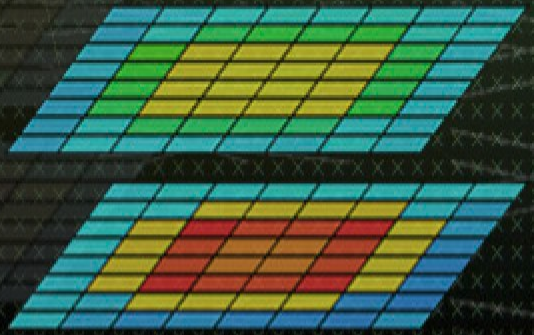


MAQAO
Hands-on exercises

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

Setup (reminder)

Login to the cluster (recommended: use ProxyCommand in your .ssh/config)

```
> ssh <username>@cshpc.rrze.fau.de
> ssh meggie
```

Copy handson material to your workspace directory

```
> export TW38=~g22c0000/VIHPS # hint: copy in ~/.bash_profile
> cd $WORK # for MAQAO handsons, may be better than $FASTTMP
> tar xvf $TW38/maqao/MAQAO_HANDSON.tgz
```

Load MAQAO environment

```
> module use $TW38/modulefiles
> module load maqao
```

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

Extract the NPB directory provided with MAQAO handsons

```
> cd $WORK
> tar xf $TW38/maqao/NPB3.4-MZ-MPI.tgz
```

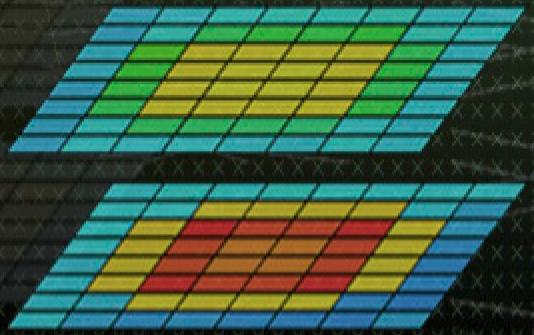
Load Intel compiler and environment

```
> module load intel64/19.0up05 intelmpi/2019up05-intel
```

Compile and run

```
> cd NPB3.4-MZ-MPI
> 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/bt_OV_sbatch.lua .
> less bt_OV_sbatch.lua
```

```
binary = "bt-mz.C.x"
...
batch_script = "bt_maqao.sbatch"
batch_command = "sbatch <batch_script>“
...
number_processes = 4
number_tasks_nodes = 2
omp_num_threads = 10
...
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 --ntasks-per-node=2<number_tasks_nodes>
#SBATCH --cpus-per-task=10<omp_num_threads>
...
export OMP_NUM_THREADS=10<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=bt_OV_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/bt_OV_interact.lua .
> less bt_OV_interact.lua
```

```
binary = "bt-mz.C.x"
...
number_processes = 4
number_tasks_nodes = 2
...
omp_num_threads = 10
...
mpi_command = "mpirun -n <number_processes>"
```

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

Request interactive session with 2 nodes

```
> srun --reservation=VIHPS1 --mpi=pmi2 --nodes=2 --pty bash
```

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin
> maqao oneview -R1 --config=bt_OV_interact.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 meggie_work
> sshfs <user>@cshpc.hhlr-gu.de:/home/woody/k_g22c/<user> \
meggie_work
> firefox meggie_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>@cshpc.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
```

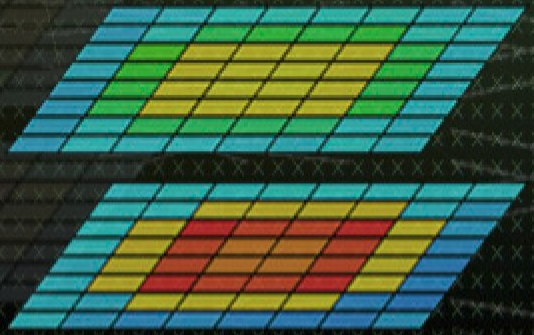
- scp is slow to copy directories (especially when containing many small files), copy a .tgz archive of the directory

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/bt_OV_scal.lua .
> less bt_OV_scal.lua
```

```
binary = "./bt-mz.C.x"
…
run_command = "<binary>"
…
batch_script = "bt_maqao.sbatch"
…
batch_command = “sbatch <batch_script>"
…
number_processes = 4
…
number_tasks_nodes = 4
…
omp_num_threads = 1
…
mpi_command = “mpirun –n <number_processes>"
…
multiruns_params = {
 {number_processes = 1, omp_num_threads = 10, number_nodes = 1, number_tasks_nodes = 1},
 {number_processes = 4, omp_num_threads =  1, number_nodes = 2, number_tasks_nodes = 2},
 {number_processes = 4, omp_num_threads = 10, number_nodes = 2, number_tasks_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 \
-c=bt_OV_scal.lua -xp=ov_scal
```

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

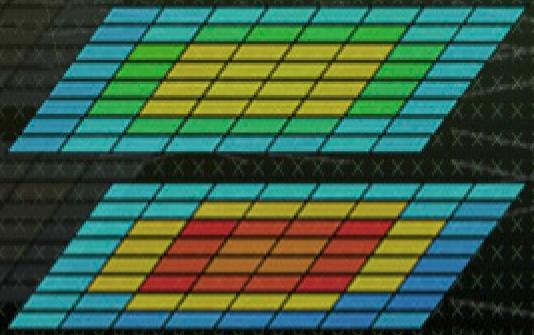
```
> firefox meggie_work/NPB3.4-MZ-MPI/bin/ov_scal/RESULTS/bt-
mz.C.x_one_html/index.html
```

OR

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

```
> scp <login>@meggie.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

Compile with GNU compiler and request interactive session

Go to the handson directory

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

Compile all variants (on login node, as requested)

```
> module load gcc/8.1.0
> make all
```

Load MAQAO environment

```
> module use $TW38/modulefiles
> module load maqao
```

Request an interactive session

```
> srun --reservation=VIHPS1 --pty bash
```

Analysing matrix multiply with MAQAO

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

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> ./matmul_orig 400 300
cycles per FMA: 2.79
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 -xp=ov_orig -- ./matmul_orig 400 300
```

OR, using a configuration script:

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

Viewing results (HTML)

On your local machine (sshfs):

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

Global Metrics

Total Time (s)		20.69
Profiled Time (s)		20.69
Time in loops (%)		100
Time in innermost loops (%)		100
Time in user code (%)		100
Compilation Options		matmul_orig: -march=(target) is missing. -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.34
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	2.13
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	8.00
	Nb Loops to get 80%	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 0.37 cycles (8.00x speedup).

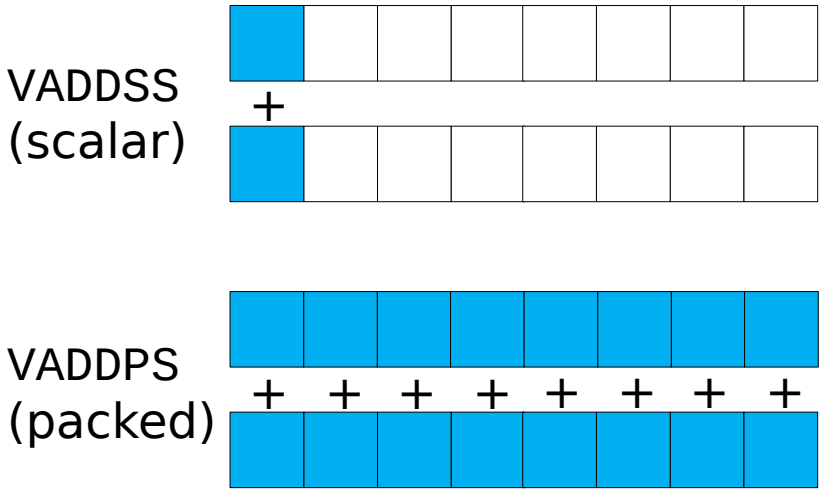
Details

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

Workaround

- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

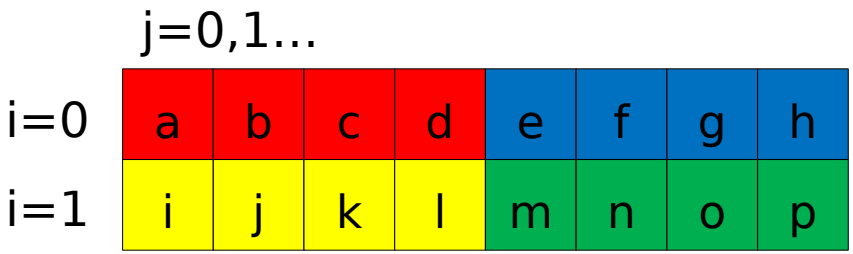
Vectorization (summing elements):



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

Impact of loop permutation on data access

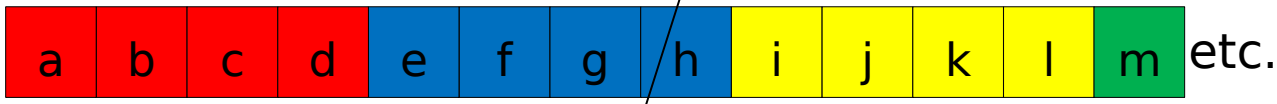
Logical mapping



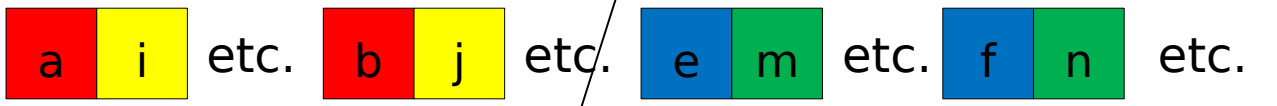
Efficient vectorization + prefetching

Physical mapping

(C stor. order: row-major)




```
for (j=0; j<n; j++)
  for (i=0; i<n; i++)
    f(a[i][j]);
```




```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    f(a[i][j]);
```

