqao_lulesh
```

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

```
> firefox
goethe_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>@goethe.hhlr-gu.de: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**