MAQAO
Hands-on exercises

Analysing a code (bt-mz)

Optimising a code

Setup

Login to the cluster with X11 forwarding

```
> ssh -Y <login>@olympe.calmip.univ-toulouse.fr
```

Load MAQAO environment

```
> module use /usr/local/trex/modulefiles
> module load maqao/2.16.0
```

Copy handson material to your TMPDIR directory

```
> export TMPDIR=/tmpdir/$USER
> cd $TMPDIR
> tar xf /usr/local/trex/maqao/MAQAO_HANDSON_20221123.tgz
> tar xf /usr/local/trex/maqao/NPB3.4-MZ-MPI.tgz
```

(If not already done) Load compiler + MPI

```
> module load intel/18.2 intelmpi/18.2
```

Setup (bt-mz compilation with debug symbols)

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

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

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

Or copy the modified file from MAQAO_HANDSON directory

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

Compile bt-mz with debug information

```
> module load intel/18.2 intelmpi/18.2
> make bt-mz CLASS=C
```

Executing bt-mz

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

Analysing bt-mz with MAQAO

Cédric Valensi

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

```
executable = "bt-mz.C.x"
...
batch_script = "maqao_bt.slurm"
...
batch_command = "sbatch <batch_script>"
...
number_processes = 4
...
number_nodes = 2
...
mpi_command = "srun -p exclusive"
...
envv_OMP_NUM_THREADS = 18
```

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

```
...
#SBATCH -N 2 <number_nodes>
#SBATCH -n 4 <number_processes>
#SBATCH -c 18 <number_threads>
...
export OMP_NUM_THREADS=18<OMP_NUM_THREADS>
...
srun ./bt-mz.C.x
<mpi_command> <run_command>
...
```

Launch ONE View

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

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

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

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
```

A sample result directory is available in
/usr/local/trex/maqao/MAQAO_HANDSON_20221123_offline.tgz

Results can also be viewed directly on the console:

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

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
```

On your local machine:

```
> scp <login>@olympe.calmip.univ-toulouse.fr:ov_html.tgz .
> tar xf ov_html.tgz
> firefox <exp-dir>/RESULTS/bt-mz.C.x_one_html/index.html
```

Or use sshfs to mount the remote drive:

```
> mkdir olympedir
> sshfs <login>@olympe.calmip.univ-toulouse.fr:/tmpdir/<user> \
olympedir
> firefox olympedir/<exp-dir>/RESULTS/bt-mz.C.x_one_html/index.html
```

Specify the additional runs to be executed in the configuration file

```
> less config_bt_oneview_sbatch.lua
```

```
multiruns_params = {
  { name="2p_on_single_node", number_nodes=1, number_processes=2,
number_processes_per_node=2, envv_OMP_NUM_THREADS=18 },
  { name="2p_on_two_nodes", number_nodes=2, number_processes=2,
number_processes_per_node=1, envv_OMP_NUM_THREADS=18 },
}
```

Launch ONE View in scalability mode using flag --with-scalability

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

Optimising a code with MAQAO

Emmanuel OSERET

```
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

Load MAQAO environment

```
> module use /usr/local/trex/modulefiles
> module load maqao/2.16.0
```

Load latest GCC compiler

```
> module load gcc/10.3.0
```

Compile naive implementation of matrix multiply

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

```
> srun -N 1 -n 1 ./matmul_orig 150 10000
cycles per FMA: 3.65
```

Analyse matrix multiply with ONE View

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

Viewing results (HTML)

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

```
> scp <login>@olympe.calmip.univ-toulouse.fr:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		53.59
Profiled Time (s)		53.56
Time in analyzed loops (%)		100
Time in analyzed innermost loops (%)		99.7
Time in user code (%)		100.0
Compilation Options Score (%)		50
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.3
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	2.80
	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 oneview -R1 -xp=ov_orig \
  --output-format=text --text-global | less
```

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

  Total Time:                          53.59 s
  Time spent in loops:                 100 %
  Time spent in innermost loops:       99.7 %
  Compilation Options:                 50
  Perfect Flow Complexity:             1.00
  Array Access Efficiency:             83.3 %
  If No Scalar Integer:
      Potential Speedup:               1.00
      Nb Loops to get 80%:             1
  If FP Vectorized:
      Potential Speedup:               2.80
      Nb Loops to get 80%:             1
    ...
```

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_orig.c:9-10           | 99.64        |
   2       | matm...  | kernel_orig.c:7-10           | 0.35         |
```

Loop ID

Viewing results (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 4.00 to 0.25 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)
```

CQA output for the baseline kernel

Vectorization

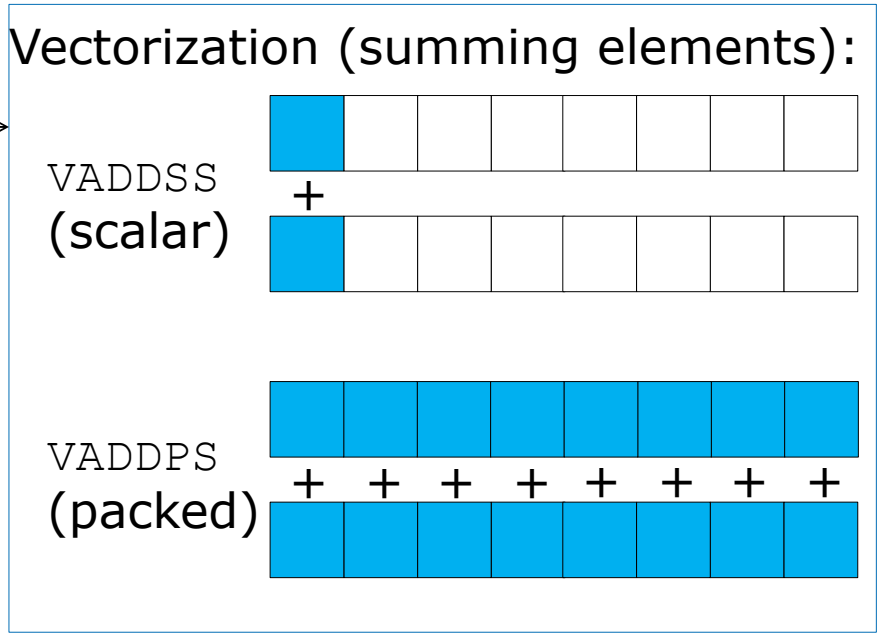
Your loop is not vectorized. 16 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 4.00 to 0.25 cycles (16.00x speedup).

