MAQAO
Hands-on exercises

Profiling bt-mz
Optimising a code

Setup

Login to the cluster with X11 forwarding

```
> ssh -Y <login>@login23-[1-4].hpc.itc.rwth-aachen.de (claix)
> ssh -Y <login>@login[1-4].barnard.hpc.tu-dresden.de (barnard)
```

Load VIHPS (incl. MAQAO) environment

```
> source /home/hpc/vihps-tw44/setup.sh (claix)
> source /projects/p_nhr_vihps/setup.sh (barnard)
```

Copy handson material to your WORK directory

```
> cd $VIHPS_WORKSPACE
> tar xf $VIHPS_ROOT/hands-on/maqao.tgz
> tar xf $VIHPS_ROOT/hands-on/NPB3.4-MZ-MPI.tgz
```

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

```
> module load MAQAO/2.19.0 intel/2022b
```

Setup (bt-mz compilation with debug symbols)

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

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

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

Compile bt-mz with debug information

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

Profiling bt-mz with MAQAO

Cédric VALENSI
Emmanuel OSERET

Setup ONE View for batch execution

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

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

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

```
“binary”: "bt-mz.C.x"
...
“batch_script”: "maqao_bt.slurm"
...
“batch_command”: "sbatch -A $SBATCH_ACCOUNT ... <batch_script>"
...
“number_processes”: 4
...
“number_nodes”: 2
...
“mpi_command”: "srun"
...
“envv_OMP_NUM_THREADS”: 24
```

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

```
...
#SBATCH -N 2 <number_nodes>
#SBATCH -n 4 <number_processes>
#SBATCH -c 24 <number_threads>
...
export OMP_NUM_THREADS=24<omp_num_threads>
...
srun ./bt-mz.C.x
<mpi_command> <run_command>
...
```

Launch MAQAO ONE View on bt-mz (batch)

Launch ONE View

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

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

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

Setup ONE View for scalability mode

Parameters for scalability mode are defined in multirun_params.

```
> less config_bt_oneview_sbatch.json
```

```
“binary”: "bt-mz.C.x"
“batch_script”: "maqao_bt.slurm"
“batch_command”: "sbatch -A $SBATCH_ACCOUNT <batch_script>"
“number_processes”: 4
“number_nodes”: 2
“mpi_command”: "srun"
“envv_OMP_NUM_THREADS”: 24
…
"multiruns_params": [
 { "name": "2P_1N", "number_nodes": 1, "number_processes": 2,
"number_processes_per_node": 2, "envv_OMP_NUM_THREADS": 24 },
 { "name": "2P_2N", "number_nodes": 2, "number_processes": 2,
"number_processes_per_node": 1, "envv_OMP_NUM_THREADS": 24 },
],
```

Launch MAQAO ONE View on bt-mz in scalability mode

Launch ONE View

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

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

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

Display MAQAO ONE View results

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

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

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

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

```
> scp <login>@:login23-[1-4].hpc.itc.rwth-aachen.de:ov_html.tgz .
OR
> scp <login>@:login[1-4].barnard.hpc.tu-dresden.de:ov_html.tgz .
> tar xf ov_html.tgz
> firefox <exp-dir>/RESULTS/bt-mz.C.x_one_html/index.html
```

A sample result directory is in **MAQAO_HANDSON/bt/bt-mz.C.x_one_html/**

Results can also be viewed directly on the console:

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

Optimising a code with MAQAO

Emmanuel OSERET

(screenshots done on barnard,
slightly different perf. on claix-2023)

Matrix Multiply code

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

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

"Naïve" dense matrix multiply implementation in C

Setup environment

Load VIHPS environment (if needed)

```
> source /projects/p_nhr_vihps/setup.sh (barnard)
> source /home/hpc/vihps-tw44/setup.sh (claix23)
```

Load MAQAO environment (if needed)

```
> module load MAQAO/2.19.0
```

Load latest GCC compiler (if not already loaded)

```
> module load development/24.04 (barnard only)
> module load GCC/13.2.0
```

Analysing matrix multiply with MAQAO

Compile naïve implementation of matrix multiply

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

Analyse matrix multiply with ONE View

```
> srun [-A $SBATCH_ACCOUNT] --reservation=$SBATCH_RESERVATION \
  maqao OV -R1 xp=ov_orig -- ./matmul_orig/matmul 150 15000
```

OR

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

Viewing results (HTML)

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

```
> scp <login>@login23-[1-4].hpc.itc.rwth-aachen.de:ov_orig.tgz .
OR
> scp <login>@login[1-4].barnard.hpc.tu-dresden.de:ov_orig.tgz .
> tar xf ov_orig.tgz
> firefox ov_orig/RESULTS/matmul_orig_one_html/index.html &
```

Global Metrics

Total Time (s)	20.40
Profiled Time (s)	20.40
Time in analyzed loops (%)	100
Time in analyzed innermost loops (%)	99.7
Time in user code (%)	100
Compilation Options Score (%)	50.0
Array Access Efficiency (%)	83.3

Potential Speedups

Perfect Flow Complexity		1.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	2.80
	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 OV -R1 -xp=ov_orig \
  --output-format=text --text-global | less
```

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

  Total Time:                     21.35 s
  Compilation Options:            binary:  -march=(target) is missing. -funroll-loops is
missing.
  Flow Complexity:                1.00
  Array Access Efficiency:        83.30 %
  If Clean:
      Potential Speedup:          1.00
      Nb Loops to get 80%:        1
  If FP Vectorized:
      Potential Speedup:          2.35
      Nb Loops to get 80%:        1
  If Fully Vectorized:
      Potential Speedup:          14.90
      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         |
   3       | matm...  | kernel_orig.c:6-10           | 0.02         |
```

Loop ID

Viewing CQA output (text)

