MAQAO
Hands-on exercises

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

Setup (reminder)

Login to the cluster

```
> ssh <username>@hawk.hww.hlrs.de
```

Load MAQAO environment

```
> module load maqao
```

Copy handson material to your workspace directory

```
> export TW35=/lustre/cray/ws9/2/ws/hpcjgrac-tw35 # optional: should already be in ~/.bash_profile
> export WORK=/lustre/cray/ws9/... # optional: should already be in ~/.bash_profile
> cd $WORK
> tar xvf $TW35/maqao/MAQAO_HANDSON.tgz
```

Setup (bt-mz compilation with openMPI & debug symbols)

Go to the NPB directory provided with MAQAO handsons

```
> cd $WORK/NPB3.4-MZ-MPI
```

Change MPI runtime from MPT to openMPI

```
> module swap mpt openmpi
```

Compile

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

Remark: with version 3.4 the generated executable supports any number of ranks (no need to generate one executable for 6 ranks, another for 8 etc.)

Profiling bt-mz with MAQAO

Salah Ibnamar

Setup ONE View for batch mode

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

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

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

```
binary = "bt-mz.C.x"
...
batch_script = "bt_maqao.pbs"
...
batch_command = "qsub <batch_script>“
...
number_processes = 8
...
omp_num_threads = 8
...
mpi_command = "mpirun -n <number_processes>"
...
```

Review jobscript for use with ONE View

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

Retrieve jobscript modified for ONE View from the MAQAO_HANDSON directory.

```
> cd $WORK/NPB3.4-MZ-MPI/bin #if current directory has changed
> cp $WORK/MAQAO_HANDSON/bt/bt_maqao.pbs .
> less bt_maqao.pbs
```

...

#PBS -l select=~~2~~<number_nodes>

...

export OMP_NUM_THREADS=~~8~~<omp_num_threads>

...

~~mpirun -n ... $EXE~~

<mpi_command> <run_command>

...

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

Launch ONE View

```
> cd $WORK/NPB3.4-MZ-MPI/bin #if current directory has changed
> maqao oneview --create-report=one \
config=config_bt_oneview_pbs.lua xp=ov_pbs
```

The -xp parameter allows to set the path to the experiment directory, where ONE View stores the analysis results and where the reports will be generated.
If -xp is omitted, the experiment directory will be named maqao_<timestamp>.
WARNINGS:
- If the directory specified with -xp already exists, ONE View will reuse its content but not overwrite it.

(OPTIONAL) Setup ONE View for interactive mode

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

```
> cd $WORK/NPB3.4-MZ-MPI/bin #if current directory has changed
> cp $WORK/MAQAO_HANDSON/bt/config_bt_oneview_interactive.lua .
> less config_bt_oneview_interactive.lua
```

```
binary = "bt-mz.C.x"
...
number_processes = 8
...
omp_num_threads = 8
...
mpi_command = "mpirun -n <number_processes>"
```

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

Request interactive session

```
> qsub -I -l select=2:mpiprocs=4:ompthreads=8,
walltime=00:10:00 -q R_tw
```

Load MAQAO environment

```
> module load maqao
```

Launch ONE View

```
> module swap mpt openmpi
> cd $WORK/NPB3.4-MZ-MPI/bin
> maqao oneview --create-report=one \
config=config_bt_oneview_interactive.lua \
xp=ov_interactive
```

Display MAQAO ONE View results

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

Mount $WORK locally:

```
> mkdir hawk_work
> sshfs <username>@hawk.hww.hlrs.de:/lustre/cray/ws9/... \
hawk_work
> firefox hawk_work/NPB3.4-MZ-MPI/bin/ov_pbs/RESULTS/bt-
mz.C.x_one_html/index.html
```

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

```
> tar czf $HOME/bt_html.tgz ov_pbs/RESULTS/bt-mz.C.x_one_html
```

```
> scp <login>@hawk.hww.hlrs.de:bt_html.tgz .
> tar xf bt_html.tgz
> firefox ov_pbs/RESULTS/bt-mz.C.x_one_html/index.html
```

Display MAQAO ONE View results (optional)

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

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

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

Scalability profiling of bt-mz with MAQAO

Salah Ibnamar

Setup ONE View for scalability analysis

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

```
> cd $WORK/NPB3.4-MZ-MPI/bin #if cur. dir. has changed
> cp $WORK/MAQAO_HANDSON/bt/config_bt_scalability.lua .
> less config_bt_scalability.lua
```

```
binary = "./bt-mz.C.x"
…
run_command = "<binary>"
…
batch_script = "bt_maqao.pbs"
…
batch_command = "qsub <batch_script>"
…
number_processes = 1
…
number_nodes = 1
…
omp_num_threads = 1
…
mpi_command = "mpirun –n <number_processes>"
…
multiruns_params = {
 {nb_processes = 1, nb_threads = 8, number_nodes = 1, number_tasks_nodes = 1},
 {nb_processes = 8, nb_threads = 1, number_nodes = 1, number_tasks_nodes = 8},
 {nb_processes = 8, nb_threads = 1, number_nodes = 2, number_tasks_nodes = 4},
 {nb_processes = 8, nb_threads = 8, number_nodes = 2, number_tasks_nodes = 4},
}
```

Launch MAQAO ONE View on lulesh (scalability mode)

Launch ONE View (execution will be longer!)

```
> maqao oneview --create-report=one --with-scalability=on \
config=config_bt_scalability.lua xp=ov_scal
```

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

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

OR

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

```
> scp <login>@hawk.hww.hlrs.de:ov_scal.tgz .
> tar xf ov_scal.tgz
> firefox ov_scal/RESULTS/bt-mz.C.x_one_html/index.html
```

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

Optimising a code with MAQAO

Emmanuel OSERET

Matrix Multiply code

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

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

“Naïve” dense matrix multiply implementation in C

Preparing interactive session with GNU compiler

Request interactive session and load MAQAO environment

```
> qsub -I -l select=1,walltime=00:20:00 -q R_tw
> module load maqao
```

