MAQAO
Hands-on exercises

Profiling bt-mz
Optimising a code

Setup

Login to the cluster

```
[LOCAL] ssh <login>@login1.amplitude.uni-due.de
```

Load VIHPS (incl. MAQAO) environment

```
> export TW47=/cluster/vi-hps_tuning_workshop/examples
Hint: copy in ~/.profile or ~/.bashrc
```

Copy handson material to your WORK directory

```
> cd $HPC_HOME
> tar xf $TW47/maqao/MAQAO_HANDSON.tgz
> tar xf $TW47/maqao/NPB3.4-MZ-MPI.tgz
```

(If not already done) Load MAQAO, compiler + MPI

```
> module load openmpi/5.0.3-intel24
> module load maqao/2025.1
```

Setup (bt-mz compilation with debug symbols)

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

```
> cd $HPC_HOME/NPB3.4-MZ-MPI
> cp $HPC_HOME/MAQAO_HANDSON/bt/make.def config
```

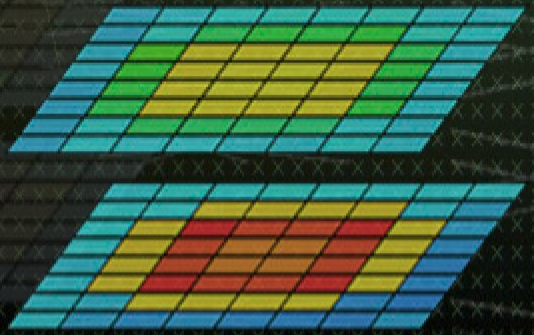
```
FFLAGS  = -O3 -qopenmp -g -fno-omit-frame-pointer
```

Compile bt-mz with debug information

```
> make bt-mz CLASS=D
```

Running bt-mz

```
> cp $HPC_HOME/MAQAO_HANDSON/bt/bt.slurm bin
> cd bin
> sbatch bt.slurm
```



Profiling bt-mz with MAQAO

Cédric VALENSI
Emmanuel OSERET

Setup ONE View for batch execution

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

```
“executable”: "bt-mz.D.x"
…
“batch_script”: "maqao_bt.slurm"
…
“batch_command”: "sbatch <batch_script>"
…
“number_processes”: 16
…
“number_nodes”: 2
…
“mpi_command”: "mpirun -n <number_processes> --bind-to core …"
…
“envv_OMP_NUM_THREADS”: 14
```

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 $HPC_HOME/NPB3.4-MZ-MPI/bin
> cp $HPC_HOME/MAQAO_HANDSON/bt/maqao_bt.slurm .
> less maqao_bt.slurm
```

```
...
#SBATCH -N 2 <number_nodes>
#SBATCH -n 16 <number_processes>
#SBATCH -c 14 <OMP_NUM_THREADS>
...
srun ./bt-mz.D.x
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on bt-mz (batch)

Launch ONE View

```
> cd $HPC_HOME/NPB3.4-MZ-MPI/bin
> maqao oneview --create-report=one \
-config=config_bt_oneview_sbatch.json -xp=ov_sbatch
```

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

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

Setup ONE View for scalability mode

Parameters for scalability mode are defined in multirun_params.

```
> less config_bt_oneview_sbatch.json
```

```
“executable”: "bt-mz.D.x"
“batch_script”: "maqao_bt.slurm"
“batch_command”: "sbatch <batch_script>"
“number_processes”: 16
“number_nodes”: 2
“mpi_command”: "mpirun -n <number_processes> --bind-to cores …"
“envv_OMP_NUM_THREADS”: 14
…
"multiruns_params": [
 { "name": "1N_8P", "number_nodes": 1, "number_processes": 8,
"number_processes_per_node": 8 },
 { "name": "2N_8P", "number_nodes": 2, "number_processes": 8,
"number_processes_per_node": 4 },
],
”scalability_reference”: “lowest-threads”
```

Launch MAQAO ONE View on bt-mz in scalability mode

Launch ONE View

```
> cd $HPC_HOME/NPB3.4-MZ-MPI/bin
> maqao oneview --create-report=one --with-scalability=strong \
-config=config_bt_oneview_sbatch.json -xp=ov_sbatch_scal
```

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 <exp-dir>/RESULTS/bt-mz.C.x_one_html/index.html
```

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

```
> tar –zcf $HOME/ov_html.tgz <exp-dir>/RESULTS/bt-mz.C.x_one_html
```

```
[LOCAL] scp <login>@login1.amplitude.uni-due.de:ov_html.tgz .
[LOCAL] tar xf ov_html.tgz
[LOCAL] firefox <exp-dir>/RESULTS/bt-mz.C.x_one_html/index.html
```

Results can also also be viewed directly on the console:

```
> maqao oneview -R1 -xp=<exp-dir> --output-format=text | less
```

The archive **MAQAO_HANDSON/bt/offline.tgz** contains a sample result directory.

Display MAQAO ONE View results using sshfs

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

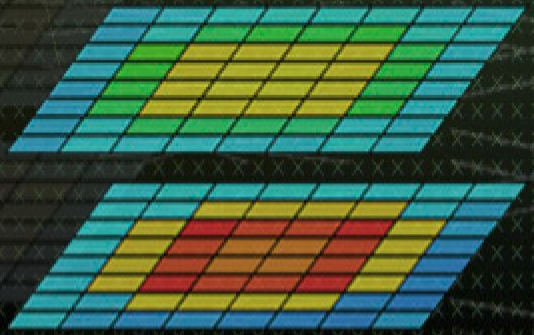
```
[LOCAL] sudo apt install sshfs
```

- Recommended to close a sshfs directory after use

```
[LOCAL] fusermount -u /path/to/sshfs/directory
```

Mount $SCRATCH locally:

```
[LOCAL] mkdir amplitude_work
[LOCAL] sshfs <login>@login1.amplitude.uni-due.de:/lustre/hpc_home/<user> \
amplitude_work
[LOCAL] firefox amplitude_work/NPB3.4-MZ-MPI/bin/ov_sbatch/RESULTS/bt-
mz.C.x_one_html/index.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

Setup environment

Load MAQAO environment (if needed)

```
> module load maqao/2025.1
```

Using GCC v11 compiler (system default)

```
> gcc -v
gcc version 11.5.0 20240719 (Red Hat 11.5.0-5) (GCC)
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

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

Execution

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
 matmul_orig/matmul 400 800
RDTSC cycles per inner iter.: 1.38
```

Analyse matrix multiply with ONE View

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
 maqao OV -R1 xp=ov_orig -- ./matmul_orig/matmul 400 800
```

OR

```
> maqao OV -R1 c=ov_orig.json xp=ov_orig
```

Viewing results (HTML)

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

```
[LOCAL] scp <login>@login1.amplitude.uni-due.de:ov_orig.tgz .
[LOCAL] tar xf ov_orig.tgz
[LOCAL] firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics	
Total Time (s)	35.40
Max (Thread Active Time) (s)	35.35
Average Active Time (s)	35.35
Activity Ratio (%)	99.9
Average number of active threads	0.999
Affinity Stability (%)	100.0
Time in analyzed loops (%)	100.0
Time in analyzed innermost loops (%)	99.8
Time in user code (%)	100.0
Compilation Options Score (%)	75.0
Array Access Efficiency (%)	83.3

Potential Speedups		
Perfect Flow Complexity		1.00
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	2.81
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	16.0
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.00
	Nb Loops to get 80%	1

Viewing results (text)

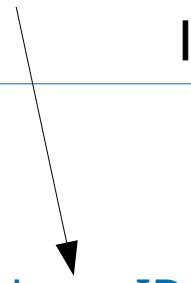
```
> maqao OV -R1 -xp=ov_orig \
  --output-format=text --text-global | less
```

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

  Total Time:                                35.40 s
  Profiled Time:                             35.35 s
  Time spent in analyzed loops:              100 %
  Time spent in analyzed innermost loops: 99.9 %
  Time spent in user code:                   100 %
  Compilation Options Score:                 75
  Array Access Efficiency:                   83.3 %

   Potential Speedups
  ------------------------------------------------
  Perfect Flow Complexity:                   1.00
  Perfect OpenMP + MPI + Pthread:            1.00
  Perfect OpenMP + MPI + Pthread + Load Distribution:  1.00
```

Viewing results (text)

```
> maqao oneview -R1 -xp=ov_orig \
  --output-format=text --text-loops | less
```