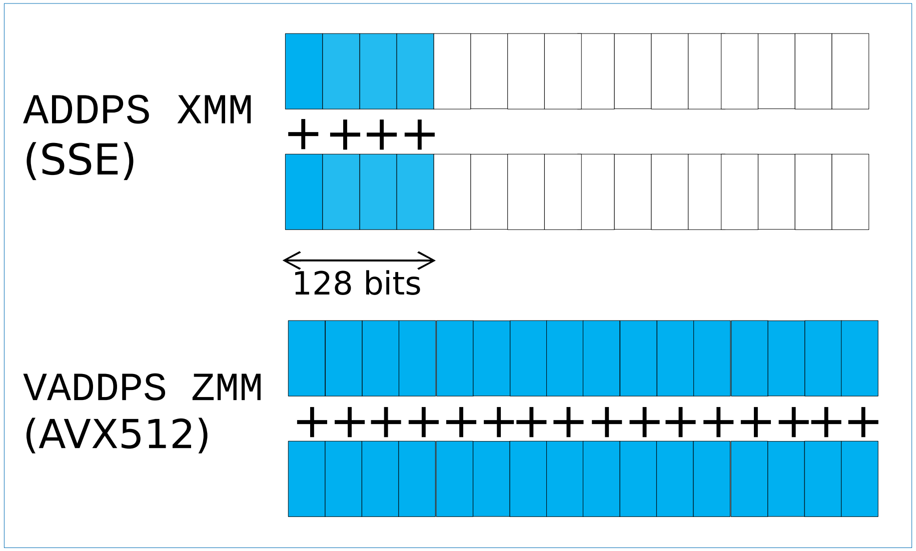
```
> maqao oneview -R1 -xp=ov_orig \
  --output-format=text --text-cqa=1
```

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 0.38 cycles
(8.00x speedup).
Workaround
 - Try another compiler or update/tune your current one:
  * recompile with fassociative-math (included in Ofast or ffast-math) to extend loop
vectorization to FP reductions.
 - Remove inter-iterations dependences from your loop and make it unit-stride:
  * If your arrays have 2 or more dimensions, check whether elements are accessed
contiguously and, otherwise, try to permute loops accordingly:
C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) =>
for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)
  * If your loop streams arrays of structures (AoS), try to use structures of arrays
instead (SoA):
for(i) a[i].x = b[i].x; (slow, non stride 1) => for(i) a.x[i] = b.x[i]; (fast, stride 1)
```

Impacts of architecture specialization: vectorization and FMA

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

Analyse matrix multiply with architecture specialisation

Compile architecture specialisation version of matrix multiply

```
> cd $VIHPS_WORKSPACE/MAQAO_HANDSON/matmul
> make matmul_opt
```

Analyse matrix multiply with ONE View

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

CQA output for the arch-specialized kernel

Global Metrics

Total Time (s)		21.00
Profiled Time (s)		21.00
Time in analyzed loops (%)		100.0
Time in analyzed innermost loops (%)		96.8
Time in user code (%)		100.0
Compilation Options Score (%)		100
Array Access Efficiency (%)		83.3
Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.01
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	2.01
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	15.9
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.01
	Nb Loops to get 80%	1

Not so faster (was 21.35) but arch-specialization will enable full vectorization

CQA output for the arch-specialized kernel

Vectorization

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

32 bits used

512 bits

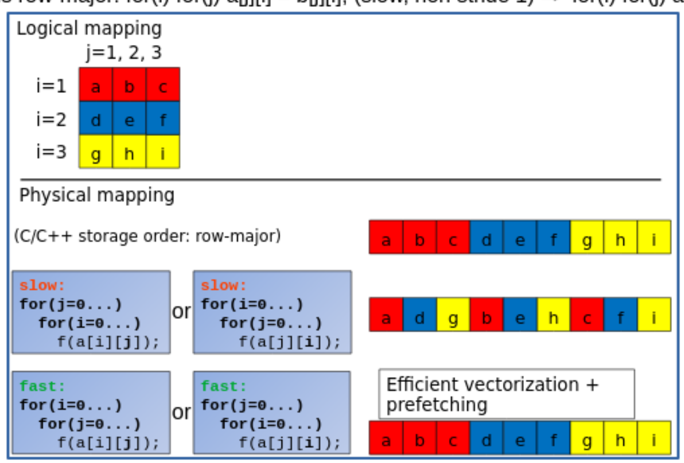
By vectorizing your loop, you can lower the cost of an iteration from 4.00 to 0.25 cycles (16.00x speedup).

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)

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

Analyse matrix multiply with ONE View

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

Loop permutation results

Global Metrics

Total Time (s)		4.36
Profiled Time (s)		4.36
Time in analyzed loops (%)		99.8
Time in analyzed innermost loops (%)		44.2
Time in user code (%)		99.8
Compilation Options Score (%)		100
Array Access Efficiency (%)		87.5
Potential Speedups		
Perfect Flow Complexity		1.00
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.26
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.27
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.80
	Nb Loops to get 80%	2

Faster (was 21.00)

Much better (was close to 16)

CQA output after loop permutation

gain | potential | hint | expert

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).

Execution units bottlenecks

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

Using comparison mode

Generating a comparison report from experiment directories

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

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

▼ Compared Reports

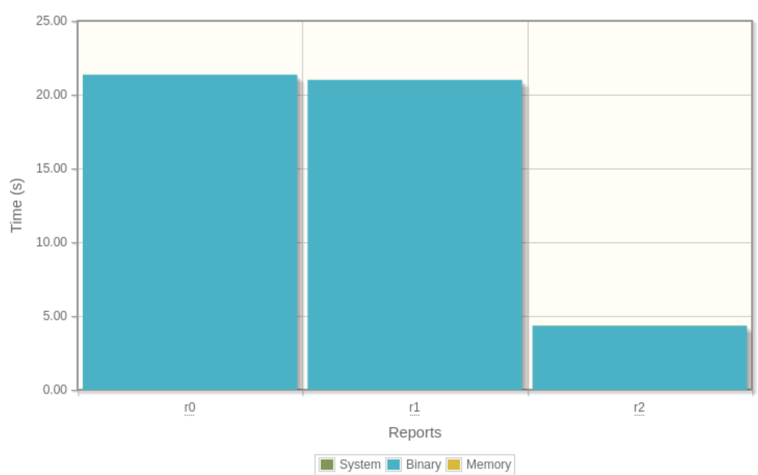
- r0: ov_orig
- r1: ov_opt
- r2: ov_perm_opt

Global Metrics

Metric		r0	r1	r2
Total Time (s)		21.35	21.00	4.36
Profiled Time (s)		21.35	21.00	4.36
Time in analyzed loops (%)		100	100.0	99.8
Time in analyzed innermost loops (%)		99.7	96.8	44.2
Time in user code (%)		100	100.0	99.8
Compilation Options Score (%)		50.0	100	100
Array Access Efficiency (%)		83.3	83.3	87.5
Potential Speedups				
Perfect Flow Complexity		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.00	1.01	1.26
	Nb Loops to get 80%	1	1	1
FP Vectorised	Potential Speedup	2.80	2.01	1.02
	Nb Loops to get 80%	1	1	1
Fully Vectorised	Potential Speedup	16.0	15.9	1.27
	Nb Loops to get 80%	1	2	1
Only FP Arithmetic	Potential Speedup	1.00	1.01	1.80
	Nb Loops to get 80%	1	1	2

Application Categorization

Time



Summary of optimizations and gains

Baseline: 21.35 seconds

1.01x speedup

Action: arch. specialization, loop unroll
Result: vectorization widened to 512b
+ FMA generation + loop unrolling

march + unroll: 21.00 seconds

4,90x speedup

Action: loop permutation
Result: vectorization

Loop perm. + march + unroll: 4.36 seconds

Hydro example

(screenshots done on barnard,
slightly different perf. on claix-2023)

Switch to the hydro handson folder

