MAQAO
Hands-on exercises

Profiling bt-mz
Optimising a code

Setup (GPP nodes)

Login to the cluster

```
> ssh nct0XXXX@glogin[12].bsc.es
```

Set WORK to the directory you want to save MAQAO results (or use HOME)

```
> export WORK=/gpfs/scratch/nct_362/users/$USER
> vim ~/.bash_profile # append the line above to make it persist
```

Copy handson material to your WORK directory

```
> cd $WORK
> tar xf /gpfs/scratch/nct_362/exercises/maqao/MAQAO_HANDSON.tgz
> tar xf /gpfs/scratch/nct_362/exercises/maqao/NPB3.4-MZ-MPI.tgz
```

Setup (bt-mz compilation with debug symbols)

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

```
> cd $WORK/NPB3.4-MZ-MPI
> vi config/make.def
```

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

Or copy the modified file from MAQAO_HANDSON directory

```
> cp $WORK/MAQAO_HANDSON/bt/make.def config
```

Compile bt-mz with debug information (on the login node)

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

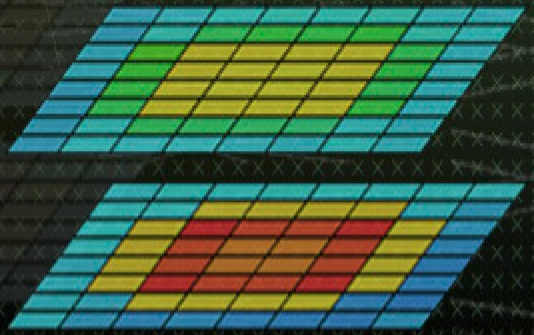
Setup (optional) run bt-mz

Copy sample jobscript file

```
> cd $WORK/NPB3.4-MZ-MPI/bin
> cp $WORK/MAQAO_HANDSON/bt/bt.slurm .
```

Launch job

```
> sbatch bt.slurm
> cat slurm-<jobid>.out
 BT-MZ Benchmark Completed.
 Class           =                        D
 Size            =          1632x 1216x 34
 Iterations      =                      250
 Time in seconds =                    61.43
 …
 Verification    =            SUCCESSFUL
```



Profiling bt-mz with MAQAO

Cédric VALENSI
Emmanuel OSERET

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.json .
> less config_bt_oneview_sbatch.json
```

```
“executable”: "bt-mz.D.x"
…
“scripts”: {
  ”command”: "sbatch <myscript>"
  ”files”: [ {“path”: "maqao_bt.slurm", “tag”: “myscript”} ]
…
“number_nodes”: 2
“number_processes_per_node”: 2
…
“mpi_command”: "srun"
…
“environment_variables”: {"name": "OMP_NUM_THREADS", "value": 56}
```

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
> cp $WORK/MAQAO_HANDSON/bt/maqao_bt.slurm .
> less maqao_bt.slurm
```

```
...
#SBATCH -N 2 <number_nodes>
#SBATCH --ntasks-per-node=2 <number_processes_per_node>
#SBATCH -c 56 <OMP_NUM_THREADS>
...
srun ./bt-mz.D.x
<mpi_command> <run_command>
...
```

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

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin
> module load maqao/2026.0.0
> maqao oneview --create-report=one --with-POP \
-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.

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

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin
> module load maqao/2026.0.0
> maqao oneview --create-report=one --with-scalability=strong \
  --with-POP -config=config_bt_oneview_sbatch.json -xp=ov_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.

(OPTIONAL) Review ONE View for interactive mode

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

```
> cp $WORK/MAQAO_HANDSON/bt/config_bt_oneview_interactive.json .
> less config_bt_oneview_interactive.json
```

```
“executable”: "bt-mz.D.x"
...
“number_nodes”: 2
“number_processes_per_node”: 2
...
“mpi_command”: "srun --reservation=POP3Tools -q gp_training -A nct_362 -N <number_nodes> --ntasks-per-node=<number_processes_per_node> -c <OMP_NUM_THREADS>"
...
“environment_variables”: {"name": "OMP_NUM_THREADS", "value": 56}
```

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

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin
> maqao OV -R1 -WS -c=config_bt_oneview_interactive.json \
-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 the experiment directory (set with -xp) and <*binary*> the name of the executable.

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

```
> tar czf $HOME/ov_html.tgz <exp-dir>/RESULTS/bt-mz.D.x_one_html
```

```
[LOCAL] scp nct0XXXX@glogin[12].bsc.es:ov_html.tgz .
[LOCAL] tar xf ov_html.tgz
[LOCAL] firefox <exp-dir>/RESULTS/bt-mz.D.x_one_html/index.html
```

A sample result directory is in **MAQAO_HANDSON/bt/offline.tgz**

Results can also be viewed directly on the console:

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

Display MAQAO ONE View results using sshfs

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

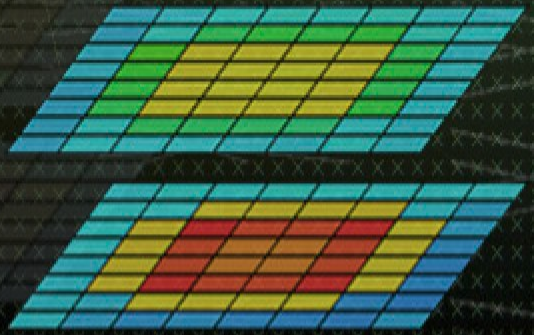
```
[LOCAL] sudo apt install sshfs
```

- Recommended to close a sshfs directory after use

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

Mount $WORK locally:

```
[LOCAL] mkdir mn5_work
[LOCAL] sshfs nct0XXXX@glogin[12].bsc.es:/gpfs/scratch/nct_362/users/nct0XXXX
mn5_work
[LOCAL] firefox mn5_work/NPB3.4-MZ-MPI/bin/ov_sbatch/RESULTS/bt-
mz.D.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 (GPP node)

Allocate a GPP socket for 1 hour (2 users per node)

```
> srun --reservation=POP3Tools -q gp_training -A nct_362 -t 60
--sockets-per-node=1 --pty bash
```

Load MAQAO and the latest GCC compiler

```
> module load maqao/2026.0.0
> module load gcc/14.1.0_binutils241
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

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

Run without MAQAO

```
> matmul_orig/matmul 200 10000
ns per inner loop iter.: 0.67
```

Analyse matrix multiply with ONE View

```
> maqao OV -R1 xp=ov_orig -- matmul_orig/matmul 200 10000
```

OR using configuration script:

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

Viewing results (HTML)

```
> tar czf $HOME/ov_orig.tgz ov_orig/RESULTS/matmul_orig_one_html
```

```
[LOCAL] scp nct0XXXX@glogin[12].bsc.es: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)	45.01
Max (Thread Active Time) (s)	43.61
Average Active Time (s)	43.61
Activity Ratio (%)	96.9
Average number of active threads	0.969
Affinity Stability (%)	100.0
Time in analyzed loops (%)	100.0
Time in analyzed innermost loops (%)	98.1
Time in user code (%)	100
Compilation Options Score (%)	50.0
Array Access Efficiency (%)	82.9

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.69
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	3.98
	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                                 +
+-------------------------------------------------------------------------------+
                                                  Potential Speedups
                                                  ------------------------------------------------
  Total Time:                          59.45 s    Perfect Flow Complexity:              1.00
  Max (Thread Active Time):            59.44 s    Perfect OpenMP/MPI/Pthread/TBB:       1.00
  Average Active Time:                 59.44 s    Perfect OpenMP/MPI/Pthread/TBB + L… : 1.00
  Activity Ratio:                      100.0 %    If No Scalar Integer:
  Average number of active threads:    1.000          Potential Speedup:                1.00
  Affinity Stability:                  100.0 %        Nb Loops to get 80%:              1
  Time spent in analyzed loops:        100.0 %    If FP Vectorized:
  Time spent in analyzed innermost loops: 99.0 %      Potential Speedup:                2.76
  Time spent in user code:             100 %          Nb Loops to get 80%:              1
  Compilation Options Score:           50         If Fully Vectorized:
  Array Access Efficiency:             83.3 %         Potential Speedup:                16.0
                                                      Nb Loops to get 80%:              1
                                                  If Only FP Arithmetic:
                                                      Potential Speedup:                1.00
                                                      Nb Loops to get 80%:              1
```

Viewing results (text)

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

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

   Loop Id | Module  | Source Location              | Coverage (%) |
  ---------+---------+------------------------------+--------------+
   1       | matm... | kernel_orig.c:9-10           | 99.64        |
   2       | matm... | kernel_orig.c:7-10           | 0.35         |
   3       | matm... | kernel_orig.c:6-10           | 0.02         |
```

Loop ID

Viewing CQA output (text)

```
> maqao OV -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).