```
+-------------------------------------------------------------------+
+                        1.1  -  Top 10 Loops                       +
+-------------------------------------------------------------------+

   Loop Id | Module  | Source Location              | Coverage (%) |
  ---------+---------+------------------------------+--------------+
   1       | matm... | kernel.c:24-25               | 99.64        |
   2       | matm... | kernel.c:7-10                | 0.35         |
```



Viewing CQA output (text)

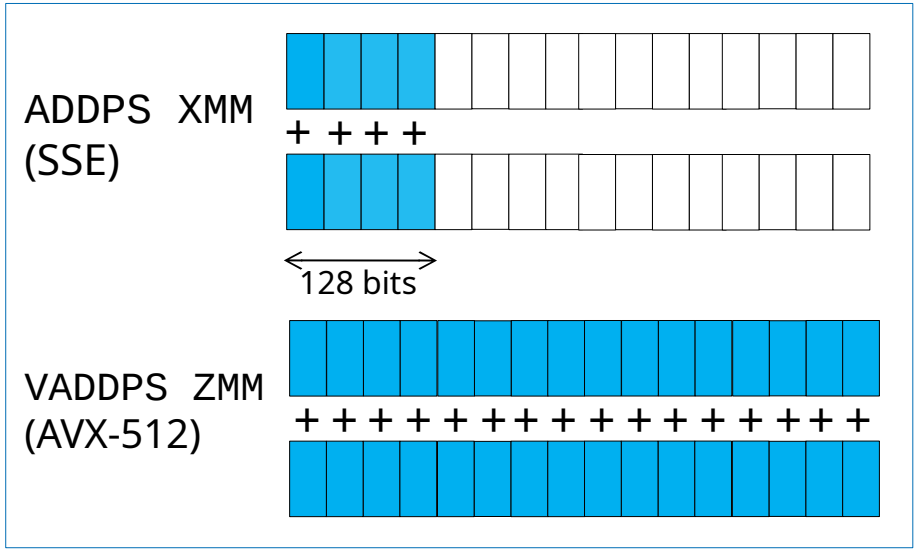
```
> maqao oneview -R1 -xp=ov_orig \
  --output-format=text --text-cqa=1
```

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 3.00 to 0.19 cycles
(16.00x speedup).
Workaround
 - Try another compiler or update/tune your current one:
  * recompile with fassociative-math (included in Ofast or ffast-math) to extend loop
vectorization to FP reductions.
 - Remove inter-iterations dependences from your loop and make it unit-stride:
  * If your arrays have 2 or more dimensions, check whether elements are accessed
contiguously and, otherwise, try to permute loops accordingly:
C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) =>
for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  * If your loop streams arrays of structures (AoS), try to use structures of arrays
instead (SoA):
for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)
```

Impacts of architecture specialization: vectorization and FMA

- Vectorization
  - SSE instructions (SIMD 128 bits) used on a processor supporting AVX 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 $HPC_HOME/MAQAO_HANDSON/matmul
> make matmul_opt
```

Execution

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
./matmul_opt/matmul 400 800
RDTSC cycles per inner iter.: 2.56
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 c=ov_opt.json xp=ov_opt
```

CQA output for the arch-specialized kernel

Global Metrics		
Total Time (s)		65.69
Max (Thread Active Time) (s)		65.56
Average Active Time (s)		65.56
Activity Ratio (%)		99.8
Average number of active threads		0.998
Affinity Stability (%)		100.0
Time in analyzed loops (%)		100.0
Time in analyzed innermost loops (%)		99.9
Time in user code (%)		100.0
Compilation Options Score (%)		100
Array Access Efficiency (%)		59.1

Potential Speedups		
Perfect Flow Complexity		1.00
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	1.67
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	16.0
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.00
	Nb Loops to get 80%	1

SLOWDOWN (from 35.35s)

CQA output for the arch-specialized kernel

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers.

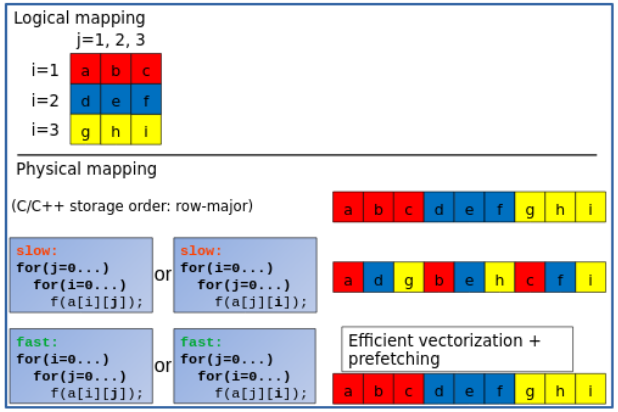
By vectorizing your loop, you can lower the cost of an iteration from 40.00 to 5.00 cycles (8.00x speedup).

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

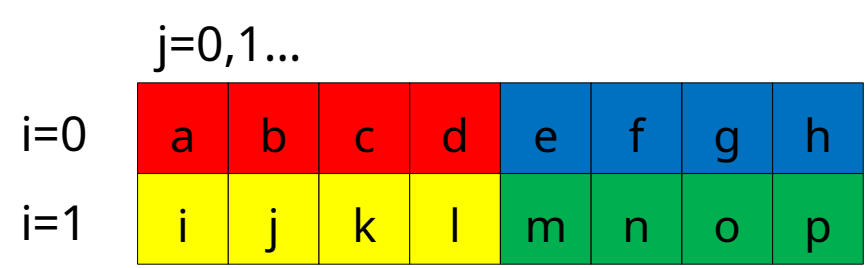
- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check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

Impact of loop permutation on data access

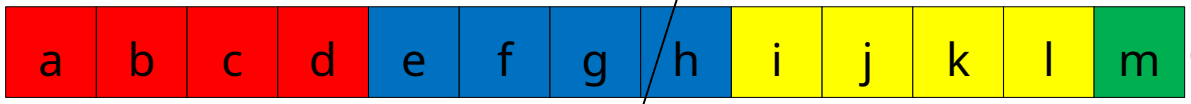
Logical mapping



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 $HPC_HOME/MAQAO_HANDSON/matmul
> make matmul_perm_opt
```

Execution

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
./matmul_perm_opt/matmul 400 800
RDTSC cycles per inner iter.: 0.22
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 c=ov_perm_opt.json xp=ov_perm_opt
```

Loop permutation results

Global Metrics		
Total Time (s)		5.62
Max (Thread Active Time) (s)		5.59
Average Active Time (s)		5.59
Activity Ratio (%)		99.5
Average number of active threads		0.995
Affinity Stability (%)		99.9
Time in analyzed loops (%)		99.7
Time in analyzed innermost loops (%)		91.3
Time in user code (%)		99.7
Compilation Options Score (%)		100
Array Access Efficiency (%)		81.3

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.10
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	2.13
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.46
	Nb Loops to get 80%	1

Much faster (orig was 35.35s)

Much better (orig was close to 16)

CQA output after loop permutation

Loop id	Source Location	Source Function	Level	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
4	matmul - kernel.c:24-25	kernel	Innermost	91.32	100	50
2	matmul - kernel.c:6-25 [...]	kernel	InBetween	8.32	26.09	18

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" report at "Potential" confidence level.