Removing inter-iteration dependences and getting stride 1 by permuting loops on j and k

```
void kernel1 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
  int i, j, k;

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++)
      c[i][j] = 0.0f;

    for (k=0; k<n; k++)
      for (j=0; j<n; j++)
        c[i][j] += a[i][k] * b[k][j];
  }
}
```

Analyse matrix multiply with permuted loops

Run permuted loops version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> ./matmul_perm 400 300
cycles per FMA: 0.49
```

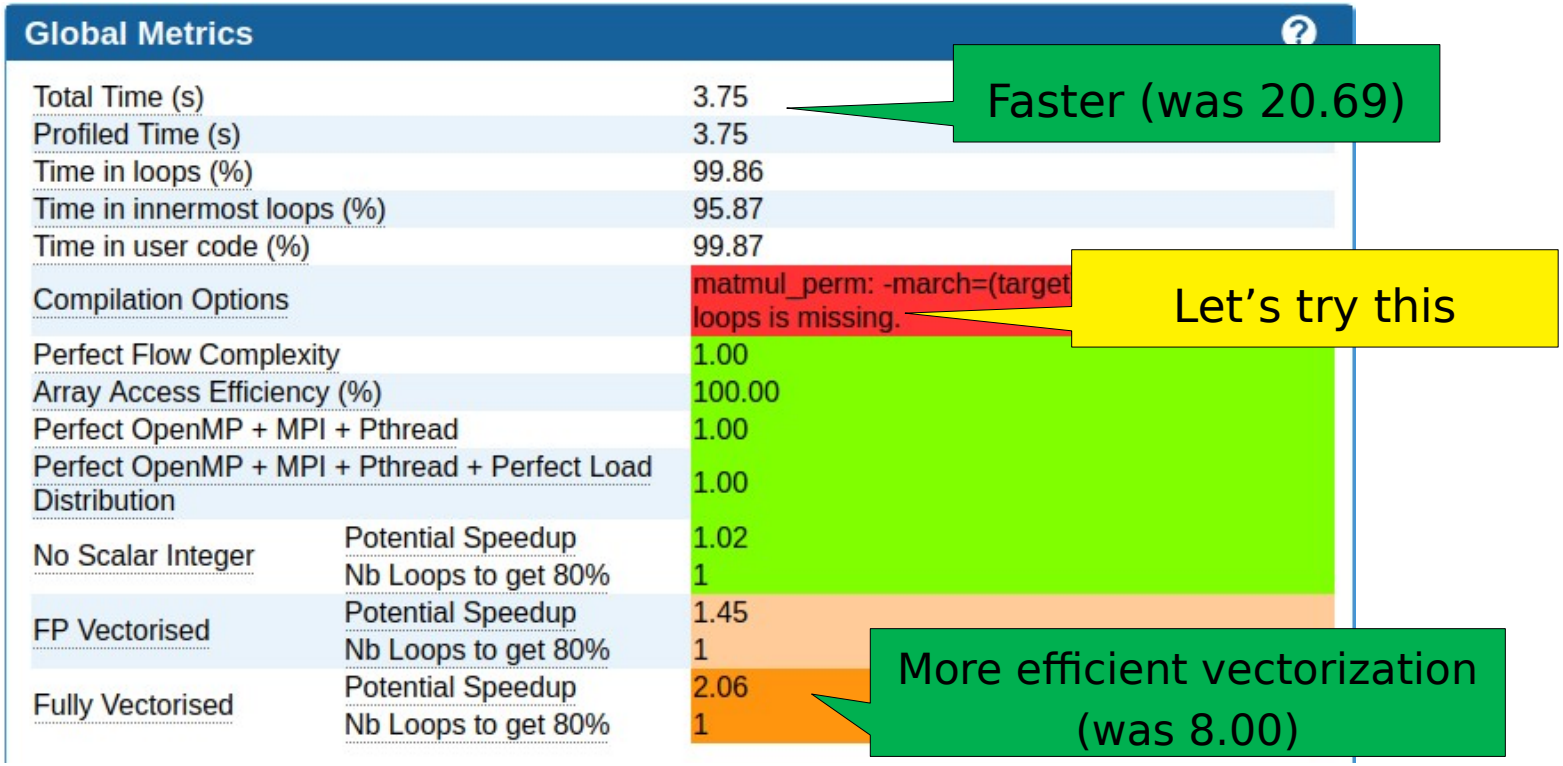
Analyse matrix multiply with ONE View

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

Viewing results (HTML)

On your local machine (sshfs):

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



CQA output after loop permutation

Vectorization

Your loop is vectorized, but using only 128 out of 256 bits (SSE/AVX-128 instructions on AVX/AVX2 processors). By fully vectorizing your loop, you can lower the cost of an iteration from 1.75 to 0.87 cycles (2.00x 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.

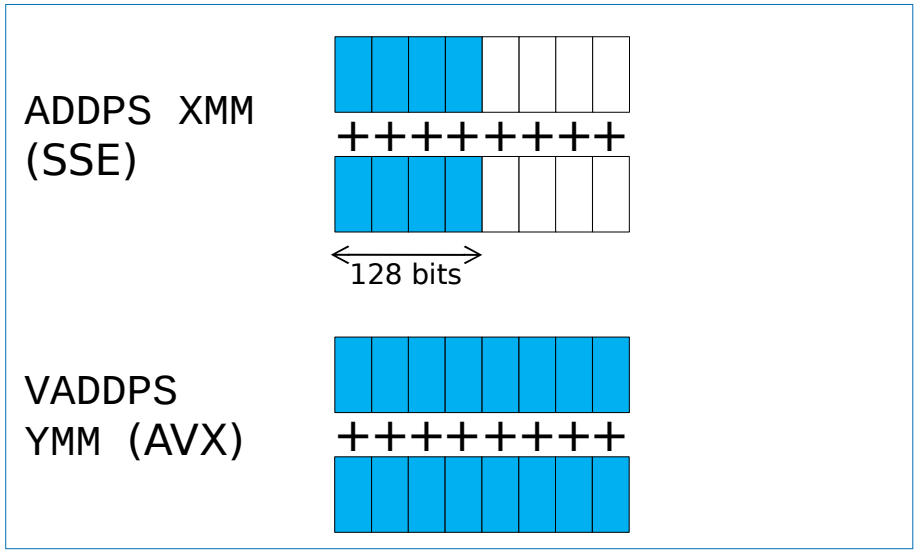
Workaround

- Recompile with march=broadwell. CQA target is Core_i7X_Xeon_E5_v4 (Intel Xeon processor E5 v4 Family microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition) but spe
- Use vector aligned instructions:
  1. align your arrays on 32 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 32, size); }.
  2. inform your compiler that your arrays are vector aligned: if array 'foo' is 32 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 32) 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 AVX256 ones (SIMD 256 bits)
  - => 50% 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 advanced compiler flags (microarchitecture-specialization and loop-unrolling)

Run compiler-optimized version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> ./matmul_flags 400 300
cycles per FMA: 0.39
```

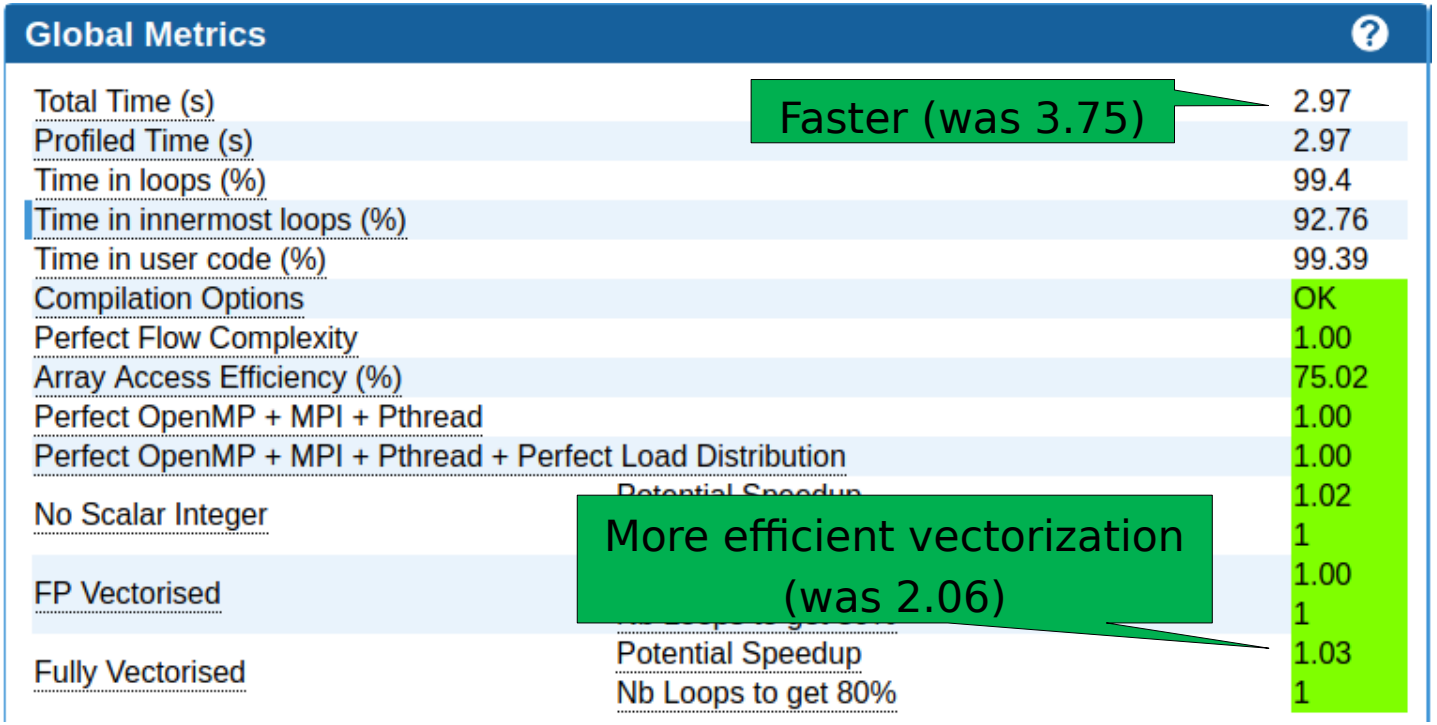
Analyse matrix multiply with ONE View

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

Viewing results (HTML)