```
> cd $VIHPS_WORKSPACE/MAQAO_HANDSON/hydro
```

Load Intel compiler

```
> module purge (barnard only)
> module load MAQAO (barnard only)
> module load intel/2022b
```

Compile

```
> make
```

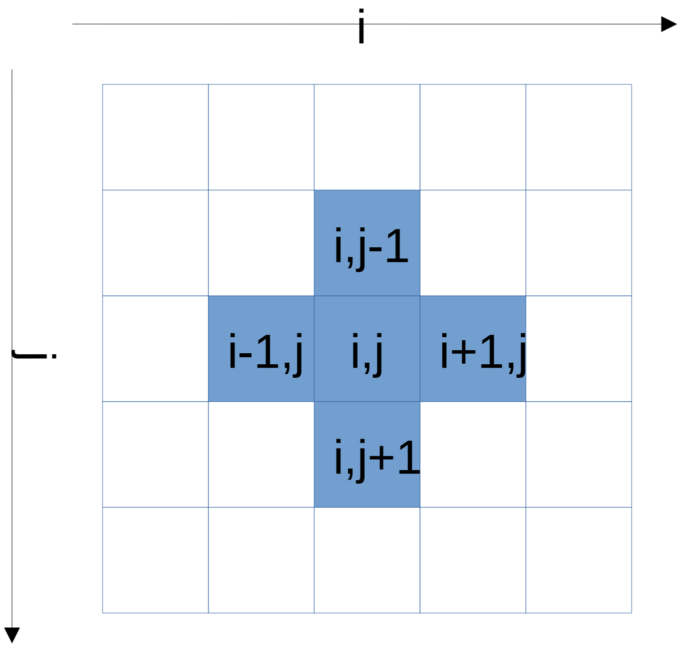
Hydro code

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

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

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

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



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

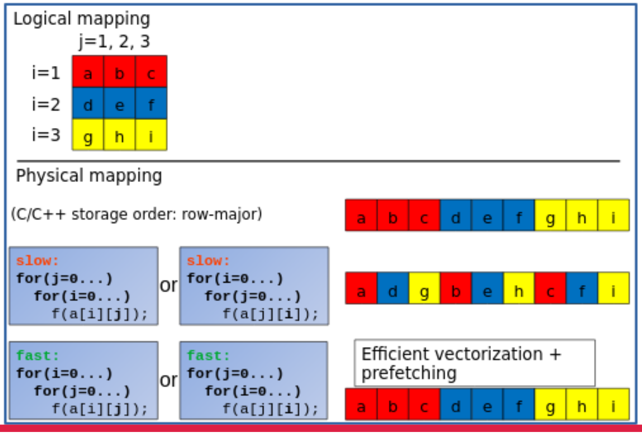
```
> maqao OV -R1 xp=ov_orig c=ov_orig.json
OR
> srun [-A $SBATCH_ACCOUNT] --reservation=$SBATCH_RESERVATION
maqao OV -R1 xp=ov_orig -- ./hydro_orig 300 200

