

MAQAO Hands-on exercises

Profiling bt-mz
Scalability profiling of lulesh
Optimising a code

Setup (reminder)

Login to the cluster

```
> ssh <username>@jureca.fz-juelich.de
```

(If not already done) Configure environment

```
> module load Architecture/KNL  
> module load Intel IntelMPI  
> jutil env activate -p cjzam11 -A jzam11
```

Setup for MAQAO hands-ons

Load MAQAO environment

```
> module use /p/scratch/share/VI-HPS/JURECA/mf  
> module load maqao/2.9.0
```

Copy handson material to your working directory

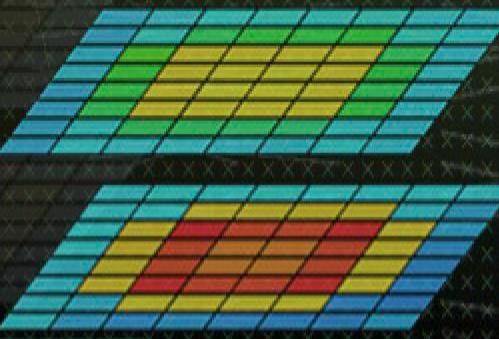
```
> cd $SCRATCH/$USER  
> tar xvf /p/scratch/share/VI-HPS/examples/MAQAO_HANDSON_TW33.tgz
```

Copy lulesh binaries to your working directory

```
> cp -r /p/scratch/share/VI-HPS/examples/LULESH $SCRATCH/$USER
```

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

```
> mkdir scratch_remote  
> sshfs <login>@jureca.fz-juelich.de:/p/scratch/cjzam11/<login> scratch_remote
```



Profiling bt-mz with MAQAO

Cédric Valensi

Setup ONE View for batch mode

The ONE View configuration file must contain all variables for executing the application.

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

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

binary = "bt-mz_C.8"
...
batch_script = "bt_maqao.sbatch"
...
batch_command = "sbatch <batch_script>"
...
number_processes = 8
...
number_nodes = 2
...
number_tasks_nodes = 4
...
mpi_command = "srun -n <number_processes>"
...
omp_num_threads = 16
```

Review jobscrip for use with ONE View

All variables in the jobscrip defined in the configuration file must be replaced with their name from it.

Retrieve jobscrip modified for ONE View from the MAQAO_HANDSON directory.

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

```
...  
#SBATCH --nodes=2<number_nodes>  
#SBATCH --ntasks-per-node=4<number_tasks_nodes>  
#SBATCH --cpus-per-task=16<omp_num_threads>  
...  
#SBATCH --disable-perfparanoid  
...  
export OMP_NUM_THREADS=16<omp_num_threads>  
...  
srun -n $PROCS $EXE  
<mpi_command> <run_command>  
...
```

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

Launch ONE View

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

The -xp parameter allows to set the path to the experiment directory, where ONE View stores the analysis results and where the reports will be generated.

If -xp is omitted, the experiment directory will be named maqao_<timestamp>.

WARNING:

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

Display MAQAO ONE View results

The HTML files are located in `<exp-dir>/RESULTS/<binary>_one_html`, where `<exp-dir>` is the path of the experiment directory (set with -xp) and `<binary>` the name of the executable.

```
> firefox maqao_bt/RESULTS/bt-mz_C.8_one_html/index.html
```

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

```
> tar -zcf $HOME/bt_html.tgz maqao_bt/RESULTS/bt-mz_C.8_one_html
```

```
> scp <login>@jureca.fz-juelich.de:bt_html.tgz .
```

```
> tar xf bt_html.tgz
```

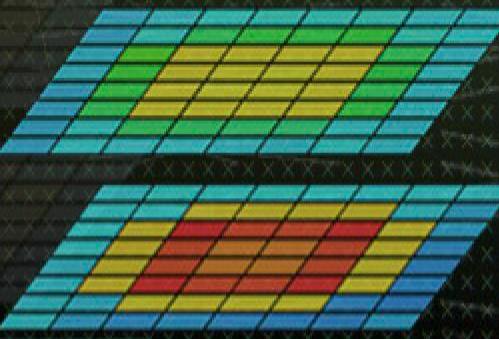
```
> firefox maqao_bt/RESULTS/bt-mz_C.8_one_html/index.html
```

A sample result directory is in

`$SCRATCH/$USER/MAQAO_HANDSON/bt/bt_html_example.tgz`

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

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



Scalability profiling of lulesh with MAQAO

Cédric Valensi

Setup ONE View for scalability analysis

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

```
> cd $SCRATCH/$USER/LULESH/mpit+omp
> cp $SCRATCH/$USER/MAQAO_HANDSON/lulesh/config_maqao_lulesh.lua .
> less config_maqao_lulesh.lua
```

```
binary = "./lulesh2.0-intel"
...
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 = "srun -n <number_processes>"
...
omp_num_threads = 1
...
scalability_params = {
    {nb_processes = 1, nb_threads = 8, nb_nodes = 1, ..., run_command = nil, ...},
    {nb_processes = 8, nb_threads = 1, nb_nodes = 1, ..., run_command = "<binary> -i 10 -p -s 65", ...},
    {nb_processes = 8, nb_threads = 1, nb_nodes = 2, ..., run_command = "<binary> -i 10 -p -s 65", ...},
    {nb_processes = 8, nb_threads = 4, nb_nodes = 2, ..., run_command = "<binary> -i 10 -p -s 65", ...},
    {nb_processes = 8, nb_threads = 8, nb_nodes = 2, ..., run_command = "<binary> -i 10 -p -s 65", ...},
}
```

Review jobscrip for use with ONE View

All variables in the jobscrip defined in the configuration file must be replaced with their name from it.

Retrieve jobscrip modified for ONE View from the MAQAO_HANDSON directory.

```
> cd $SCRATCH/$USER/LULESH/mpi+omp
> cp $SCRATCH/$USER/MAQAO_HANDSON/lulesh/job_lulesh_maqao.sbatch .
> less job_lulesh_maqao.sbatch
```

```
...
#SBATCH --nodes=2<number_nodes>
#SBATCH --cpus-per-tasks=4<omp_num_threads>
...
#SBATCH --disable-perfparanoid
...
srun ./lulesh2.0-intel -i 10 -p -s 130
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on lulesh (scalability mode)

Launch ONE View

```
> cd $SCRATCH/$USER/LULESH/mpitomp  
> maqao onerview --create-report=one --with-scalability=on \  
--config=config_maqao_lulesh.lua -xp=maqao_lulesh
```

Execution will be longer!

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

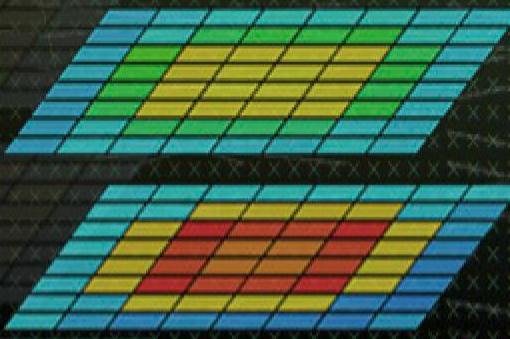
```
> firefox maqao_lulesh/RESULTS/lulesh2.0-intel_one_html/index.html
```

Or

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

```
> scp <login>@jureca.fz-juelich.de:lulesh_html.tgz .  
> tar xf lulesh_html.tgz  
> firefox maqao_lulesh/RESULTS/lulesh2.0-intel_one_html/index.html
```

A sample result directory is in `MAQAO_HANDSON/lulesh/lulesh_html_example.tgz`



Optimising a code with MAQAO

Emmanuel OSERET

Matrix Multiply code

```
void kernel0 (int n,
              float a[n][n],
              float b[n][n],
              float c[n][n]) {
    int i, j, k;

    for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            c[i][j] = 0.0f;
            for (k=0; k<n; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

“Naïve” dense matrix multiply implementation in C

Preparing interactive session (KNL)

Request interactive session (exclusive access to one node)

```
> jutil env activate -p cjzam11 -A jzam11  
> srun -N 1 --exclusive --disable-perfparanoid -t 30 \  
-p booster --reservation tw_booster --pty bash
```

Load KNL and MAQAO environment

```
> module use /p/scratch/share/VI-HPS/JURECA/mf  
> module load maqao/2.9.0
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

```
> cd $SCRATCH/$USER/MAQAO_HANDSON/matmul  
> make matmul_orig
```