On your local machine (sshfs):

```
> firefox meggie_work/MAQAO_HANDSON/matmul/ov_flags/RESULTS/
matmul_flags_one_html/index.html &
```



CQA output with advanced compiler flags (microarchitecture-specialization and loop-unrolling)

gain | potential | hint | expert

Vectorization

Your loop is fully vectorized, using full register length.

Details

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

gain | potential | hint | expert

FMA

Detected 8 FMA (fused multiply-add) operations.

Vector unaligned load/store instructions

Detected 16 optimal vector unaligned load/store instructions.

Details

- VMOVUPS: 16 occurrences

Workaround

Use vector aligned instructions:

Let’s try this

1. align your arrays on 32 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 32, size); }.
2. inform your compiler that your arrays are vector aligned: if array 'foo' is 32 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 32) and use it instead of 'foo' in the loop.

Analyse matrix multiply with array alignment

Run aligned version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> ./matmul_align 400 300 # remark: size%8 has to equal 0
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.29
```

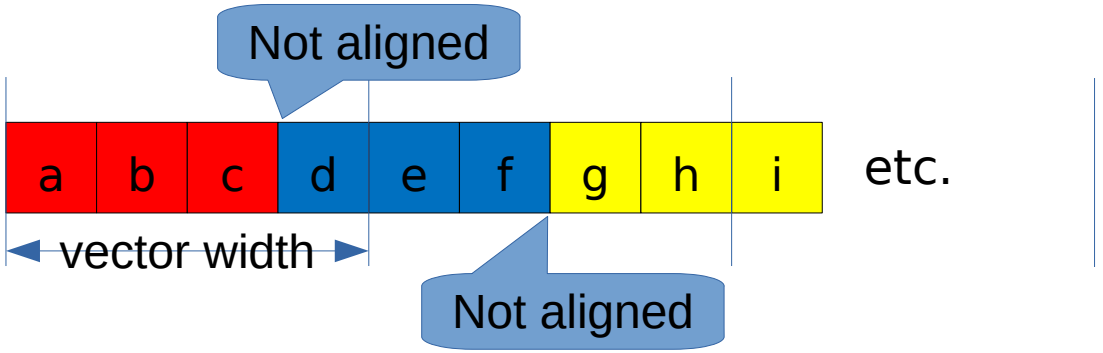
Analyse matrix multiply with ONE View

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

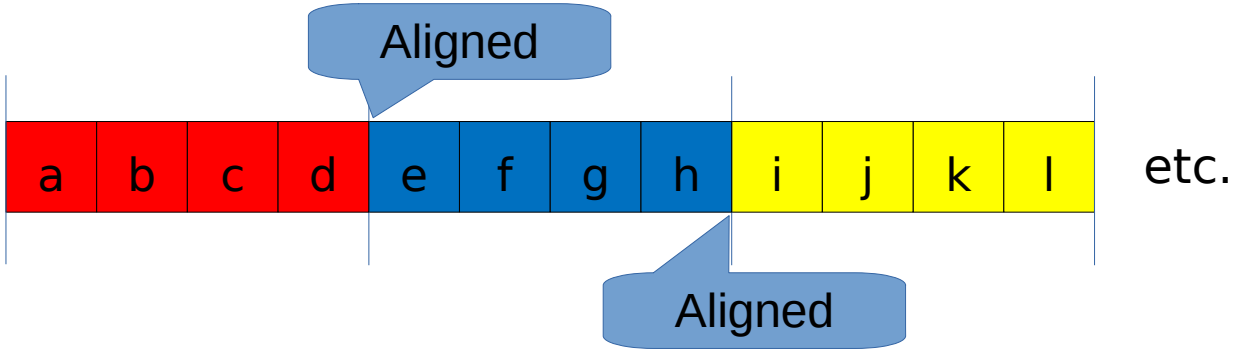
Multidimensional array alignment

Data organized as a 2D array: n lines of 3 columns
Each vector can hold 4 consecutive elements

a[n][3], only 1st element is aligned



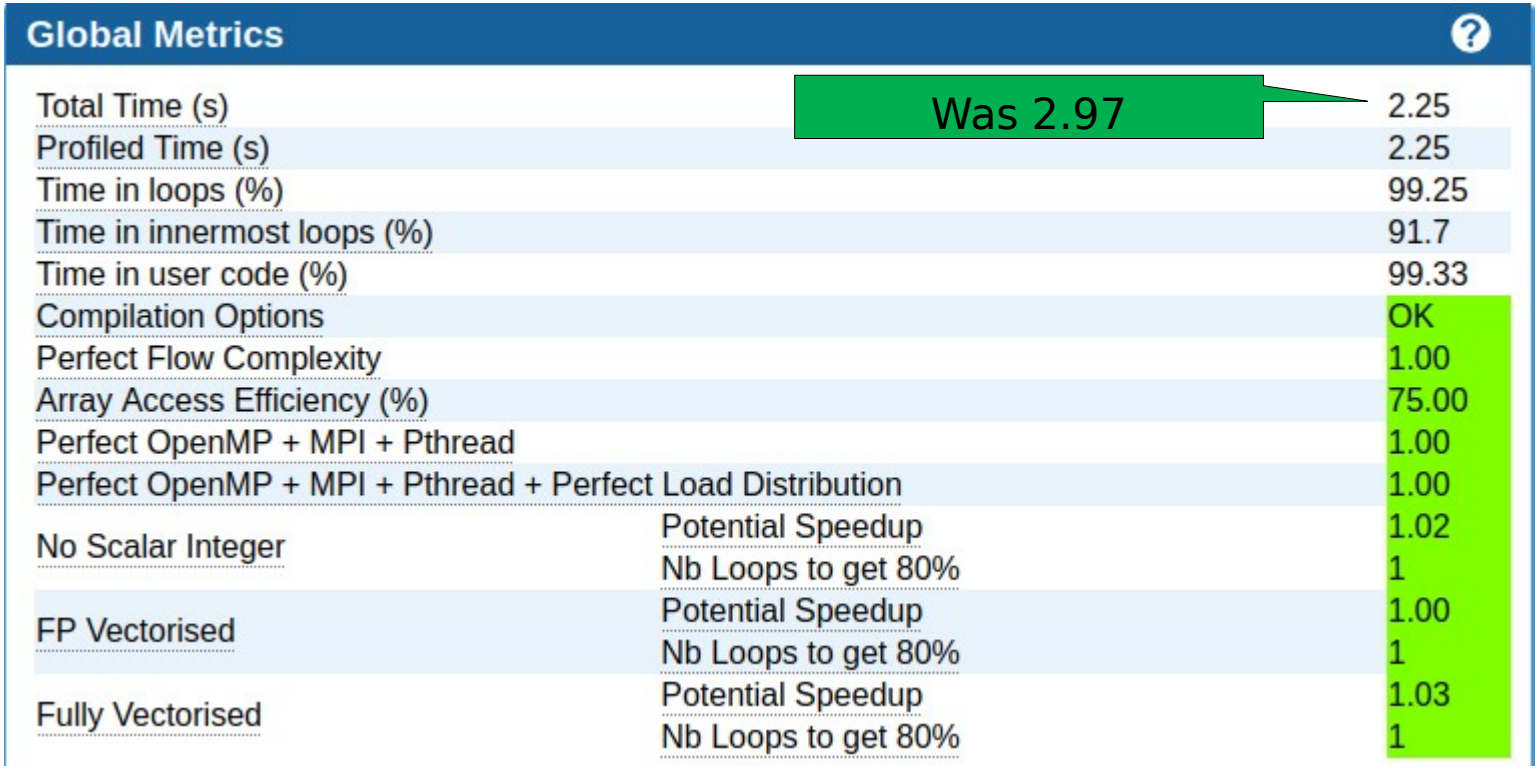
a[n][4], 1st element of each line are aligned



Viewing results (HTML)

On your local machine (sshfs):

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



Global Metrics

Was 2.97

Total Time (s)		2.25
Profiled Time (s)		2.25
Time in loops (%)		99.25
Time in innermost loops (%)		91.7
Time in user code (%)		99.33
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		75.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.03
	Nb Loops to get 80%	1

Using comparison mode

```
> maqao oneview --compare-reports --inputs=ov_orig,ov_align \
-xp=ov_orig_vs_align
```

On your local machine (sshfs):