Details
All VPU 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…
```

CQA output for the baseline kernel

Vectorization

Your loop is not vectorized. 16 data elements could be processed at once in vector registers.

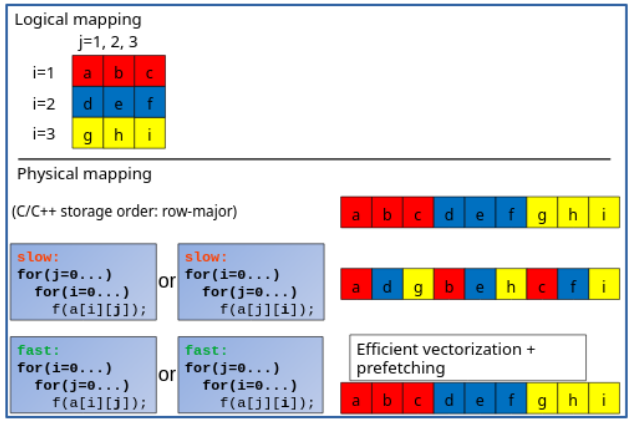
By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.19 cycles (16.00x speedup).

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

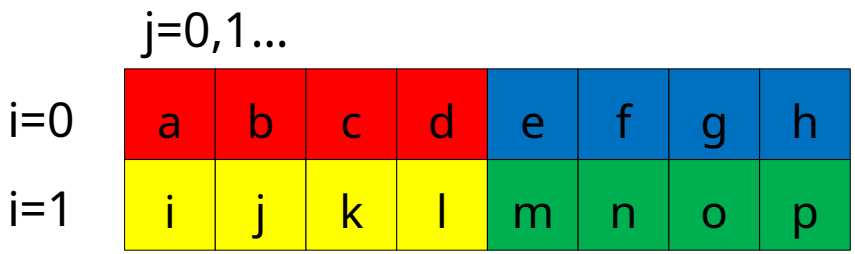
- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check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

Impact of loop permutation on data access

Logical mapping

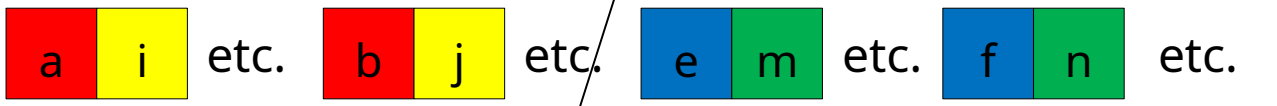


Efficient vectorization + prefetching

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
> make matmul_perm
```

Run without MAQAO

```
> matmul_perm/matmul 200 10000
ns per inner loop iter.: 0.13
```

Analyse matrix multiply with ONE View

```
> maqao OV -R1 xp=ov_perm -- matmul_perm/matmul 200 10000
```

OR using configuration script:

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

Loop permutation results

Faster (was 59.44)

Global Metri		
Total Time (s)		11.44
Max (Thread Active Time) (s)		11.06
Average Active Time (s)		11.06
Activity Ratio (%)		96.7
Average number of active threads		0.967
Affinity Stability (%)		96.6
Time in analyzed loops (%)		99.5
Time in analyzed innermost loops (%)		99.5
Time in user code (%)		94.4
Compilation Options Score (%)		50.0
Array Access Efficiency (%)		100

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.78
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.08
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.20
	Nb Loops to get 80%	2

Compilation Options

Source Object	Issue
▼ matmul	
▼ kernel.c	
○	-march=x86-64 is used but it should be replaced by a more architecture specific option or -march=native.
○	-funroll-loops is missing.

Let's try this

More efficient vectorization (was 16.00)

Loops/CQA output after loop permutation

Loop id	Source Location	Source Function	Level	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
1	matmul - kernel.c:24-25	kernel	Innermost	99.03	0	6.25
2	matmul - kernel.c:7-25 [...]	kernel	InBetween	0.95	25	14.06

Vectorization

Your loop is vectorized, but using only 128 out of 512 bits (SSE/AVX-128 instructions on AVX-512 processors).

By fully vectorizing your loop, you can lower the cost of an iteration from 1.17 to 0.29 cycles (4.00x speedup).

Details

All SSE/AVX instructions are used in vector version (process two or more data elements in vector registers). Since your execution units are vector units, only a fully vectorized loop can use their full power.

Workaround

- Recompile with march=sapphirerapids. CQA target is Sapphire_Rapids_8f (Intel(R) Xeon(R) Sapphire Rapids) but specialization flags are -march=x86-64
- Use vector aligned instructions:
  1. align your arrays on 64 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 64, size); }.
  2. inform your compiler that your arrays are vector aligned: if array 'foo' is 64 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 64) and use it instead of 'foo' in the loop.

Impacts of architecture specialization: AVX512 and FMA

- Vectorization
  - Default compilation ensures compatibility for any x86-64 processor (SSE2: 128 bits)
  - Extension to 256 or 512 bits with AVX512
- FMA : AxB+C for the cost of AxB

Analyse matrix multiply with unrolling and architecture specialisation

Compile architecture specialisation version of matrix multiply

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

Run without MAQAO

```
> matmul_perm_opt/matmul 200 10000
ns per inner loop iter.: 0.11
```

Analyse matrix multiply with ONE View

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

Loop permutation + (-march=sapphirerapids -funroll-loops)

Global Metrics		
Total Time (s)		9.67
Max (Thread Active Time) (s)		9.01
Average Active Time (s)		9.01
Activity Ratio (%)		93.1
Average number of active threads		0.931
Affinity Stability (%)		63.0
Time in analyzed loops (%)		99.4
Time in analyzed innermost loops (%)		84.3
Time in user code (%)		99.5
Compilation Options Score (%)		100
Array Access Efficiency (%)		81.3

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.08
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.16
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	2.23
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.50
	Nb Loops to get 80%	2

Faster (was 11.06)

Now 100% (-funroll-loops -march=native/sapphirerapids prev. missing)

Loops/CQA output after microarch specialization and loop unrolling

Loop id	Source Location	Source Function	Level	Max Thread Time / Walltime run_0 (%)	Exclusive Coverage run_0 (%)	Max Exclusive Time Over Threads run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vector Length Use (%)
5	matmul - kernel.c:24-25	kernel	Innermost	78.52	84.30	7.59	1	100	50
4	matmul - kernel.c:3-25 [...]	kernel	InBetween	14.01	15.04	1.36	1	44.44	25.58

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

512-bits vectorization

On some x86 processors supporting 512-bits vectorization, compilers are often too conservative and limit vectorization to 256 bits. Performance can then be improved by enforcing 512-bits vectorization, especially with many vectorized and high trip count loops. 512-bits vectorization performance overhead (compared to 256-bits) is generally lower on newer processors.

Workaround

Recompile with -mprefer-vector-width=512

Analyse matrix multiply with enforcing 512b vectorization

Compile architecture specialisation version of matrix multiply

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

Run without MAQAO

```
> matmul_perm_opt512/matmul 200 10000
ns per inner loop iter.: 0.11
```

Analyse matrix multiply with ONE View

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

After enforcing 512b vectorization

Global Metrics		
Total Time (s)		9.18
Max (Thread Active Time) (s)		8.82
Average Active Time (s)		8.82
Activity Ratio (%)		96.1
Average number of active threads		0.961
Affinity Stability (%)		96.1
Time in analyzed loops (%)		99.9
Time in analyzed innermost loops (%)		44.4
Time in user code (%)		99.9
Compilation Options Score (%)		100
Array Access Efficiency (%)		81.3

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.35
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.21
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.84
	Nb Loops to get 80%	2

Faster (was 9.01)

Roughly 2x better (was 2.23)

CQA output after enforcing 512b vectorization

Loop id	Source Location	Source Function	Level	Max Thread Time / Walltime run_0 (%)	Exclusive Coverage run_0 (%)	Max Exclusive Time Over Threads run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vector Length Use (%)
4	matmul - kernel.c:3-25 [...]	kernel	InBetween	53.29	55.44	4.89	1	32.43	35.56
5	matmul - kernel.c:24-25	kernel	Innermost	42.72	44.44	3.92	1	100	100

More time spent in the inbetween loop than in the innermost one:
512b vec. is too wide for input size and unroll factor

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).