Define variable for workspace directory (if not already done)

```
> export WORK=/lustre/cray/ws9/...
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

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

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

```
> ./matmul_orig 400 300
cycles per FMA: 2.30
```

Analyse matrix multiply with ONE View

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

OR, using a configuration script:

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

Viewing results (HTML)

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

```
> scp <login>@hawk.hww.hlrs.de:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)		17.13
Time in loops (%)		100
Time in innermost loops (%)		99.89
Time in user code (%)		99.99
Compilation Options		**binary**: -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.33
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	2.13
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	8.00
	Nb Loops to get 80%	1

Viewing results (text)

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

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

  Total Time:              17.13 s
  Time spent in loops:          100 %
  Compilation Options:          binary: -funroll-loops is missing.
  Flow Complexity:         1.00
  Array Access Efficiency:      83.32 %
  If Clean:
      Potential Speedup: 1.00
      Nb Loops to get 80%:      1
  If FP Vectorized:
      Potential Speedup: 2.00
      Nb Loops to get 80%:      1
  If Fully Vectorized:
      Potential Speedup: 8.00
      Nb Loops to get 80%:      1
```

Viewing results (text)

```
+----------------------------------------------------------------------------------+
+                                1.3  -  Potential Speedups                        +
+----------------------------------------------------------------------------------+


  If No Scalar Integer:
...
  If FP Vectorized:
...

  If Fully Vectorized:
      Number of loops   | 1      | 2      | 3      |
      Cumulated Speedup | 7.9334 | 7.9944 | 8.0000 |
  Top 5 loops:
    matmul_orig - 1:    7.9334
    matmul_orig - 2:    7.9944
    matmul_orig - 4:    8
```

Loop ID

Viewing CQA output (text)

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

Loop ID

```
      Vectorization
  --------------------
Your loop is not vectorized.
8 data elements could be processed at once in vector registers.
By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.37 cycles
(8.00x speedup).

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

Workaround
 - Try another compiler or update/tune your current one:
  * recompile with fassociative-math (included in Ofast or ffast-math) to extend loop
vectorization to FP reductions.
 - (…)
```

CQA output for the baseline kernel

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 3.00 to 0.37 cycles (8.00x speedup).

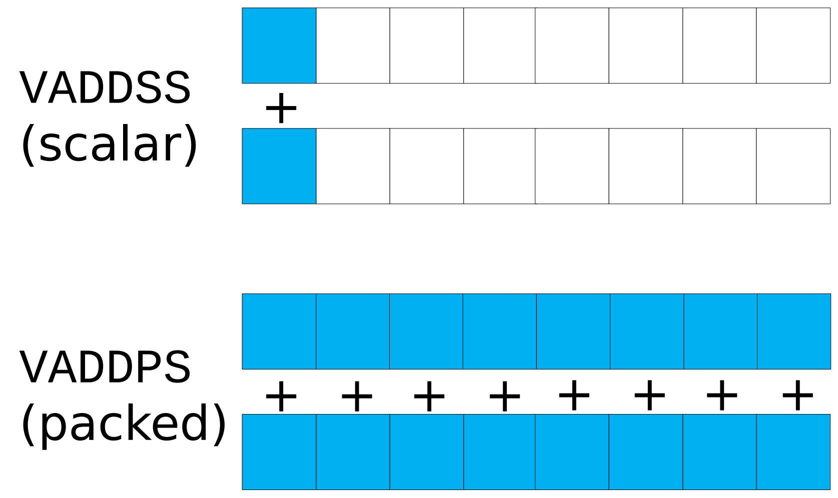
Details

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

Workaround

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

Vectorization (summing elements):



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

Impact of loop permutation on data access

Logical mapping

j=0,1…

i=0	a	b	c	d	e	f	g	h
i=1	i	j	k	l	m	n	o	p

Efficient vectorization + prefetching

Physical mapping

(C stor. order: row-major)

a	b	c	d	e	f	g	h	i	j	k	l	m	etc.

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

a	i	etc.	b	j	etc.	e	m	etc.	f	n	etc.

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

a	b	c	d	e	f	g	h	i	j	k	l	m	etc.

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 #if cur. directory has changed
> make matmul_perm
```

```
> ./matmul_perm 400 300
cycles per FMA: 0.22
```

Analyse matrix multiply with ONE View

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

OR, using a configuration script:

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

Viewing results (HTML)

```
> tar czf $HOME/ov_perm.tgz ov_perm/RESULTS/matmul_perm_one_html
```

```
> scp <login>@hawk.hww.hlrs.de:ov_perm.tgz .
> tar xf ov_perm.tgz
> firefox ov_perm/RESULTS/matmul_perm_one_html/index.html &
```

Global Metrics

Total Time (s)		1.63
Time in loops (%)		99.69
Time in innermost loops (%)		96.43
Time in user code (%)		99.69
Compilation Options		**binary**: -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		1[illegible]
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.01
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	4
Fully Vectorised	Potential Speedup	1.03
	Nb Loops to get 80%	1

Faster (was 17.13)

Let’s try this

More efficient vectorization (was 8.00)

CQA output after loop permutation

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).

Vector unaligned load/store instructions

Detected 2 optimal vector unaligned load/store instructions.