> maqao OV -R1 xp=ov_orig \
--output-format=text --text-global --text-loops | less
> maqao oneview -R1 xp=ov_orig \
--output-format=text --text-global --text-cqa=18
> ...
> Total time: 8.61s
```

CQA output for original kernel

Workaround

- Try another compiler or update/tune your current one:
  - recompile with O2 or higher to enable loop vectorization and with ffast-math (included in Ofast) to extend vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: for(i) for(j) a[j][i] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)



As for matmul, loops should be permuted.
CF build_index

Unroll opportunity

Loop is data access bound.

Workaround

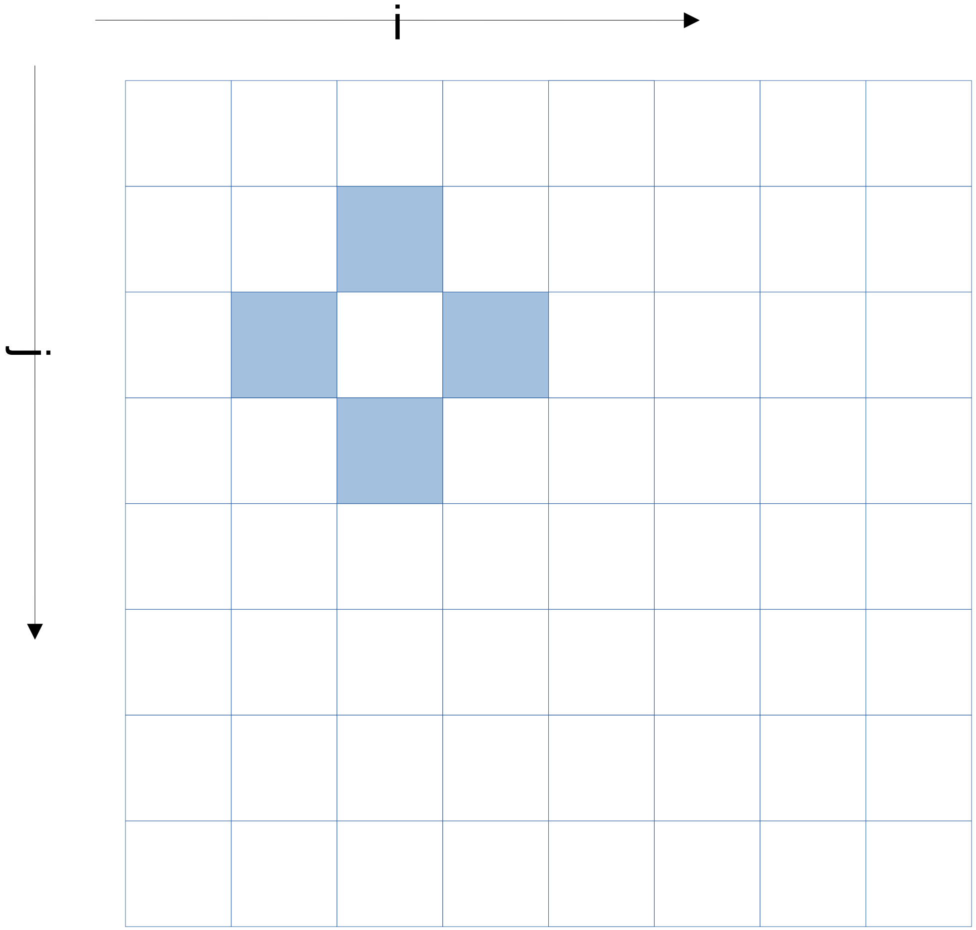