Details

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

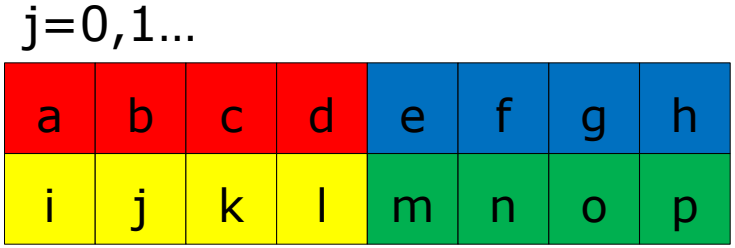
Workaround

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



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

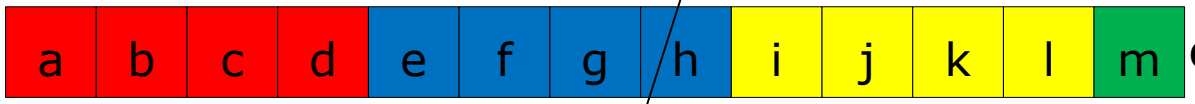
Logical mapping



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];
  }
}
```

Compile permuted loops version of matrix multiply

```
> cd $TMPDIR/MAQAO_HANDSON/matmul
> make matmul_perm
```

```
> srun -N 1 -n 1 ./matmul_perm 150 10000
cycles per FMA: 0.60
```

Analyse matrix multiply with ONE View

```
> maqao oneview -R1 mpi-command="srun -p exclusive -N 1 -n 1" \
xp=ov_perm -- ./matmul_perm 150 10000
```

OR using configuration script:

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

Viewing new results

```
> maqao oneview -R1 -xp=ov_perm \
  --output-format=text --text-global --text-loops | 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)		8.76
Profiled Time (s)		8.72
Time in analyzed loops (%)		99.7
Time in analyzed innermost loops (%)		89.8
Time in user code (%)		99.7
Compilation Options Score (%)		50
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		100
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.05
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.71
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	4.25
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.81
	Nb Loops to get 80%	2

Faster (was 53.59)

Compilation Options

Source Object	Issue
▼ matmul_orig	
▼ kernel_orig.c	
○	-march=(target) is missing.
○	-funroll-loops is missing.

Let's try this

More efficient vectorization (was 16.00)

CQA output after loop permutation

Vectorization

Your loop is vectorized, but using only 128 out of 512 bits (SSE/AVX-128 instructions on AVX-512 processors).
By fully vectorizing your loop, you can lower the cost of an iteration from 1.75 to 0.44 cycles (4.00x speedup).

Details

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

Workaround

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

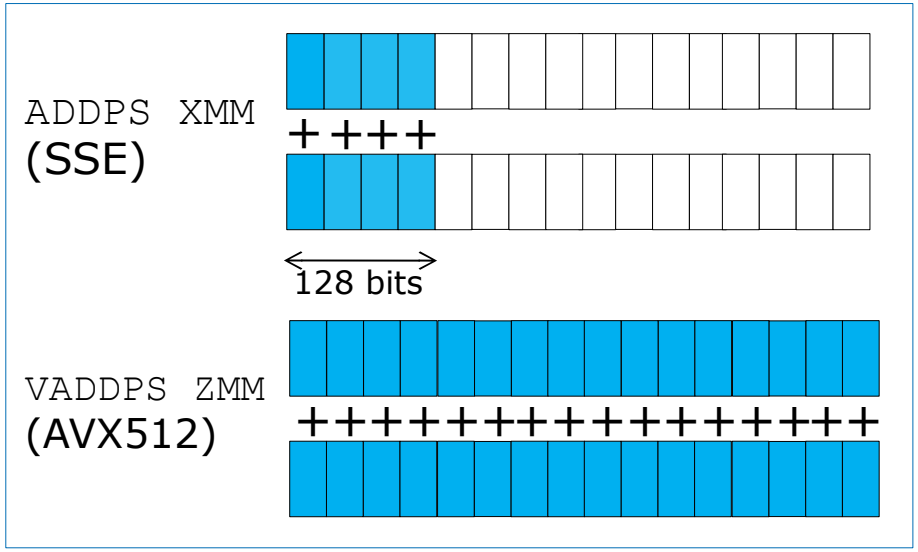
Execution units bottlenecks

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

Let's add –march= skylake-avx512

```
> maqao oneview -R1 -xp=ov_perm \
  --output-format=text --text-cqa=4 | less
```

- Vectorization
  - SSE instructions (SIMD 128 bits) used on a processor supporting AVX-512 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>
```

Compile architecture specialisation version of matrix multiply

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

```
> ./matmul_perm_opt 150 10000
cycles per FMA: 0.37
```

Analyse matrix multiply with ONE View

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

Viewing new results:

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

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

Loop permutation + (-march=skylake-avx512 -funroll-loops)

Global Metrics		
Total Time (s)		5.40
Profiled Time (s)		5.39
Time in analyzed loops (%)		99.6
Time in analyzed innermost loops (%)		51.5
Time in user code (%)		99.6
Compilation Options Score (%)		100
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.24
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.24
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	2.58
	Nb Loops to get 80%	2

Faster (was 8.76)

Now OK (-funroll-loops prev. missing)

Better vectorization (was 4.25)

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 -R1 -xp=ov_perm_opt \
  --output-format=text --text-cqa=4 | less
```

Compile aligned array version of matrix multiply

```
> cd $TMPDIR/MAQAO_HANDSON/matmul
> make matmul_align
```

Checking aligned version:

```
> srun -N 1 -n 1 ./matmul_align 150 10000
Cannot call kernel on matrices with size%16 != 0 (data not
aligned on 64B boundaries)
Aborted
```

=> Alignment imposes restrictions on input parameters.

```
> srun -N 1 -n 1 ./matmul_align 160 10000
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.17
```

Analyse matrix multiply with ONE View

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

Viewing new results

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

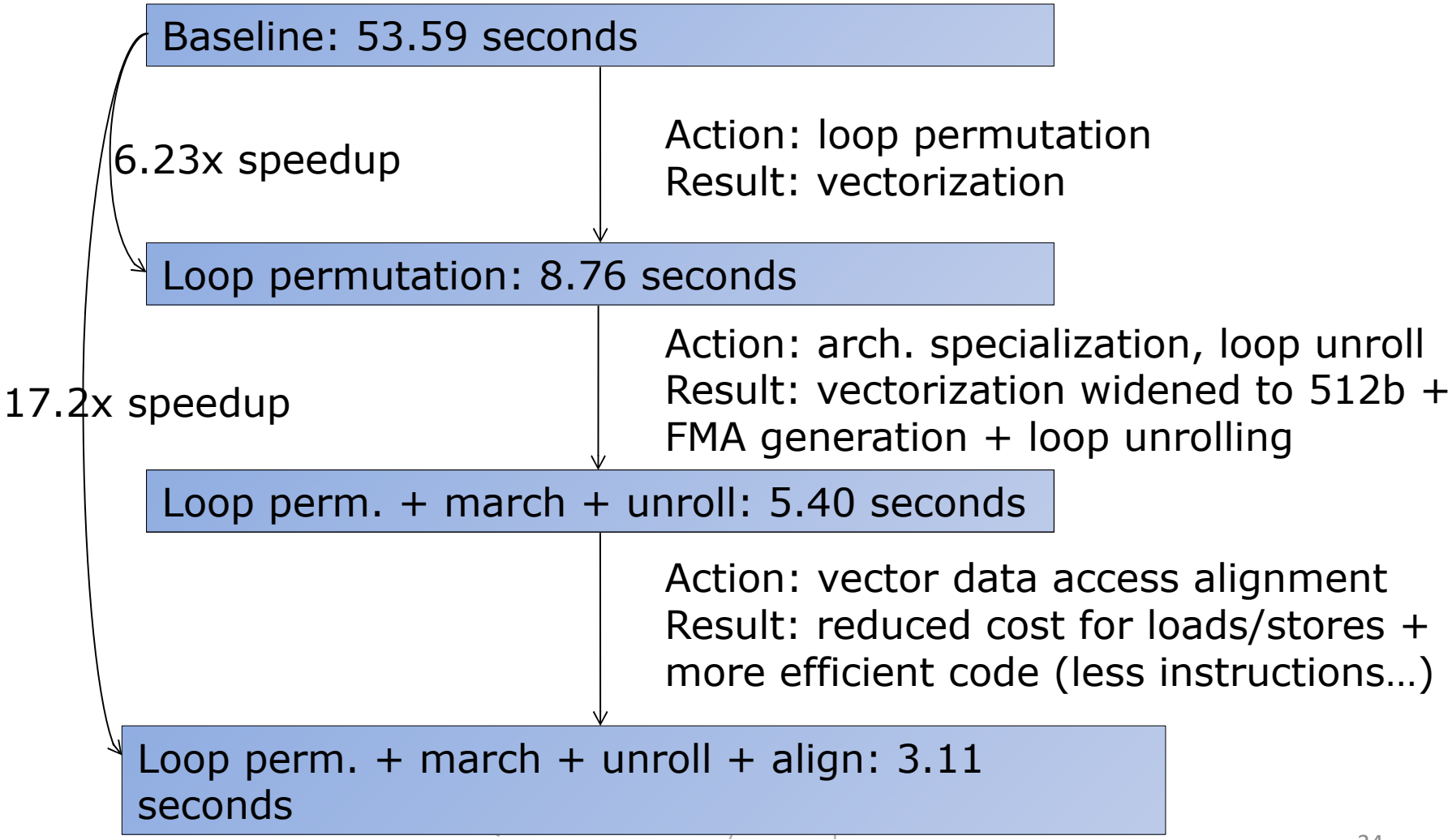
(Or download the **ov_align/RESULTS/matmul_align_one_html** folder locally and open **ov_align/RESULTS/matmul_align_one_html/index.html** in your browser)