Details

- VMOVUPS: 2 occurrences

Workaround

Use vector aligned instructions:

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

Analyse matrix multiply with loop unrolling & array alignment

Compile array-aligned version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> make matmul_align
```

```
> ./matmul_align 400 300
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.18
```

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 results (HTML)

```
> tar czf $HOME/ov_align.tgz ov_align/RESULTS/matmul_align_one_html
```

```
> scp <login>@hawk.hww.hlrs.de:ov_align.tgz .
> tar xf ov_align.tgz
> firefox ov_align/RESULTS/matmul_align_one_html/index.html &
```

Global Metrics

Total Time (s)		1.28
Time in loops (%)		99.61
Time in innermost loops (%)		90.63
Time in user code (%)		99.61
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.38
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.02
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.05
	Nb Loops to get 80%	1

Faster (was 1.63)

Now OK (-funroll-loops prev. missing)

Summary of optimizations and gains

Baseline: 2.29 cycles/FMA

9.95x speedup

Action: loop permutation
Result: vectorization

Loop permutation: 0.23 cycles/FMA

12.72x speedup

Action: unrolling + alignment
Result: more efficient array accesses

Loop perm. + align: 0.18 cycles/FMA

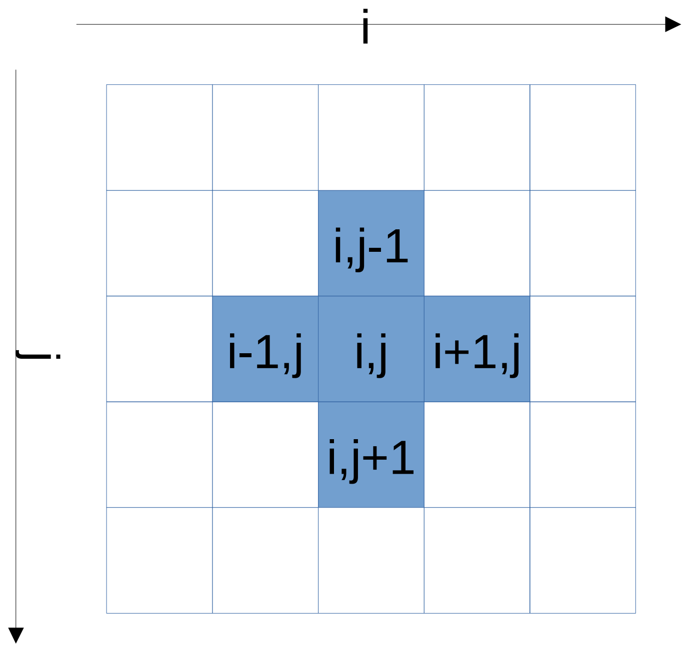
Hydro code

```
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



Preparing (new) interactive session with Intel compiler

If necessary, exit any active interactive session

```
> exit
logout
```

Request interactive session and load MAQAO environment

```
> qsub -I -l select=1,walltime=00:20:00 -q R_tw
> module load maqao
```

Load AMD compiler

```
> module swap gcc aocc
```

Define variable for workspace directory (if not already done)

```
> export WORK=/lustre/cray/ws9/...
```

Hydro example

Switch to the hydro handson folder

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

Compile

```
> make
```

Running and analyzing kernel0

```
> ./hydro_k0 300 50
Cycles per element for solvers: 2056.49
```

- 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:                    4.25 s
  Time spent in loops:           99.46 %
  Compilation Options:           OK
  Flow Complexity:               1.04
  Array Access Efficiency:       50.34 %
```

Running and analyzing kernel0

Source Code