Reports comparison

```
> maqao OV -CR inputs=ov_orig,ov_perm,ov_perm_opt,ov_perm_opt512
xp=ov_cmp [-include-detailed]
```

▼ **Compared Reports**

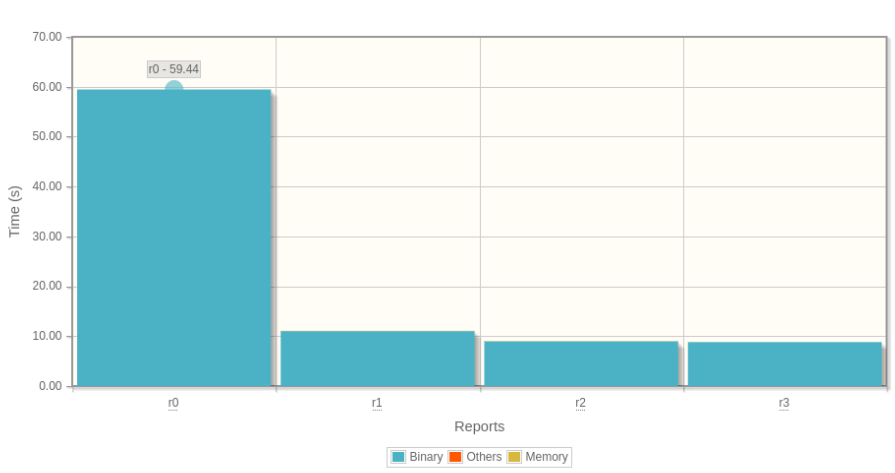
- r0: ov_orig
- r1: ov_perm
- r2: ov_perm_opt
- r3: ov_perm_opt512

Global Metrics

Metric		r0	r1	r2	r3
Total Time (s)		59.45	11.44	9.67	9.18
Max (Thread Active Time) (s)		59.44	11.06	9.01	8.82
Average Active Time (s)		59.44	11.06	9.01	8.82
Activity Ratio (%)		100.0	96.7	93.1	96.1
Average number of active threads		1.000	0.967	0.931	0.961
Affinity Stability (%)		100.0	96.6	63.0	96.1
Time in analyzed loops (%)		100.0	99.5	99.4	99.9
Time in analyzed innermost loops (%)		99.0	94.4	84.3	44.4
Time in user code (%)		100	99.5	99.5	99.9
Compilation Options Score (%)		50.0	50.0	100	100
Array Access Efficiency (%)		83.3	100	81.3	81.3
Potential Speedups					
Perfect Flow Complexity		1.00	1.00	1.00	1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00	1.00	1.00	1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00	1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.00	1.03	1.08	1.35
	Nb Loops to get 80%	1	1	1	1
FP Vectorised	Potential Speedup	2.76	1.78	1.16	1.01
	Nb Loops to get 80%	1	1	2	1
Fully Vectorised	Potential Speedup	16.0	4.08	2.23	1.21
	Nb Loops to get 80%	1	2	2	1
Only FP Arithmetic	Potential Speedup	1.00	1.20	1.50	1.84
	Nb Loops to get 80%	1	2	2	2

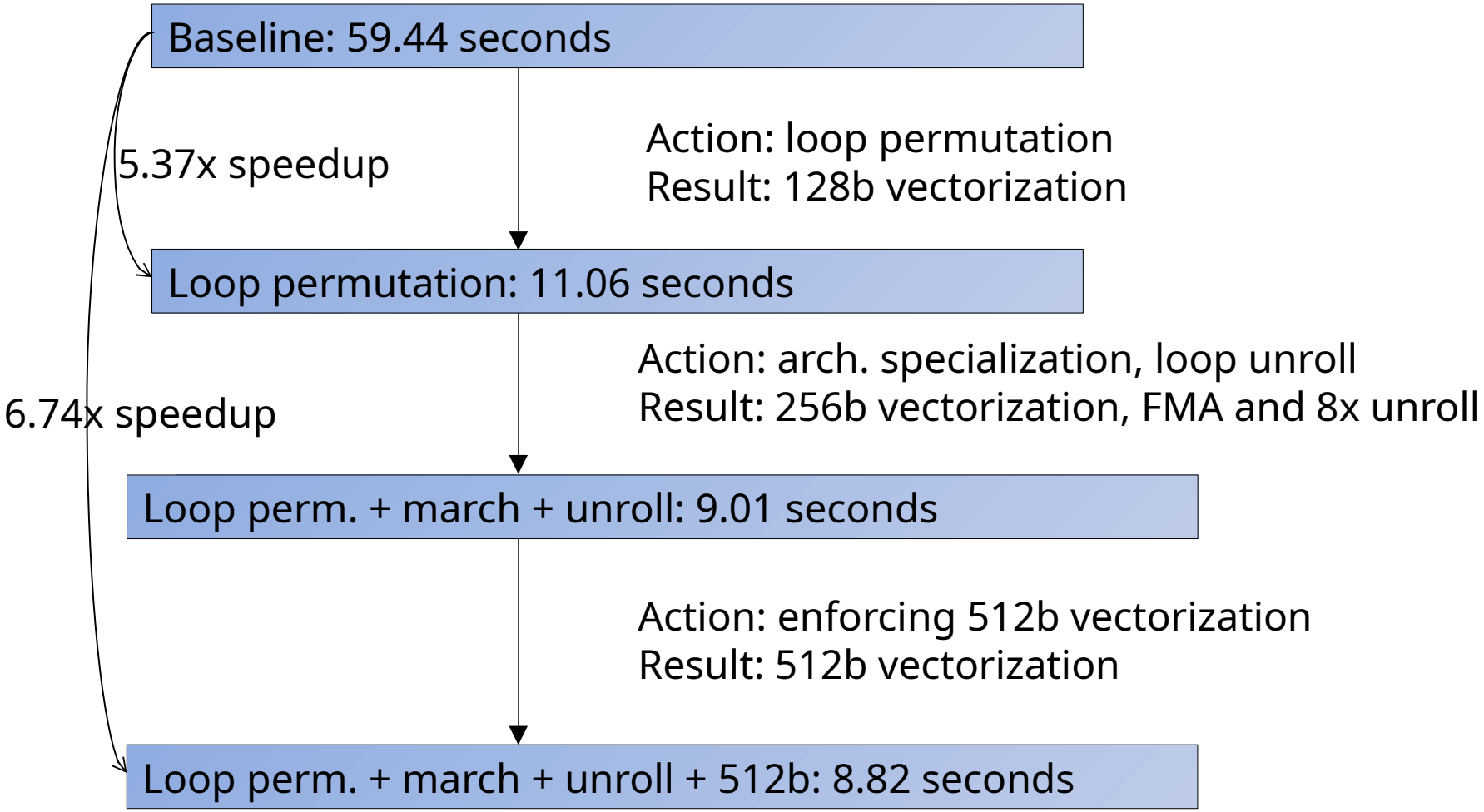
Application Categorization

Time



Coverage

Summary of optimizations and gains



Hydro example

Switch to the hydro handson folder

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

Load MAQAO and the latest vendor compiler (oneAPI 2025.2)

```
> module load maqao/2026.0.0
> module load oneapi/2025.2
```