Vector-aligning array accesses

Global Metrics		
Total Time (s)		3.11
Profiled Time (s)		3.10
Time in analyzed loops (%)		98.9
Time in analyzed innermost loops (%)		55.2
Time in user code (%)		98.9
Compilation Options Score (%)		100
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		75.0
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.20
	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	2.53
	Nb Loops to get 80%	2

Extra speedup (was 5.40)

Summary of optimizations and gains



Switch to the hydro handson folder

```
> cd $TMPDIR/MAQAO_HANDSON/hydro
```

Load Intel compiler environment

```
> module load intel/18.2
```

Compile

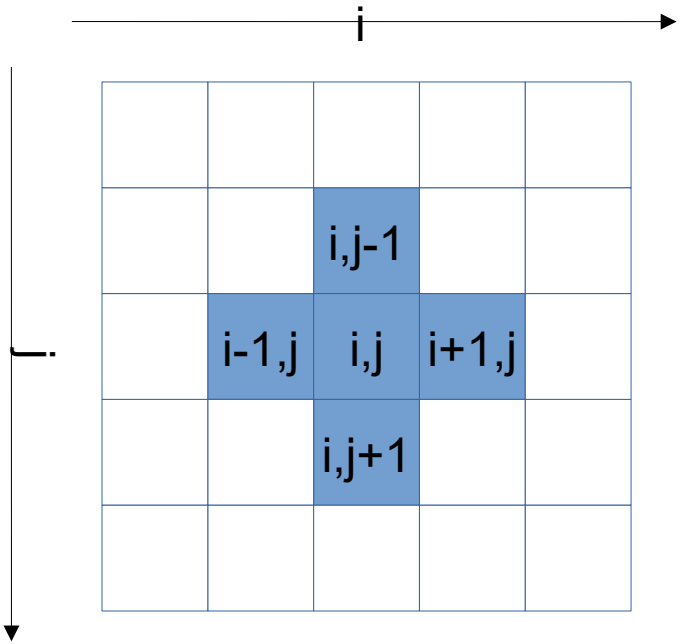
```
> make
```

```
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

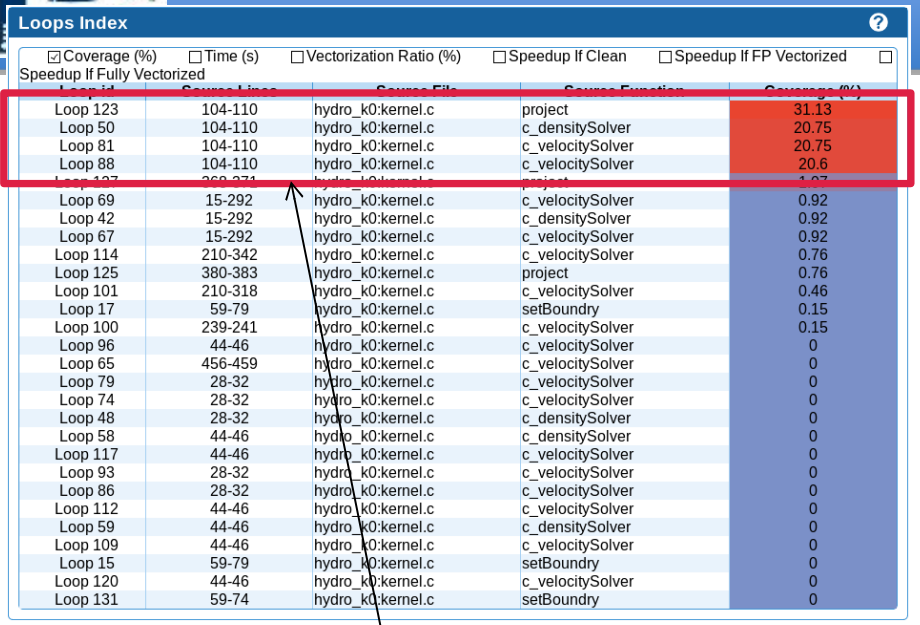


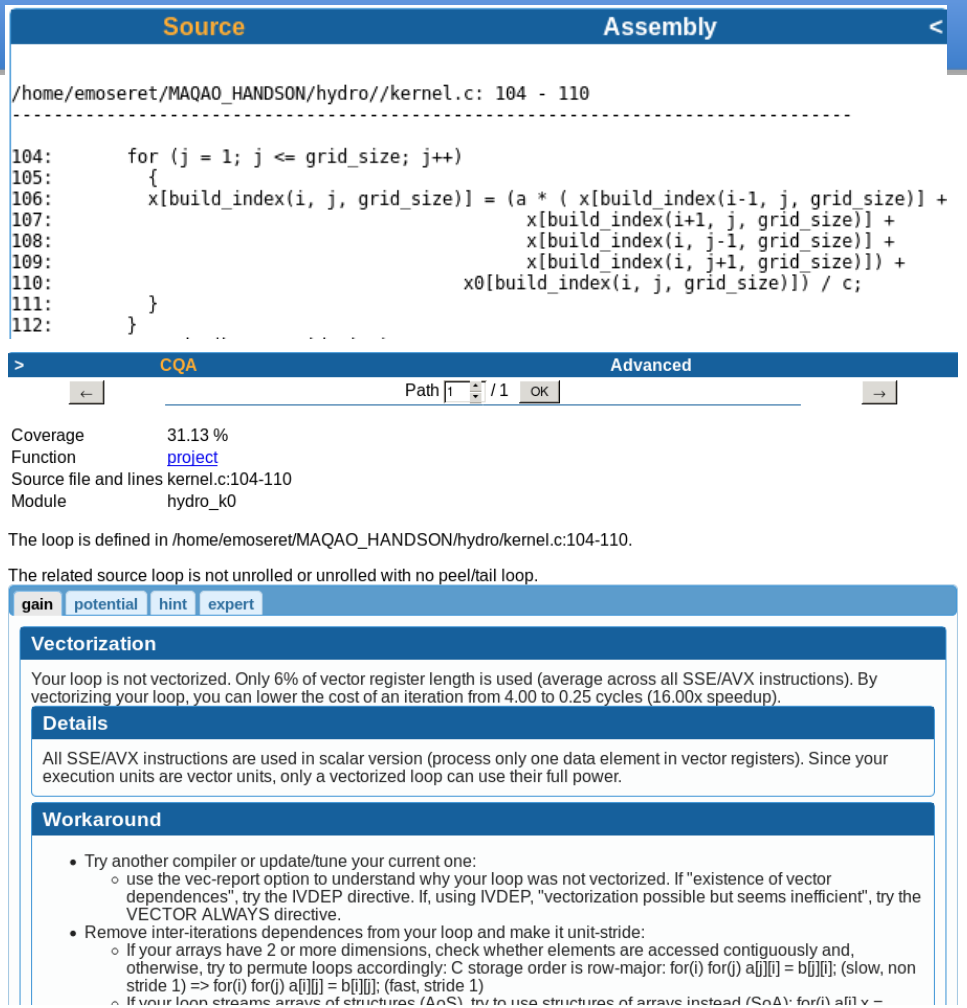
```
> srun -N 1 -n 1 ./hydro_k0 300 100
Cycles per element for solvers: 2733.88
```

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

> maqao oneview -R1 xp=ov_k0 \
--output-format=text --text-global --text-loops | less
> ...
> Total time: 10.71s
```

```
> maqao oneview -R1 xp=ov_k0 \
--output-format=text --text-cqa=123 | less
```