```
/lustre/cray/ws9/6/ws/xhpeo-TW35/MAQAO_HANDSON/hydro//kernel.c: 104 - 110
-----------------------------------------------------------------------------
104:      for (j = 1; j <= grid_size; j++)
105:        {
106:        x[build_index(i, j, grid_size)] = (a * ( x[build_index(i-1, j, grid_size)] +
107:                                          x[build_index(i+1, j, grid_size)] +
108:                                          x[build_index(i, j-1, grid_size)] +
109:                                          x[build_index(i, j+1, grid_size)]) +
110:                                    x0[build_index(i, j, grid_size)]) / c;
111:        }
112:      }
```

CQA

← Path 0 / 1 OK →

Average path: Display a virtual path defined by average values of all real paths

Coverage	95.91 %
Function	linearSolver0
Source file and lines	kernel.c:104-110
Module	hydro_k0

The loop is defined in /lustre/cray/ws9/6/ws/xhpeo-TW35/MAQAO_HANDSON/hydro/kernel.c:104-110.

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 17% 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 5.00 to 0.62 cycles (8.00x speedup).

Details

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

Workaround

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

Execution units bottlenecks

Performance is limited by execution of divide and square root operations (the divide/square root unit is a bottleneck). By removing all these bottlenecks, you can lower the cost of an iteration from 5.00 to 3.00 cycles (1.67x speedup).

