

MAQAO

Hands-on exercises

ONE View: Performance View Aggregator
LProf: Lightweight Profiler
CQA: Code Quality Analyzer

Setup

Login to the cluster with X11 forwarding

```
> ssh -Y <login>@lxlogin8.lrz.de
```

Load MAQAO environment

```
> source /home/hpc/a2c06/lu23vof/load_maqao.sh
```

Copy handson material to your SCRATCH_LEGACY directory

```
> cp $MAQAO_DIR/MAQAO_HANDSON_TW27.tgz $SCRATCH_LEGACY
> cd $SCRATCH_LEGACY
> tar xf MAQAO_HANDSON_TW27.tgz
```

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

```
FFLAGS = -O3 $(COMPFLAGS) -g
```

```
> cp MAQAO_HANDSON/make.def NPB3.3-MZ-MPI/config
> cd $SCRATCH_LEGACY/NPB3.3-MZ-MPI
> make bt-mz CLASS=C NPROCS=32
```

MAQAO ONE View Hands-on exercises

Cédric Valensi

Setup: Generating ONE View config file

Generate a Oneview config file template

```
> cd $SCRATCH_LEGACY  
> maqao oneview --create-config
```

Modify it to specify the execution parameters.

```
> vim config.lua
```

/!\ Environment variables are **not** accepted, but variables can be used

```
SCRATCH_LEGACY="/naslx/..."  
binary = SCRATCH_LEGACY.."/NPB3.3-MZ-MPI/bin/bt-mz_C.32"
```

Sample config files are located in the MAQAO_HANDSON/oneview directory.

Setup ONE View for batch mode

Modify configuration file (sample file ***config_bt_oneview_sbatch.lua***)

```
binary = "NPB3.3-MZ-MPI/bin/bt-mz_C.32"  
...  
sbatch_script = "bt_oneview.sbatch"  
...  
mpi_command = "mpiexec"  
...  
omp_num_threads = 4
```

Modify batch script to replace execution command

```
> cp NPB3.3-MZ-MPI/jobscript/coolmuc3/reference.sbatch bt_oneview.sbatch  
> vim bt_oneview.sbatch
```

```
...  
mpiexec ./NPB3.3-MZ-MPI/bin/bt-mz_C.32  
<run_command>  
...
```

A sample batch script is located in the MAQAO_HANDSON/oneview directory.

Setup ONE View for interactive mode

Modify configuration file (sample file ***config_bt_oneview_interactive.lua***)

```
> vim config.lua
```

```
binary = "NPB3.3-MZ-MPI/bin/bt-mz_C.32"  
...  
mpi_command = "mpiexec -n 32"  
...  
omp_num_threads = 4
```

Request interactive session

```
> salloc --nodes=2 --tasks-per-node=32
```

Launching MAQAO ONE View on bt-mz

Launch ONE View

```
> cd $SCRATCH_LEGACY  
> maqao oneview --create-report=one --config=config.lua --format=html
```

By default, the experiment directory `<exp_dir>` is named `maqao_<timestamp>`. Directory name can be specified using `-xp=<exp_dir>` (if `<exp_dir>` already exists, ONE View will reuse its content but not overwrite it).

HTML files are located in `<exp-dir>/RESULTS/bt-mz_one_html`

```
> firefox <exp-dir>/RESULTS/bt-mz_one_html/index.html &
```

It is also possible to download the results locally

```
> tar -zcf one_view.tar.gz <exp-dir>/RESULTS/bt-mz_one_html
```

Then download the archive locally, inflate it and view the `bt-mz_one_html/index.html` file through your browser.

MAQAO LProf Hands-on exercises

Cédric Valensi

LProf standalone execution

Modify batch script to replace execution command

```
> cd $SCRATCH_LEGACY
> cp NPB3.3-MZ-MPI/jobscript/coolmuc3/reference.sbatch bt_lprof.sbatch
> vim bt_lprof.sbatch

...
source /home/hpc/a2c06/lu23vof/load_maqao.sh
...
EXE="maqao lprof -- ./NPB3.3-MZ-MPI/bin/bt-mz_${CLASS}.${PROCS}"
...
```

A sample batch script is located in the MAQAO_HANDSON/lprof directory.