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

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

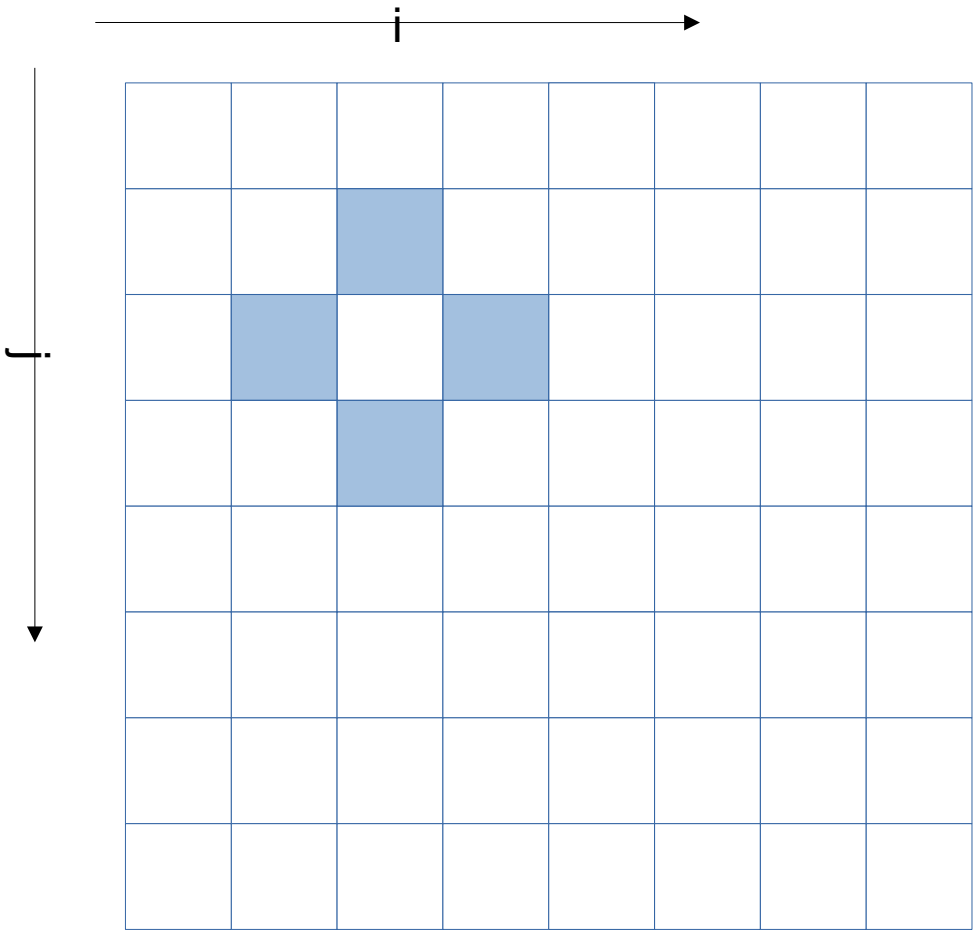
Loop is potentially data access bound.

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).

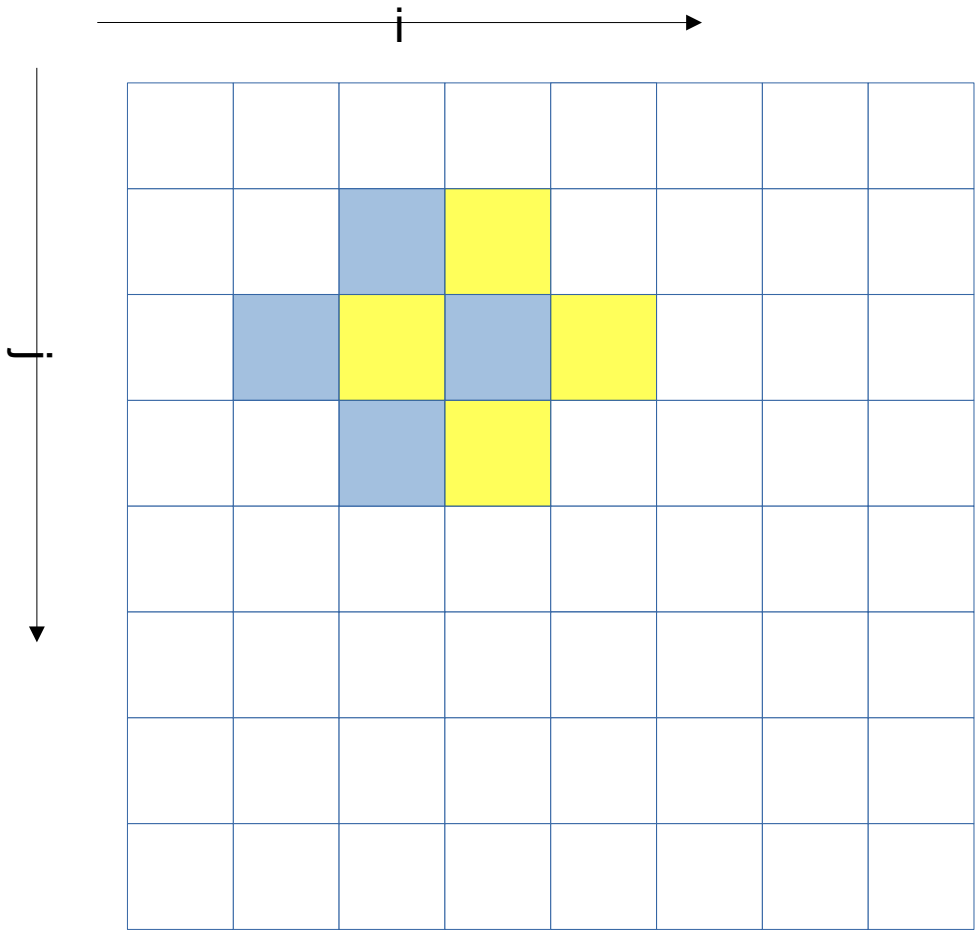
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