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

```
> ./matmul_orig 390 300  
cycles per FMA: 6.26
```

Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \  
binary=./matmul_orig run-command="<binary> 390 300" \  
xp=ov_orig
```

OR, using a configuration script:

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

Viewing results (HTML)

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

```
> scp <login>@jureca.fz-juelich.de:ov_orig.tgz .
```

```
> tar xf ov_orig.tgz
```

```
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics	
Total Time (s)	72.39
Time in loops (%)	98.17
Compilation Options	binary: -march=(target) is missing. -funroll-loops is missing.
Flow Complexity	1.00
Array Access Efficiency (%)	66.58
Potential Speedup	1.00
Clean	
Nb Loops to get 80%	1
FP Vectorised	
Potential Speedup	2.30
Nb Loops to get 80%	1
Fully Vectorised	
Potential Speedup	12.55
Nb Loops to get 80%	1

Viewing results (text)

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

```
+-----+
+                               1.2  -  Global Metrics
+-----+
Total Time:                      72.39 s
Time spent in loops:             98.17 %
Compilation Options:             binary: -march=native is missing. -funroll-loops
is missing.
Flow Complexity:                 1.00
Array Access Efficiency:          66.58 %
If Clean:
  Potential Speedup:            1.00
  Nb Loops to get 80%:          1
If FP Vectorized:
  Potential Speedup:            2.30
  Nb Loops to get 80%:          1
If Fully Vectorized:
  Potential Speedup:            12.55
  Nb Loops to get 80%:          1
```

Viewing results (text)

```
+-----+
+          1.3 - Potential Speedups
+-----+
+-----+
If Clean:
Number of loops | 1      | 2      |
Cumulated Speedup | 1.0009 | 1.0009 |
Top 5 loops:
matmul_orig - 2:    1.0009
matmul_orig - 1:    1.0009

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

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



Loop ID

Viewing CQA output (text)

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

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 6.00 to 0.38 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.
 - (...)



Loop ID

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 6.00 to 0.38 cycles (16.00x speedup).

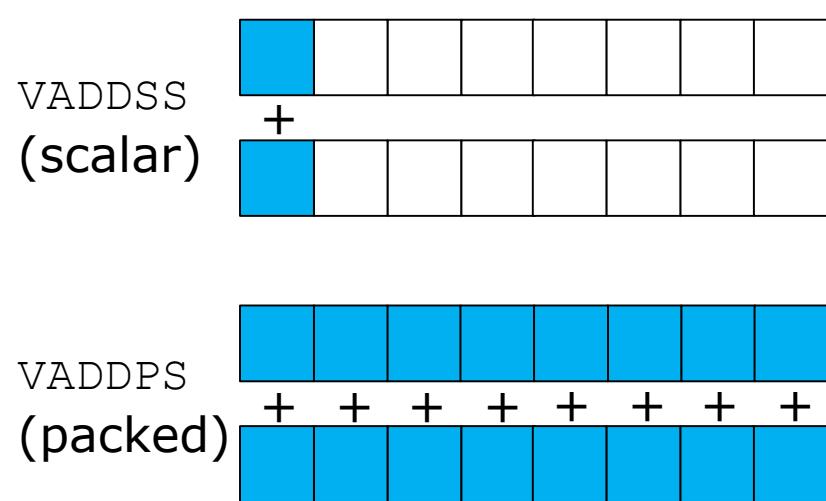
Details

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: $\text{for}(i) \text{ for}(j) a[j][i] = b[j][i]$; (slow, non stride 1) $\Rightarrow \text{for}(i) \text{ for}(j) a[i][j] = b[i][j]$; (fast, stride 1)
 - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): $\text{for}(i) a[i].x = b[i].x$; (slow, non stride 1) $\Rightarrow \text{for}(i) a.x[i] = b.x[i]$; (fast, stride 1)

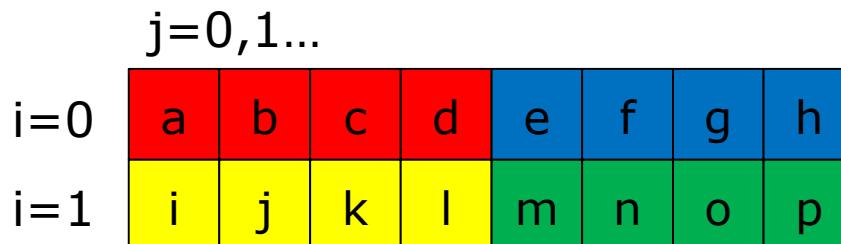
Vectorization (summing elements):



- 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 kernell1 (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 $SCRATCH/$USER/MAQAO_HANDSON/matmul  
> make matmul_perm  
> ./matmul_perm 390 300  
cycles per FMA: 1.22
```

Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \  
binary=./matmul_perm run-command="<binary> 390 300" \  
xp=ov_perm
```

OR using configuration script:

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

Viewing new results

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

(Or download the ov_perm/RESULTS/matmul_perm_one_html folder locally and open ov_perm/RESULTS/matmul_perm_one_html/index.html)

Loop permutation results

Global Metrics		
Total Time (s)	14.66	Faster (was 72.39)
Time in loops (%)	97.03	Let's try this
Compilation Options	binary: -march=(target) is missing. -funroll-loops is missing.	
Flow Complexity	1.00	
Array Access Efficiency (%)	75.01	
	Potential Speedup	1.06
Clean	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.84
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	3.95
	Nb Loops to get 80%	2

CQA output after loop permutation

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

Vectorization

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

By fully vectorizing your loop, you can lower the cost of an iteration from 4 to 1.00 cycles (4.00x speedup).

Details

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

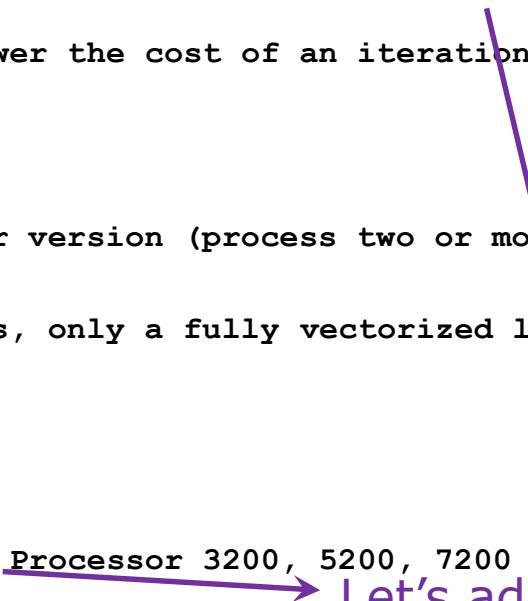
Since your execution units are vector units, only a fully vectorized loop can use their full power.

Workaround

- Recompile with `march=knl`.

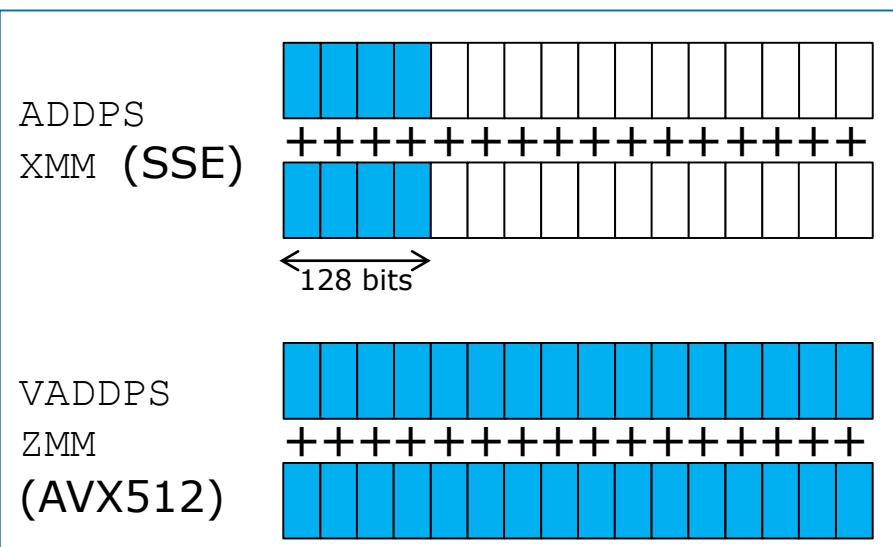
CQA target is Xeon_Phi_KNL (Intel Xeon Phi Processor 3200, 5200, 7200 Series) but specialization flags are `-march=x86-64`

- (...)



Impacts of architecture specialization: vectorization and FMA

- Vectorization
 - SSE instructions (SIMD 128 bits) used on a processor supporting AVX512 ones (SIMD 512 bits)
 - => 75% efficiency loss
- FMA
 - Fused Multiply-Add ($A + BC$)
 - Intel architectures: supported on MIC/KNC/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 architecture specialisation

Compile architecture specialisation version of matrix multiply