Analyse matrix multiply after enforcing 512 bits vectorization

Compile permuted loops version of matrix multiply

```
> cd $HPC_HOME/MAQAO_HANDSON/matmul
> make matmul_perm_opt512
```

Execution

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
./matmul_perm_opt512/matmul 400 800
RDTSC cycles per inner iter.: 0.20
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 c=ov_perm_opt512.json xp=ov_perm_opt512
```

512 bits vectorization results

Global Metrics	
Total Time (s)	5.06
Max (Thread Active Time) (s)	5.04
Average Active Time (s)	5.04
Activity Ratio (%)	99.6
Average number of active threads	0.996
Affinity Stability (%)	99.8
Time in analyzed loops (%)	99.7
Time in analyzed innermost loops (%)	88.6
Time in user code (%)	99.7
Compilation Options Score (%)	100
Array Access Efficiency (%)	81.3

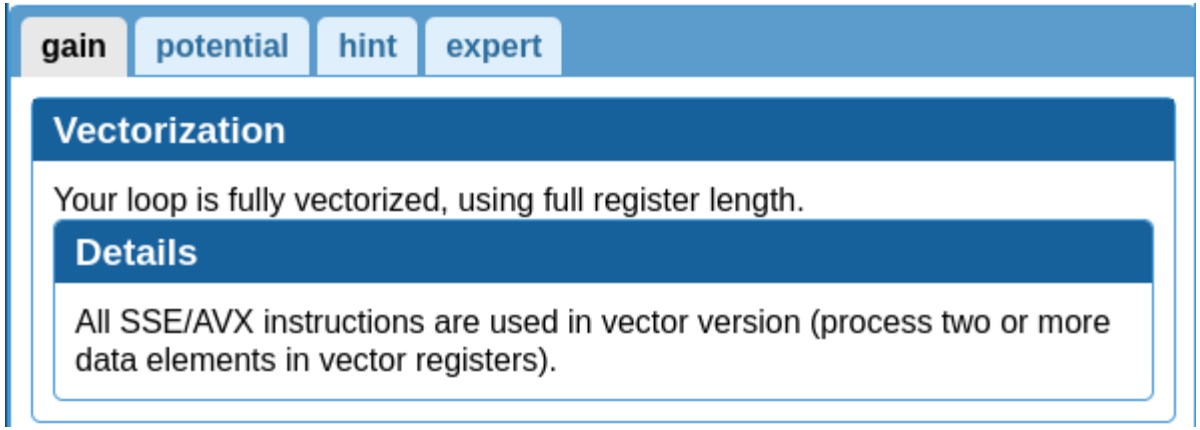
Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.04
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.05
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.48
	Nb Loops to get 80%	2

Much faster (orig was 35.35s)

Much better (orig was close to 16)

CQA output after 512 bits vectorization

Loop id	Source Location	Source Function	Level	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
4	matmul - kernel.c:24-25	kernel	Innermost	88.59	100	100
2	matmul - kernel.c:6-25 [...]	kernel	InBetween	11.11	21.43	25.95



Using comparison mode

Generating a comparison report from experiment directories

```
> maqao oneview --compare-reports -xp=ov_matmul_cmp \
-inputs=ov_orig,ov_opt,ov_perm_opt,ov_perm_opt512
```

Open `ov_matmul_cmp/RESULTS/ov_matmul_cmp/index.html`

▼ **Compared Reports**

- r0: ov_orig
- r1: ov_opt
- r2: ov_perm_opt
- r3: ov_perm_opt512

Global Metrics

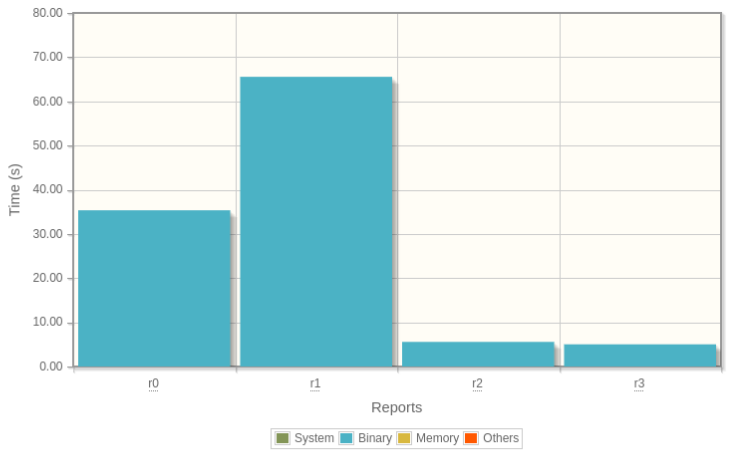
Metric	r0	r1	r2	r3
Total Time (s)	35.40	65.69	5.62	5.06
Max (Thread Active Time) (s)	35.35	65.56	5.59	5.04
Average Active Time (s)	35.35	65.56	5.59	5.04
Activity Ratio (%)	99.9	99.8	99.5	99.6
Average number of active threads	0.999	0.998	0.995	0.996
Affinity Stability (%)	100.0	100.0	99.9	99.8
Time in analyzed loops (%)	100.0	100.0	99.7	99.7
Time in analyzed innermost loops (%)	99.8	99.9	91.3	88.6
Time in user code (%)	100.0	100.0	99.7	99.7
Compilation Options Score (%)	75.0	100	100	100
Array Access Efficiency (%)	83.3	59.1	81.3	81.3

Potential Speedups

		r0	r1	r2	r3
Perfect Flow Complexity		1.00	1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.00	1.00	1.03	1.04
	Nb Loops to get 80%	1	1	1	1
FP Vectorised	Potential Speedup	2.81	1.67	1.10	1.00
	Nb Loops to get 80%	1	1	2	1
Fully Vectorised	Potential Speedup	16.0	16.0	2.13	1.05
	Nb Loops to get 80%	1	1	2	1
Only FP Arithmetic	Potential Speedup	1.00	1.00	1.46	1.48
	Nb Loops to get 80%	1	1	1	2

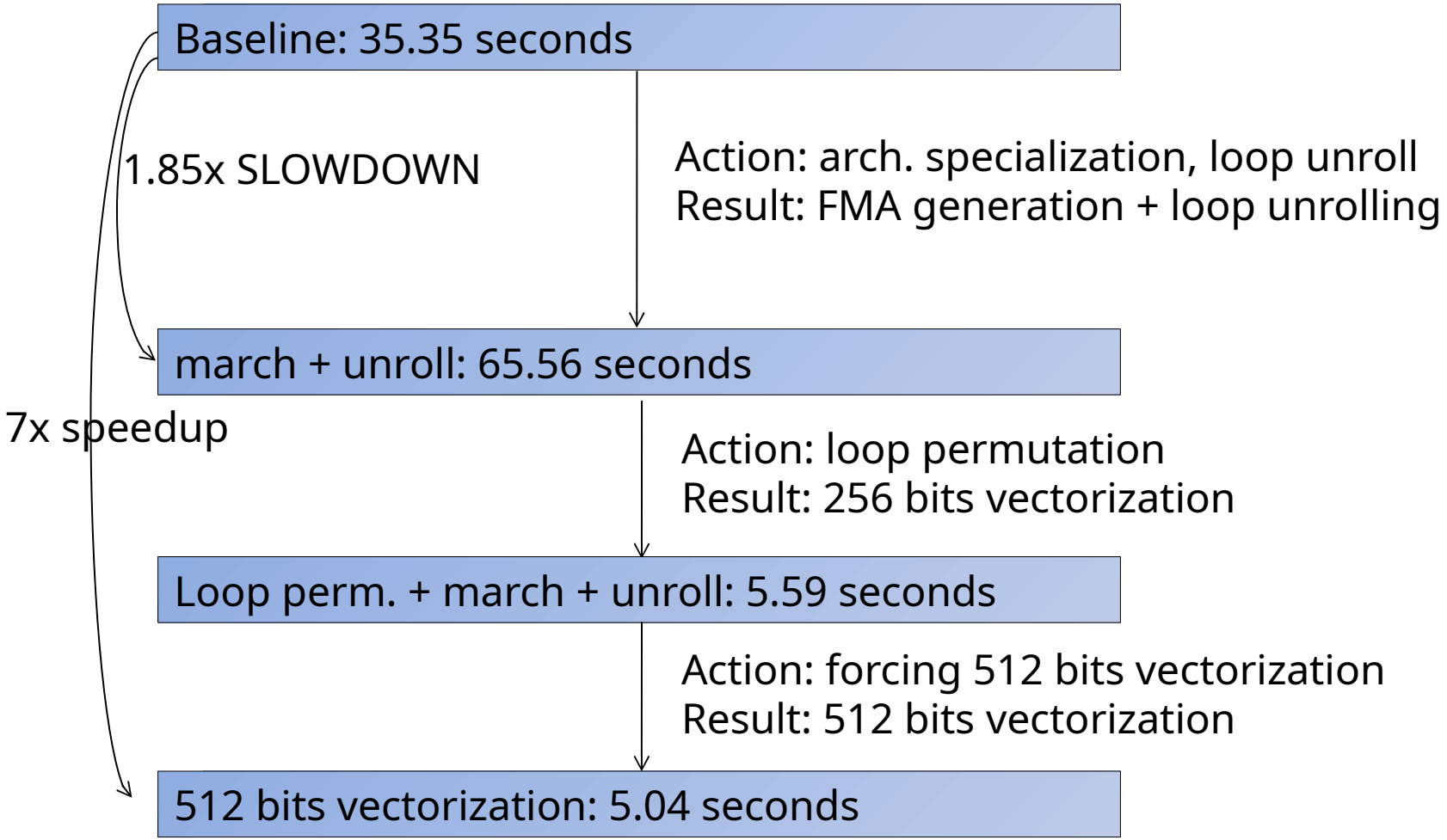
Application Categorization

Time



Coverage

Summary of optimizations and gains



Hydro example

Switch to the hydro handson folder