```
> firefox meggie_work/MAQAO_HANDSON/matmul/ov_orig_vs_align/
index.html &
```

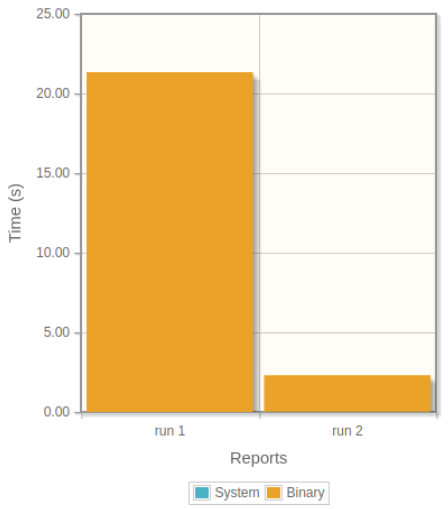
Using comparison mode

Global Metrics

Metric		run 1	run 2
Total Time (s)		21.34	2.31
Profiled Time (s)		21.34	2.31
Time in loops (%)		99.97	99.40
Time in innermost loops (%)		99.88	90.76
Time in user code (%)		99.98	99.44
Compilation Options		matmul_orig: -march=(target) is missing. -funroll-loops is missing.	OK
Perfect Flow Complexity		1.00	1.00
Array Access Efficiency (%)		83.33	83.33
Perfect OpenMP + MPI + Pthread		1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00
No Scalar Integer	Potential Speedup	1.00	1.03
	Nb Loops to get 80%	1	1
FP Vectorised	Potential Speedup	2.13	1.00
	Nb Loops to get 80%	1	1
Fully Vectorised	Potential Speedup	7.98	1.04
	Nb Loops to get 80%	1	1

Application Categorization

Time



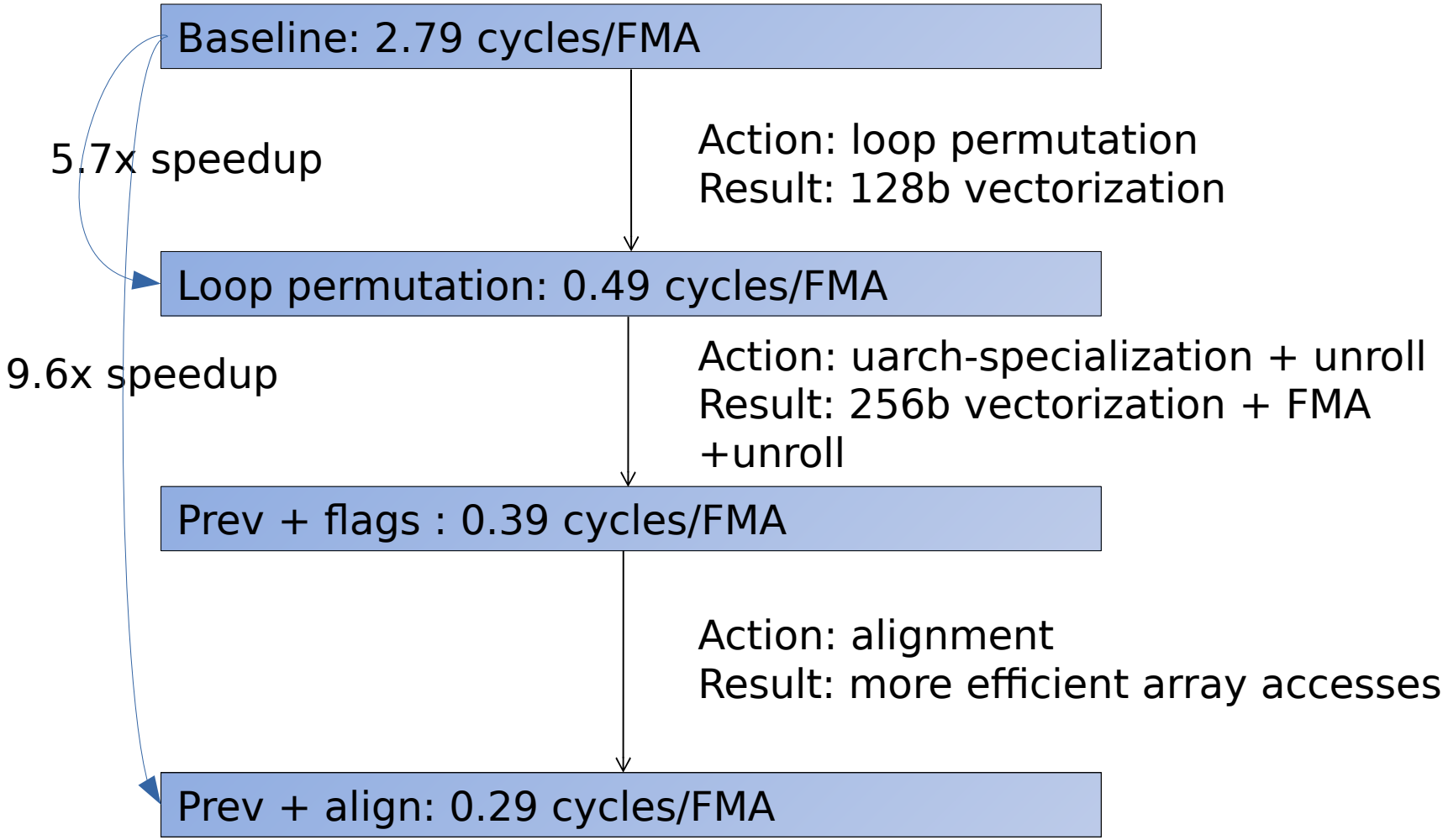
Experiment Summaries

	run 1	run 2
Application	./matmul_orig	./matmul_align
Timestamp		
Experiment Type	Sequential	Sequential
Machine	m0102.rrze.uni-erlangen.de	m0102.rrze.uni-erlangen.de
Architecture	x86_64	x86_64
Micro Architecture	BROADWELL	BROADWELL
Model Name	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
Cache Size	25600 KB	25600 KB
Number of Cores	10	10
Maximal Frequency	2.201 GHz	2.201 GHz
OS Version	Linux 3.10.0-1160.6.1.el7.x86_64 #1 SMP Tue Nov 17 13:59:11 UTC 2020	Linux 3.10.0-1160.6.1.el7.x86_64 #1 SMP Tue Nov 17 13:59:11 UTC 2020
Architecture used during static analysis	x86_64	x86_64
Micro Architecture used during static analysis	BROADWELL	BROADWELL
Compilation Options	matmul_orig: GNU 8.1.0 -mtune=generic -march=x86-64 -g -O3 -fno-omit-frame-pointer	matmul_align: GNU 8.1.0 -march=broadwell -g -O3 -fno-omit-frame-pointer -funroll-loops
Number of processes observed	1	1
Number of threads observed	1	1
MAQAO version	2.12.7	2.12.7
MAQAO build	4dbfb0c5ad8da9d9015ba1c31ba75b8344 2c7c28::20210224-171504	4dbfb0c5ad8da9d9015ba1c31ba75b8344 2c7c28::20210224-171504

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	99.98	99.4	21.34	2.3	1	1	0.00	0.00
__GI_memset	libc-2.17.so	0	0.3		0.01		1		0.00
__random	libc-2.17.so	0.02	0.13	0	0	1	1	0.00	0.00
__random_r	libc-2.17.so	0	0.13		0		1		0.00
Unknown function	binary	0	0.04		0		1		0.00

Summary of optimizations and gains



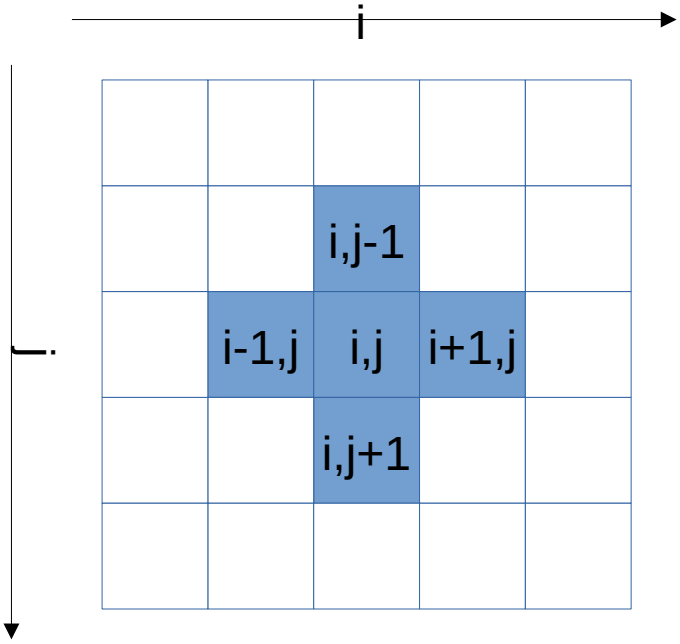
Hydro code

```
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

Go back to login node (if necessary)

```
> exit
```

Switch to the hydro handson folder

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

Load MAQAO (if no more loaded)

```
> module use $TW38/modulefiles
> module load maqao
```

Load Intel 19 compiler

```
> module load intel64/19.0up05
```

Compile

```
> make
```

Hydro example: interactive session

Request interactive session

```
> srun --reservation=VIHPS1 --pty bash
```

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)