Compile

```
> make
```

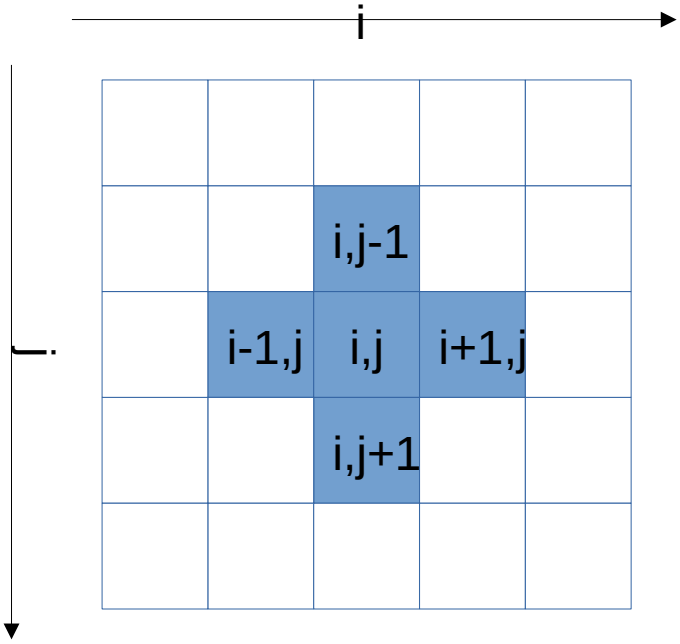
Hydro code

```
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



Hydro example : original version

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

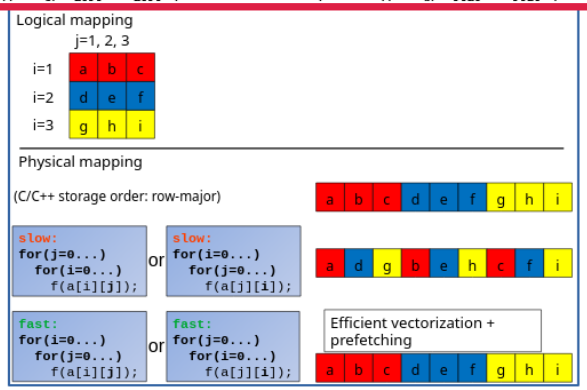
Global Metrics	
Total Time (s)	9.60
Max (Thread Active Time) (s)	9.30
Average Active Time (s)	9.30
Activity Ratio (%)	96.9
Average number of active threads	0.969
Affinity Stability (%)	96.4
Time in analyzed loops (%)	99.9
Time in analyzed innermost loops (%)	99.8
Time in user code (%)	100.0
Compilation Options Score (%)	16.7
Array Access Efficiency (%)	49.6

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.01
	Nb Loops to get 80%	3
FP Vectorised	Potential Speedup	1.64
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	13.7
	Nb Loops to get 80%	5
FP Arithmetic Only	Potential Speedup	1.06
	Nb Loops to get 80%	3