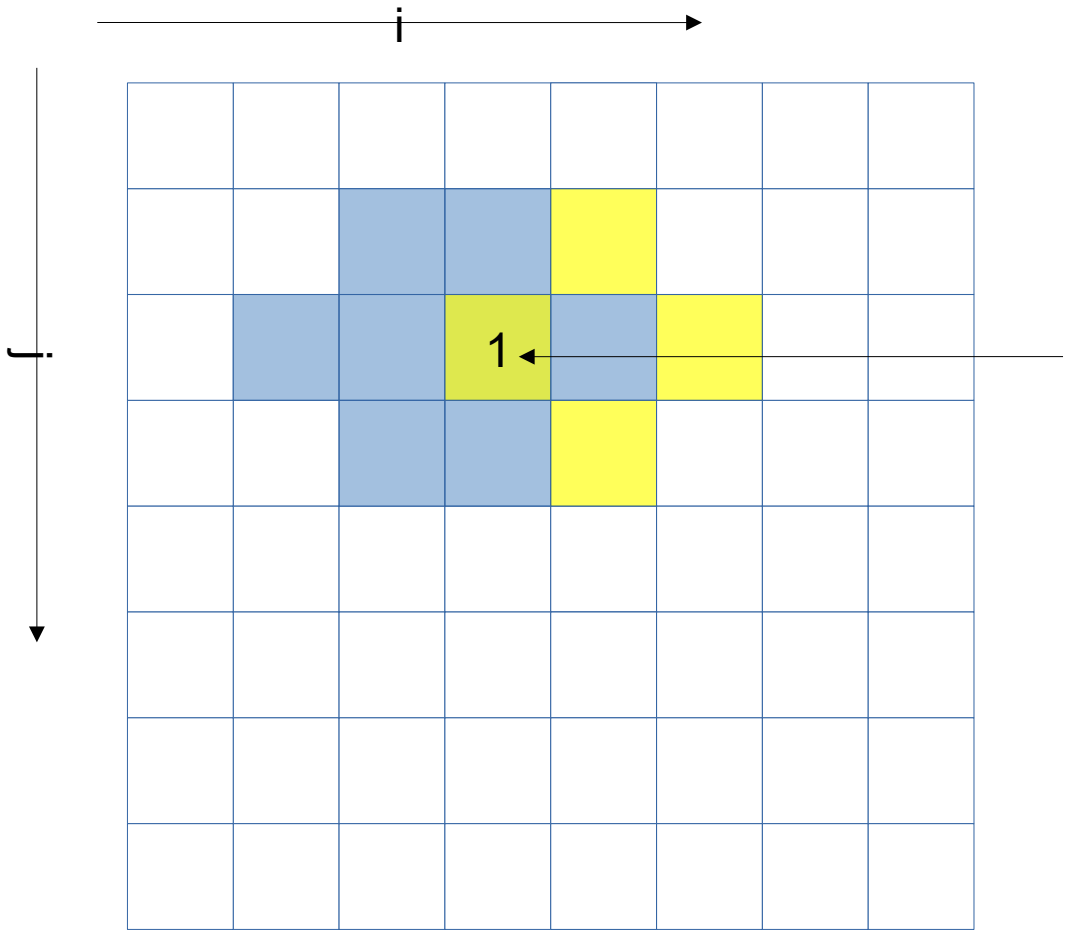
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



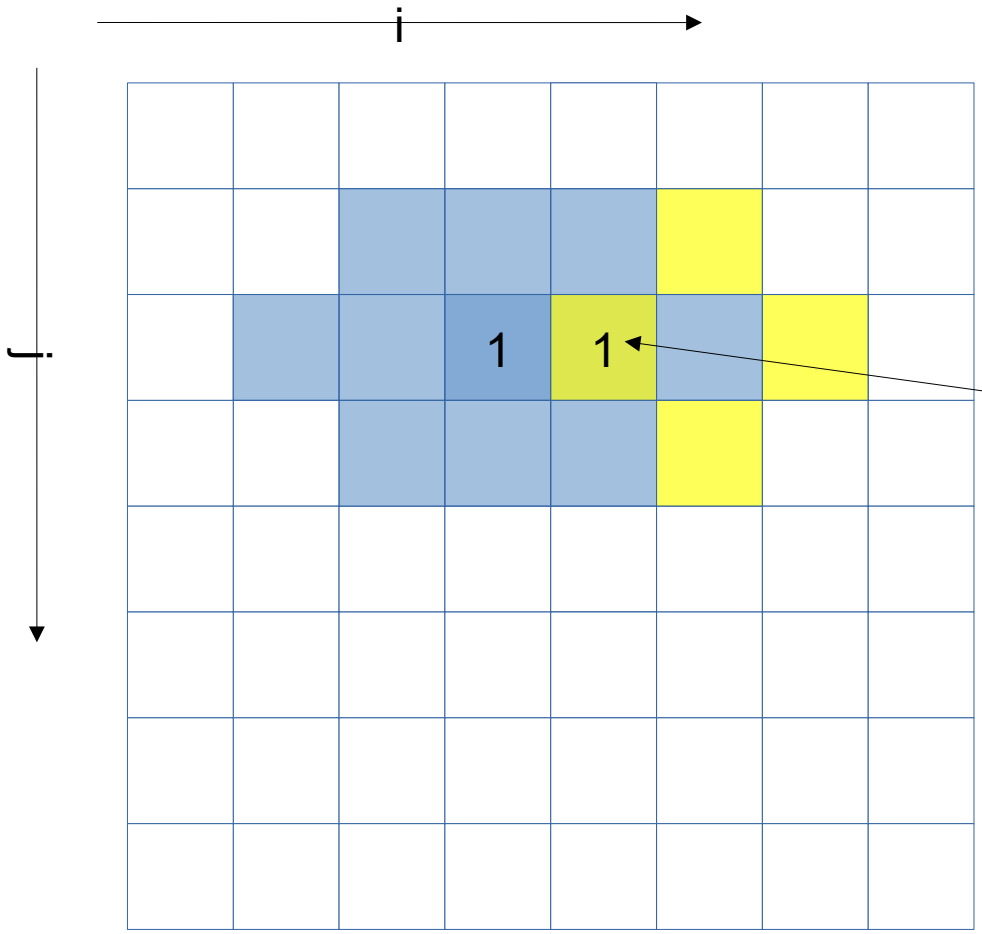
LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)

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)