```
> cd $HPC_HOME/MAQAO_HANDSON/hydro
```

Load MAQAO (if not alread loaded) and Intel 24 compiler

```
> module load maqao/2025.1 openmpi/5.0.3-intel24
```

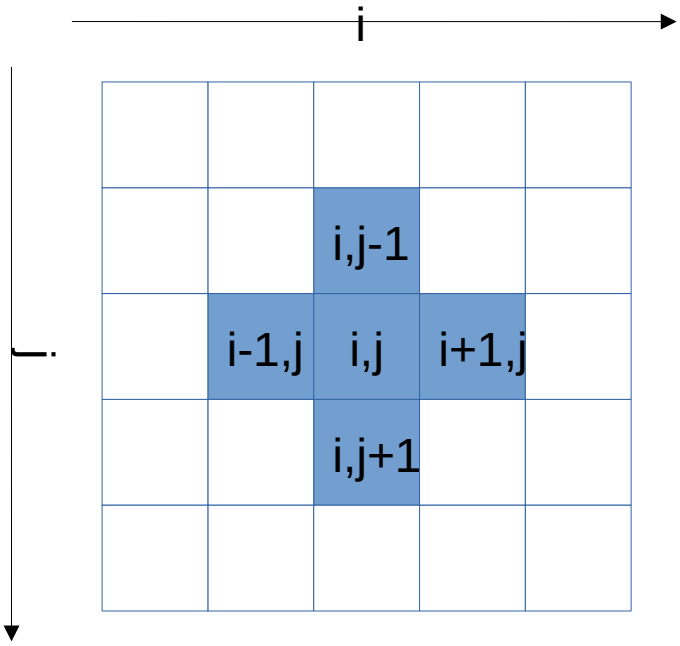
Compile

```
> make
```

Hydro code

```
int build_index (int i, int j, int grid_size)
{
  return (i + (grid_size + 2) * j);
}

void linearSolver0 (...) {
  int i, j, k;

  for (k=0; k<20; k++)
    for (i=1; i<=grid_size; i++)
      for (j=1; j<=grid_size; j++)
        x[build_index(i, j, grid_size)] =
  (a * ( x[build_index(i-1, j, grid_size)] +
         x[build_index(i+1, j, grid_size)] +
         x[build_index(i, j-1, grid_size)] +
         x[build_index(i, j+1, grid_size)]
       ) + x0[build_index(i, j, grid_size)]
  ) / c;
}
```

Iterative linear system solver using the Gauss-Siedel relaxation technique. « Stencil » code



Running and analyzing original kernel (icx -O3 -xHost)

Execution

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
./hydro_orig 300 400
Cycles per element for solvers: 971.72
```

Analysing with MAQAO

```
> maqao OV -R1 xp=ov_orig c=ov_orig.json
```

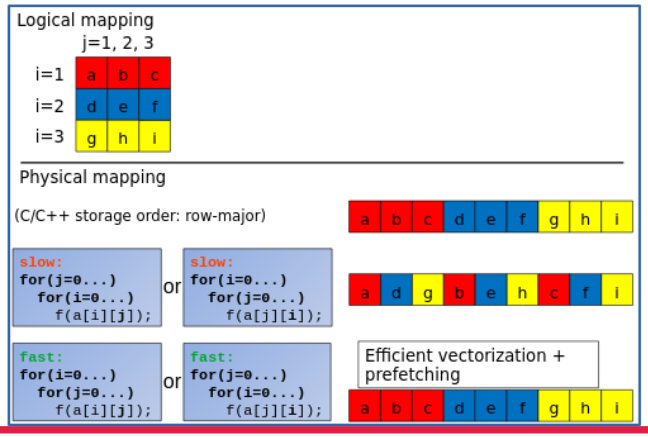
OR

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
 maqao OV -R1 xp=ov_orig -- ./hydro_orig 300 400
```

CQA output for original kernel

Workaround

- Try another compiler or update/tune your current one:
  - recompile with O2 or higher to enable loop vectorization and with ffast-math (included in Ofast) to extend 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)



As for matmul, loops should be permuted.
CF build_index

Unroll opportunity

Loop is 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 recompiling with -funroll-loops and/or -floop-unroll-and-jam.

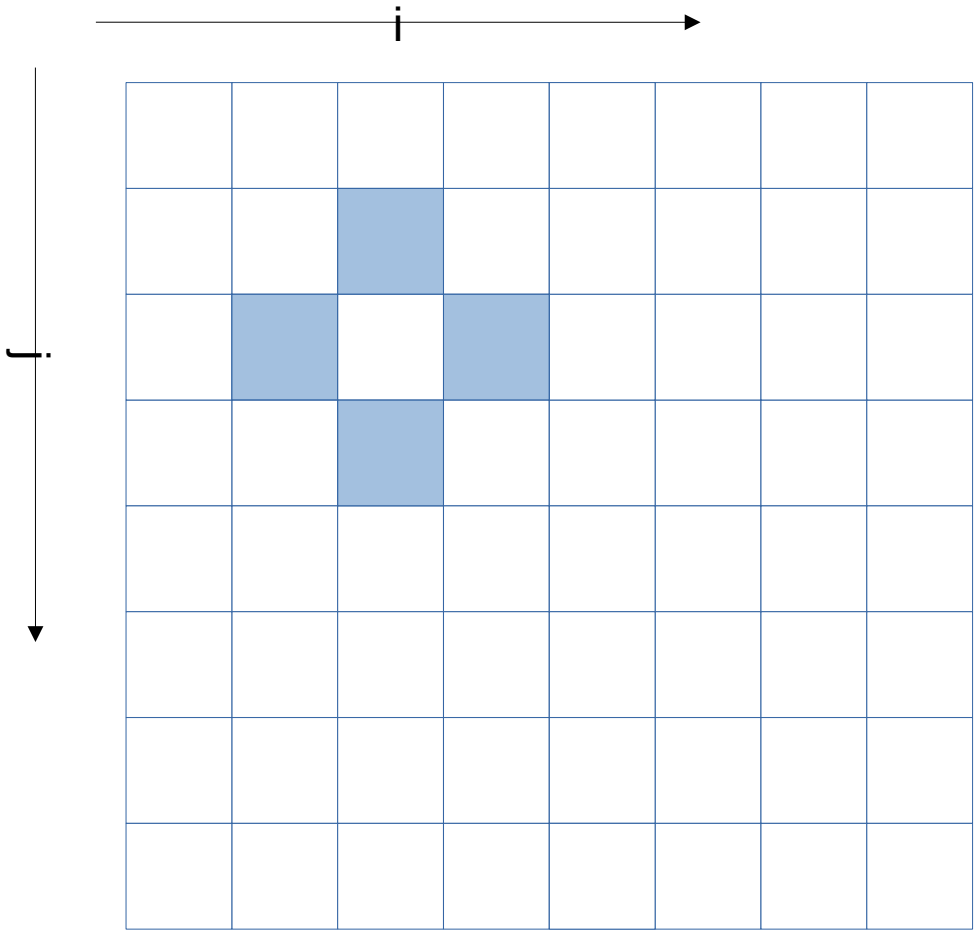
Consider loop unrolling

Kernel with loop permutation

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
./hydro_perm 300 400
Cycles per element for solvers: 966.03
```