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 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 with the unroll (resp. unroll_and_jam) directive on top of the inner (resp. surrounding) loop. You can enforce an unroll factor: #pragma unroll_and_jam N, unroll_and_jam(N), unroll N or unroll(N)

Consider loop unrolling

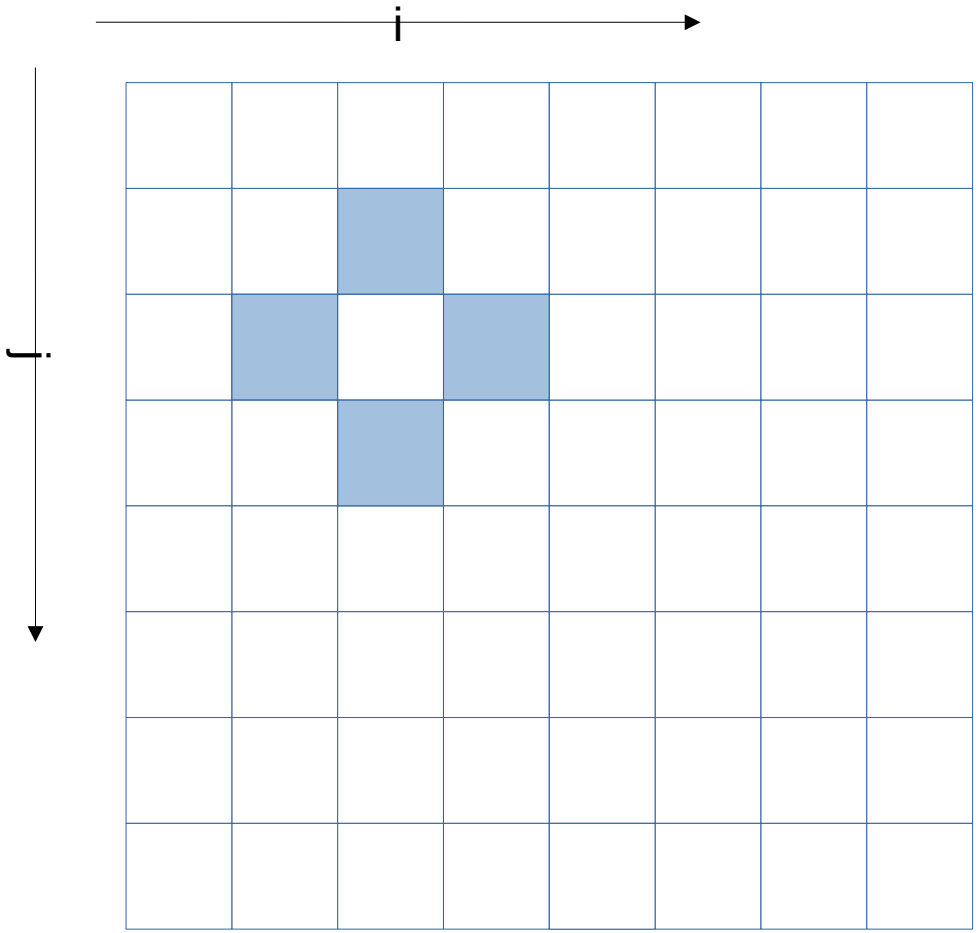
Hydro example : loop permutation

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

Global Metrics	
Total Time (s)	12.13
Max (Thread Active Time) (s)	11.89
Average Active Time (s)	11.89
Activity Ratio (%)	98.0
Average number of active threads	0.980
Affinity Stability (%)	98.0
Time in analyzed loops (%)	99.9
Time in analyzed innermost loops (%)	99.8
Time in user code (%)	100.0
Compilation Options Score (%)	16.7
Array Access Efficiency (%)	72.9

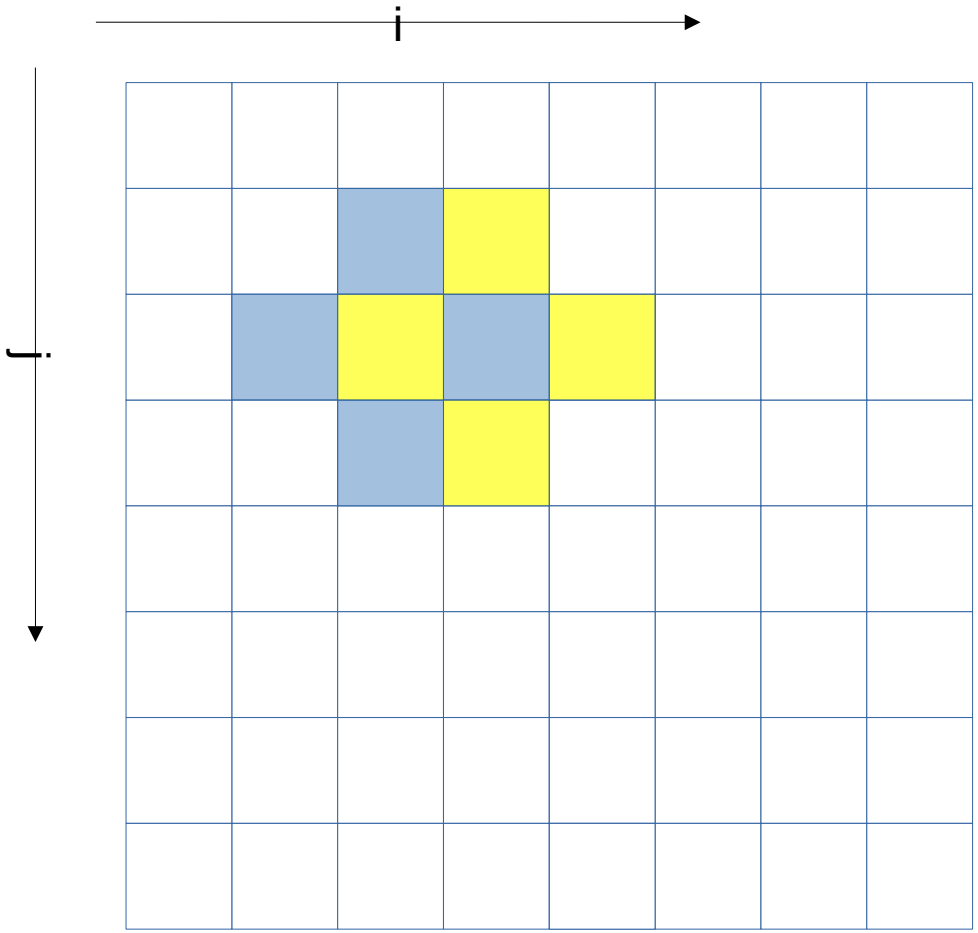
Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.01
	Nb Loops to get 80%	3
FP Vectorised	Potential Speedup	1.49