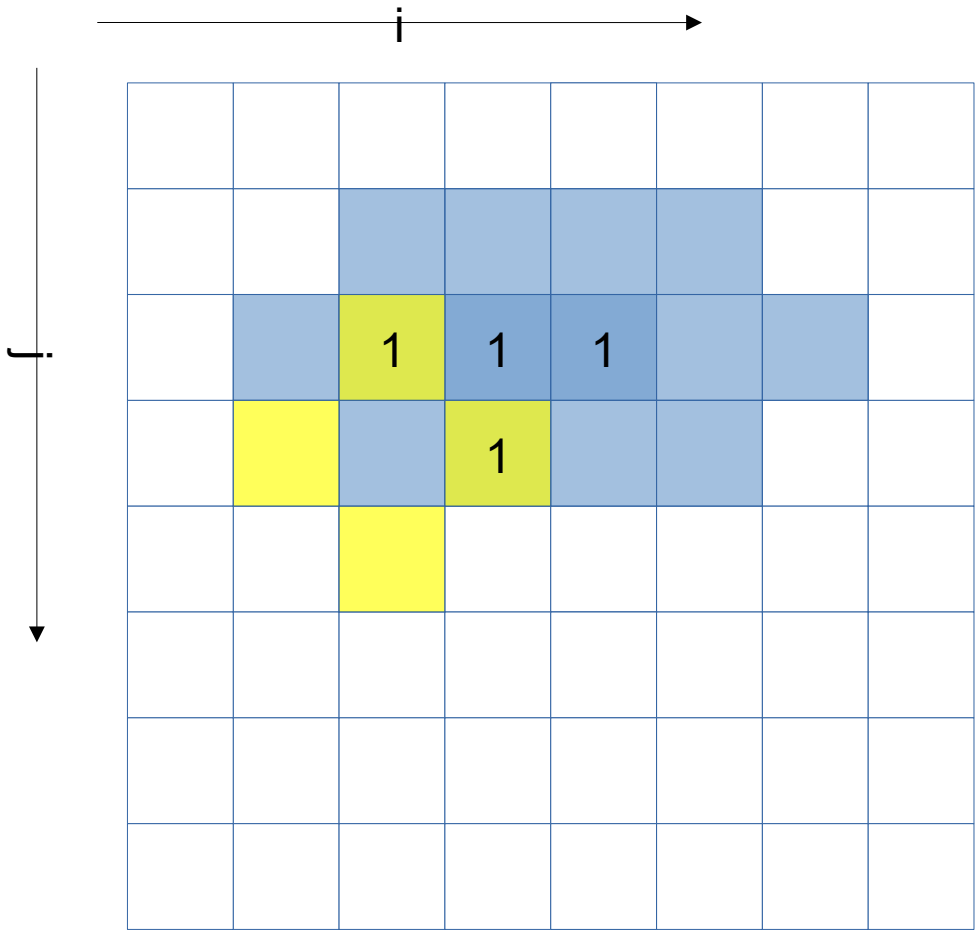
Memory references reuse : 4x4 unroll footprint on loads