```
> cd $SCRATCH/$USER/MAQAO_HANDSON/matmul  
> make matmul_perm_opt  
> ./matmul_perm_opt 390 300  
cycles per FMA: 0.45
```

Analyse matrix multiply with ONE View

```
> maqao oneview create-report=one \  
binary=./matmul_perm_opt run-command="<binary> 390 300" \  
xp=ov_perm_opt
```

OR using configuration script:

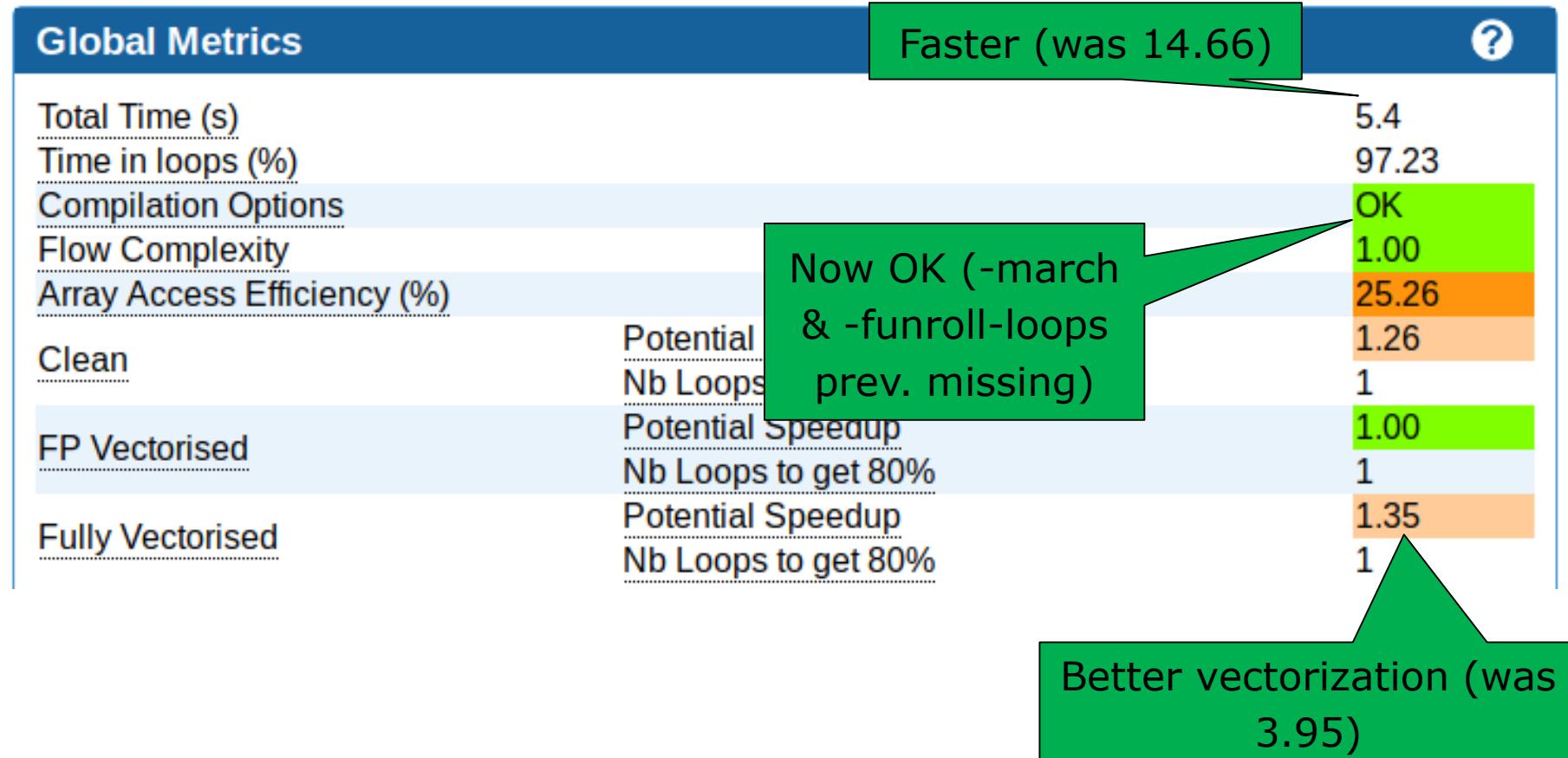
```
> maqao oneview create-report=one c=ov_perm_opt.lua xp=ov_perm_opt
```

Viewing new results:

```
> maqao oneview create-report=one xp=ov_perm_opt \  
output-format=text --text-global | less
```

(or download the ov_perm/RESULTS/matmul_perm_opt_one_html folder locally and open `index.html` in your browser)

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



CQA output with (-march=knl -funroll-loops)

Workaround

Use vector aligned instructions:

1. align your arrays on 64 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 64, size); }.
2. inform your compiler that your arrays are vector aligned: if array 'foo' is 64 bytes-aligned, define a pointer 'p_foo' as __builtin_assume_aligned (foo, 64) and use it instead of 'foo' in the loop.

Let's switch to the next proposal: vector aligned instructions

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

Using aligned arrays in matrix multiply

Compile aligned array version of matrix multiply

```
> cd $SCRATCH/$USER/MAQAO_HANDSON/matmul  
> make matmul_align
```

Checking aligned version:

```
> ./matmul_align 390 300  
Cannot call kernel on matrices with size%16 != 0 (data not  
aligned on 64B boundaries)  
Aborted
```

=> Alignment imposes restrictions on input parameters.

```
> ./matmul_align 400 300  
driver.c: Using posix_memalign instead of malloc  
cycles per FMA: 0.22
```

Analysing matrix multiply with aligned arrays

Analyse matrix multiply with ONE View

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

OR using configuration script:

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

Viewing new results