Unroll your loop if trip count is significantly higher than target unroll factor and if some data references are common to consecutive iterations. This can be done manually. Or by recompiling with -funroll-loops and/or -floop-unroll-and-jam.

Consider loop unrolling

Kernel with loop permutation

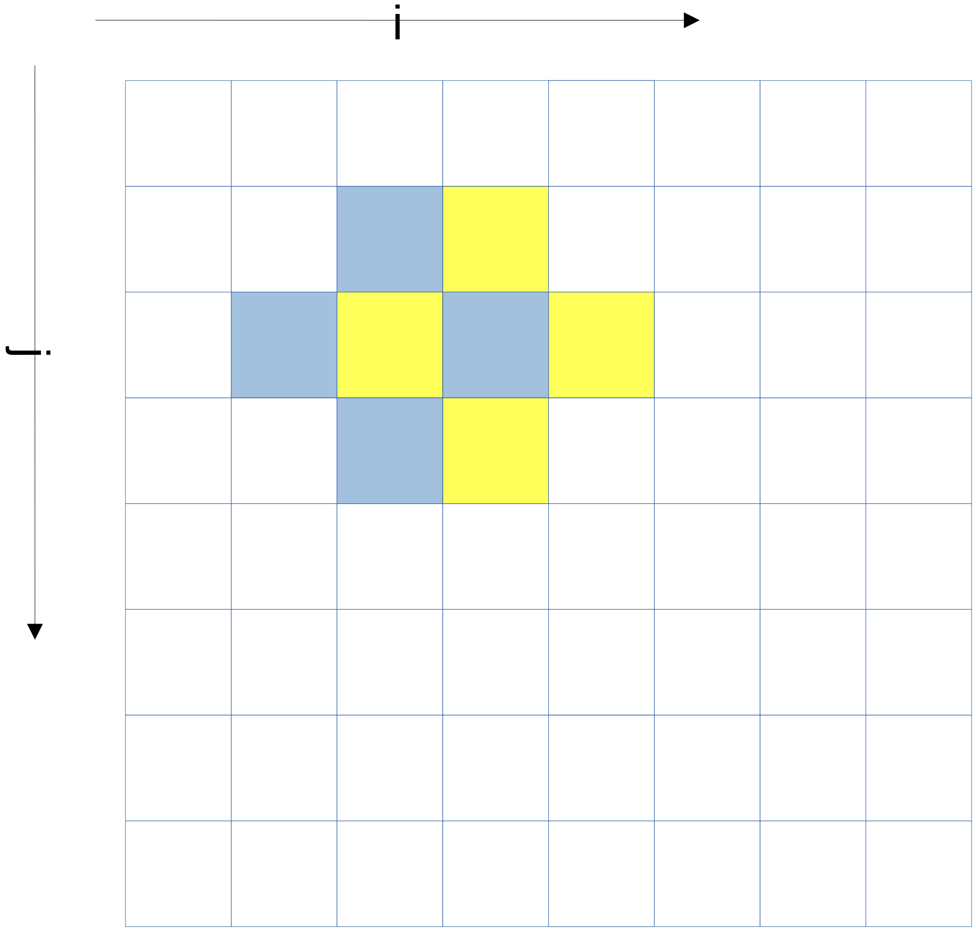
```
> maqao oneview -R1 xp=ov_perm c=ov_perm.json
> ...
> Total time: 9.28s
```

Memory references reuse : 4x4 unroll footprint on loads



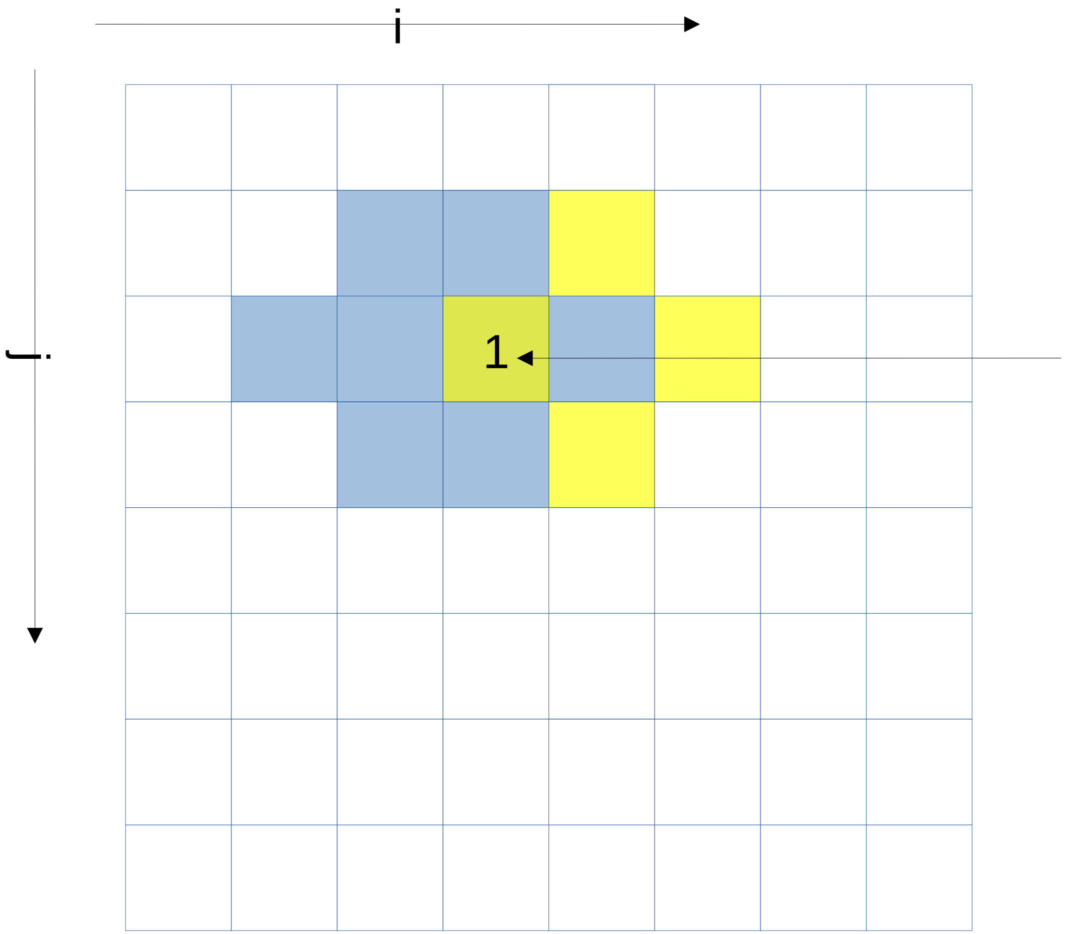
LINEAR_SOLVER(i+0,j+0)

Memory references reuse : 4x4 unroll footprint on loads



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

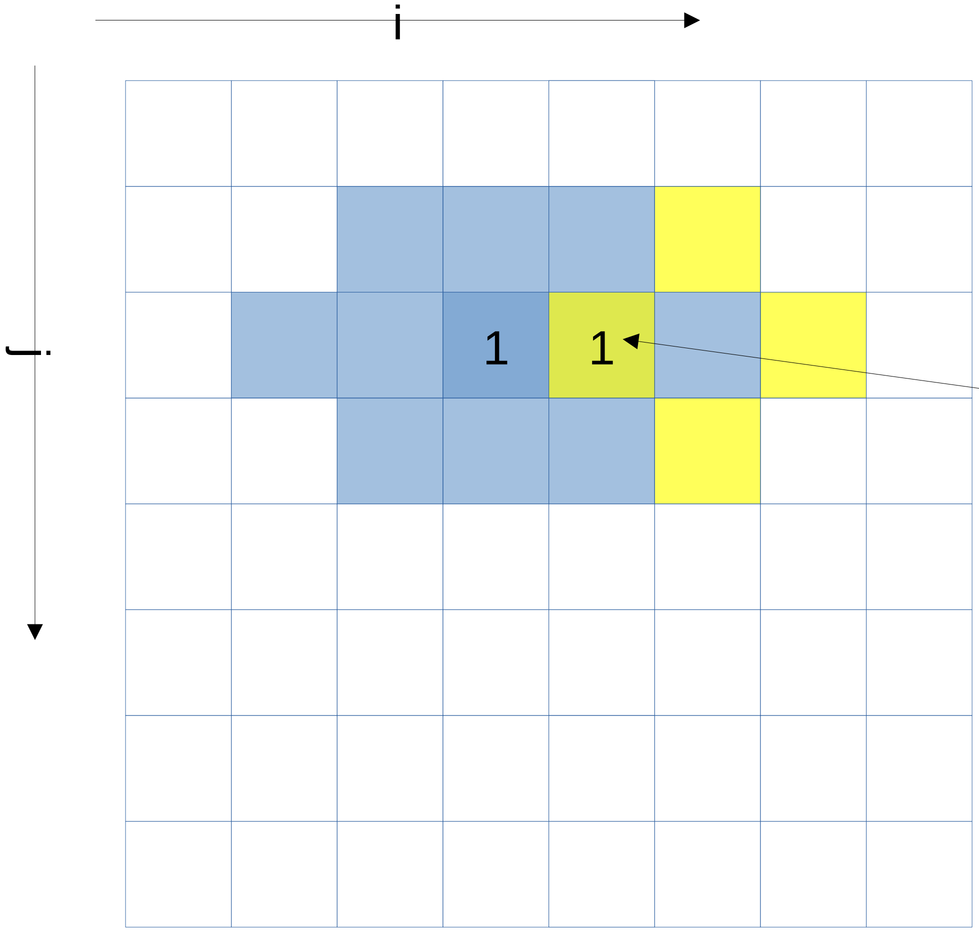
Memory references reuse : 4x4 unroll footprint on loads



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

1 reuse