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

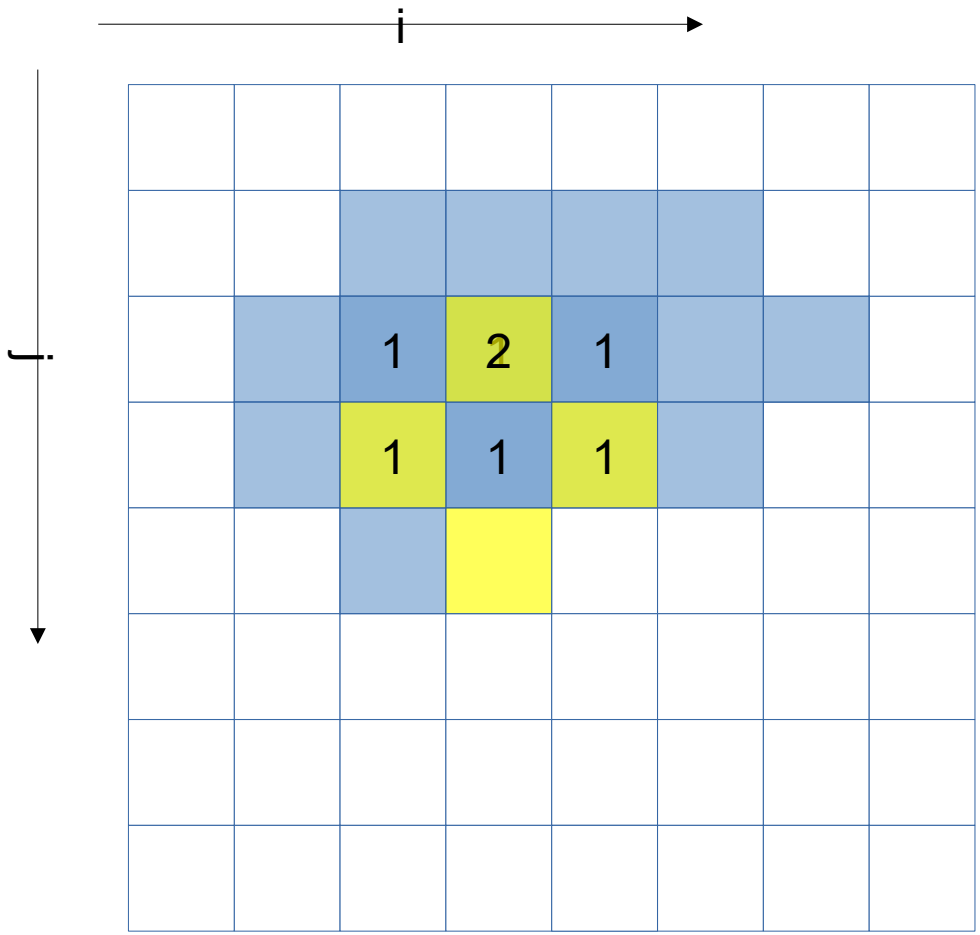


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

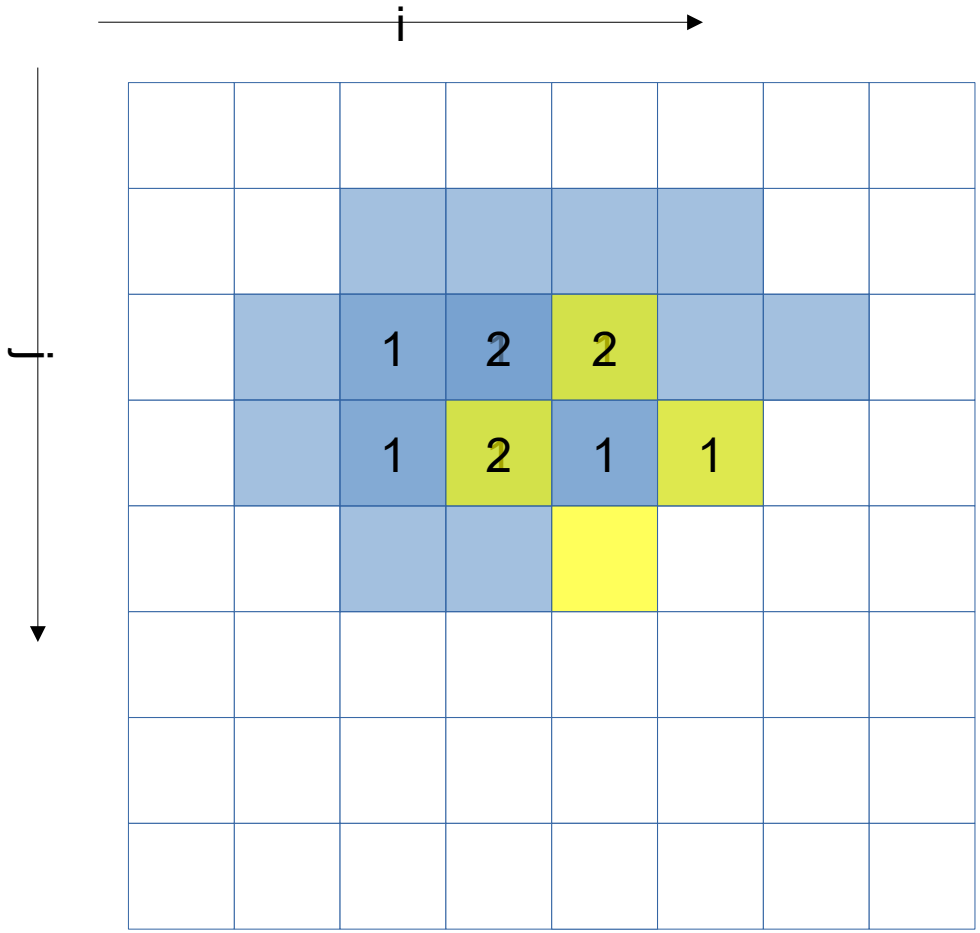


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

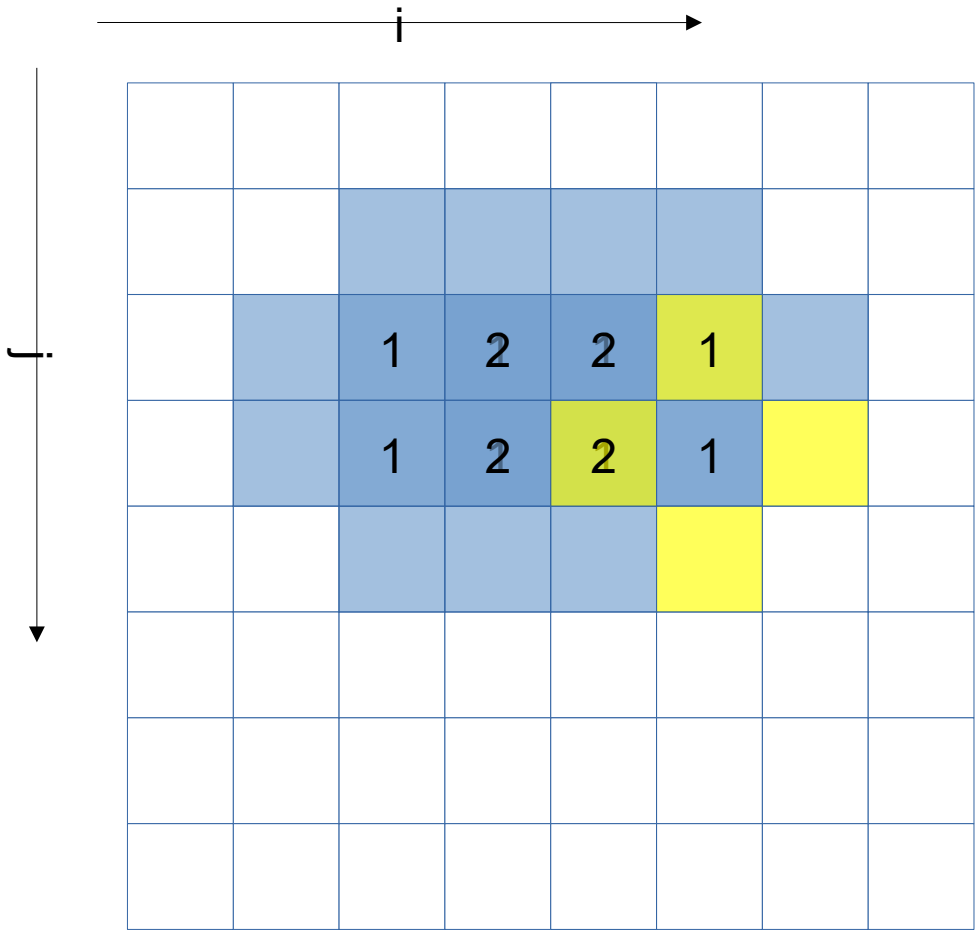


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

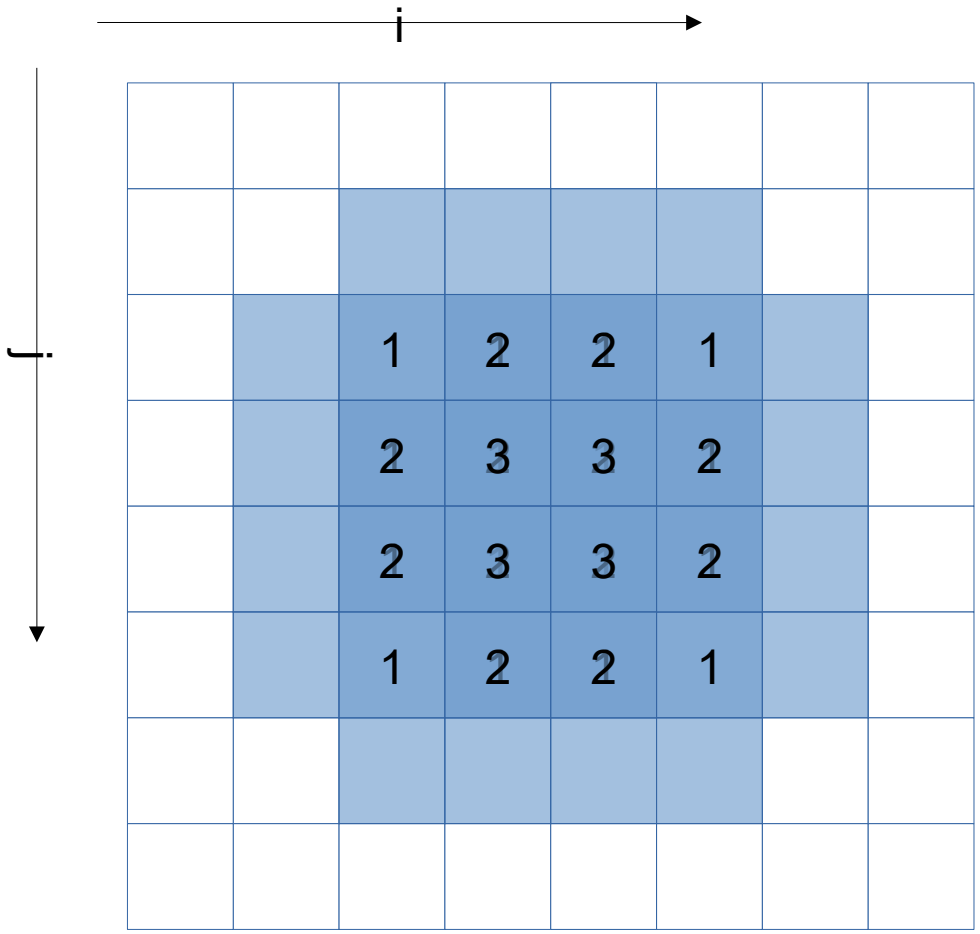


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

- For the x array, instead of 4x4x4 = 64 loads, now only 32 (32 loads avoided by reuse)
- For the x0 array no reuse possible : 16 loads
- Total loads : 48 instead of 80

4x4 unroll

```
#define LINEARSOLVER(...) x[build_index(i, j, grid_size)] = …

void linearSolver2 (...) {
  (...)

  for (k=0; k<20; k++)
    for (i=1; i<=grid_size-3; i+=4)
      for (j=1; j<=grid_size-3; j+=4) {
        LINEARSOLVER (…, i+0, j+0);
        LINEARSOLVER (…, i+0, j+1);
        LINEARSOLVER (…, i+0, j+2);
        LINEARSOLVER (…, i+0, j+3);

        LINEARSOLVER (…, i+1, j+0);
        LINEARSOLVER (…, i+1, j+1);
        LINEARSOLVER (…, i+1, j+2);
        LINEARSOLVER (…, i+1, j+3);

        LINEARSOLVER (…, i+2, j+0);
        LINEARSOLVER (…, i+2, j+1);
        LINEARSOLVER (…, i+2, j+2);
        LINEARSOLVER (…, i+2, j+3);

        LINEARSOLVER (…, i+3, j+0);
        LINEARSOLVER (…, i+3, j+1);
        LINEARSOLVER (…, i+3, j+2);
        LINEARSOLVER (…, i+3, j+3);
      }
}
```