```
> maqao oneview create-report=one xp=ov_align \
output-format=text --text-global | less
```

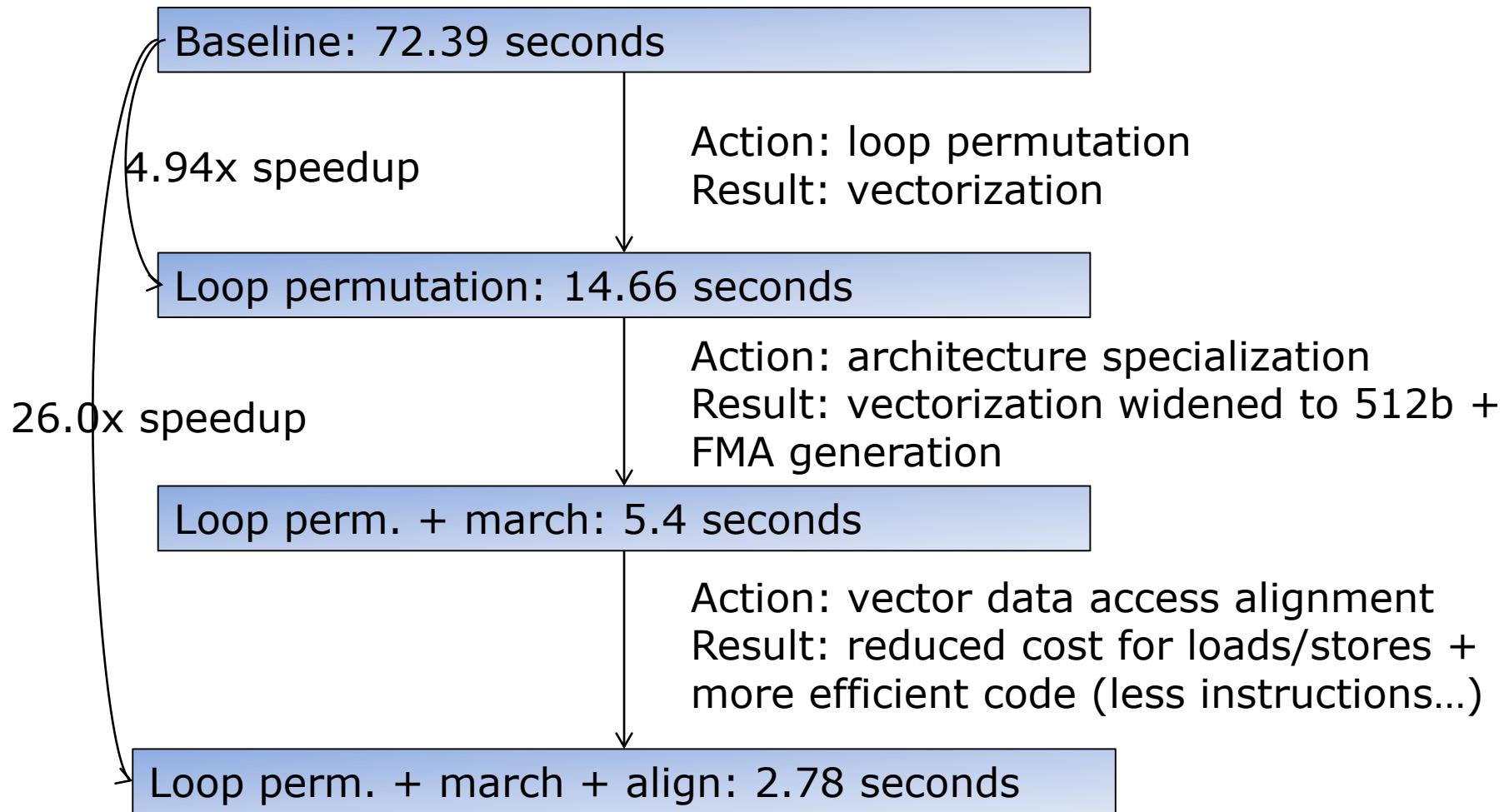
(Or download the ov_align/RESULTS/matmul_align_one_html folder locally and open ov_align/RESULTS/matmul_align_one_html/index.html in your browser)

Vector-aligning array accesses

Global Metrics		?
Total Time (s)	2.78	
Time in loops (%)	95.68	
Compilation Options	OK	
Flow Complexity	1.00	
Array Access Efficiency (%)	25.32	
Clean	Potential Speedup Nb Loops to get 80%	1.09 1
FP Vectorised	Potential Speedup Nb Loops to get 80%	1.00 1
Fully Vectorised	Potential Speedup Nb Loops to get 80%	1.14 1

Extra speedup
(was 5.4)

Summary of optimizations and gains



Hydro example

(If necessary) Request interactive session and load KNL/MAQAO environment

```
> jutil env activate -p cjzam11 -A jzam11
> srun -N 1 --exclusive --disable-perfparanoid -t 30 \
-p booster --reservation tw_booster --pty bash
> module use /p/scratch/share/VI-HPS/JURECA/mf
> module load maqao/2.9.0
```

Switch to the hydro handson folder

```
> cd $SCRATCH/$USER/MAQAO_HANDSON/hydro
```

Make sure Intel compiler environment is loaded

```
> module load Intel
```

Compile and execute sample program

```
> make
```

```
> ./hydro_k0 300 30
```

```
Cycles per element for solvers: 4799.16
```

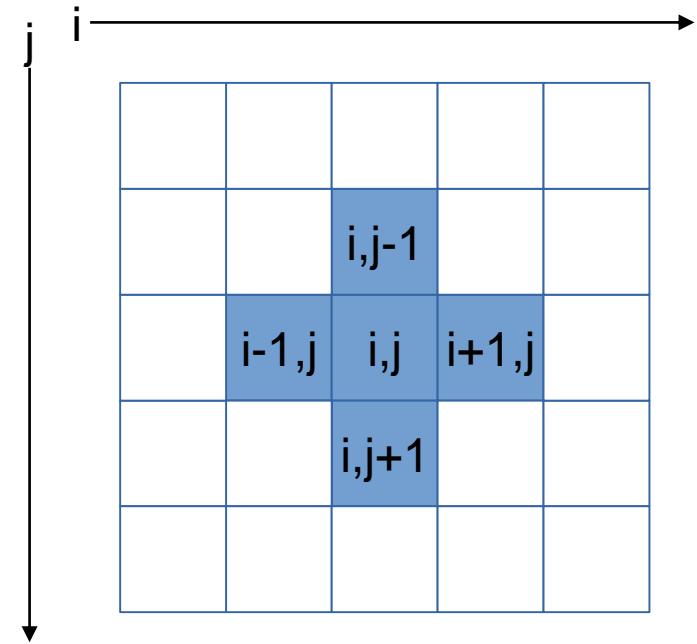
Hydro code

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

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

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

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



Running and analyzing kernel0 (icc -O3 -xHost)

- Profile with MAQAO

```
> maqao oneview create-report=one xp=ov_k0 c=ov_k0.lua
```

- Display results