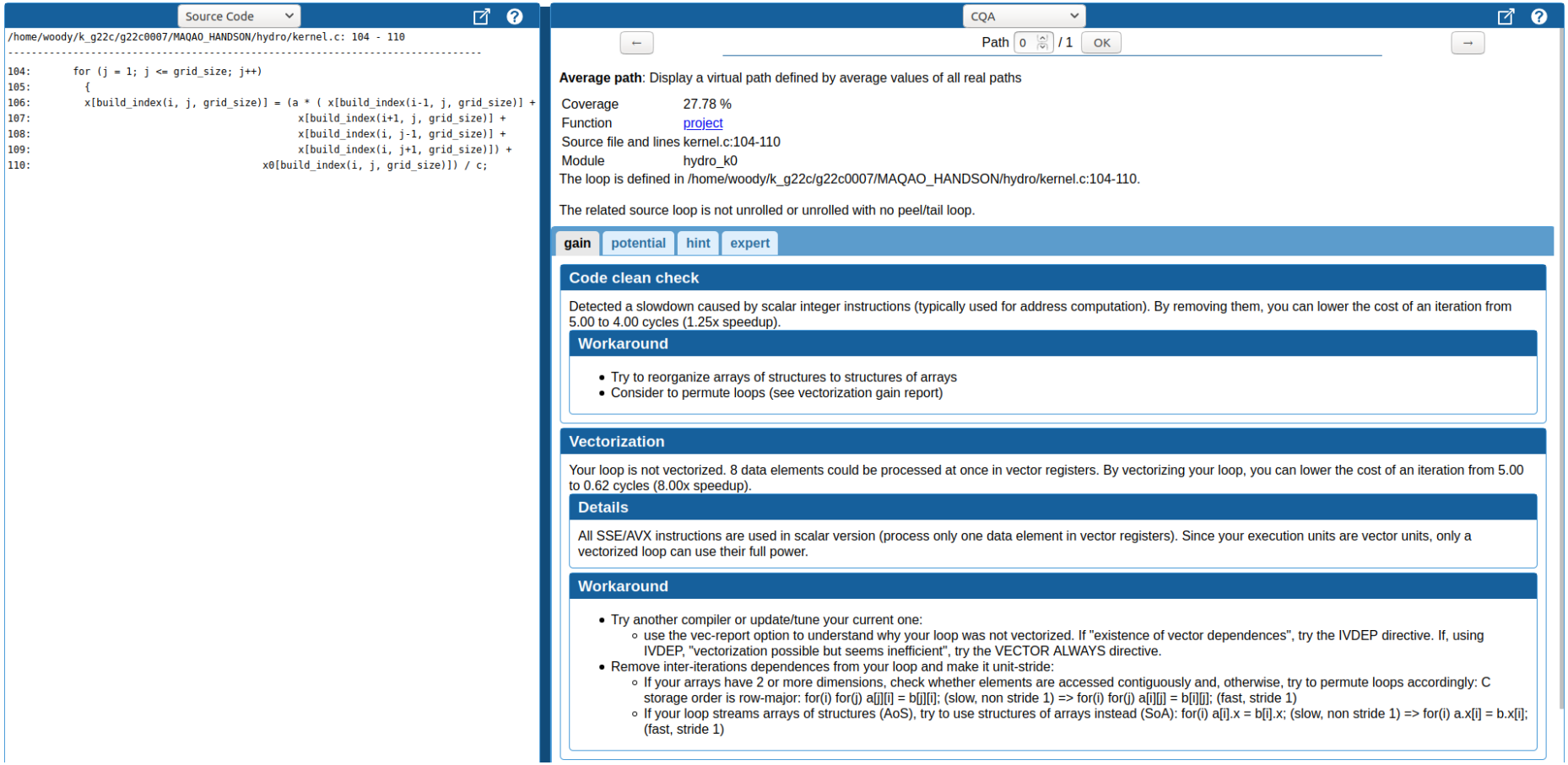
On your local machine (sshfs):

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

Global Metrics

Total Time (s)		15.06
Profiled Time (s)		15.06
Time in loops (%)		99.94
Time in innermost loops (%)		99.86
Time in user code (%)		99.97
Compilation Options		OK
Perfect Flow Complexity		1.02
Array Access Efficiency (%)		50.55
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.06
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.64
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	6.39
	Nb Loops to get 80%	7

Running and analyzing kernel0



CQA output for kernel0

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

gain | potential | **hint** | expert

Slow data structures access

Detected data structures (typically arrays) that cannot be efficiently read/written

Details

- Constant unknown stride: 4 occurrence(s)

Non-unit stride (uncontiguous) accesses are not efficiently using data caches

Workaround

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

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

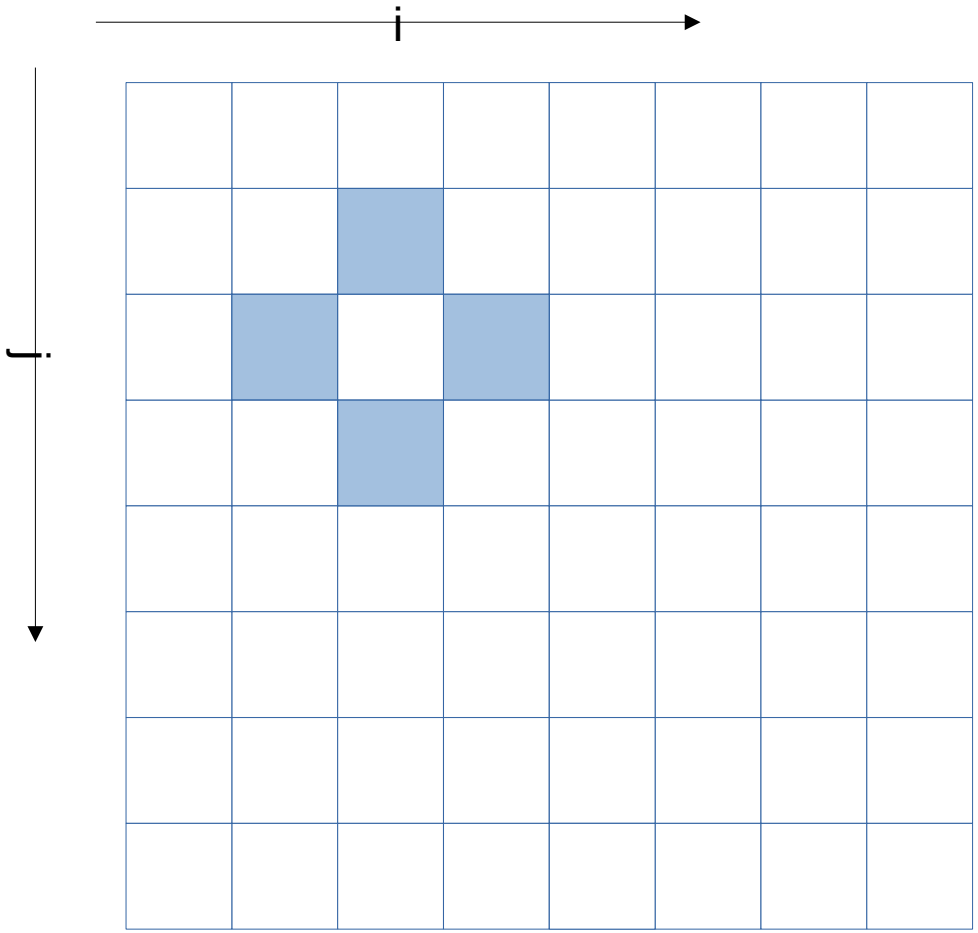
Loop is potentially data access bound.

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).

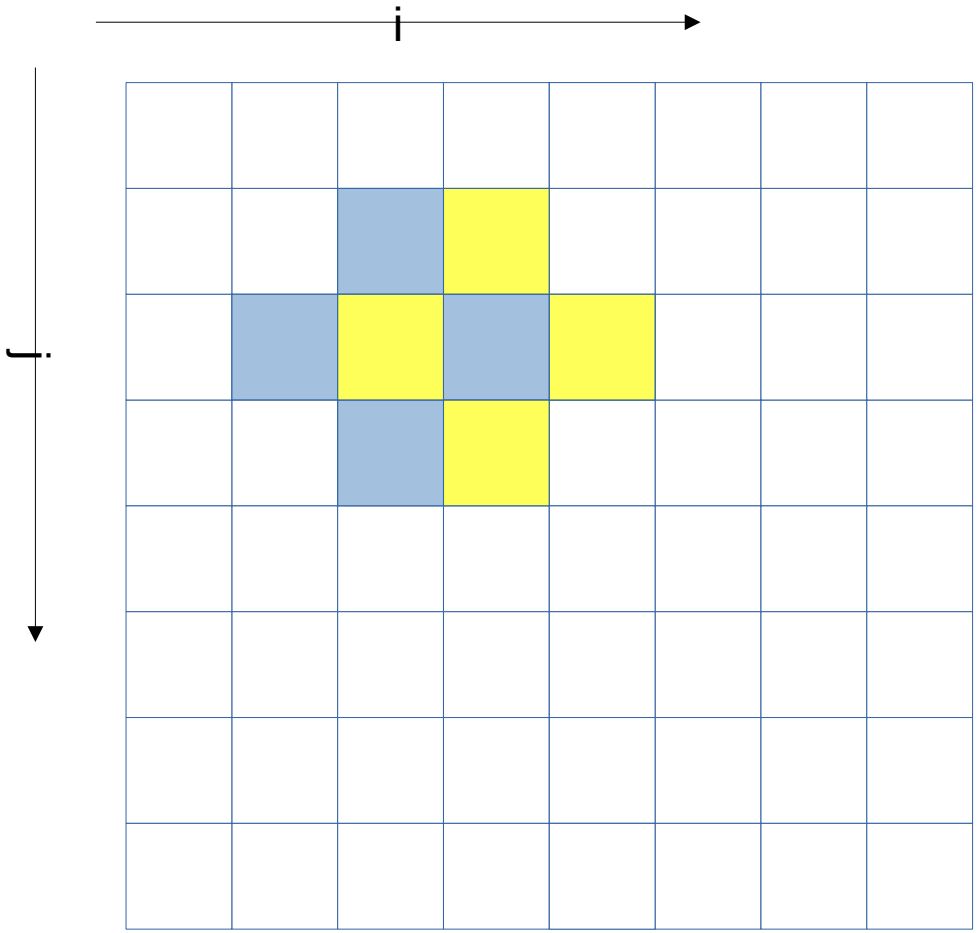
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