Submit the modified batch script

```
> sbatch bt_lprof.sbatch
```

By default, the experiment directory <exp_dir> is named
maqao_lprof_<timestamp>

Directory name can be specified using -xp=<exp_dir> (if <exp_dir> already exists, LProf will refuse to overwrite it).

Advanced profiling display with LProf

Display the results from an experiment stored in `<exp_dir>` in the console

```
> maqao lprof xp=<exp_dir> -df          #functions
> maqao lprof xp=<exp_dir> -dl          #loops
> maqao lprof xp=<exp_dir> -df -dt      #functions, thread breakdown
> maqao lprof xp=<exp_dir> -df -cv=full #categorization details
```

Generate and display the results in HTML format

```
> maqao lprof xp=<exp_dir> of=html      #html
```

HTML files are located in `<exp-dir>/html`

```
> firefox <exp-dir>/html/index.html &
```

It is also possible to display the profiling results from a ONE View experiment:

```
> maqao lprof of=html xp=<oneview-exp-dir>/lprof_npsu
```

MAQAO / CQA Hands-on exercises

Emmanuel OSERET

Setup for a Broadwell node

Remark: CQA can also be executed directly on a login node because it uses static analysis. Yet it is possible to cross-analyze to KNL (explained later in this presentation)

Load MAQAO (if not already done)

```
> source /home/hpc/a2c06/lu23vof/load_maqao.sh
```

Load a recent GCC compiler

```
> module load gcc/7
```

Switch to CQA handson folder

```
> cd $SCRATCH_LEGACY/MAQAO_HANDSON/cqa/matmul
```

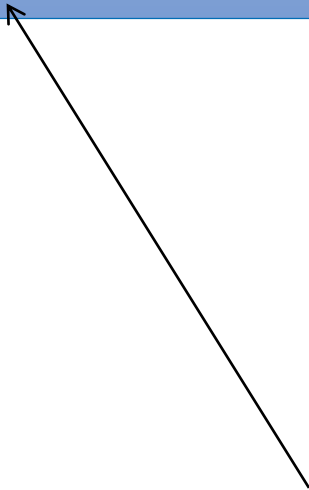
Matrix Multiply code

```
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

Compiling, running and analyzing kernel0 in -O3