```
> maqao oneview create-report=one xp=ov_k0 \
output-format=text --text-global | less
```

+-----+ + 1.2 - Global Metrics +-----+	
Total Time:	10.02 s
Time spent in loops:	99.46 %
Compilation Options:	OK
Flow Complexity:	1.09
Array Access Efficiency:	25.24 %

Loops Index

Filters

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

Loop id	Source Location	Source Function	Coverage (%)	Level
140	hydro_k0:kernel.c:104-110	project	25.74	Innermost
95	hydro_k0:kernel.c:104-108	c_velocitySolver	21.55	Innermost
89	hydro_k0:kernel.c:104-108	c_velocitySolver	21.55	Innermost
58	hydro_k0:kernel.c:104-108	c_densitySolver	21.45	Innermost
51	hydro_k0:kernel.c:15-292	c_densitySolver	1.4	Innermost
149	hydro_k0:kernel.c:15-371	project	1.3	Innermost
77	hydro_k0:kernel.c:15-292	c_velocitySolver	1.2	Innermost
79	hydro_k0:kernel.c:15-292	c_velocitySolver	1.1	Innermost
142	hydro_k0:kernel.c:380-383	project	1.1	Innermost
130	hydro_k0:kernel.c:15-318	c_velocitySolver	0.3	Innermost

The kernel routine, linearSolver, was inlined in caller functions. Moreover, there is direct mapping between source and binary loop. Consequently the 4 hot loops are identical and only one needs analysis.

Loop Id: 140

Module: hydro_k0

Source: kernel.c:104-110

Coverage: 25.74%

Source Code



CQA