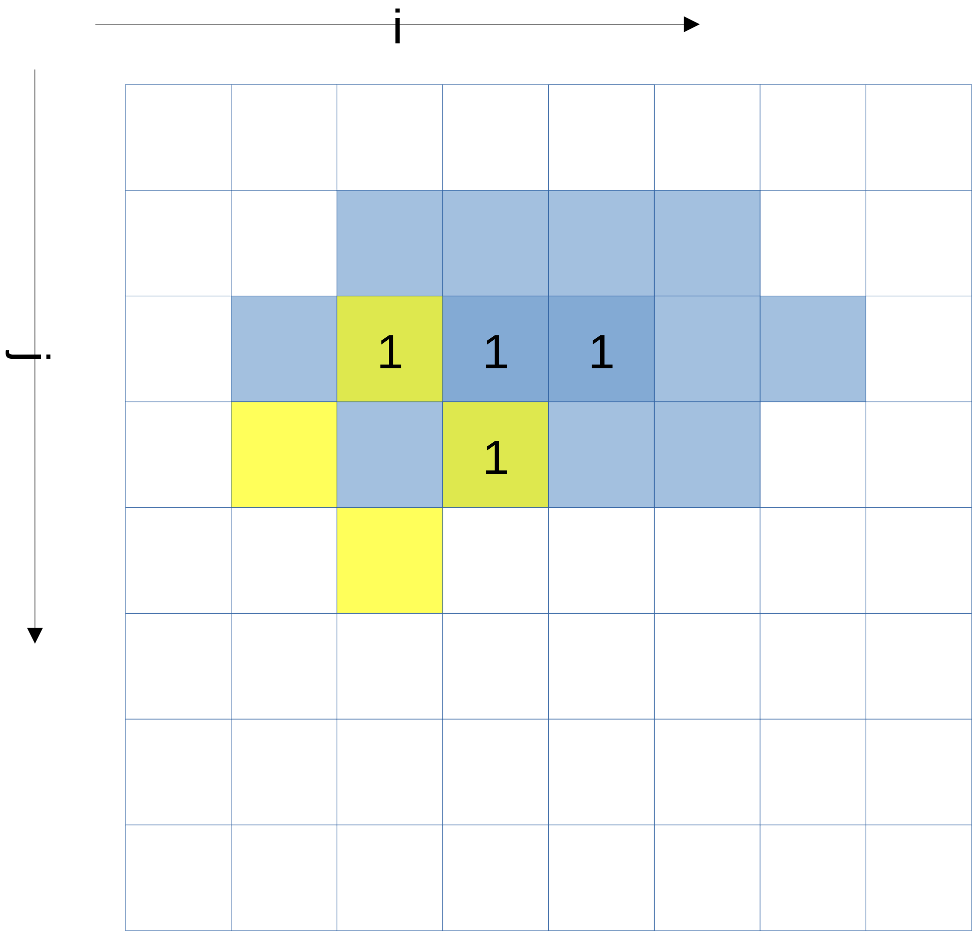
Memory references reuse : 4x4 unroll footprint on loads



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

2 reuses

Memory references reuse : 4x4 unroll footprint on loads

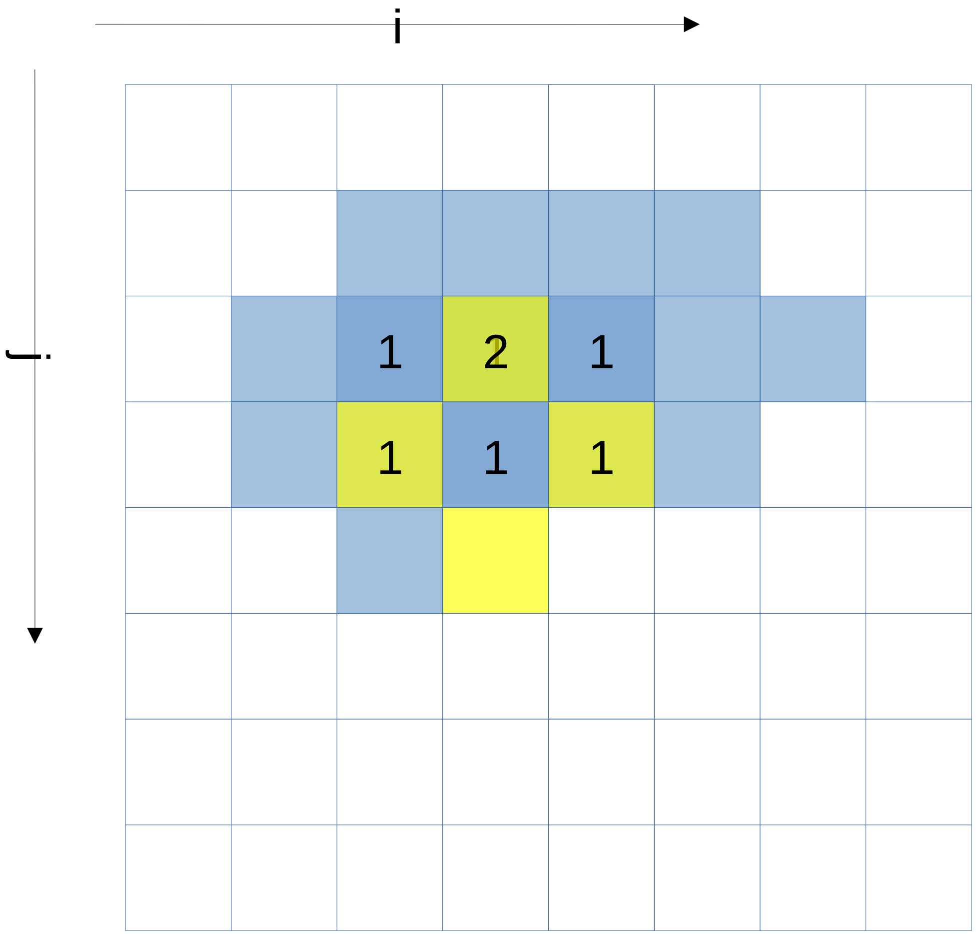


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

LINEAR_SOLVER(i+0,j+1)

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

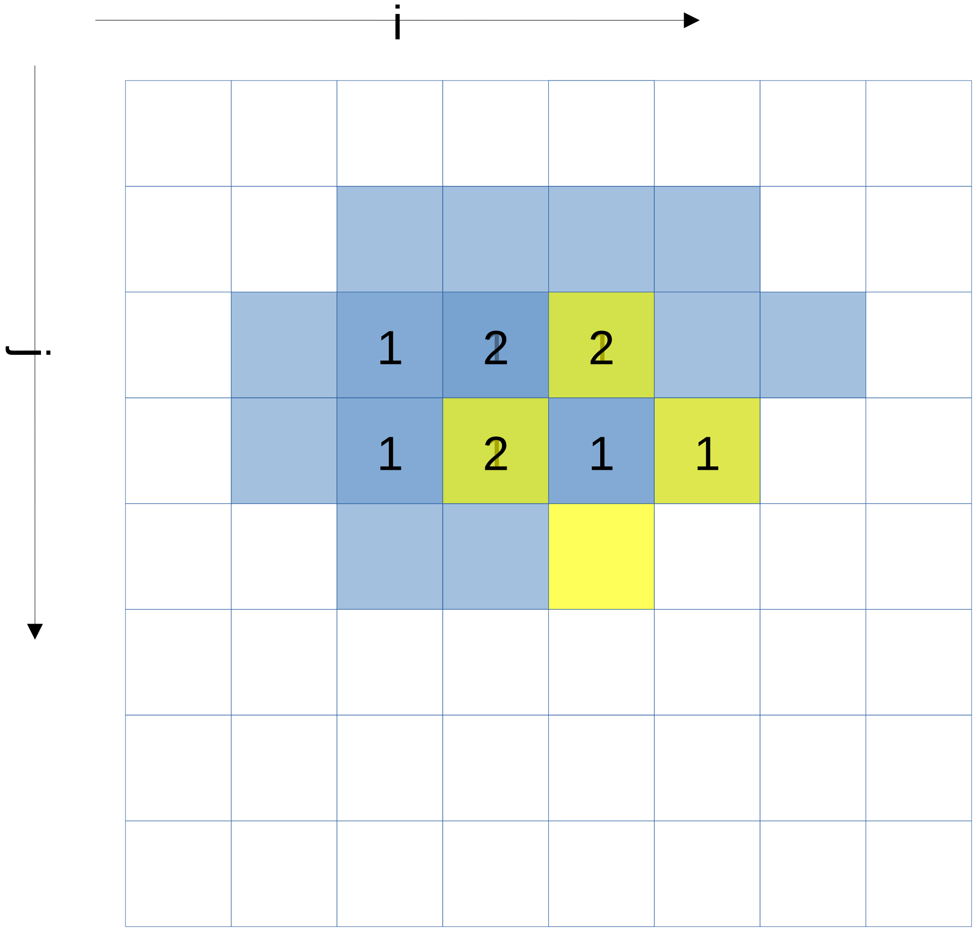


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

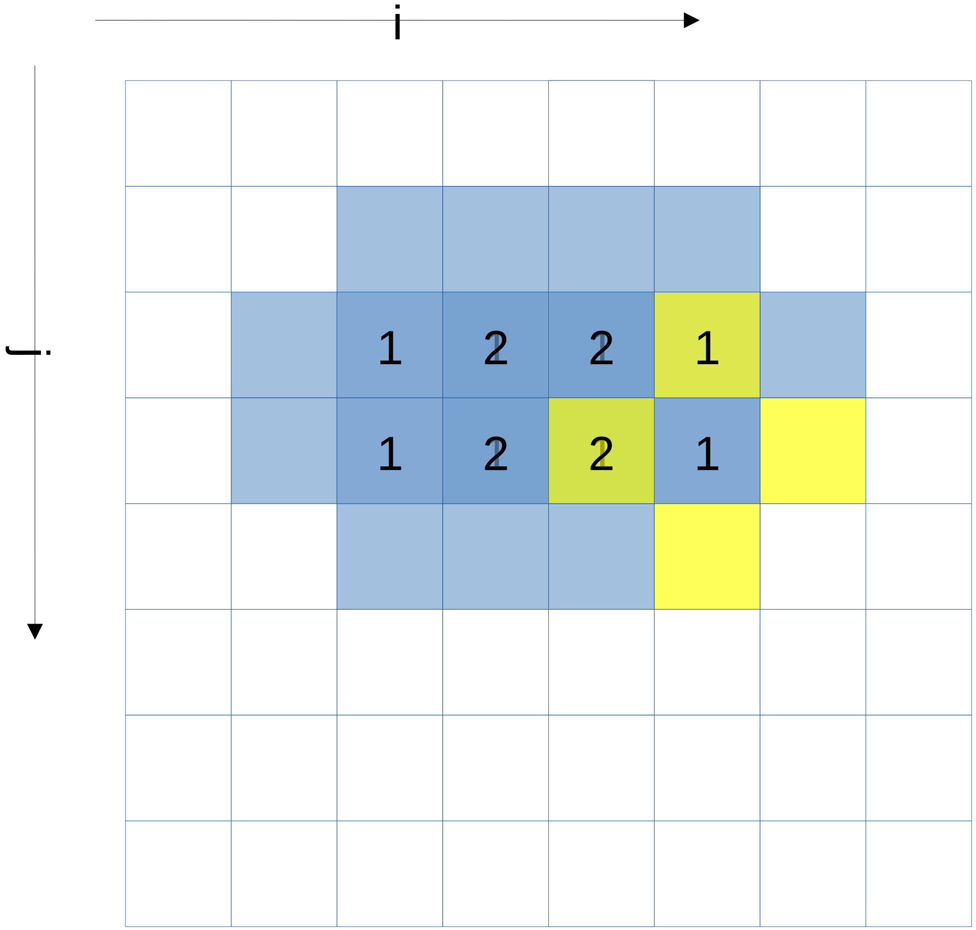


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

10 reuses

Memory references reuse : 4x4 unroll footprint on loads

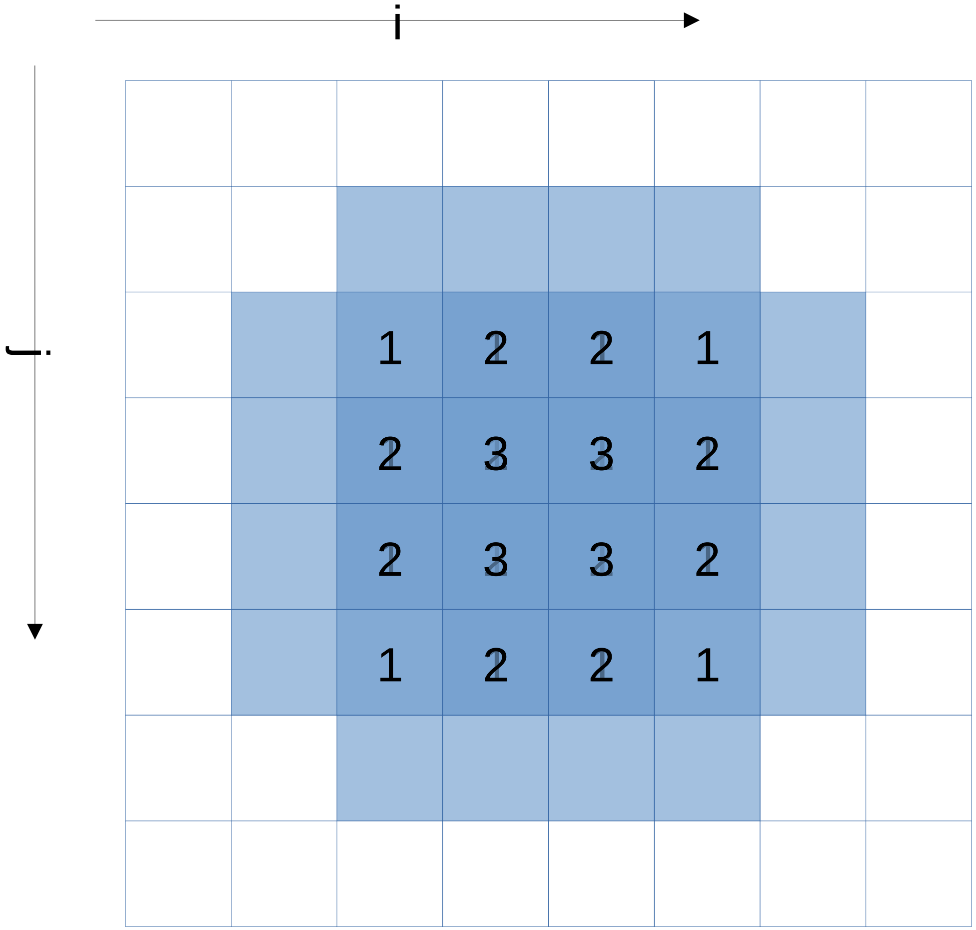


LINEAR_SOLVER(i+0,j+0)
LINEAR_SOLVER(i+1,j+0)
LINEAR_SOLVER(i+2,j+0)
LINEAR_SOLVER(i+3,j+0)

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

12 reuses

Memory references reuse : 4x4 unroll footprint on loads



LINEAR_SOLVER(i+0-3,j+0)

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

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

LINEAR_SOLVER(i+0-3,j+3)

32 reuses

4x4 unroll

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

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

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

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

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

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

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

Kernel with manual 4x4 unroll and jam