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

```
> srun -N 1 -n 1 ./hydro_k1 300 100
Cycles per element for solvers: 872.08
```

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

> maqao oneview -R1 xp=ov_k1 \
--output-format=text --text-global --text-loops | less
> ...
> Total time: 3.37s
```

```
> maqao oneview -R1 xp=ov_k1 \
--output-format=text --text-cqa=129 | less
```

Loops Index

☑ Coverage (%) ☐ Time (s) ☐ Vectorization Ratio (%) ☐ Speedup If Clean ☐ Speedup If FP Vectorized ☐ Speedup If Fully Vectorized

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 129	15-176	hydro_k1:kernel.c	linearSolver1	62.13
Loop 51	15-176	hydro_k1:kernel.c	c_densitySolver	15.27
Loop 43	15-292	hydro_k1:kernel.c	c_densitySolver	3.16
Loop 70	15-292	hydro_k1:kernel.c	c_velocitySolver	3.16
Loop 68	15-292	hydro_k1:kernel.c	c_velocitySolver	3.16
Loop 117	210-342	hydro_k1:kernel.c	c_velocitySolver	2.63
Loop 104	210-318	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 66	380-383	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 72	368-371	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 86	368-371	hydro_k1:kernel.c	c_velocitySolver	1.58
Loop 84	380-383	hydro_k1:kernel.c	c_velocitySolver	1.05
Loop 103	239-241	hydro_k1:kernel.c	c_velocitySolver	0.53
Loop 123	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 64	456-459	hydro_k1:kernel.c	c_velocitySolver	0
Loop 77	28-32	hydro_k1:kernel.c	c_velocitySolver	0
Loop 109	226-230	hydro_k1:kernel.c	c_velocitySolver	0
Loop 120	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 110	226-230	hydro_k1:kernel.c	c_velocitySolver	0
Loop 115	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 17	59-79	hydro_k1:kernel.c	setBoundry	0
Loop 91	28-32	hydro_k1:kernel.c	c_velocitySolver	0
Loop 127	59-79	hydro_k1:kernel.c	setBoundry	0
Loop 112	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 82	28-32	hydro_k1:kernel.c	c_velocitySolver	0
Loop 99	44-46	hydro_k1:kernel.c	c_velocitySolver	0
Loop 49	28-32	hydro_k1:kernel.c	c_densitySolver	0
Loop 59	44-46	hydro_k1:kernel.c	c_densitySolver	0
Loop 96	28-32	hydro_k1:kernel.c	c_velocitySolver	0

```
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:     }
```

> CQA Advanced

← Path 1 / 1 OK →

Coverage 62.13 %
Function linearSolver1
Source file and lines kernel.c:15-176
Module hydro_k1

The loop is defined in /home/emoseret/MAQAO_HANDSON/hydro/kernel.c:15-176.

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

gain | potential | hint | expert

Vectorization

Your loop is not vectorized. Only 6% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 41.50 to 2.59 cycles (16.00x speedup).

Details

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

Workaround

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

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

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

The binary loop is composed of 96 FP arithmetical operations:

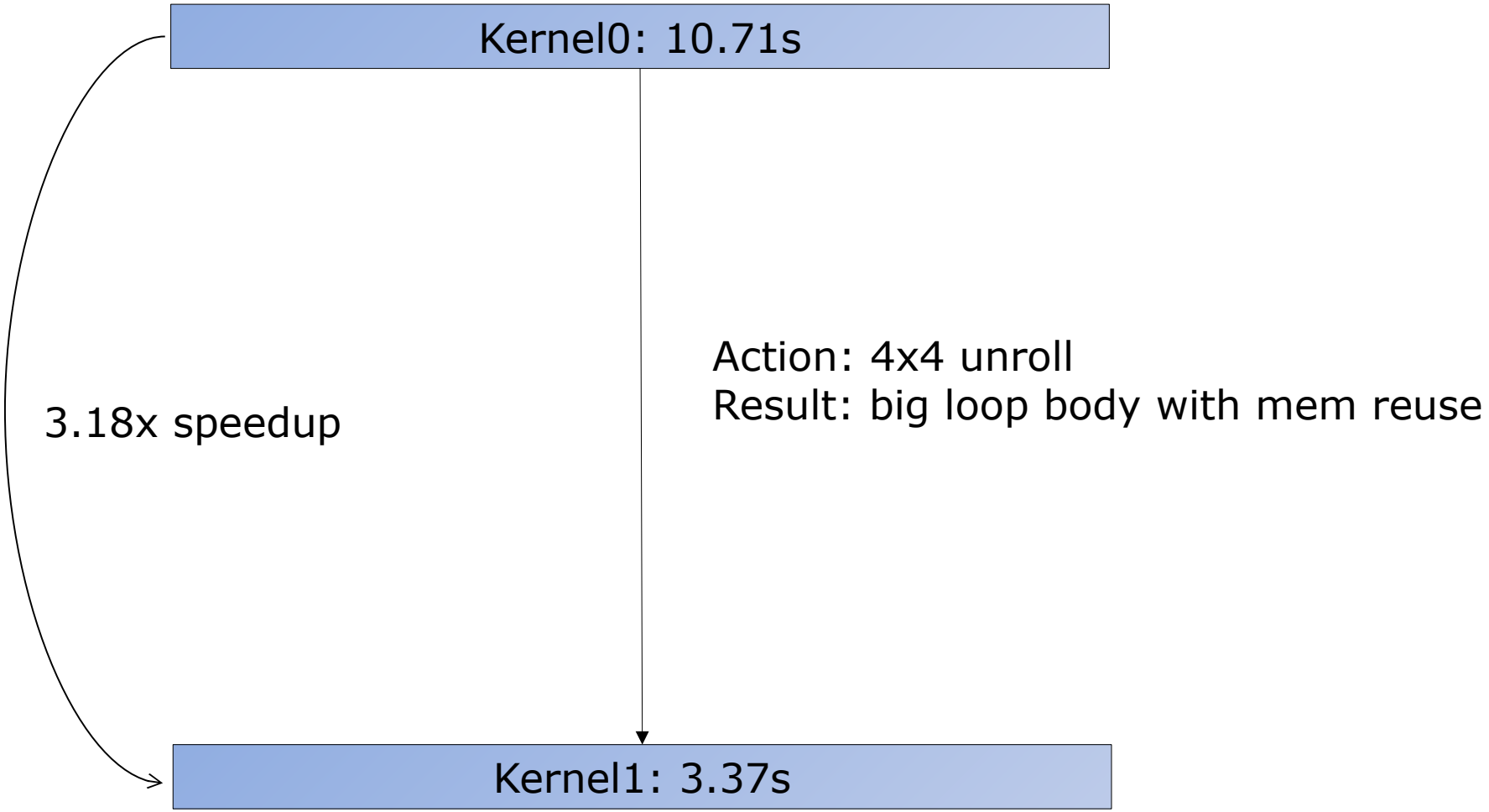
- 64: addition or subtraction
- 32: multiply

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



```
/usr/local/trex/maqao/loop_optim_tutorial.tgz
```

Thanks for your attention

QUESTIONS ?