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

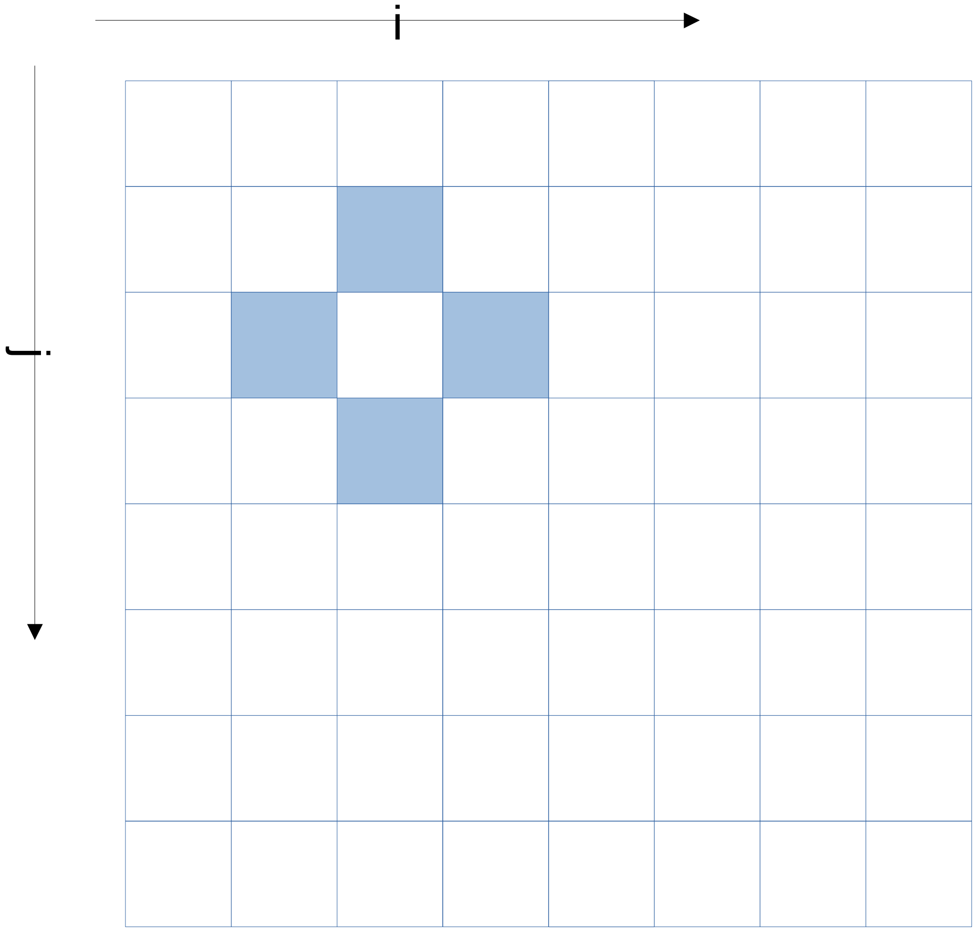
Loop is potentially data access bound.

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).

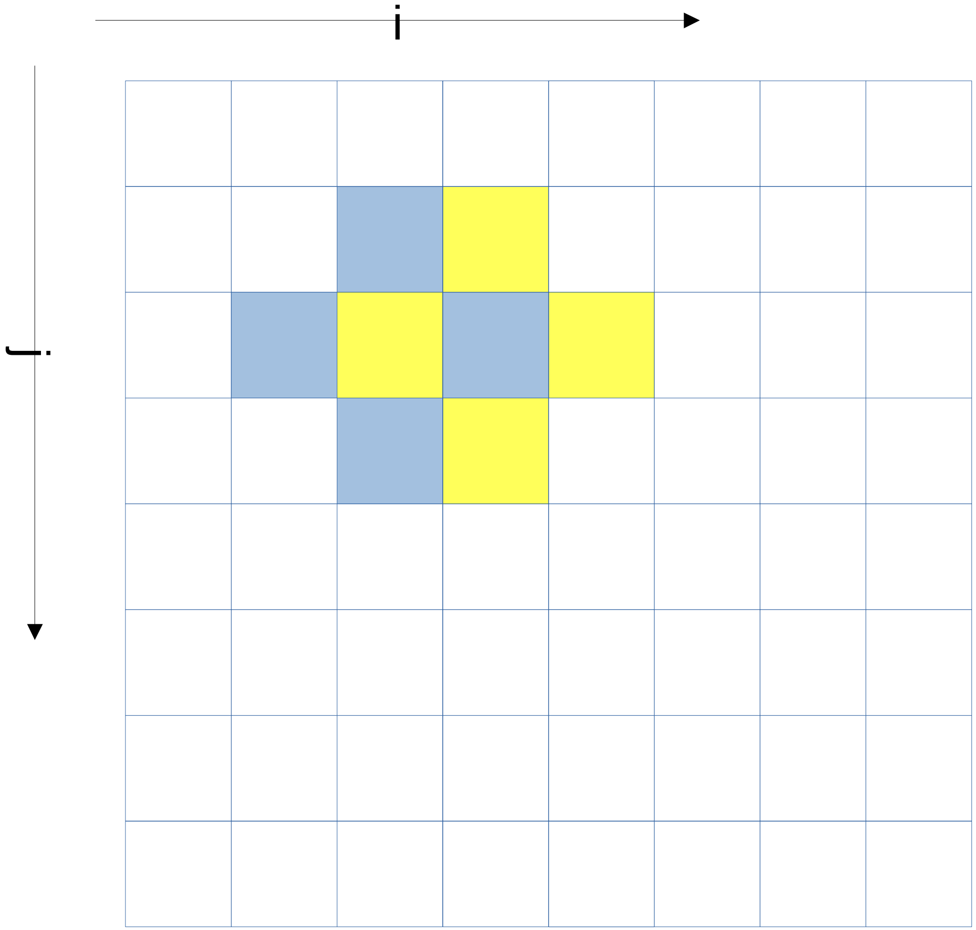
Unrolling is generally a good deal: fast to apply and often provides gain.
Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



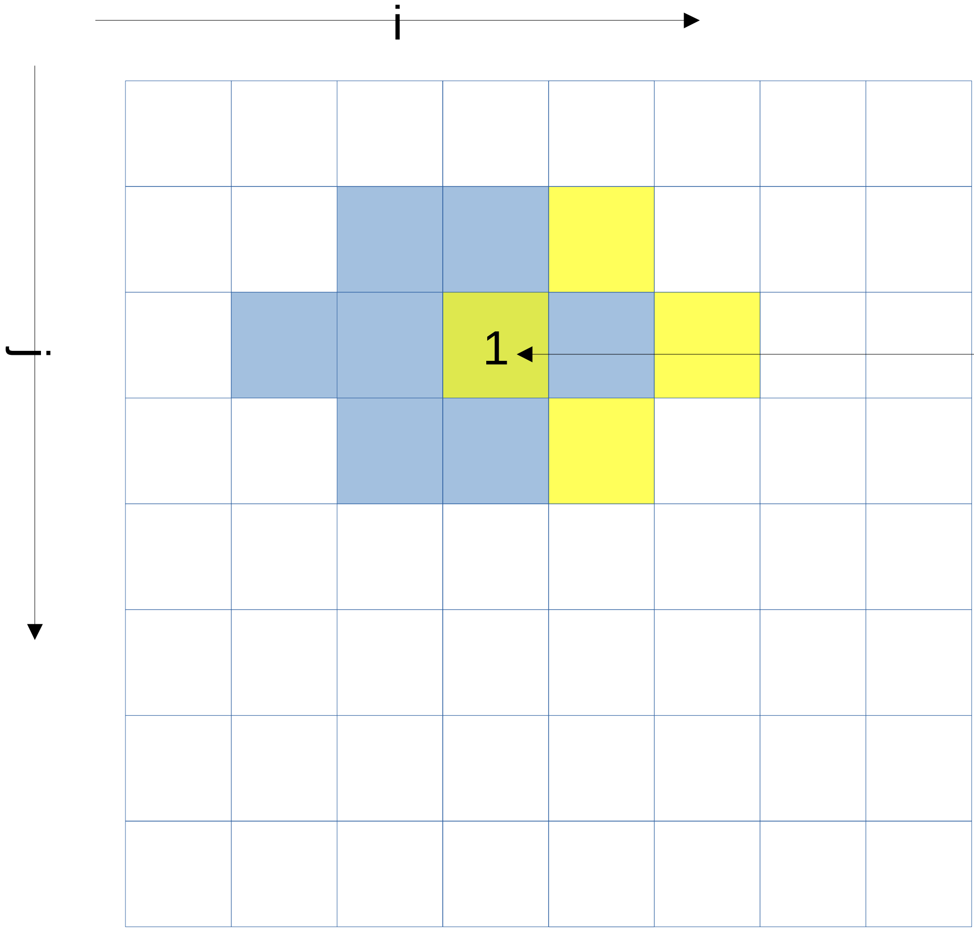
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

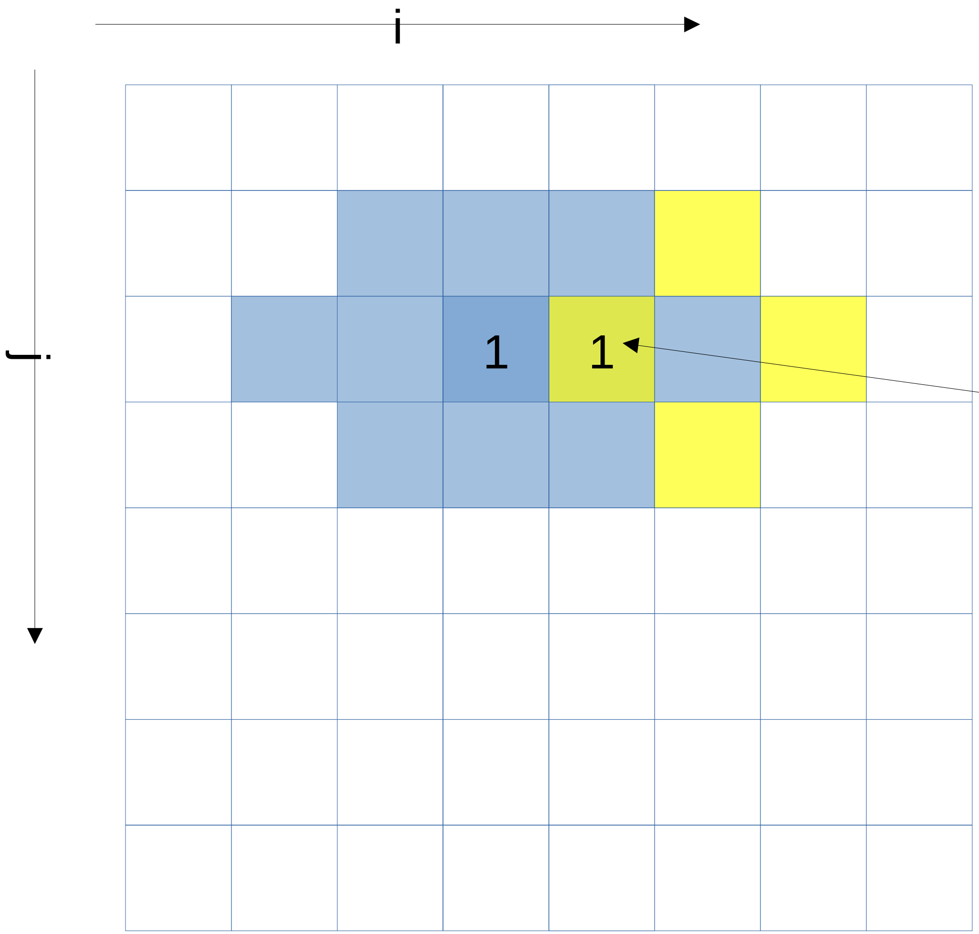
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

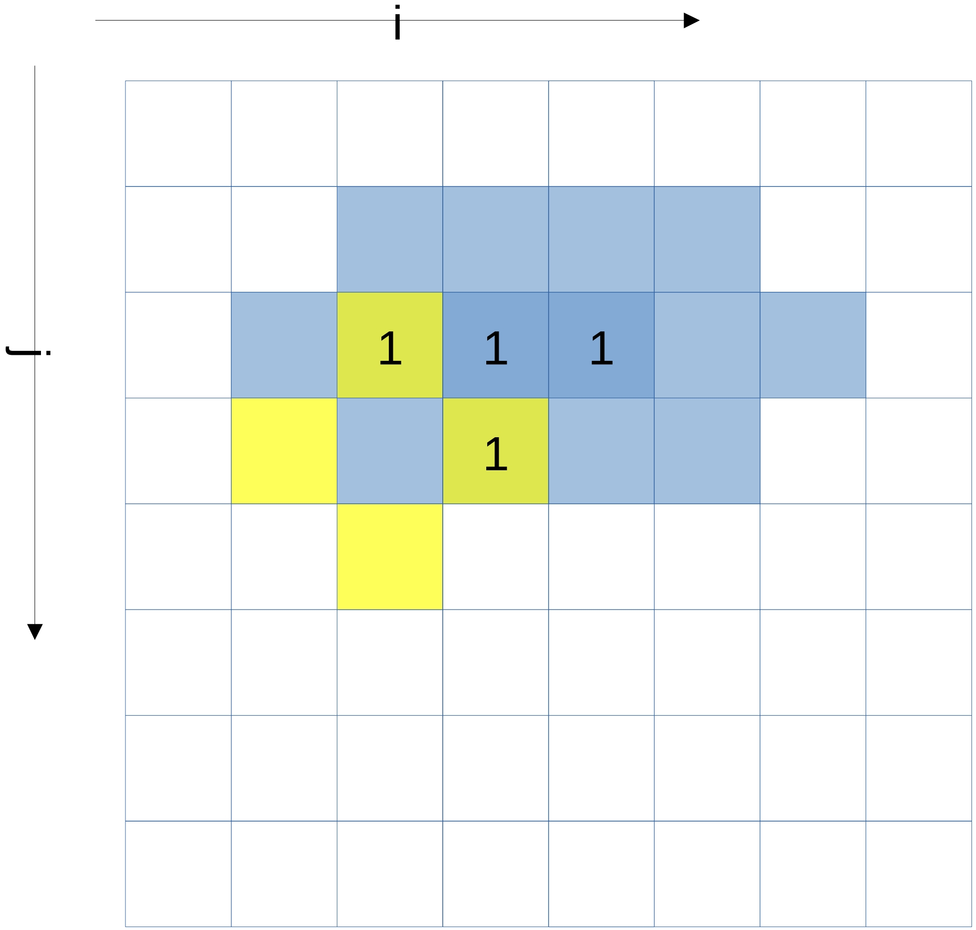
Memory references reuse : 4x4 unroll footprint on loads



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

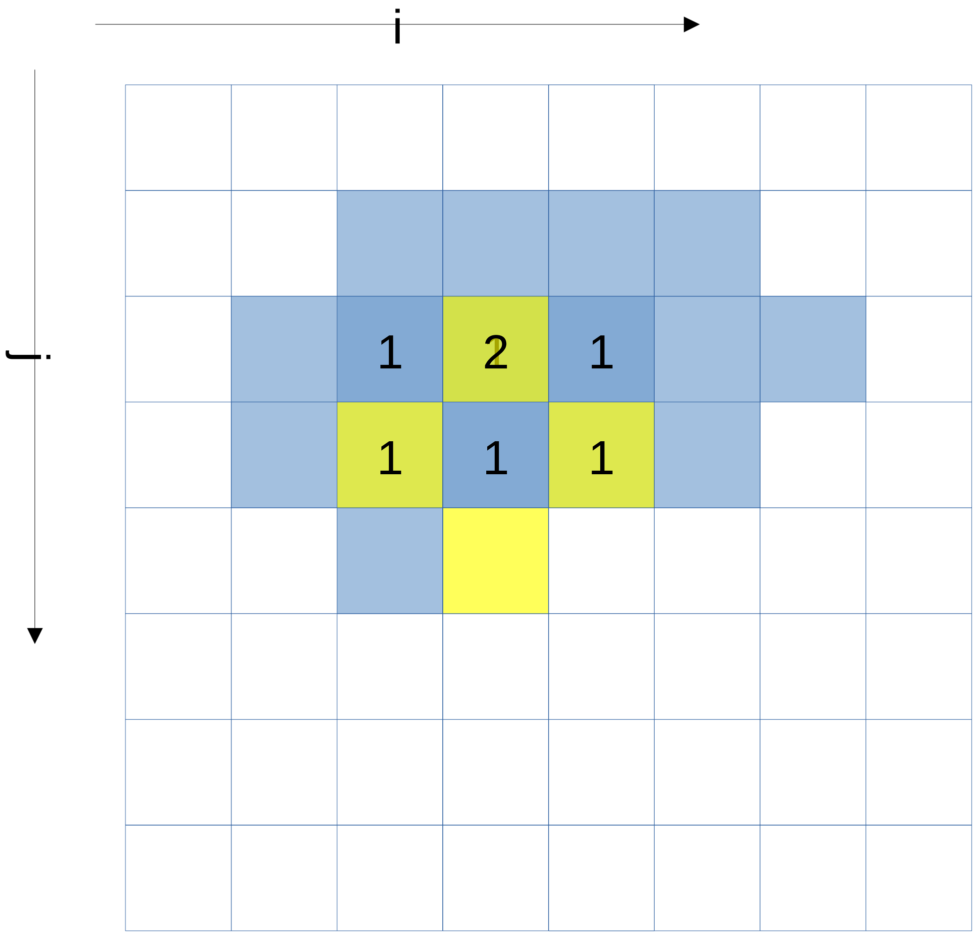