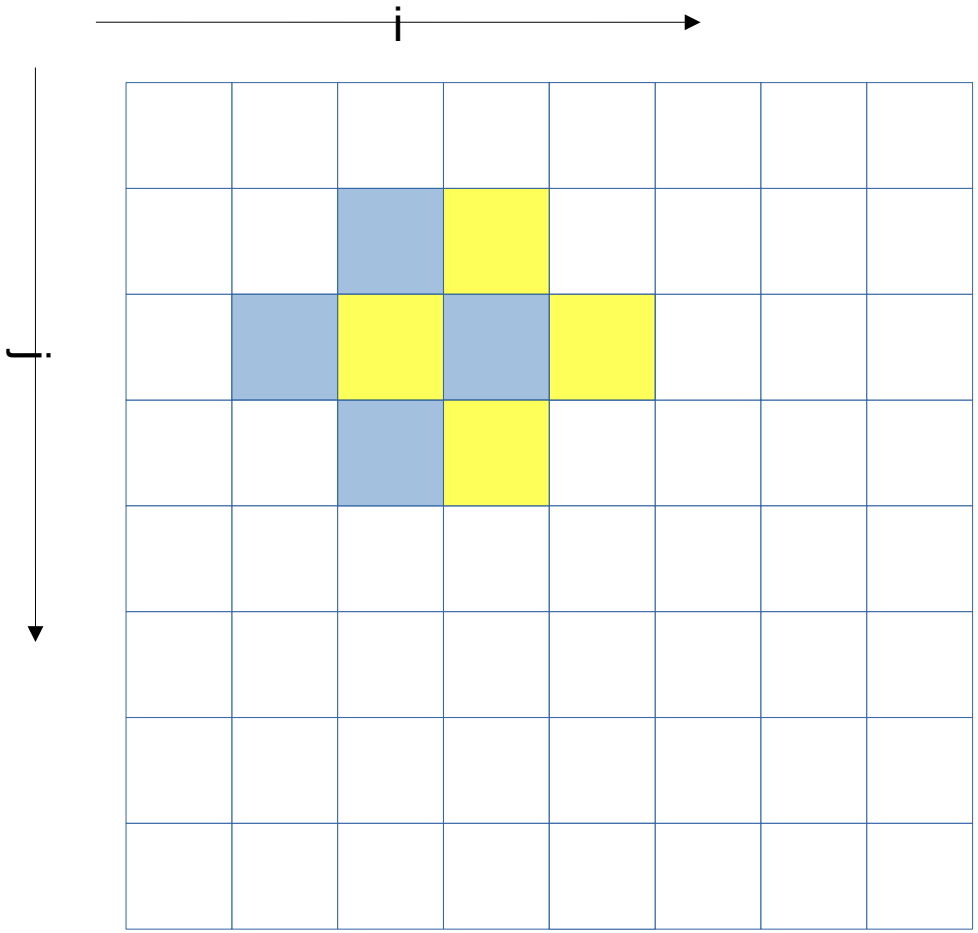
```
> maqao oneview -R1 xp=ov_perm c=ov_perm.json
```

Memory references reuse : 4x4 unroll footprint on loads



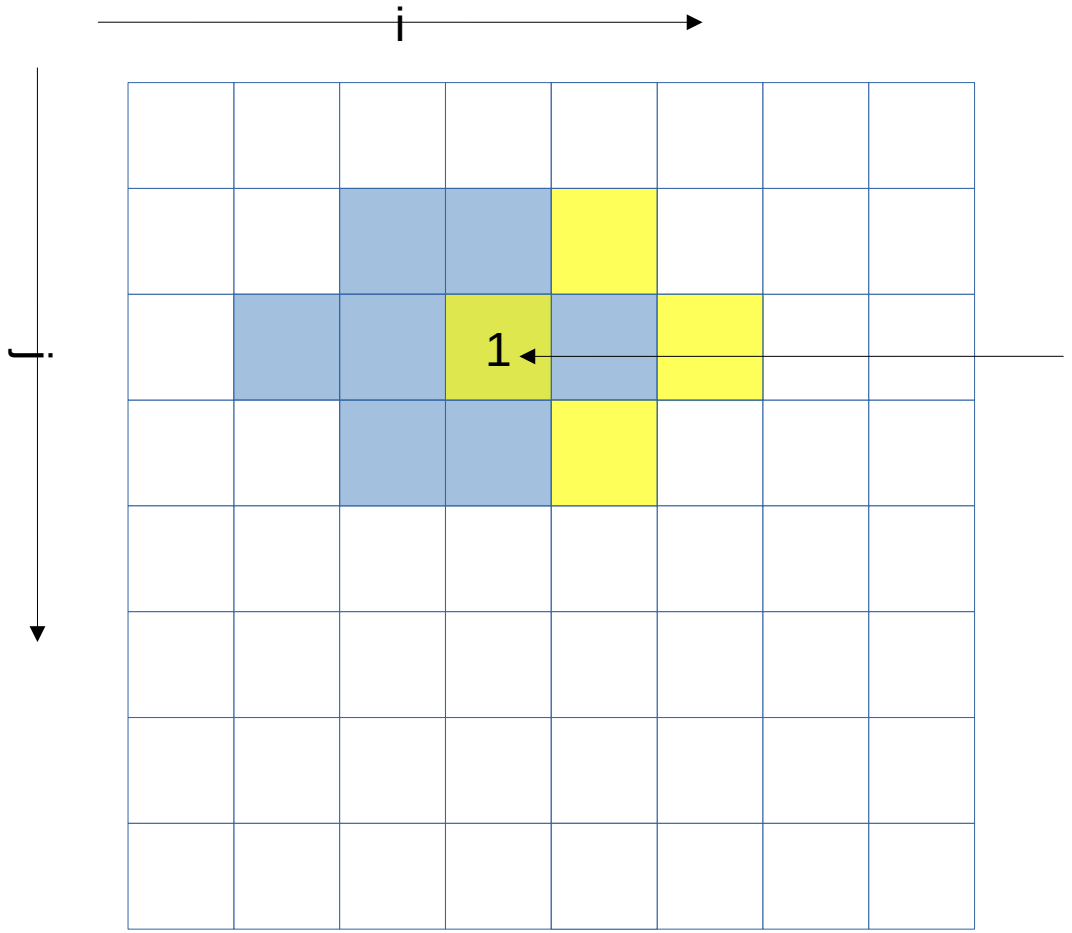
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

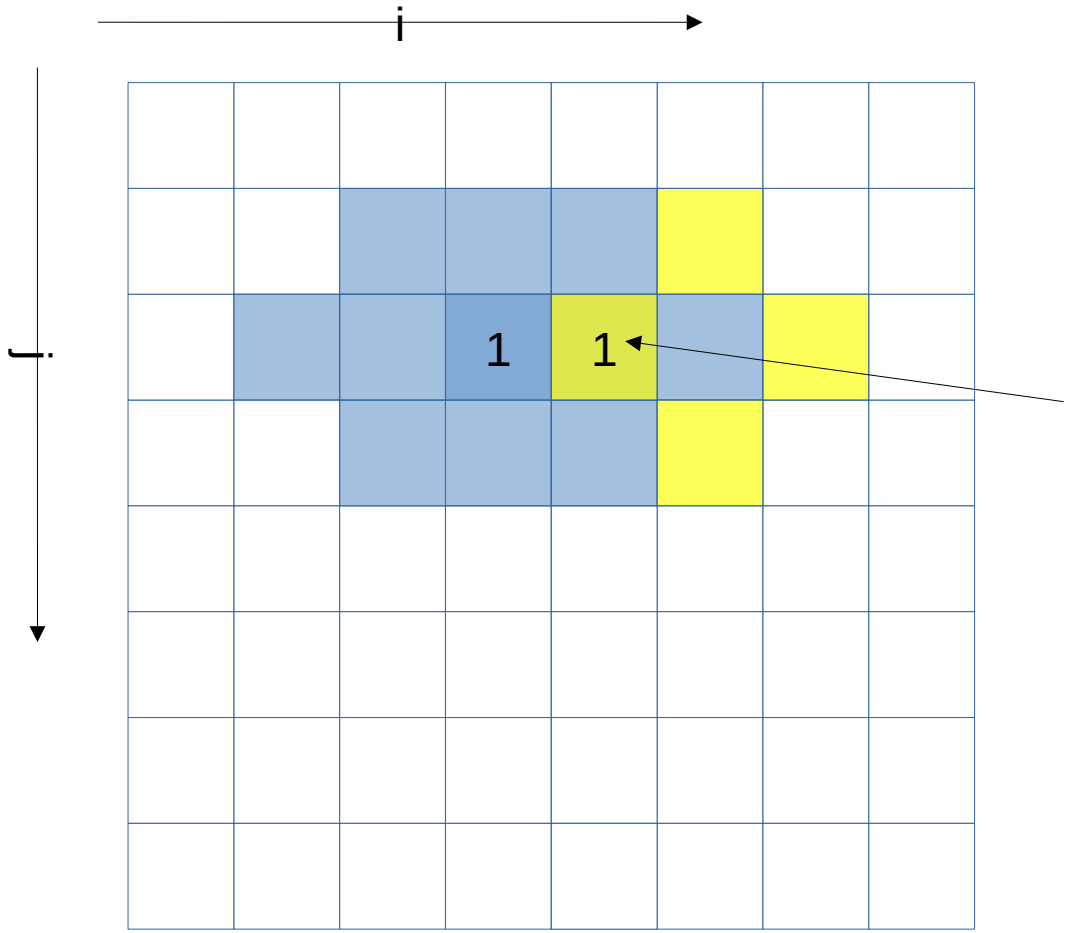
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