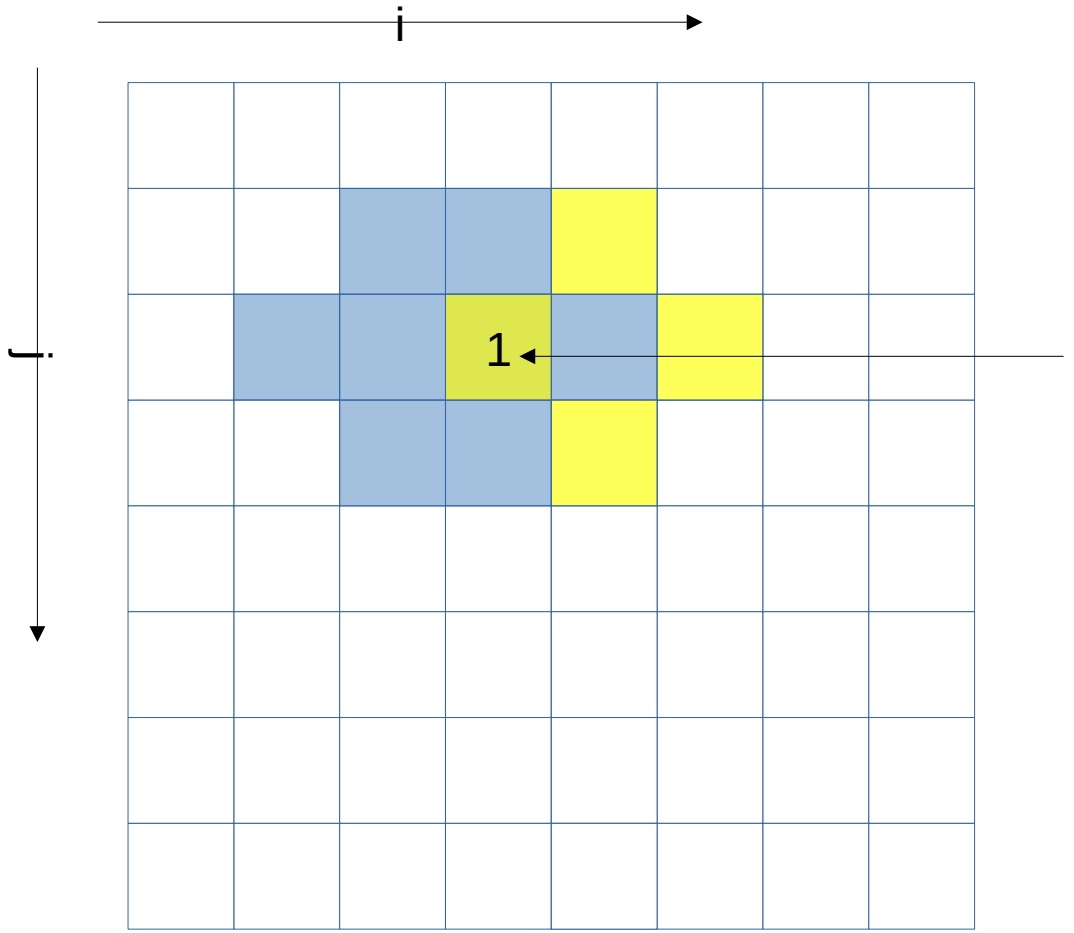
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

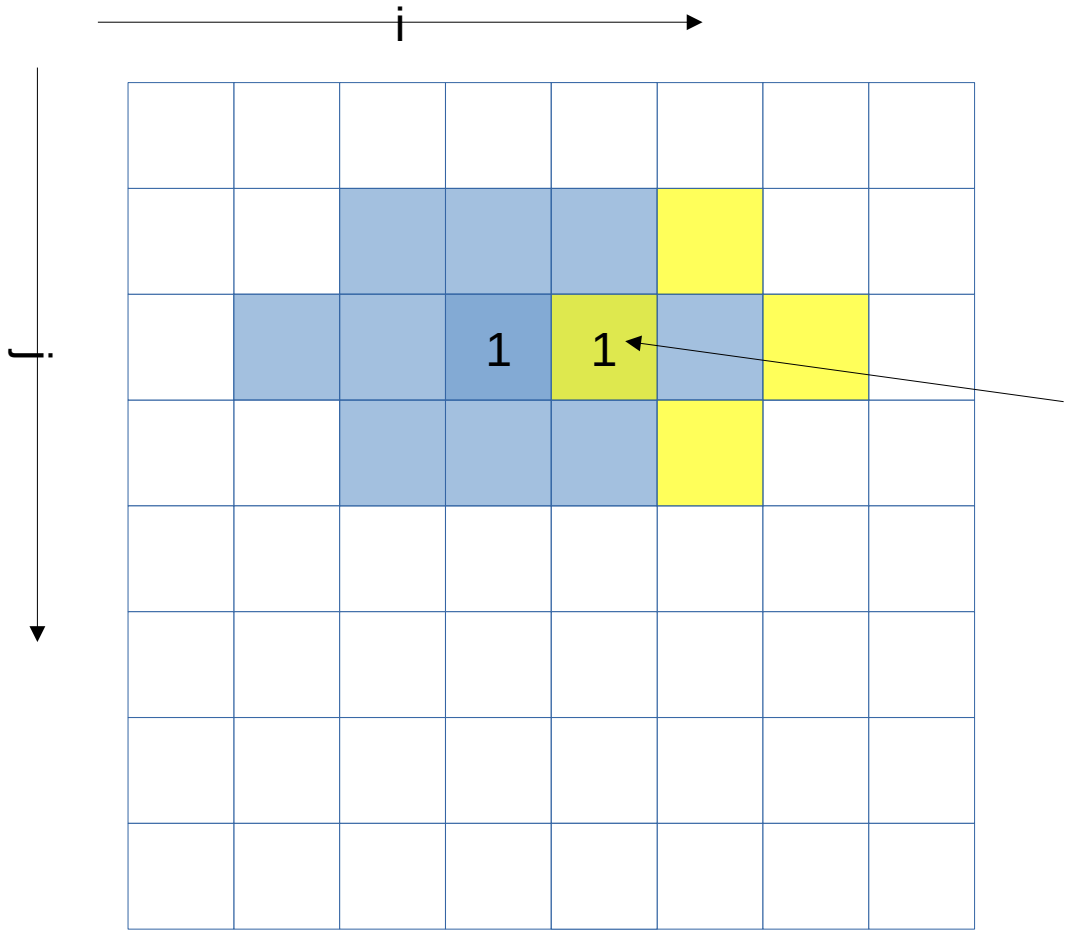
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

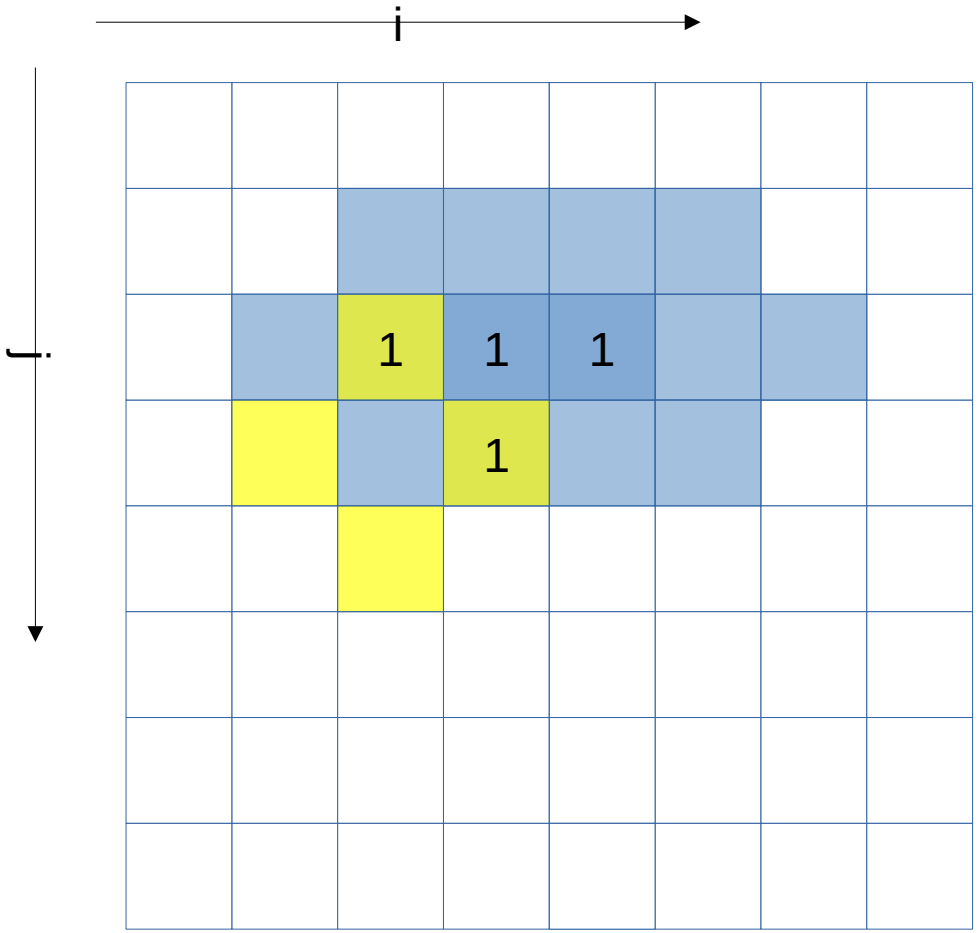
Memory references reuse : 4x4 unroll footprint on loads