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

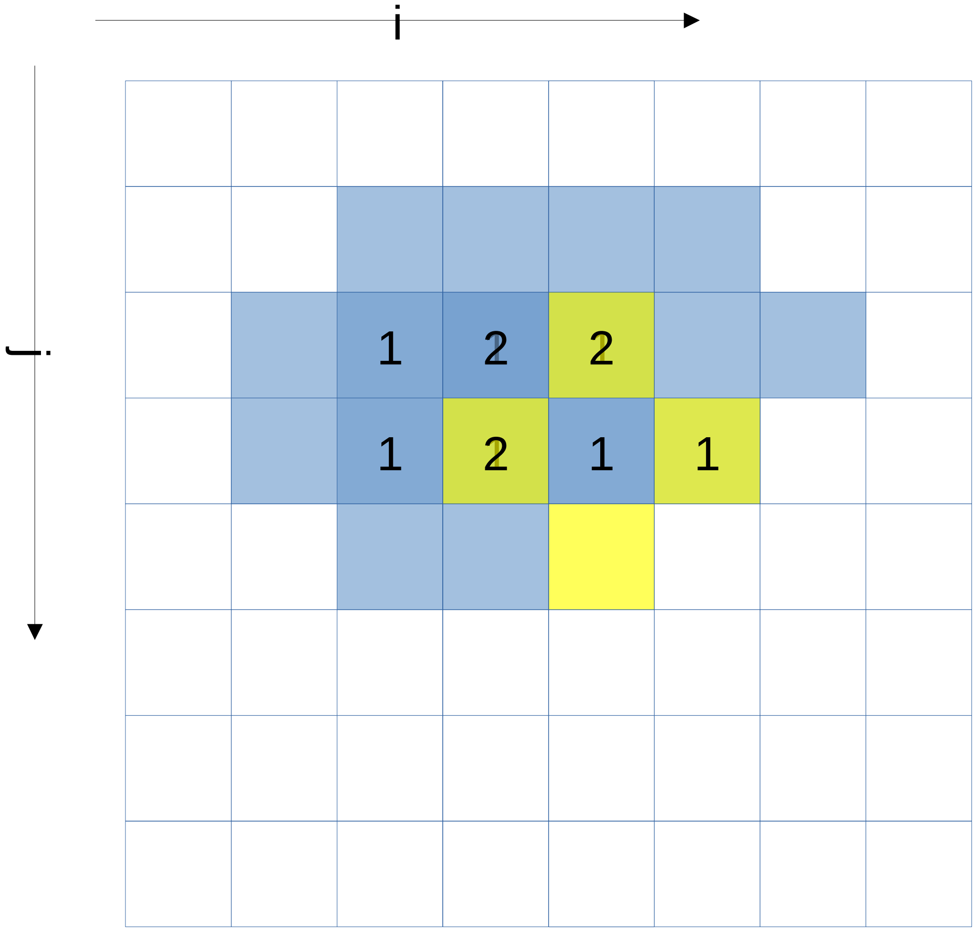


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

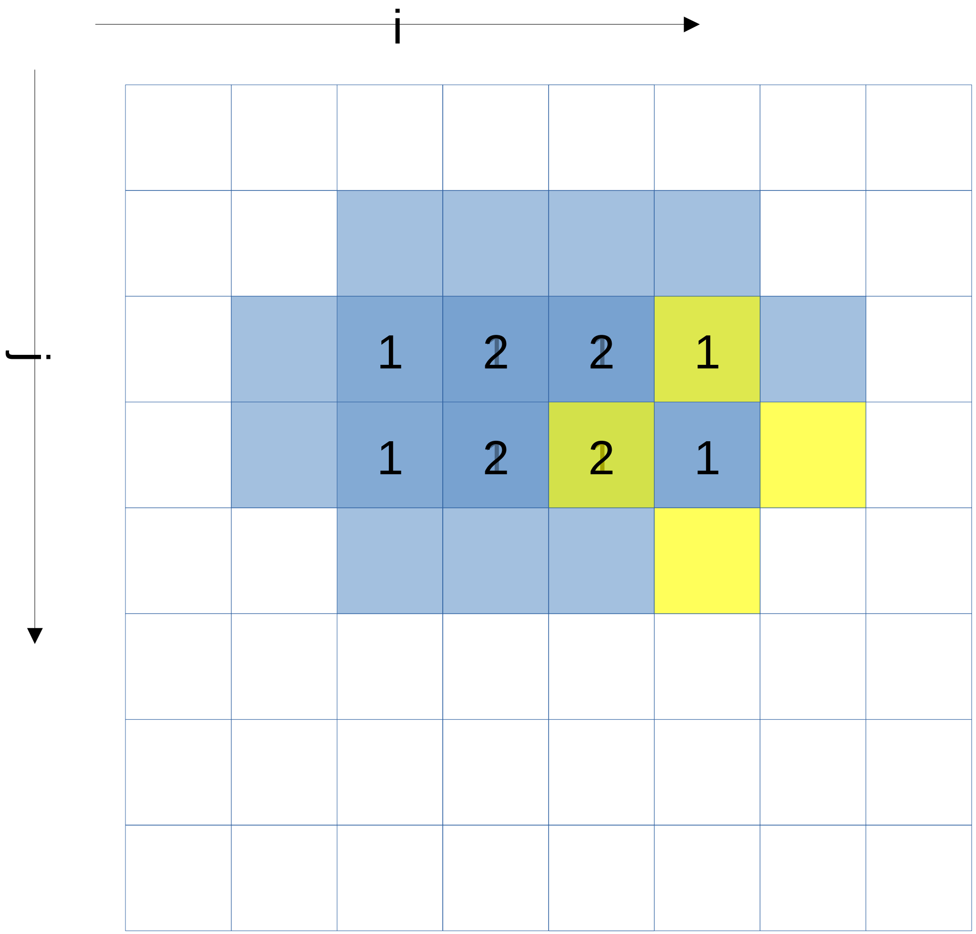


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

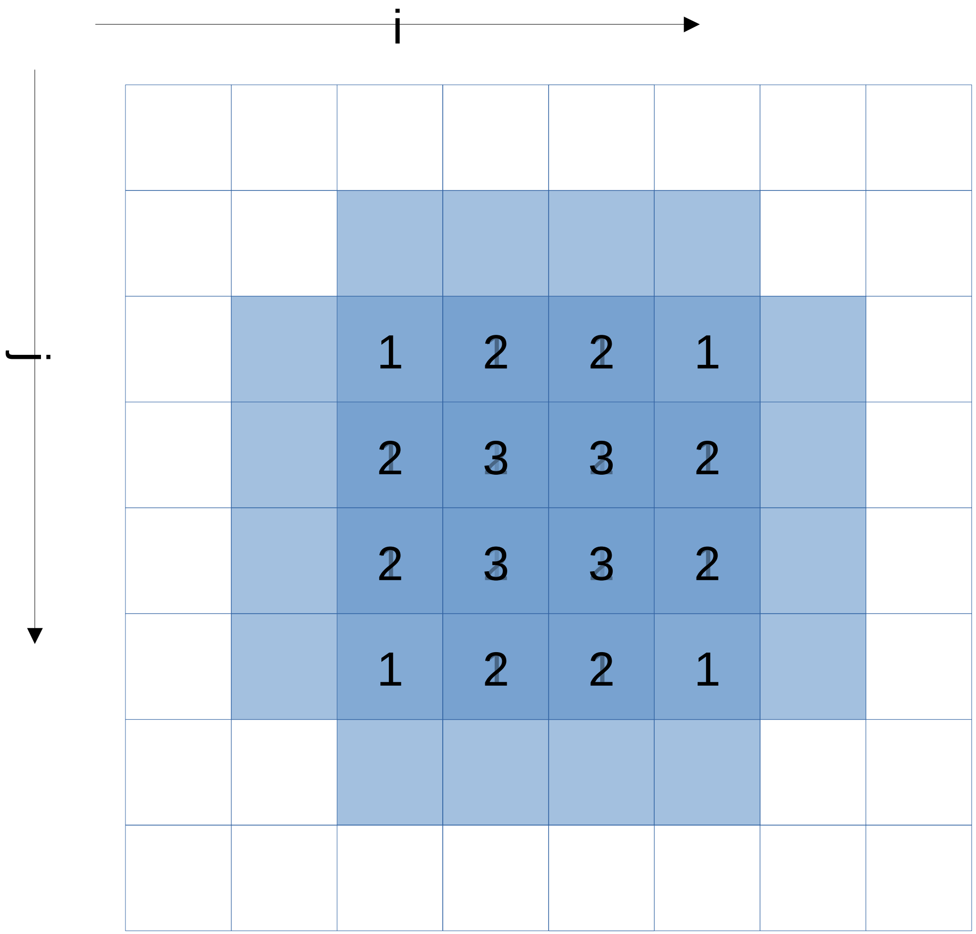


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

4x4 unroll

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

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

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

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

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

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

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

Running and analyzing kernel1