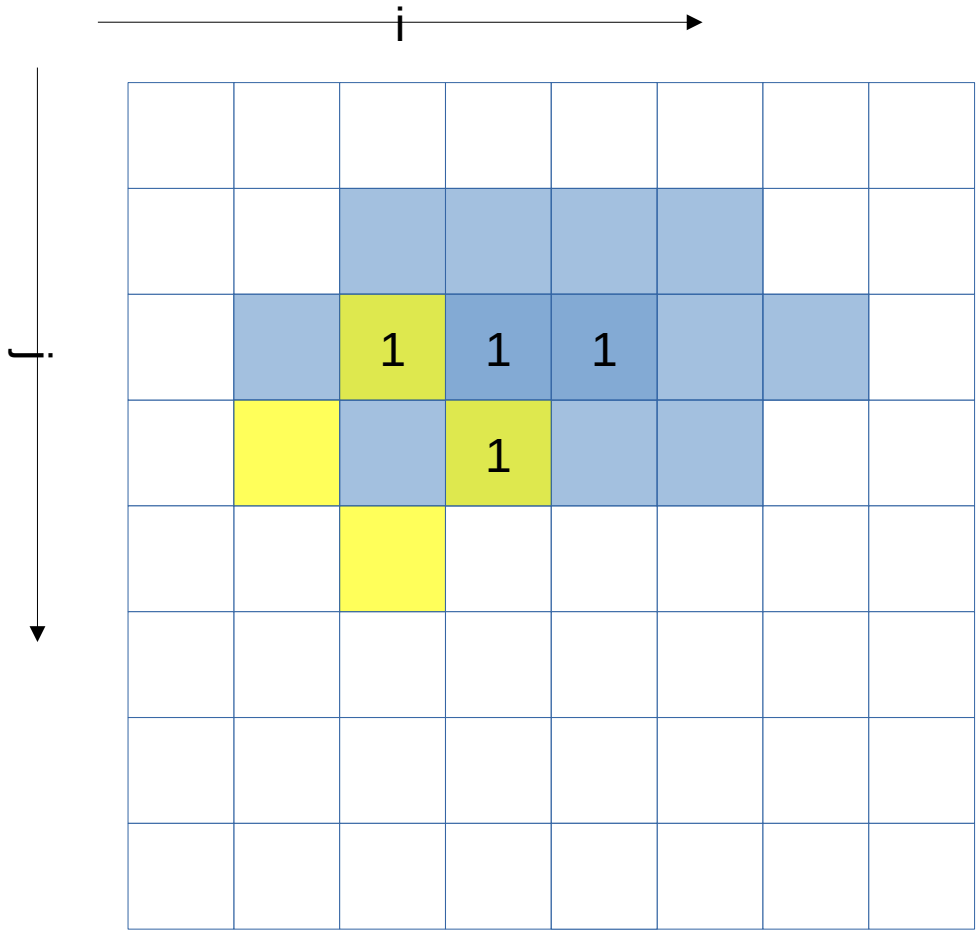
Memory references reuse : 4x4 unroll footprint on loads