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

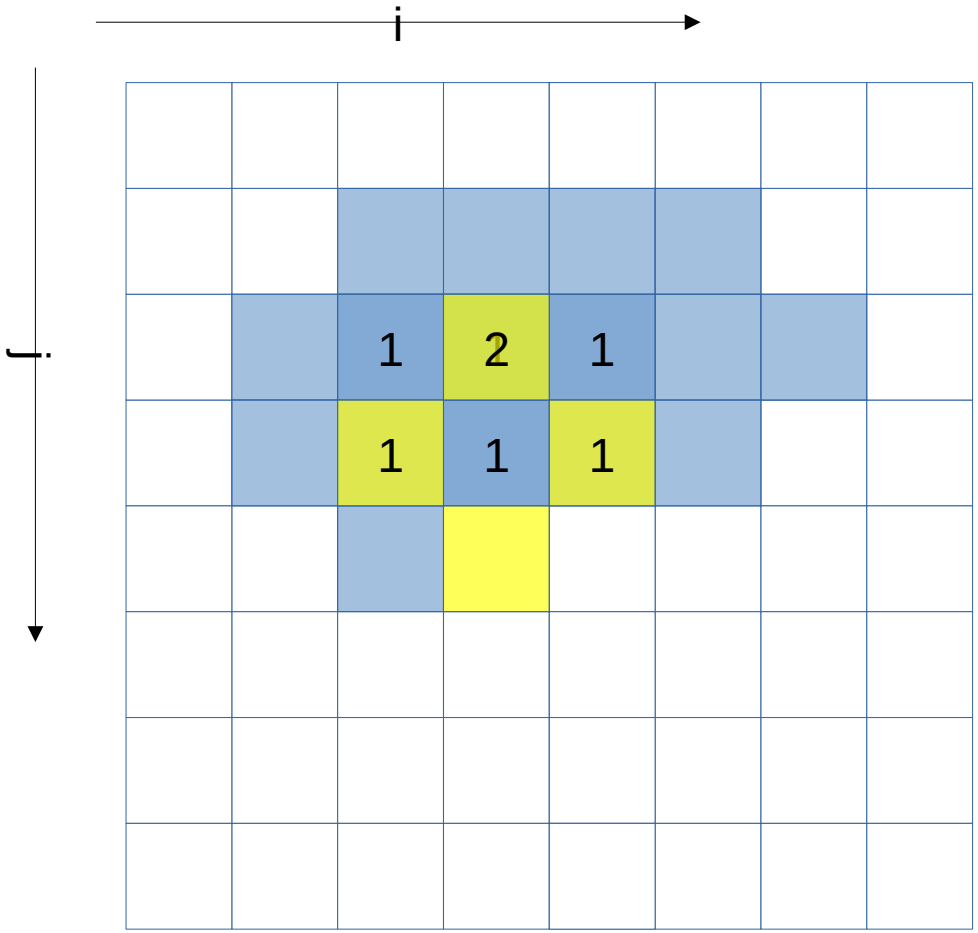


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

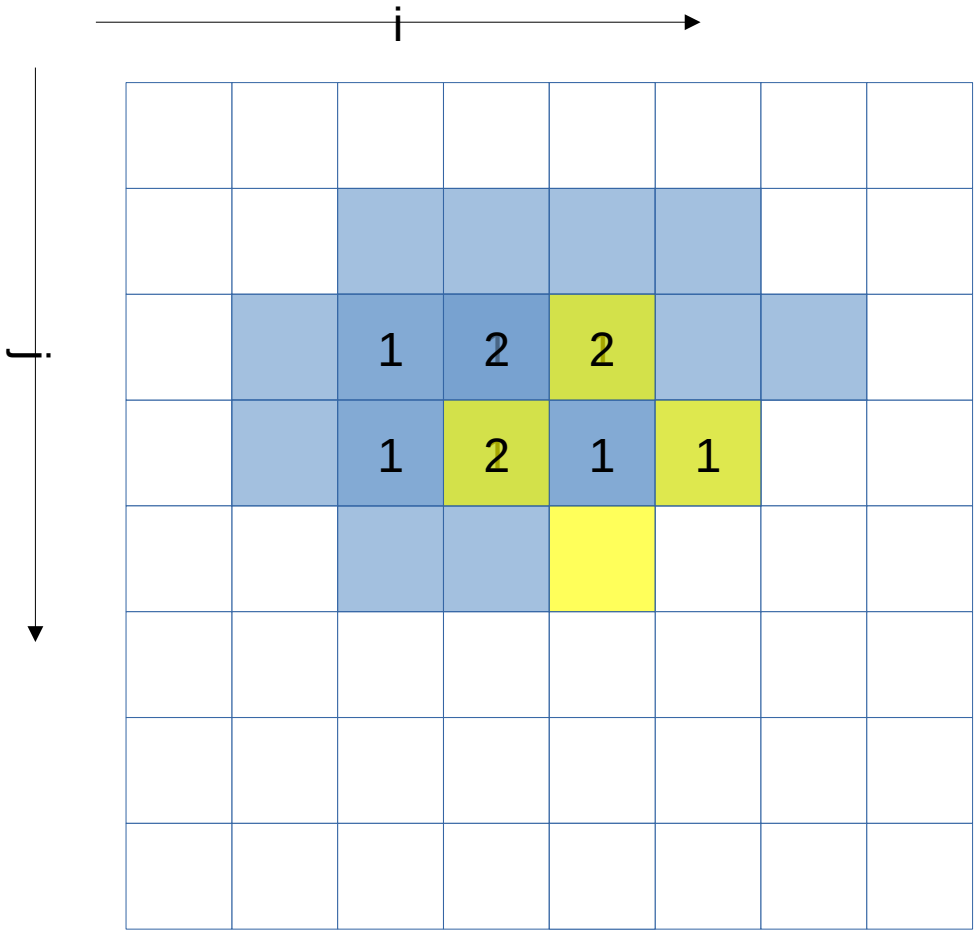


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

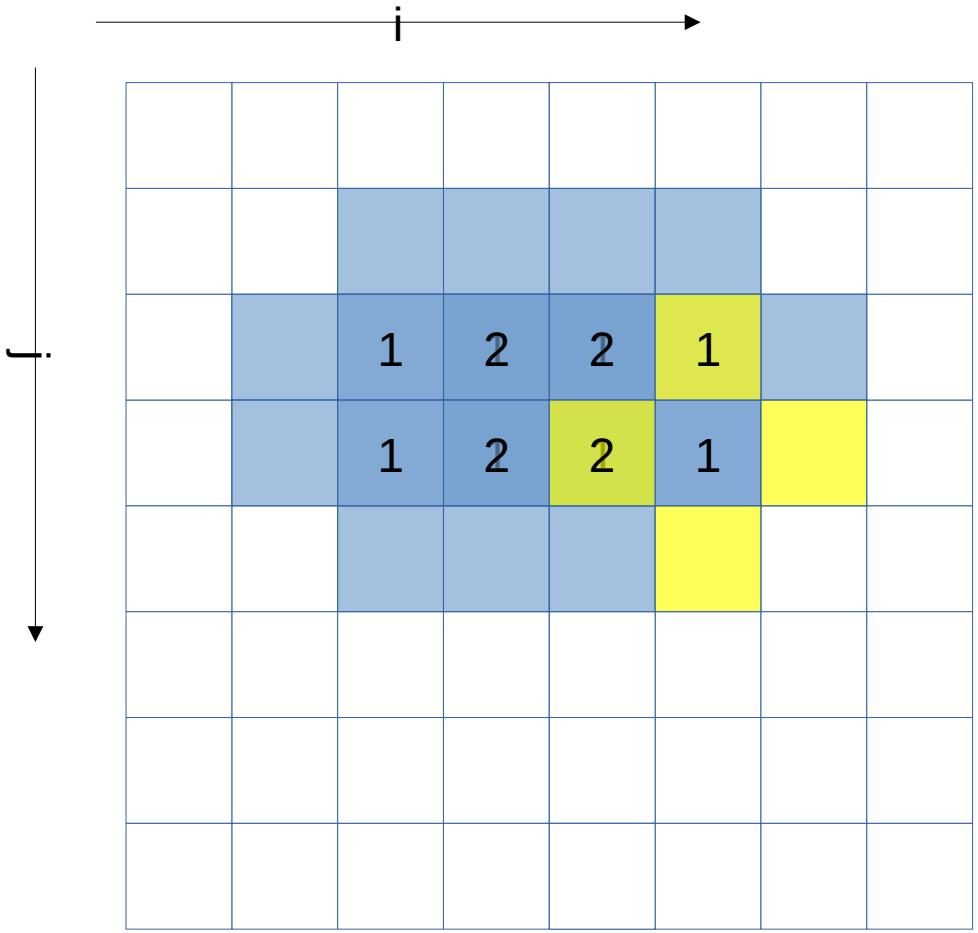


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

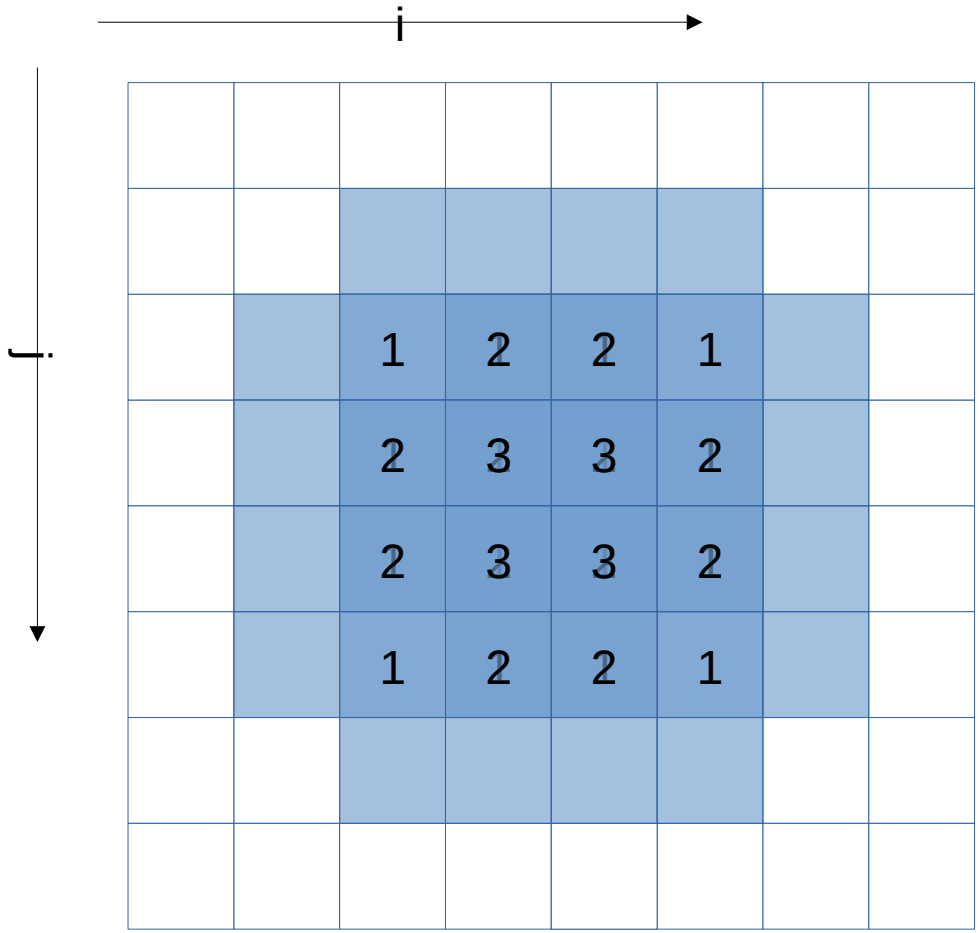


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

Running and analyzing kernel1

```
> ./hydro_k1 300 50
Cycles per element for solvers: 603.86
```

Profile with MAQAO (200 repetitions)

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

Viewing results (HTML)

On your local machine (sshfs):

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

Global Metrics

Total Time (s)		5.04
Profiled Time (s)		5.04
Time in loops (%)		99.83
Time in innermost loops (%)		99.81
Time in user code (%)		99.88
Compilation Options		OK
Perfect Flow Complexity		1.08
Array Access Efficiency (%)		48.01
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	4
FP Vectorised	Potential Speedup	1.35
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	6.32
	Nb Loops to get 80%	10

Running and analyzing kernel1

Source Code

```
/home/woody/k_g22c/g22c0007/MAQAO_HANDSON/hydro/kernel.c: 15 -
-------------------------------------------------------------
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);
```

CQA

Coverage	61.22 %
Function	linearSolver1
Source file and lines	kernel.c:15-176
Module	hydro_k1

The loop is defined in /home/woody/k_g22c/g22c0007/MAQAO_HANDSON/hydro/kernel.c:15,156-176.

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

gain | potential | hint | expert

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 48.00 to 6.00 cycles (8.00x speedup).