```
> ./hydro_k1 300 50
Cycles per element for solvers: 498.49
```

- 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:                     1 s
  Time spent in loops:            99.43 %
  Compilation Options:            OK
  Flow Complexity:                1.09
  Array Access Efficiency:        50.20 %
```

Running and analyzing kernel1

Source Code

/lustre/cray/ws9/6/ws/xhpeo-TW35/MAQAO_HANDSON/hydro//kernel.c: 156 - 176

```
------------------------------------------------------------------------------------
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:      }
```

CQA

← Path 0 / 1 OK →

Average path: Display a virtual path defined by average values of all real paths

Coverage	82.79 %
Function	linearSolver1
Source file and lines	kernel.c:156-176
Module	hydro_k1

The loop is defined in /lustre/cray/ws9/6/ws/xhpeo-TW35/MAQAO_HANDSON/hydro/kernel.c:156-176.

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

gain | potential | hint | expert

Vectorization

Your loop is not vectorized. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 40.00 to 5.00 cycles (8.00x speedup).

Details

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

Workaround

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

Execution units bottlenecks

Performance is limited by:

CQA output for kernel1

```
> maqao oneview create-report=one xp=ov_k1 \
output-format=text text-cqa=17 | less
```

Type of elements and instruction set

96 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

kernel0: 4.25s

4.25x speedup

Action: 4x4 unroll
Result: big loop body with mem reuse

Kernel1: 1.00s

More sample codes

More codes to study with MAQAO in

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

Scalability profiling of lulesh with MAQAO

Salah Ibnamar

Compiling Lulesh

Copy Lulesh sources to your working directory

```
> cd $WORK
> tar xvf $TW35/maqao/examples/lulesh2.0.3.tgz
```

Compile Lulesh

```
> cd lulesh
> module swap mpt openmpi
> make
```

(Optional) To execute a sample run of Lulesh:

```
> less job_lulesh.pbs
> qsub job_lulesh.pbs
```

Setup ONE View for scalability analysis

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

```
> cd $WORK/lulesh #if current directory has changed
> cp $WORK/MAQAO_HANDSON/lulesh/config_maqao_lulesh.lua .
> less config_maqao_lulesh.lua
```

```
binary = "./lulesh2.0"
…
run_command = "<binary> -i 10 -p -s 130"
…
batch_script = "job_lulesh_maqao.pbs"
…
batch_command = “qsub <batch_script>"
…
number_processes = 1
…
number_nodes = 1
…
mpi_command = “mpirun –n <number_processes>"
…
omp_num_threads = 1
…
multiruns_params = {
 {nb_processes = 1, nb_threads = 8, number_nodes = 1, …, run_command = nil, …},
 {nb_processes = 8, nb_threads = 1, number_nodes = 1, …, run_command = "<binary> -i 10 –p –s 65“},
 {nb_processes = 8, nb_threads = 1, number_nodes = 2, …, run_command = "<binary> -i 10 –p –s 65“},
 {nb_processes = 8, nb_threads = 8, number_nodes = 2, …, run_command = "<binary> -i 10 –p –s 65“},
}
```

Review jobscript for use with ONE View

All variables in the jobscript defined in the configuration file must be replaced with their name from it.
Retrieve jobscript modified for ONE View from the MAQAO_HANDSON directory.

```
> cd $WORK/lulesh #if current directory has changed
> cp $WORK/MAQAO_HANDSON/lulesh/job_lulesh_maqao.pbs .
> less job_lulesh_maqao.pbs
```

```
...
#PBS -l select=2<number_nodes>
...
export OMP_NUM_THREADS=8<omp_num_threads>
...
mpirun -n ... $EXE
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on lulesh (scalability mode)

Launch ONE View (execution will be longer!)

```
> module load maqao
> maqao oneview --create-report=one --with-scalability=on \
config=config_maqao_lulesh.lua xp=maqao_lulesh
```

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

```
> firefox
hawk_work/lulesh/maqao_lulesh/RESULTS/lulesh2.0_one_html/index.html
```

OR

```
> tar czf $HOME/lulesh_html.tgz \
maqao_lulesh/RESULTS/lulesh2.0_one_html
```

```
> scp <login>@hawk.hww.hlrs.de:lulesh_html.tgz .
> tar xf lulesh_html.tgz
> firefox maqao_lulesh/RESULTS/lulesh2.0_one_html/index.html
```

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