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

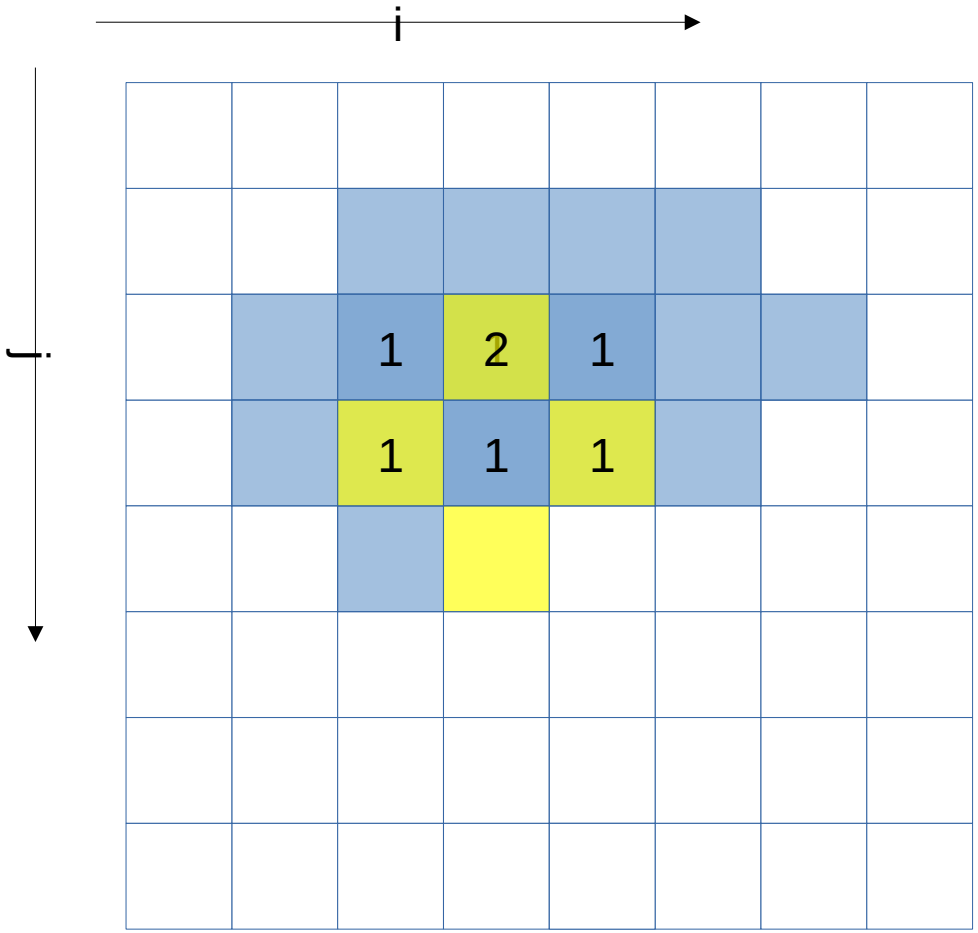


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

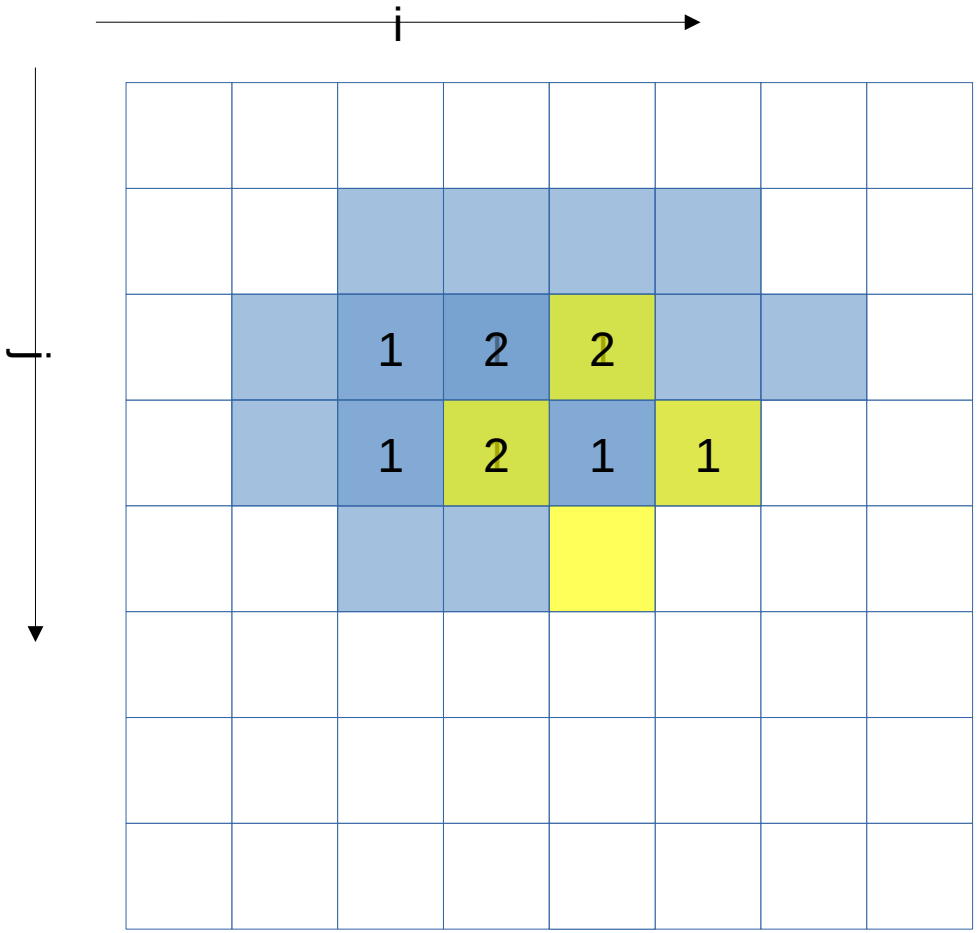


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

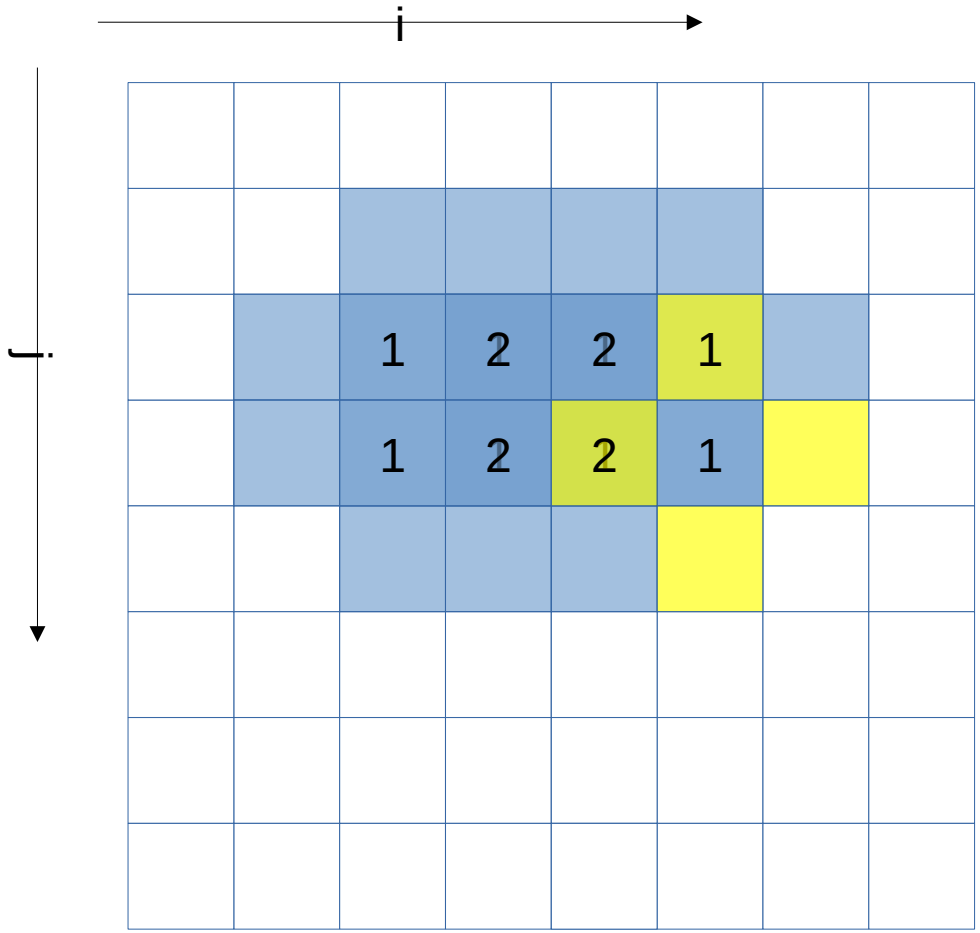


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

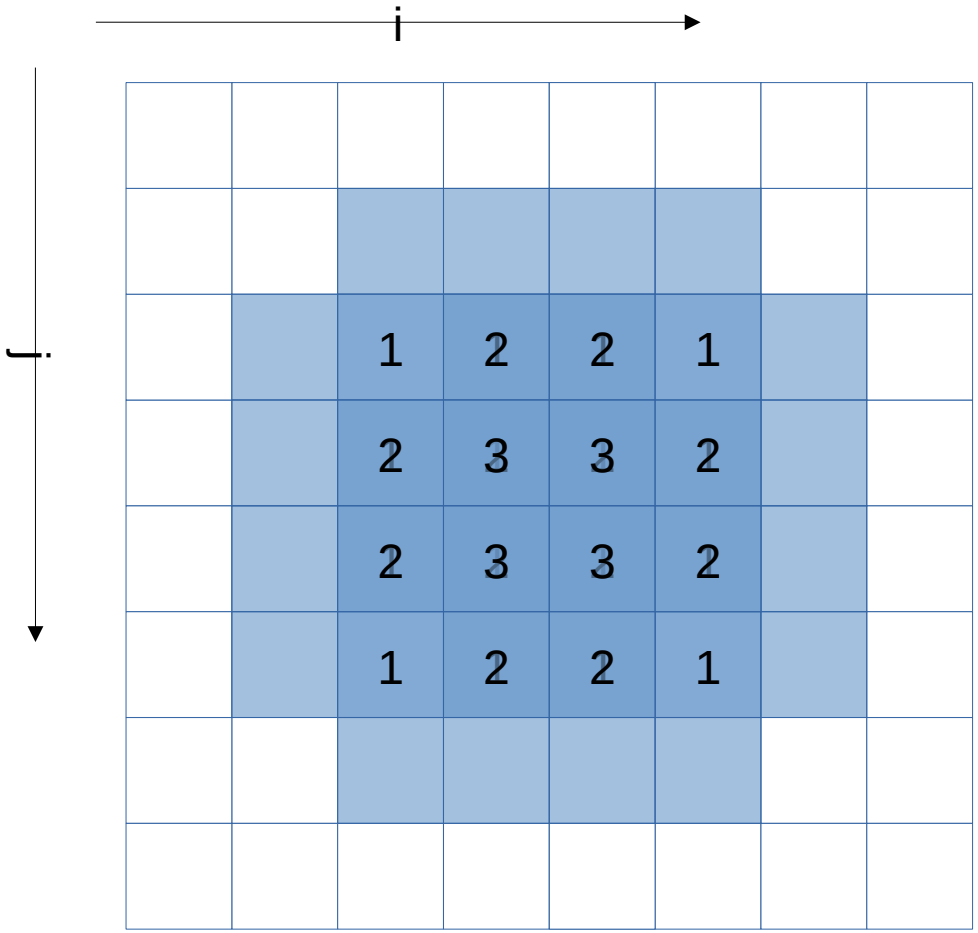


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

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);
      }
}
```

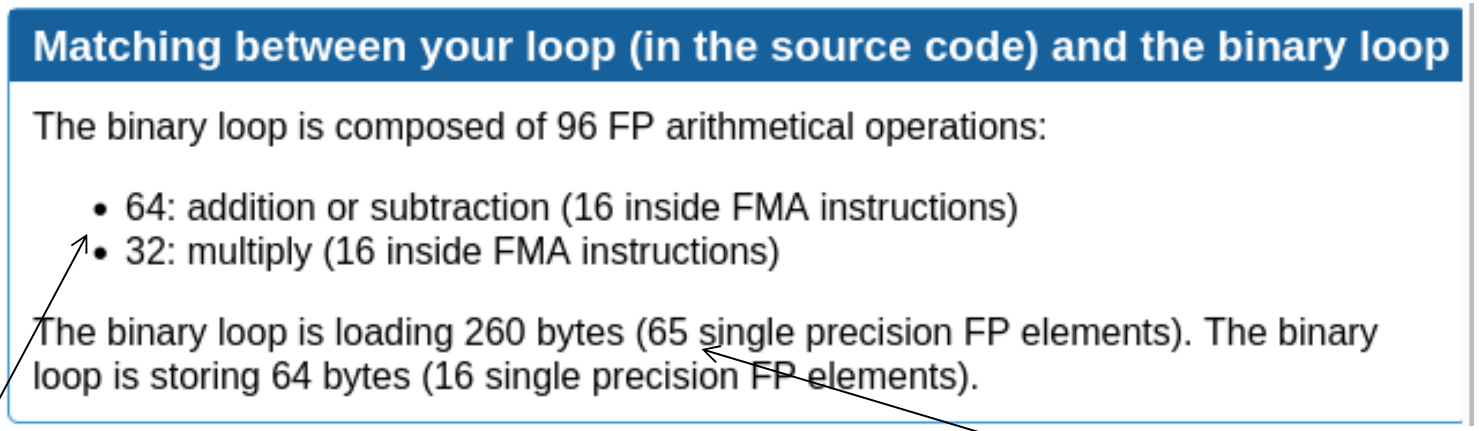
grid_size must now be multiple of 4. Or loop control must be adapted (much less readable) to handle leftover iterations

Kernel with manual 4x4 unroll and jam