```
> maqao oneview -R1 xp=ov_unroll c=ov_unroll.json
> ...
> Total time: 4.02s
```

CQA output for unrolled kernel

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

The binary loop is composed of 96 FP arithmetical operations:

- 64: addition or subtraction (16 inside FMA instructions)
- 32: multiply (16 inside FMA instructions)

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

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

4x4 Unrolling were applied

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

Using comparison mode

Generating a comparison report from experiment directories

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

Open `ov_hydro_cmp/RESULTS/ov_hydro_cmp/index.html`

▼ Compared Reports

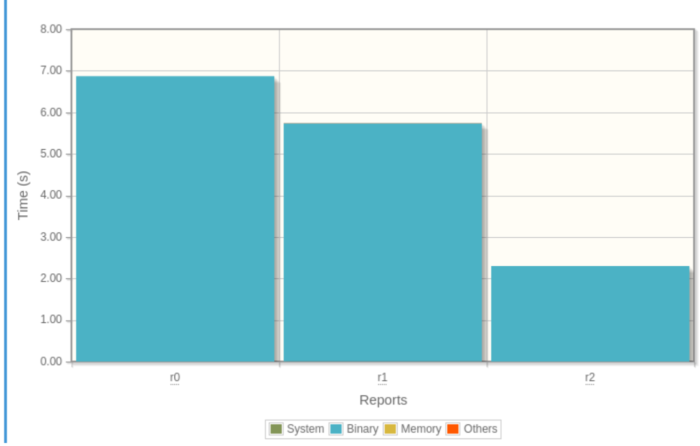
- r0: ov_orig
- r1: ov_perm
- r2: ov_unroll

Global Metrics

Metric		r0	r1	r2
Total Time (s)		6.87	5.75	2.50
Profiled Time (s)		6.87	5.74	2.30
Time in analyzed loops (%)		99.9	99.8	100.0
Time in analyzed innermost loops (%)		99.8	99.7	100.0
Time in user code (%)		99.9	99.8	100
Compilation Options Score (%)		100.0	100	100
Array Access Efficiency (%)		48.9	90.1	55.2
Potential Speedups				
Perfect Flow Complexity		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread		1.00	1.00	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00	1.00	1.00
No Scalar Integer	Potential Speedup	1.01	1.01	1.03
	Nb Loops to get 80%	3	4	4
FP Vectorised	Potential Speedup	1.63	2.27	1.68
	Nb Loops to get 80%	2	2	2
Fully Vectorised	Potential Speedup	13.7	12.9	10.3
	Nb Loops to get 80%	5	5	5
Only FP Arithmetic	Potential Speedup	1.07	1.08	1.21
	Nb Loops to get 80%	3	3	4

Application Categorization

Time



Summary of optimizations and gains

kernel0: 8.61s

2.14x speedup

Actions: loop perm, 4x4 unroll
Result: big loop body with mem reuse

Kernel1: 4.02s

More sample codes

More codes to study with MAQAO in

```
$VIHPS_WORKSPACE/MAQAO_HANDSON/loop_optim_tutorial.tgz
```