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	14.1
	Nb Loops to get 80%	5
FP Arithmetic Only	Potential Speedup	1.04
	Nb Loops to get 80%	3

Memory references reuse : 4x4 unroll footprint on loads



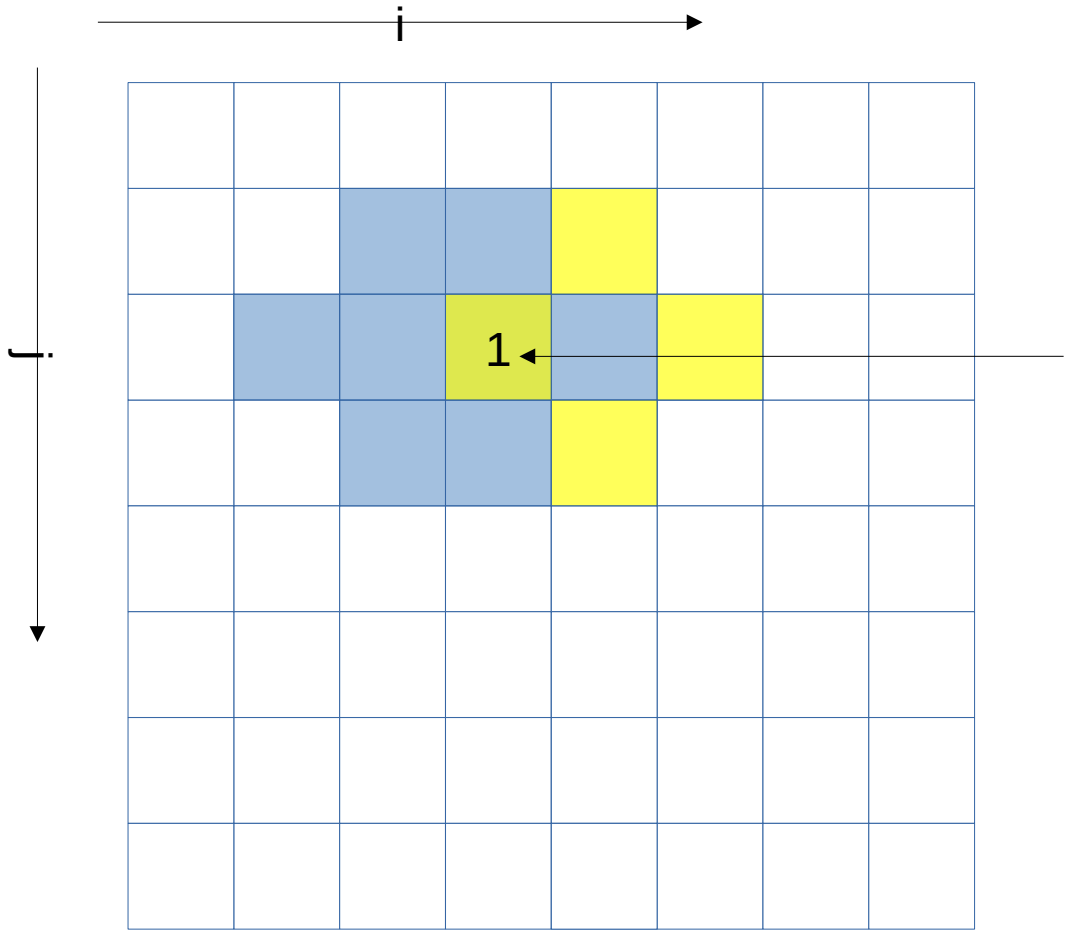
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

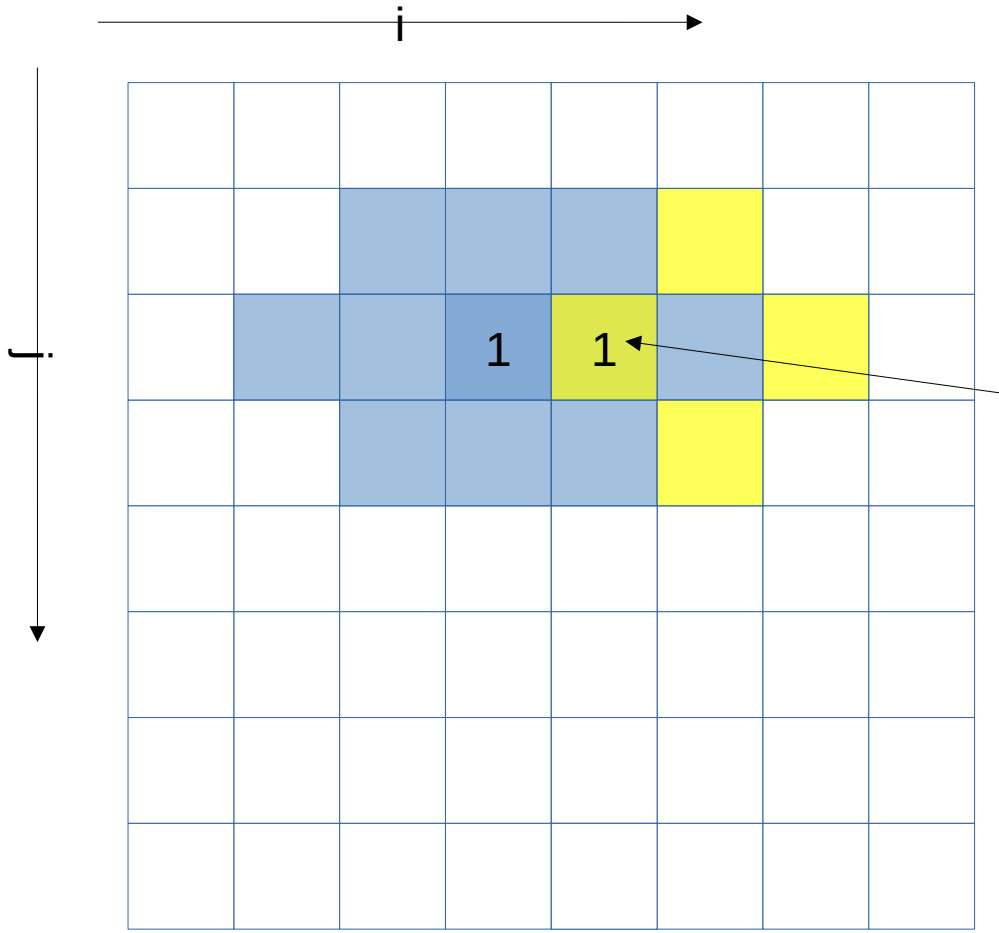
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

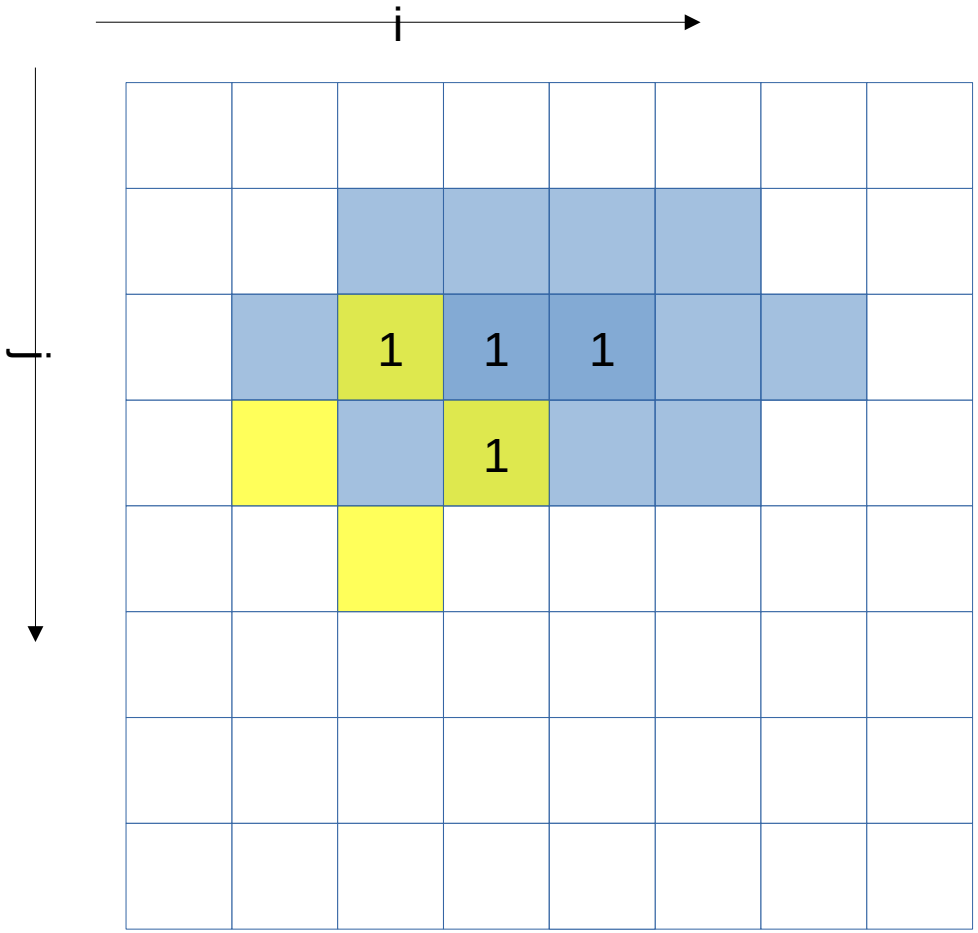
Memory references reuse : 4x4 unroll footprint on loads



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

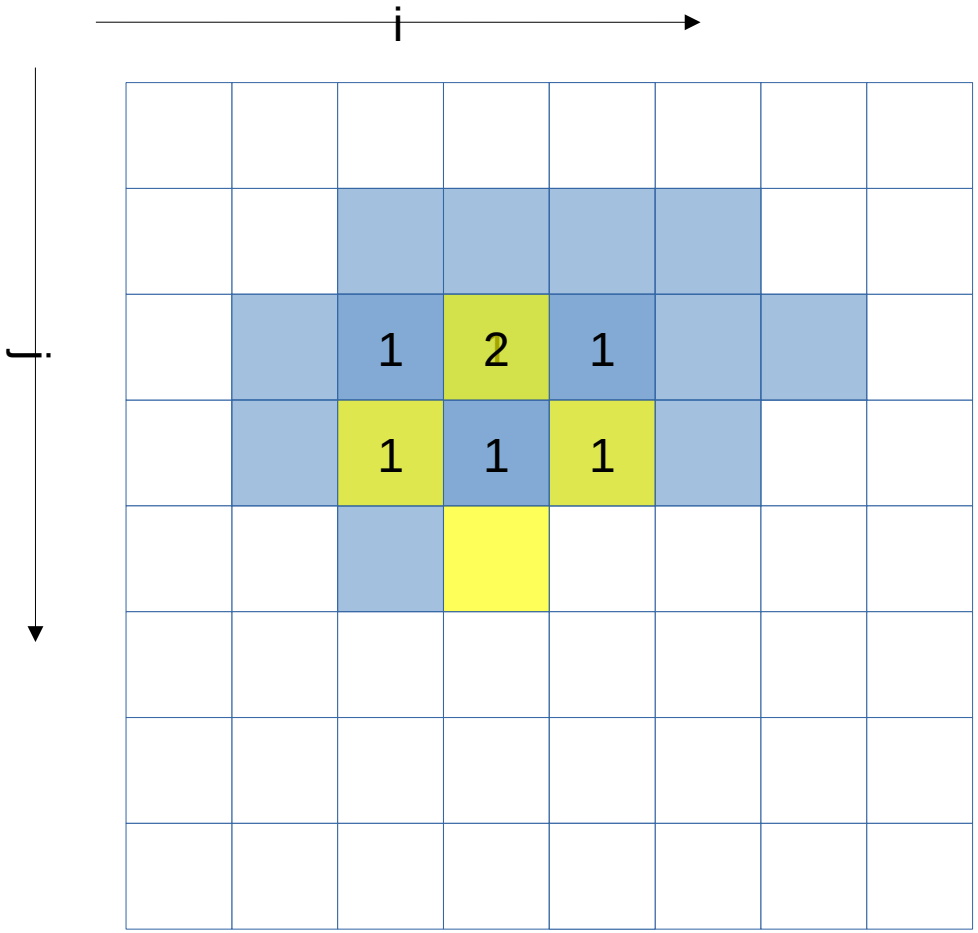


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

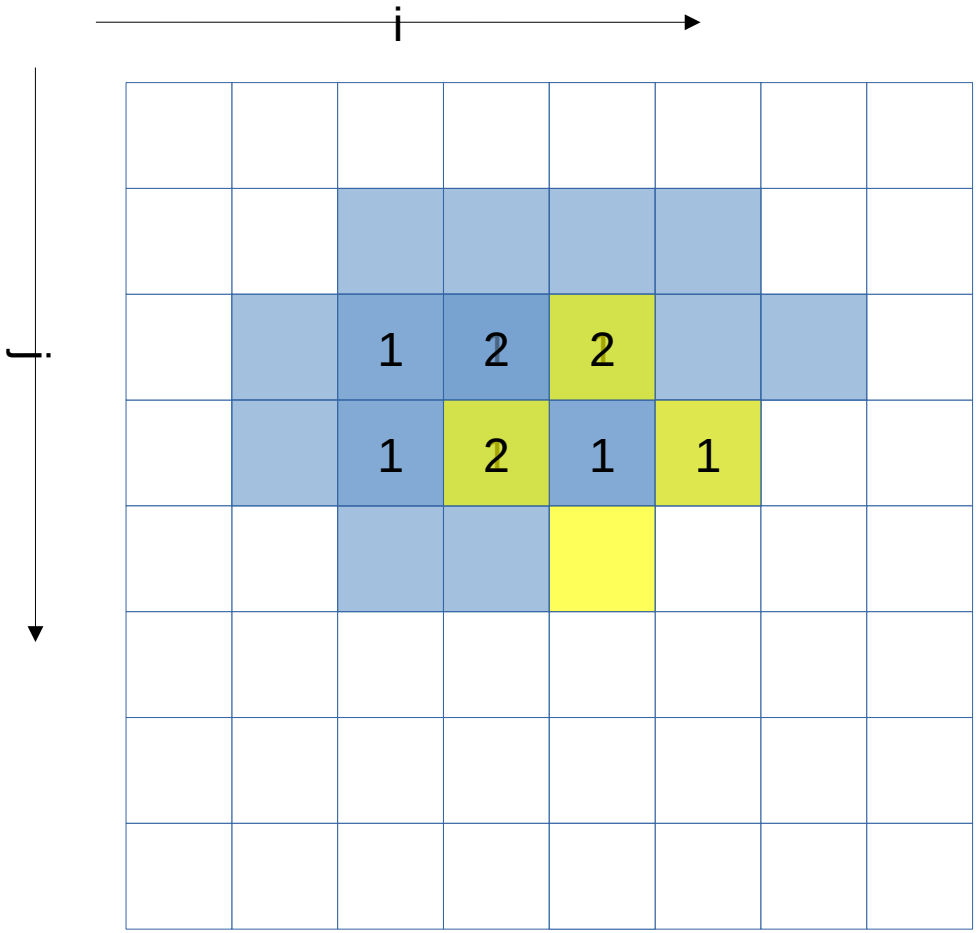


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

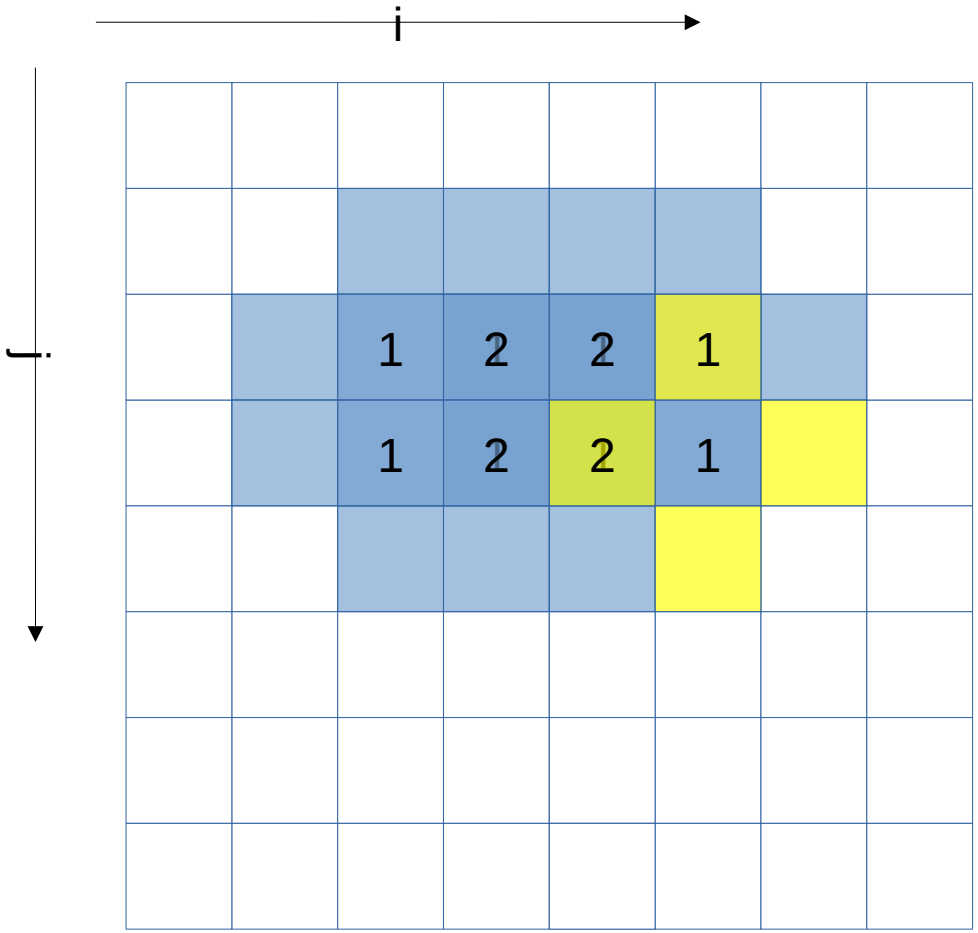


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

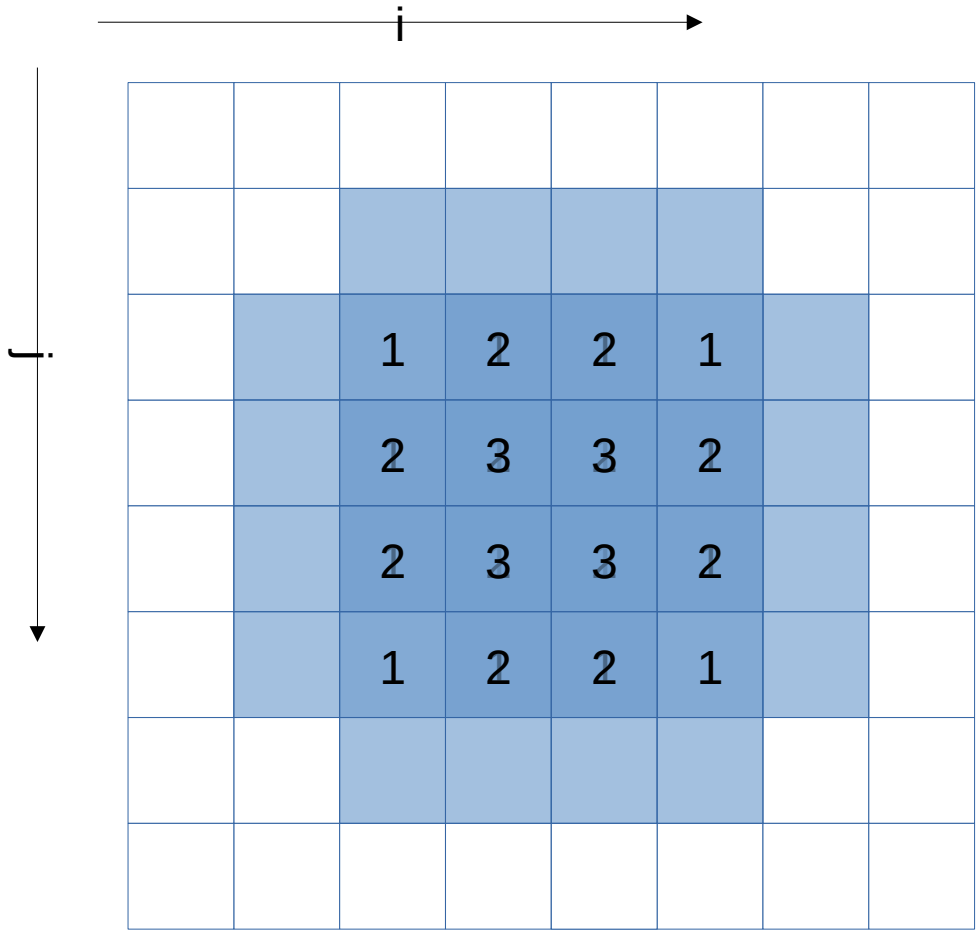


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