Details

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

Workaround

- Try another compiler or update/tune your current one:
  - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)

Execution units bottlenecks

Performance is limited by:

- execution of FP add operations (the FP add unit is a bottleneck)
- execution of FP multiply or FMA (fused multiply-add) operations (the FP multiply/FMA unit is a bottleneck)

By removing all these bottlenecks, you can lower the cost of an iteration from 48.00 to 45.00 cycles (1.07x speedup).

Workaround

- Reduce the number of FP add instructions
- Reduce the number of FP multiply/FMA instructions

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:

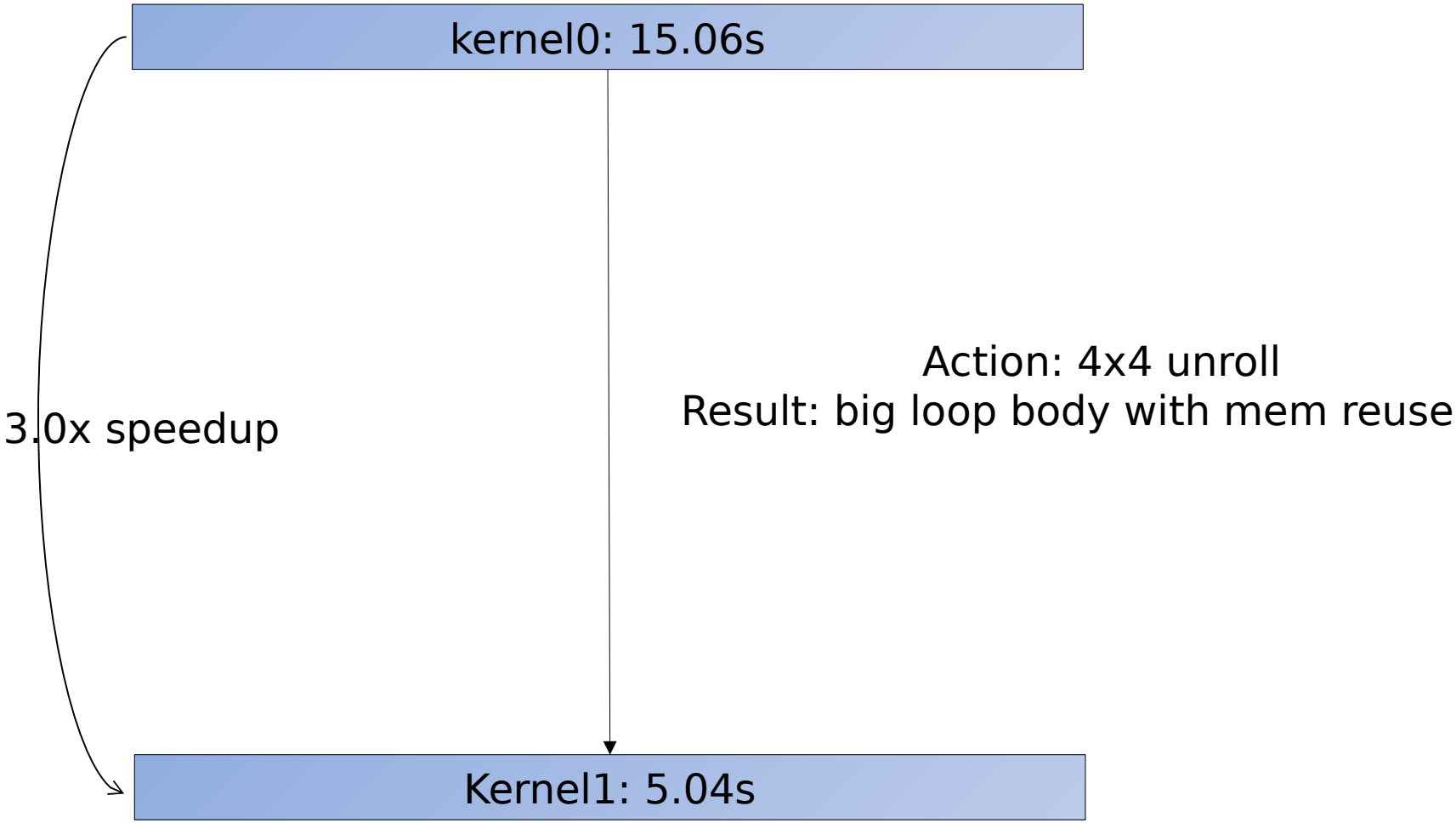
- 64: addition or subtraction
- 32: multiply

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

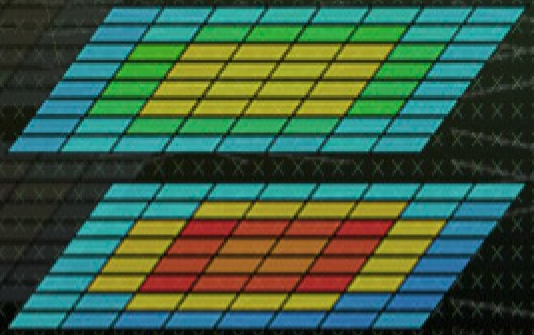
Summary of optimizations and gains



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 $TW38/maqao/lulesh2.0.3.tgz
```

Compile Lulesh

```
> cd lulesh
> module purge
> module load intel64/19.0up05 intelmpi/2019up05-intel
> 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 = {
{number_processes = 1, omp_num_threads = 10, number_nodes = 1},
{number_processes = 8, omp_num_threads =  1, number_nodes = 1, run_command = "<binary> -i 10 –p –s 65“},
{number_processes = 8, omp_num_threads =  1, number_nodes = 2, run_command = "<binary> -i 10 –p –s 65“},
{number_processes = 8, omp_num_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 $TW38/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
meggie_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>@meggie.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**