```
> srun --reservation=VI-HPS_Tuning_Workshop_Duisburg-Essen -t 2 \
  ./hydro_unroll 300 400
Cycles per element for solvers: 378.28
```

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

CQA output for unrolled kernel



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 (16 inside FMA instructions)
- 32: multiply (16 inside FMA instructions)

The binary loop is loading 260 bytes (65 single precision FP elements). The binary loop is storing 64 bytes (16 single precision FP elements).

4x4 Unrolling were applied

Lower than 80: 64 (from x) + 16 (from x0)

Now divides appears: compiler failed to remove common divide across macros

Using comparison mode

Generating a comparison report from experiment directories

```
> maqao oneview --compare-reports -xp=ov_hydro_cmp \
-inputs=ov_orig,ov_perm,ov_unroll
```

Open ov_hydro_cmp/RESULTS/ov_hydro_cmp/index.html

▼ Compared Reports

- r0: ov_orig
- r1: ov_perm
- r2: ov_unroll

Global Metrics

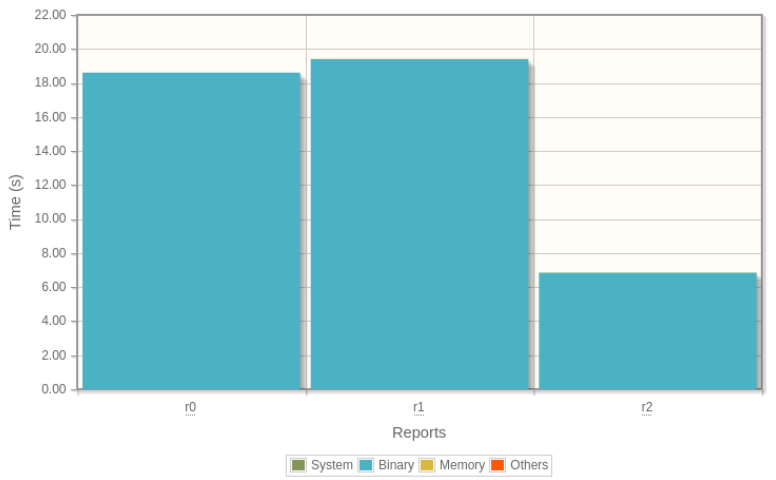
Metric		r0	r1	r2
Total Time (s)		18.66	19.52	6.90
Max (Thread Active Time) (s)		18.62	19.43	6.88
Average Active Time (s)		18.62	19.43	6.88
Activity Ratio (%)		99.8	99.5	99.7
Average number of active threads		0.998	0.995	0.997
Affinity Stability (%)		100.0	100.0	99.9
Time in analyzed loops (%)		99.9	99.8	99.6
Time in analyzed innermost loops (%)		99.9	99.7	99.6
Time in user code (%)		99.9	99.8	99.6
Compilation Options Score (%)		100	100	100
Array Access Efficiency (%)		49.3	93.3	48.3

Potential Speedups

Perfect Flow Complexity		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.01	1.01	1.03
	Nb Loops to get 80%	2	3	2
FP Vectorised	Potential Speedup	1.66	1.67	1.76
	Nb Loops to get 80%	2	2	2
Fully Vectorised	Potential Speedup	14.7	14.3	12.8
	Nb Loops to get 80%	6	6	6
Only FP Arithmetic	Potential Speedup	1.04	1.04	1.12
	Nb Loops to get 80%	2	2	1

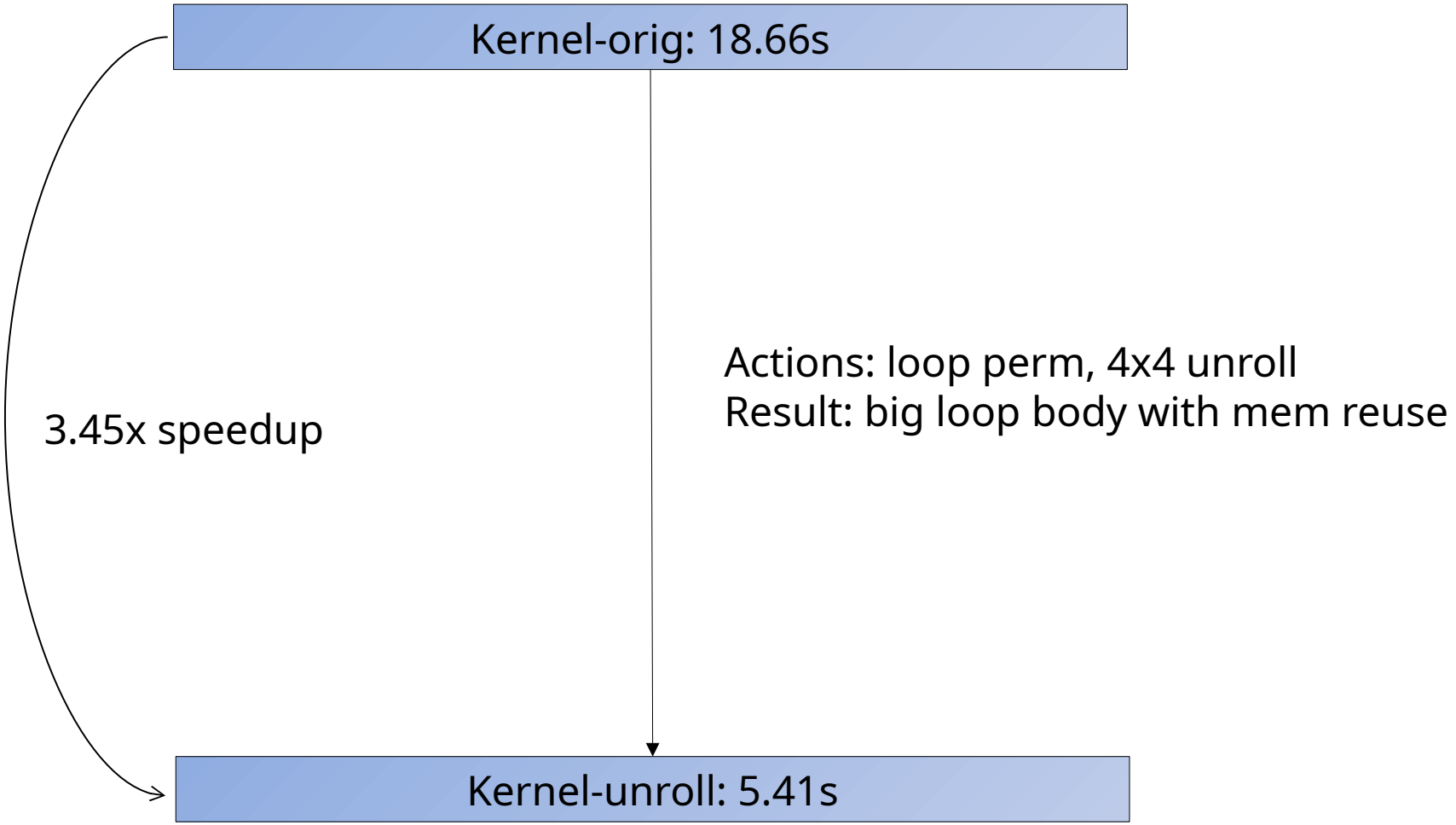
Application Categorization

Time



Coverage

Summary of optimizations and gains



More sample codes

More codes to study with MAQAO in

```
$HPC_HOME/MAQAO_HANDSON/loop_optim_tutorial.tgz
```