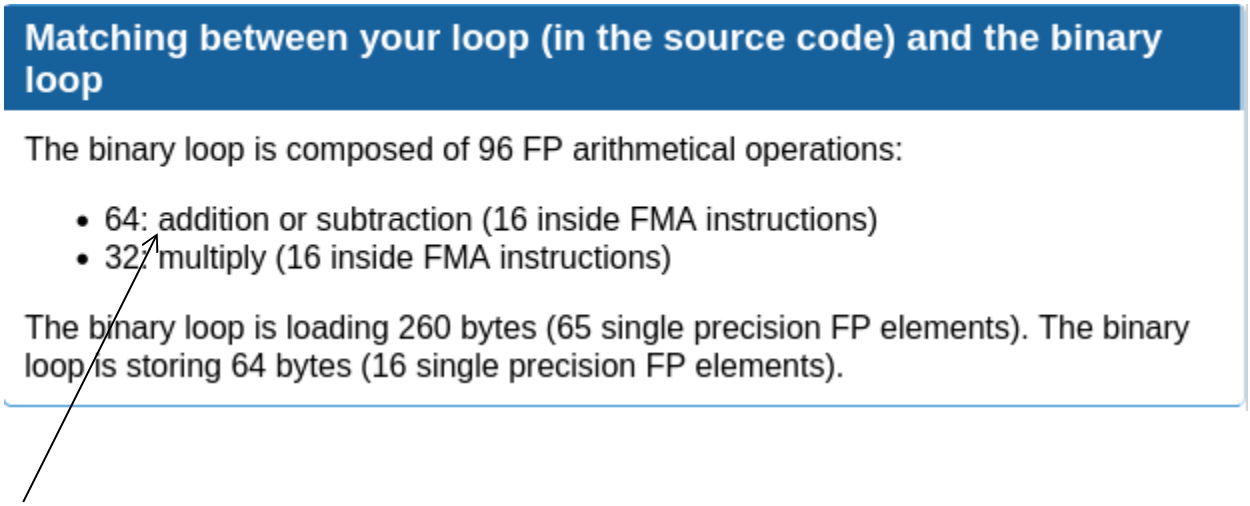
Hydro example : manual 4x4 unroll and jam

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

Global Metrics	
Total Time (s)	3.66
Max (Thread Active Time) (s)	3.31
Average Active Time (s)	3.31
Activity Ratio (%)	90.6
Average number of active threads	0.906
Affinity Stability (%)	90.5
Time in analyzed loops (%)	99.7
Time in analyzed innermost loops (%)	99.2
Time in user code (%)	99.7
Compilation Options Score (%)	16.7
Array Access Efficiency (%)	48.9

Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00
Perfect OpenMP/MPI/Pthread/TBB + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	3
FP Vectorised	Potential Speedup	1.69
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	10.9
	Nb Loops to get 80%	5
FP Arithmetic Only	Potential Speedup	1.18
	Nb Loops to get 80%	3

CQA output for unrolled kernel



4x4 Unrolling were applied

Hydro example : comparison