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

gain potential hint expert

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 7.00 to 6.00 cycles (1.17x speedup).

Workaround

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

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 7.00 to 0.44 cycles (16.00x speedup).

Details

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

Workaround

- Try another compiler or update/tune your current one:
 - use the vec-report option to understand why your loop was not vectorized. If "existence of vector dependences", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.

CQA output for kernel0

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

gain potential hint expert

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

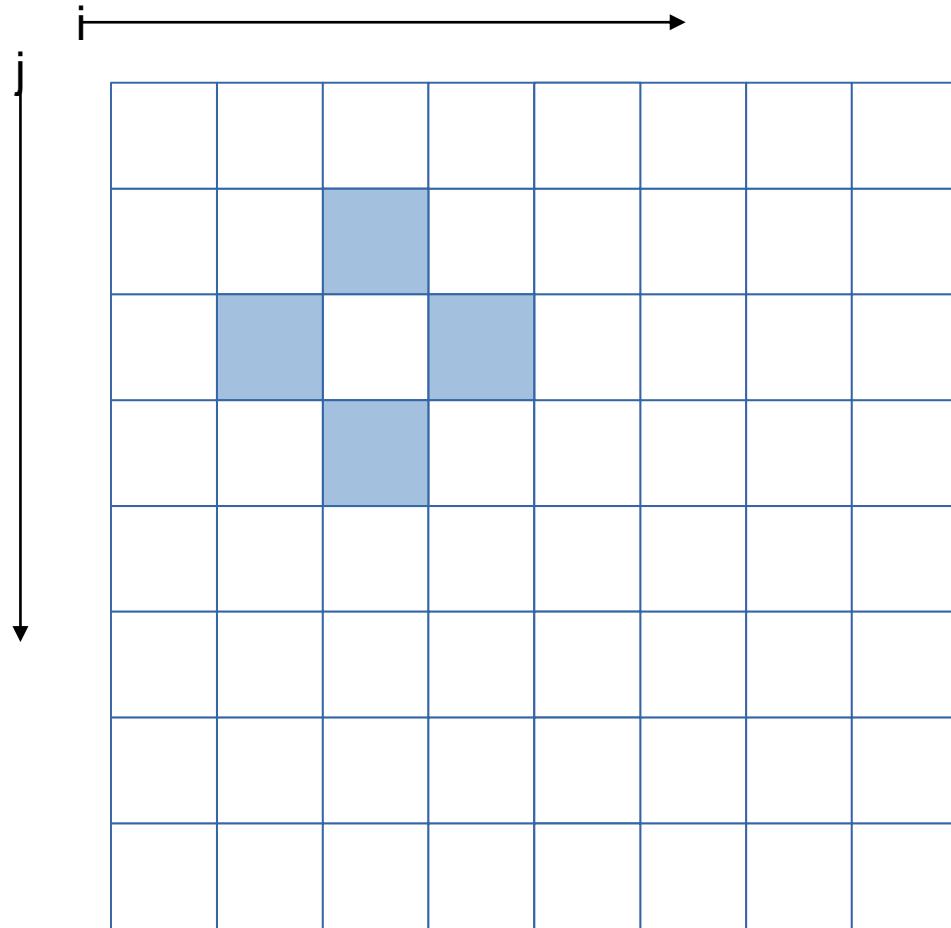
Loop is potentially data access bound.

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).

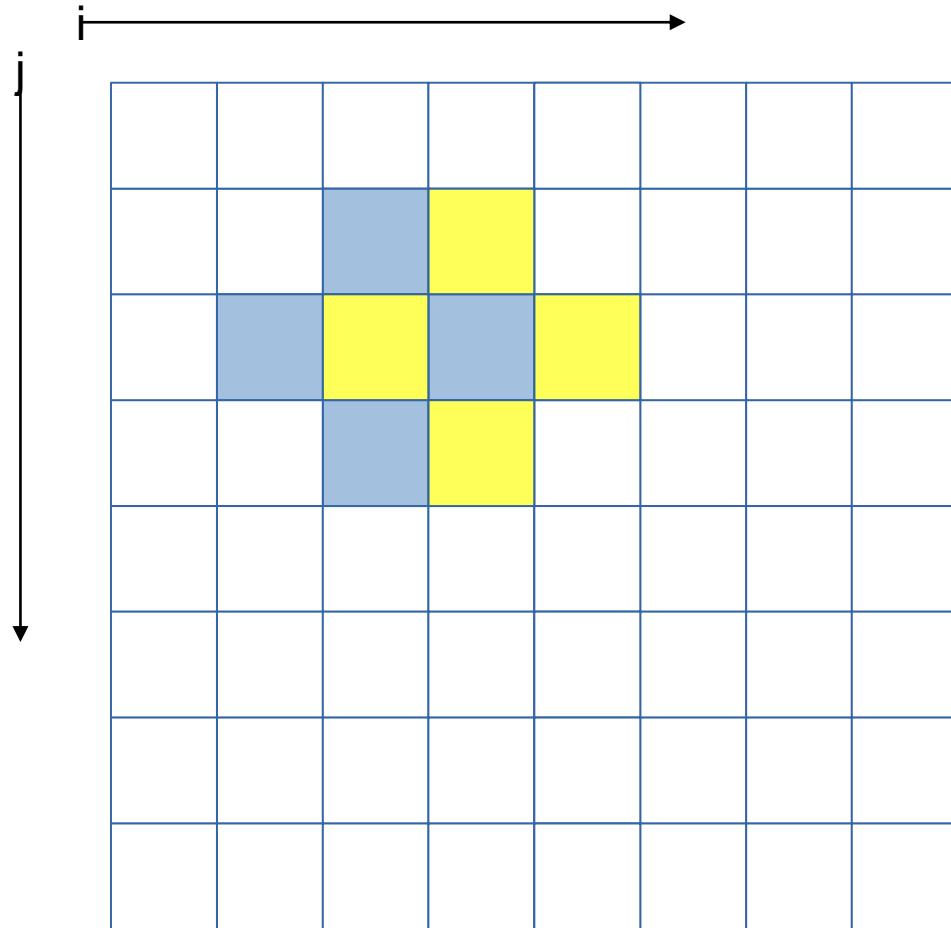
Unrolling is generally a good deal: fast to apply and often provides gain.
Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



LINEAR_SOLVER(i+0, j+0)

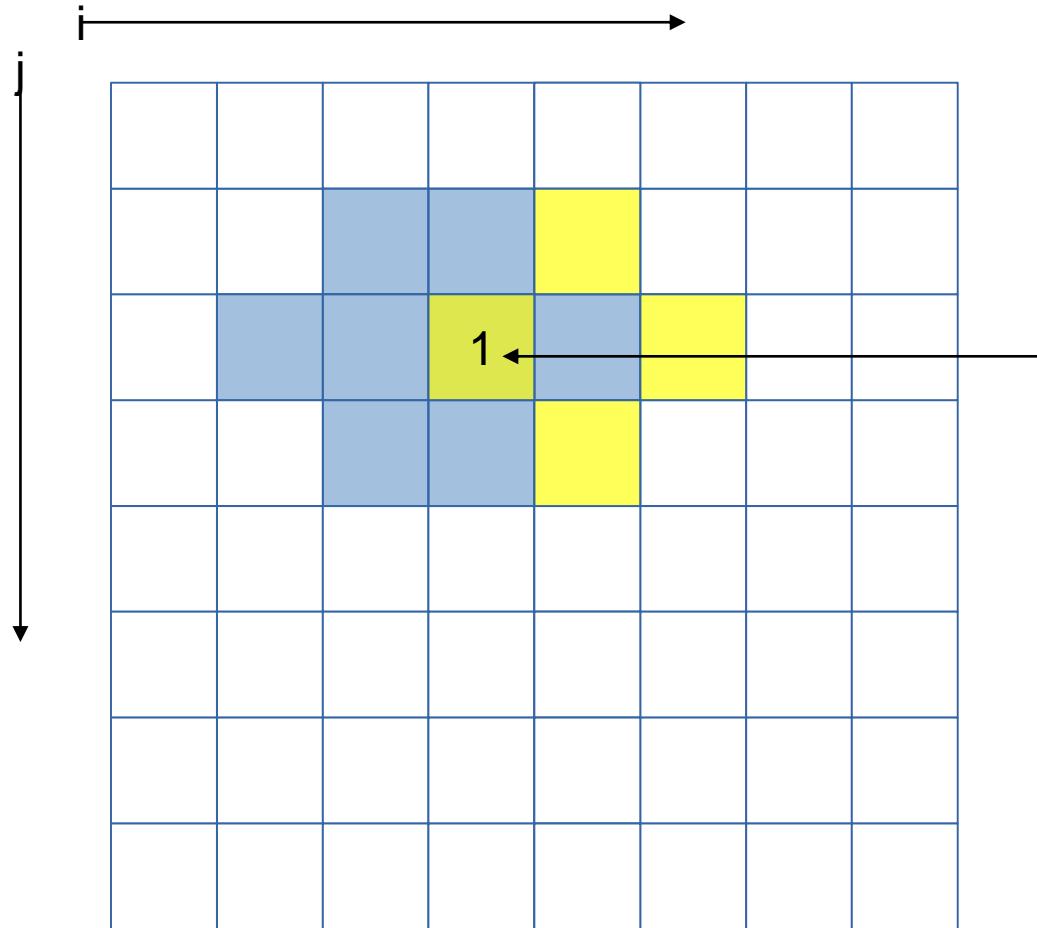
Memory references reuse : 4x4 unroll footprint on loads



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