```
> maqao OV -CR inputs=ov_orig,ov_perm,ov_unroll \
  xp=ov_cmp [-include-detailed]
```

▼ **Compared Reports**

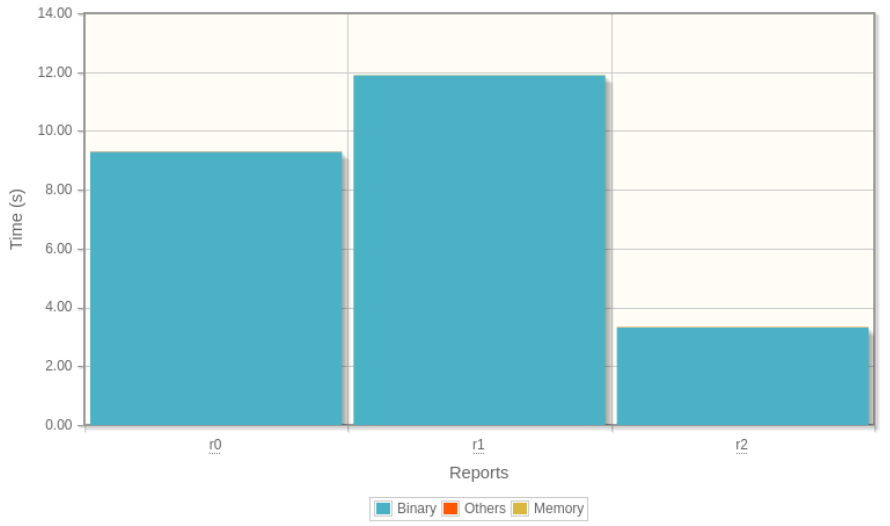
- r0: ov_orig
- r1: ov_perm
- r2: ov_unroll

Global Metrics

Metric		r0	r1	r2
Total Time (s)		9.60	12.13	3.66
Max (Thread Active Time) (s)		9.30	11.89	3.31
Average Active Time (s)		9.30	11.89	3.31
Activity Ratio (%)		96.9	98.0	90.6
Average number of active threads		0.969	0.980	0.906
Affinity Stability (%)		96.4	98.0	90.5
Time in analyzed loops (%)		99.9	99.9	99.7
Time in analyzed innermost loops (%)		99.8	99.8	99.2
Time in user code (%)		100.0	100.0	99.7
Compilation Options Score (%)		16.7	16.7	16.7
Array Access Efficiency (%)		49.6	72.9	48.9
Potential Speedups				
Perfect Flow Complexity		1.00	1.00	1.00
Perfect OpenMP/MPI/Pthread/TBB		1.00	1.00	1.00
Perfect OpenMP/MPI/Pthread/TBB + 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%	3	3	3
FP Vectorised	Potential Speedup	1.64	1.49	1.69
	Nb Loops to get 80%	2	2	2
Fully Vectorised	Potential Speedup	13.7	14.1	10.9
	Nb Loops to get 80%	5	5	5
Only FP Arithmetic	Potential Speedup	1.06	1.04	1.18
	Nb Loops to get 80%	3	3	3

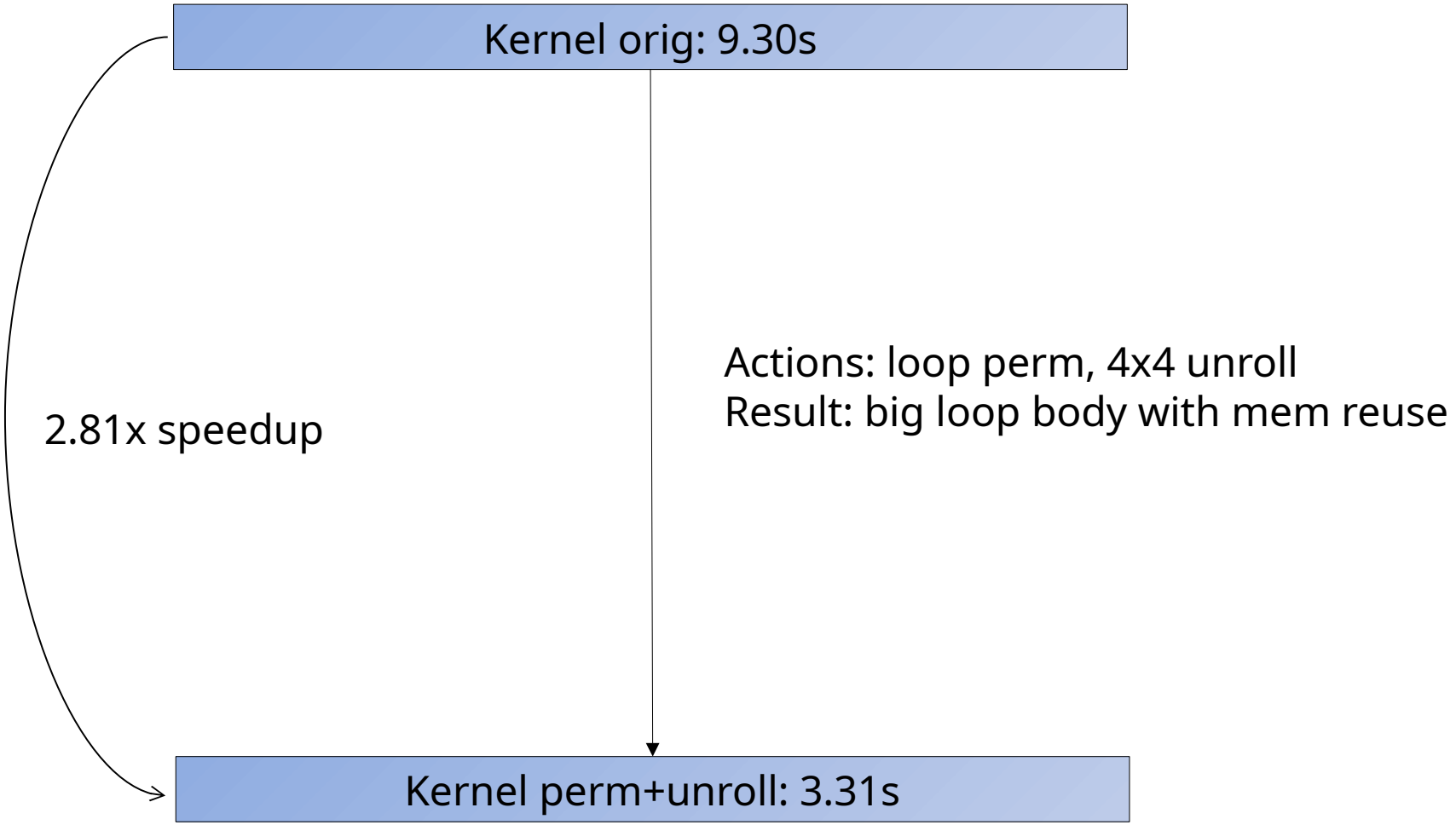
Application Categorization

Time



Coverage

Summary of optimizations and gains



More sample codes

More codes to study with MAQAO in

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