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

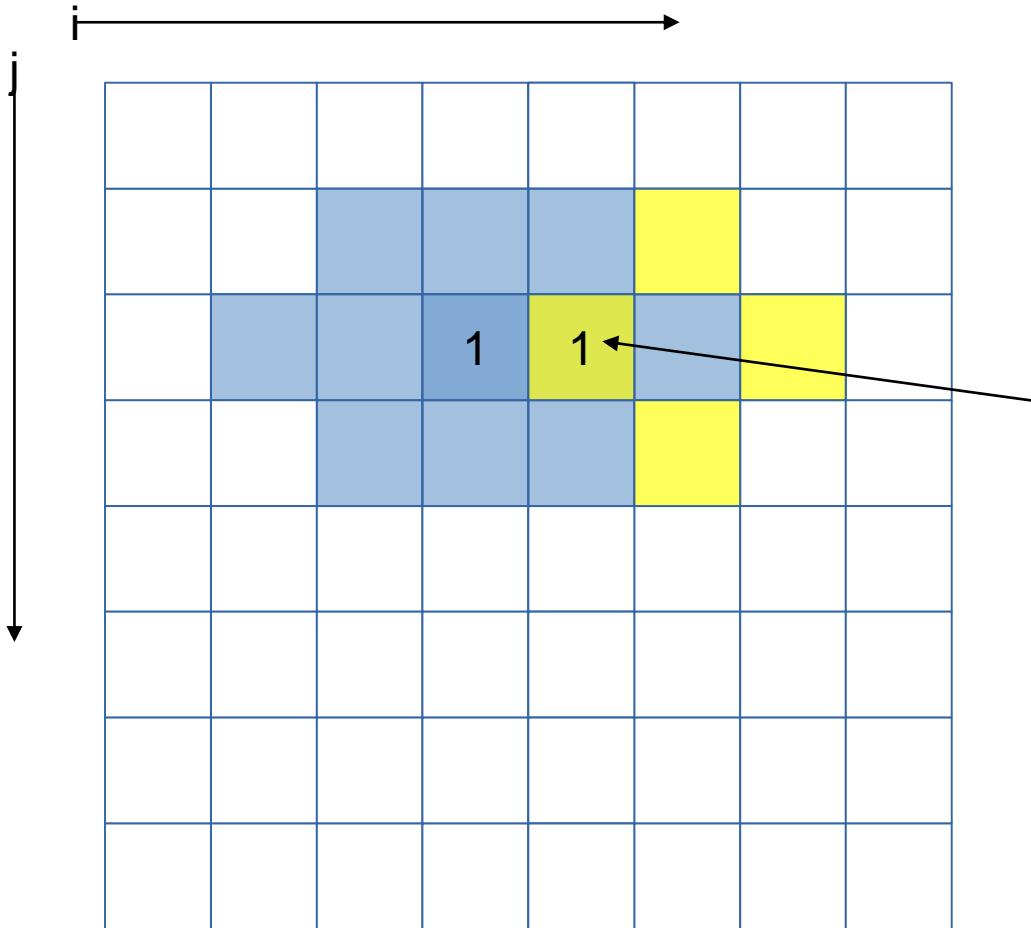
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

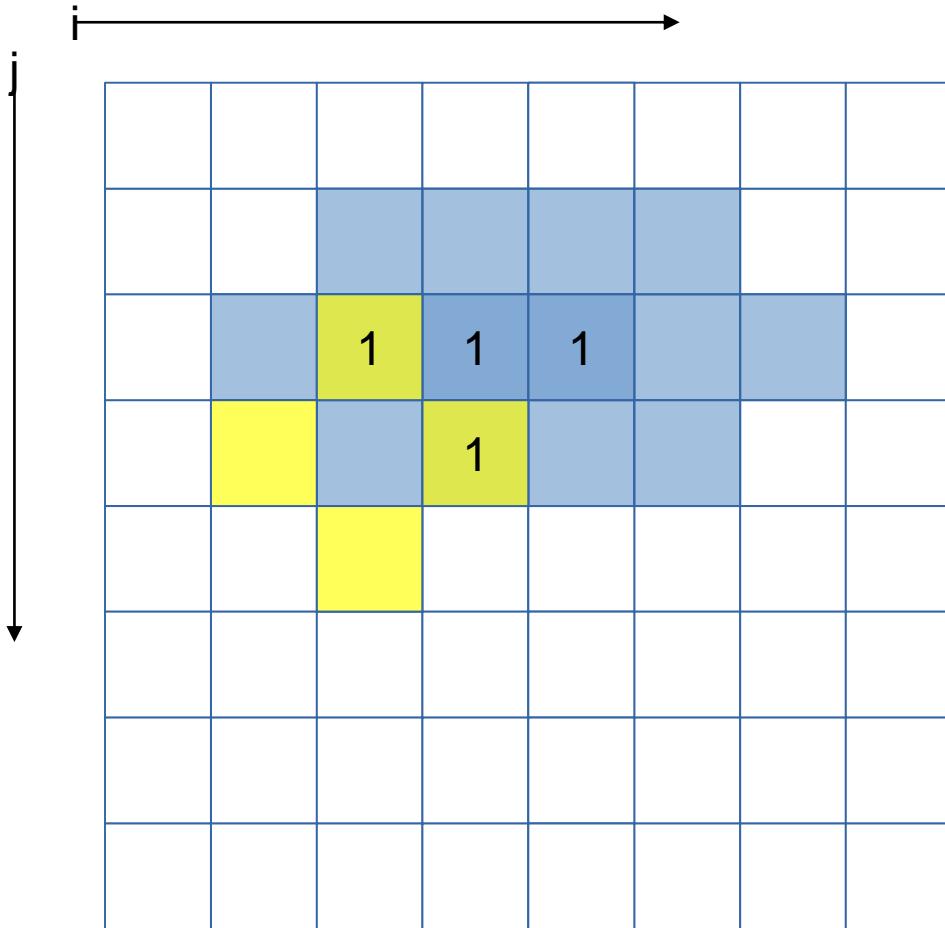
Memory references reuse : 4x4 unroll footprint on loads



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

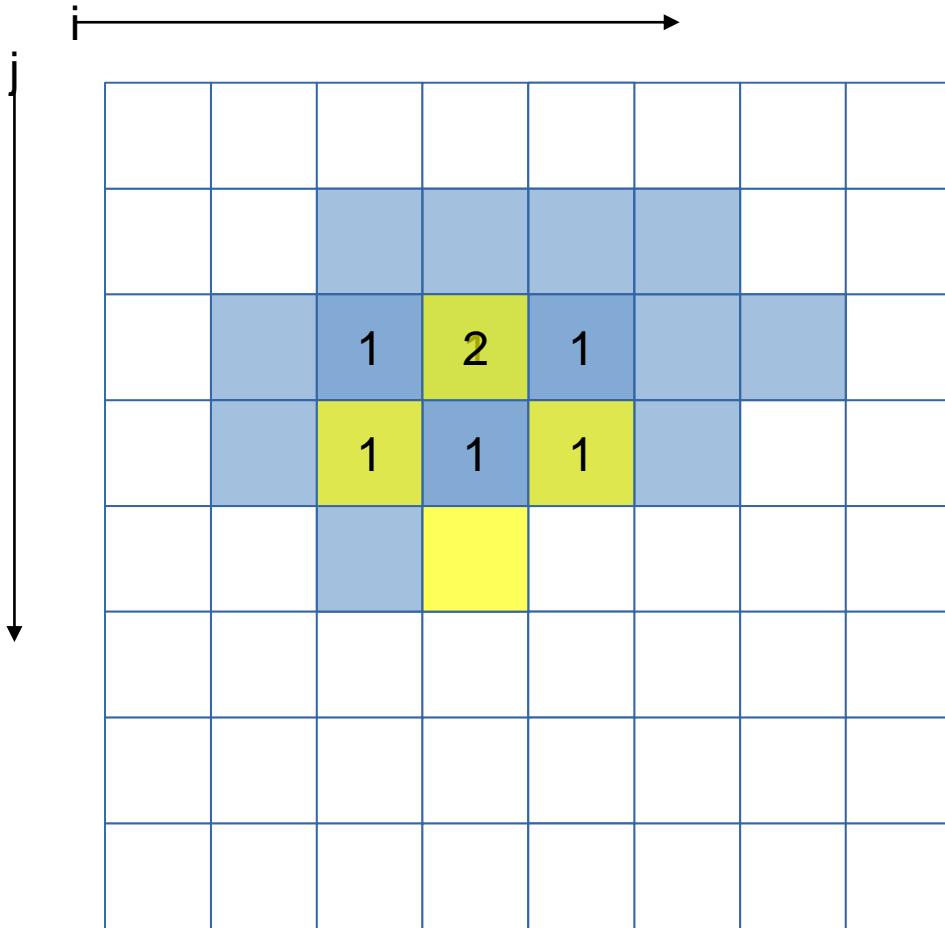


`LINEAR_SOLVER(i+0, j+0)`
`LINEAR_SOLVER(i+1, j+0)`
`LINEAR_SOLVER(i+2, j+0)`
`LINEAR_SOLVER(i+3, j+0)`

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

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

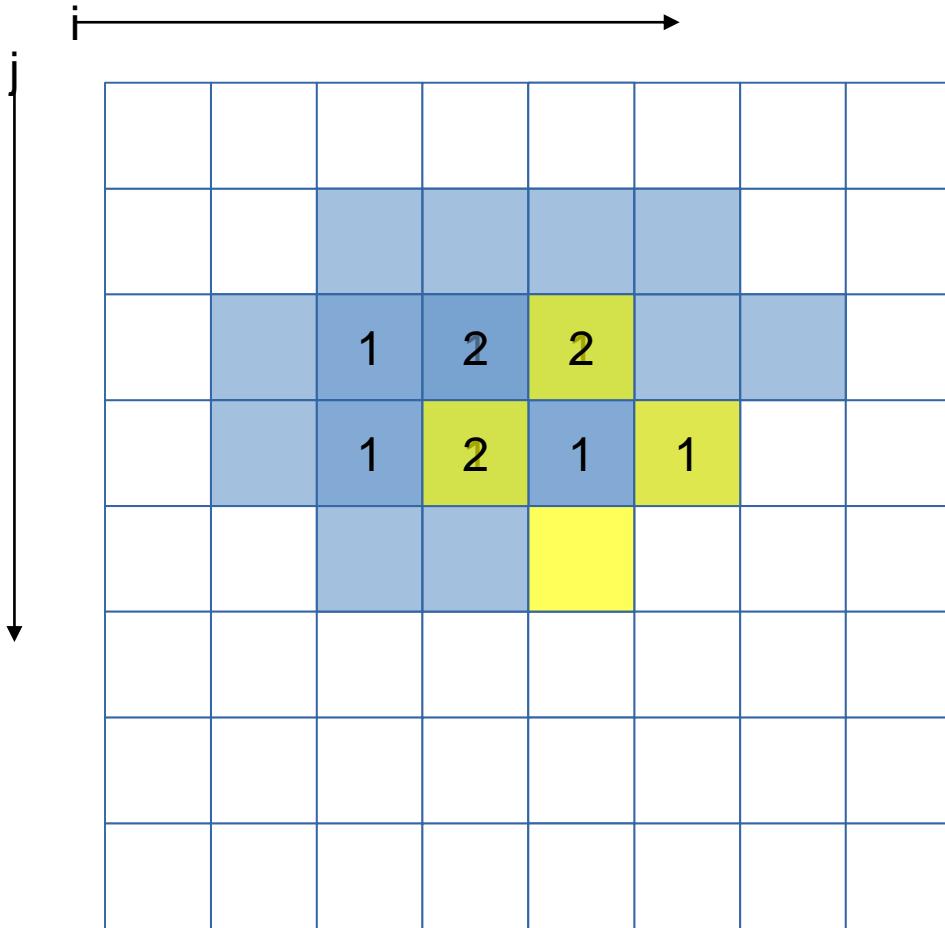


`LINEAR_SOLVER(i+0, j+0)`
`LINEAR_SOLVER(i+1, j+0)`
`LINEAR_SOLVER(i+2, j+0)`
`LINEAR_SOLVER(i+3, j+0)`

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

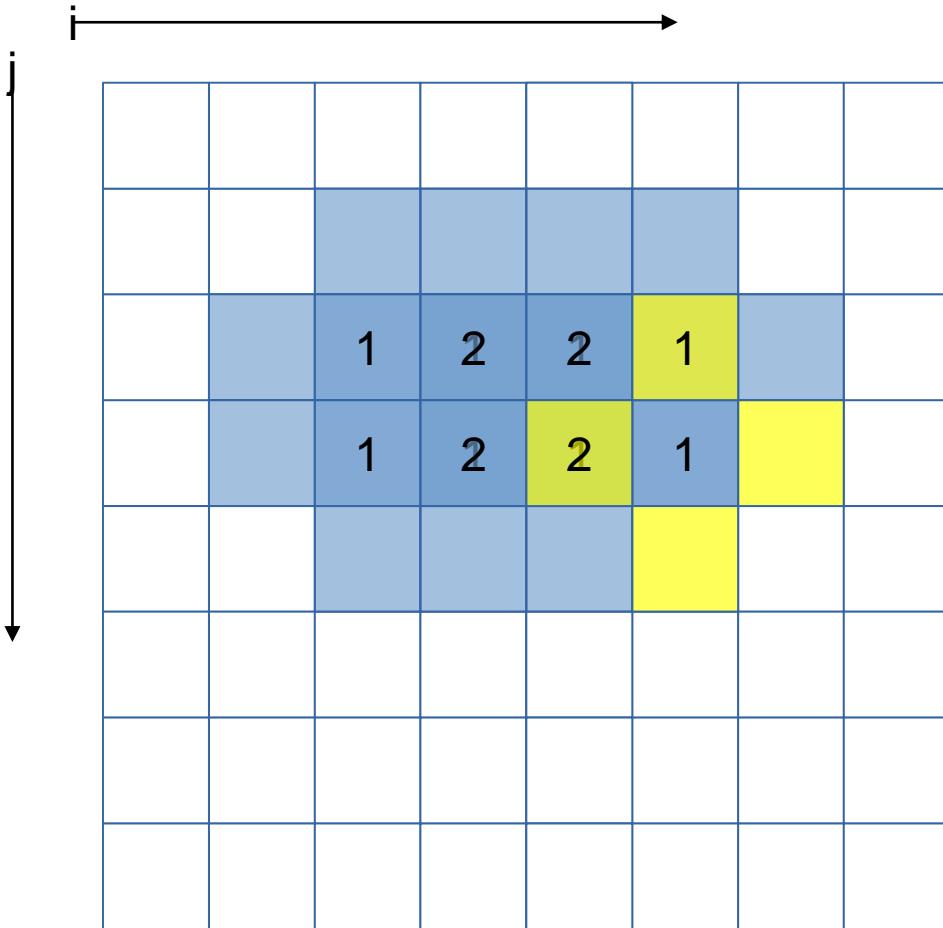


LINEAR_SOLVER(i+0, j+0)
LINEAR_SOLVER(i+1, j+0)
LINEAR_SOLVER(i+2, j+0)
LINEAR_SOLVER(i+3, j+0)

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

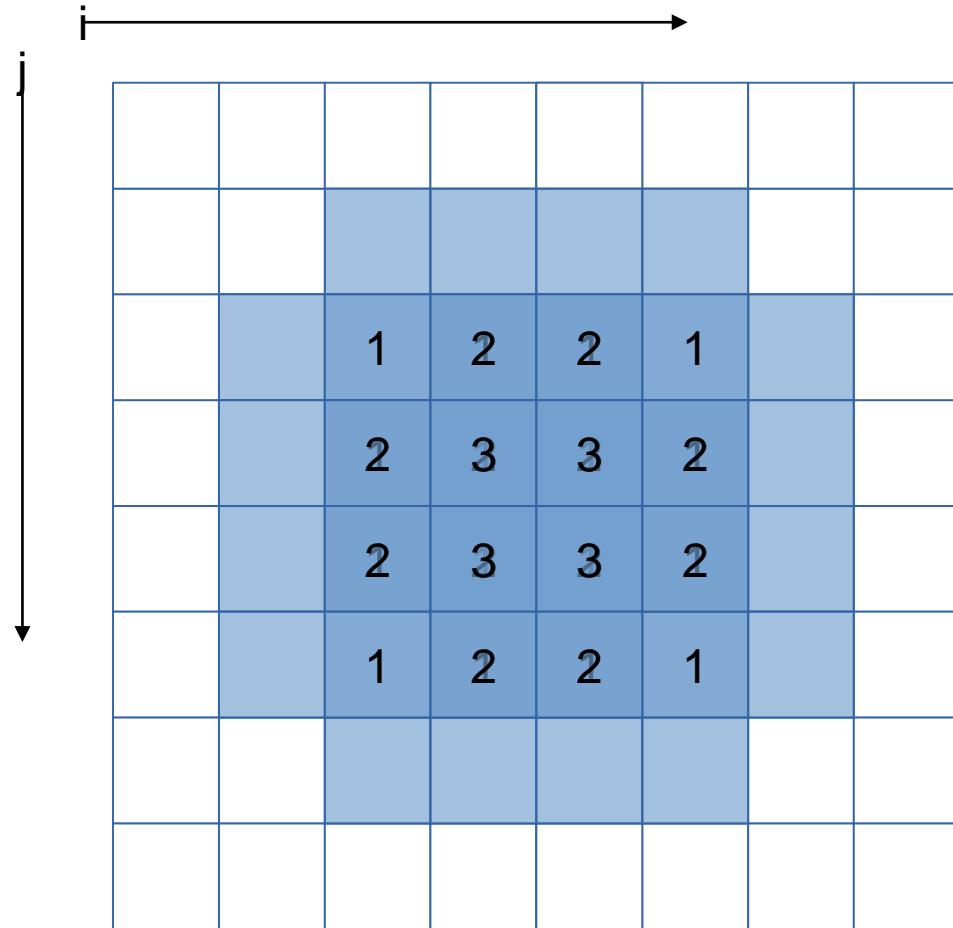


LINEAR_SOLVER($i+0, j+0$)
LINEAR_SOLVER($i+1, j+0$)
LINEAR_SOLVER($i+2, j+0$)
LINEAR_SOLVER($i+3, j+0$)

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 $4 \times 4 \times 4 = 64$ 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 30
Cycles per element for solvers: 1422.86
```

- Profile with MAQAO

```
> maqao oneview create-report=one xp=ov_k1 c=ov_k1.lua
```

- Display results

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

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

Total Time:	3.2 s
Time spent in loops:	97.97 %
Compilation Options:	OK
Flow Complexity:	1.29
Array Access Efficiency:	26.23 %

Loops Index

Filters

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

Loop id	Source Location	Source Function	Coverage (%)	Level
158	hydro_k1:kernel.c:15-176	linearSolver1	58.44	Innermost
63	hydro_k1:kernel.c:15-176	c_densitySolver	12.5	Innermost
50	hydro_k1:kernel.c:15-292	c_densitySolver	4.09	Innermost
82	hydro_k1:kernel.c:15-292	c_velocitySolver	3.75	Innermost
84	hydro_k1:kernel.c:15-292	c_velocitySolver	3.44	Innermost
80	hydro_k1:kernel.c:15-383	c_velocitySolver	2.19	Innermost
91	hydro_k1:kernel.c:15-371	c_velocitySolver	1.87	Innermost
107	hydro_k1:kernel.c:15-371	c_velocitySolver	1.87	Innermost
100	hydro_k1:kernel.c:15-383	c_velocitySolver	1.87	Innermost
145	hydro_k1:kernel.c:15-318	c_velocitySolver	1.25	Innermost

Loop Id: 158

Module: hydro_k1

Source: kernel.c:15-176

Coverage: 58.44%

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

Source Code

```

143: (a * (x[build_index(i-1, j, grid_size)] + x[build_index(i+1, j, grid_size)] + \
144: x[build_index(i, j-1, grid_size)] + x[build_index(i, j+1, grid_size)]) + \
145: x[build_index(i, j, grid_size)]) * inv_c;
146:
147: void linearSolver1(int b, float* x, float* x0, float a, float c, float dt, int grid_s:
148: {
149:     int i,j,k;
150:     const float inv_c = 1.0f / c;
151:
152:     for (k = 0; k < 20; k++)
153:     {
154:         for (i = 1; i <= grid_size-3; i+=4)
155:         {
156:             for (j = 1; j <= grid_size-3; j+=4)
157:             {
158:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+0);
159:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+1);
160:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+2);
161:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+3);
162:
163:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+0);
164:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+1);
165:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+2);
166:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+3);
167:
168:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+0);
169:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+1);
170:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+2);
171:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+3);
172:
173:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+0);
174:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+1);
175:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+2);
176:                 LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+3);
177:             }
178:         }

```

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

gain potential hint expert

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 67.00 to 63.00 cycles (1.06x speedup).

Workaround

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

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 67.00 to 4.19 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:
 - 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.

CQA output for kernel1

```
> maqao oneview create-report=one xp=ov_k1 \
output-format=text text-cqa=158 | less
```

gain potential hint expert

Type of elements and instruction set

80 SSE or AVX instructions are processing arithmetic or math operations on single precision FP elements in scalar mode (one at a time).

Matching between your loop (in the source code) and the binary loop

The binary loop is composed of 96 FP arithmetical operations:

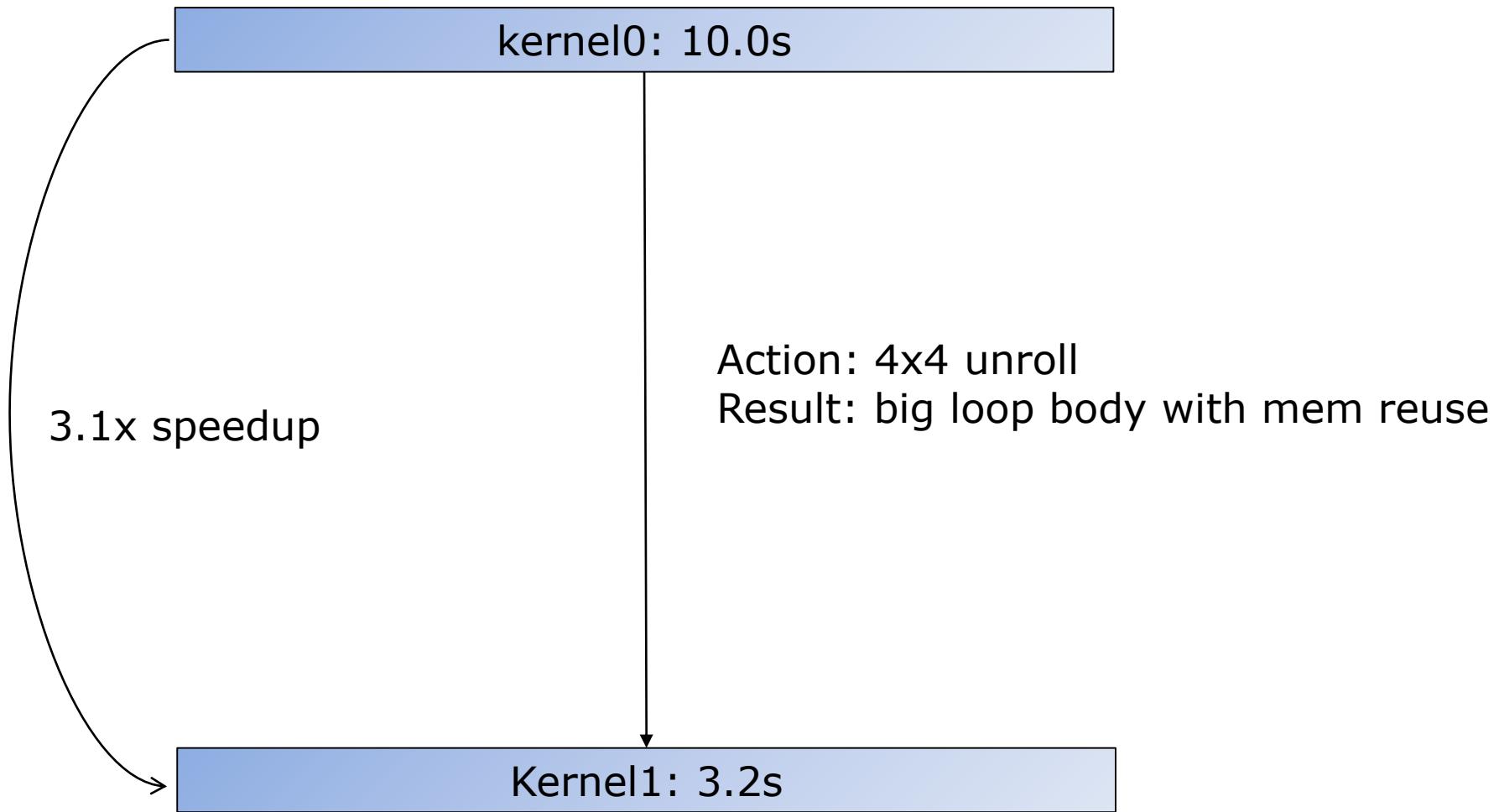
- 64: addition or subtraction
- 32: multiply

The binary loop is loading 272 bytes (68 single precision FP elements). The binary loop is storing 64 bytes (16 single precision FP elements).

4x4 Unrolling were applied

Expected 48... But still better than 80

Summary of optimizations and gains



More sample codes

More codes to study with MAQAO in

```
$SCRATCH/$USER/MAQAO_HANDSON/loop_optim_tutorial.tgz
```