```
> make OPTFLAGS=-O3 KERNEL=0
> ./matmul 100 1000
Cycles per FMA: 1.89
> maqao cqa matmul fct-loops=kernel0 [of=html]
```



NB: the usual way to use CQA consists in finding IDs of hot loops with the MAQAO profiler and forwarding them to CQA (loop=17,42...), or using oneview. To simplify this hands-on, we will bypass profiling and directly requesting CQA to analyze all innermost loops in functions (max 2-3 loops/function for this hands-on).

CQA output for kernel0 (from the "gain" confidence level)

Vectorization

(...) By fully vectorizing your loop, **you can lower the cost of an iteration from 3.00 to 0.38 cycles (8.00x speedup)** . (...)

- Remove inter-iterations dependences from your loop and make it unit-stride.

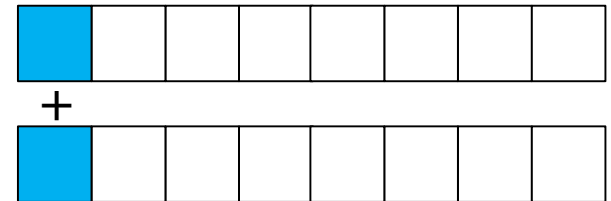
- * If your arrays have 2 or more dimensions, **check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly:**

C storage order is row-major: `for(i)`
`a[j][i] = b[j][i]; (slow, non stride 1)`
=> `for(i) for(j) a[i][j] = b[i][j]; (fast, stride 1)`

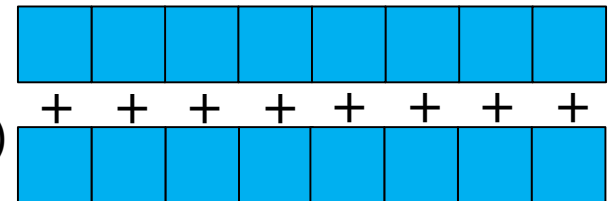
- * If your loop streams arrays of structures (AoS), try to use (...) SoA

Vectorization (summing elements):

VADDSS
(scalar)



VADDPS
(packed)



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

CQA output for kernel0 (from the "gain" confidence level)

Performance Evaluation - Profi... x +

file:///home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/matmul/cqa_html/index.ht Search

MAQAO

Code quality analysis

Target processor is: Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition (x86_64 architecture).

Source loop ending at line 10 in ...AO_HANDSON/cqa/matmul/kernel.c

It is composed of the loop 1

MAQAO binary loop id: 1

The loop is defined in /home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/matmul/kernel.c:9-10
The related source loop is not unrolled or unrolled with no peel/tail loop
2% of peak computational performance is used (0.67 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

Gain Potential gain Hints Experts only

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).
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.

Proposed solution(s):

- 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[i][j] = 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

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

Impact of loop permutation on data access

Logical mapping

$j=0,1\dots$

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

Efficient vectorization +
prefetching

Physical mapping

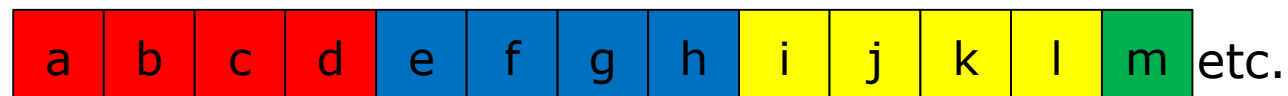
(C stor. order: row-major)



```
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 kernell (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];  
    }  
}
```


Kernel1: loop interchange

```
> make clean
> make OPTFLAGS=-O3 KERNEL=1
> ./matmul 100 1000
Cycles per FMA: 0.41
> maqao cqa matmul fct-loops=kernel1
```

CQA output for kernel1

Vectorization

Your loop is vectorized, but using only 128 out of 256 bits (SSE/AVX-128 instructions on AVX/AVX2 processors).

Workaround(s):

- **Recompile with march=broadwell.** CQA target is Core_i7X_Xeon_E5_v4... but specialization flags are -march=x86_64
- Use vector aligned instructions...

FMA

Presence of both ADD/SUB and MUL operations.

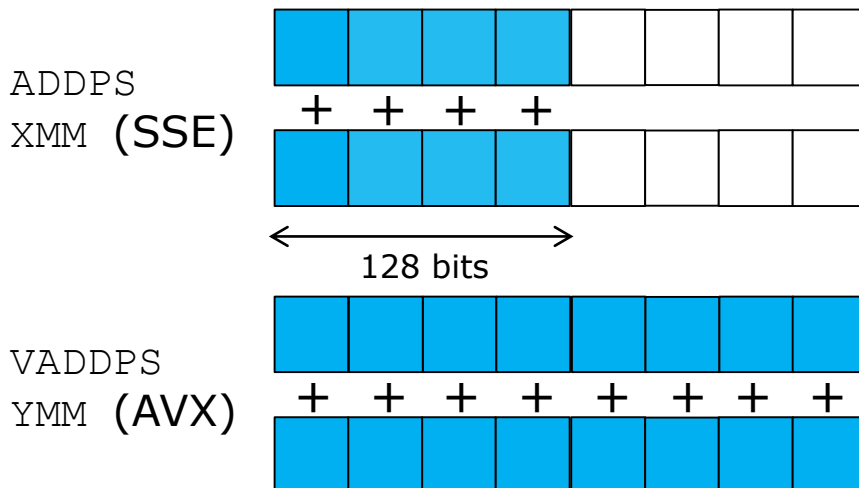
- **Recompile with march=broadwell.**
- Try to change order in which...

- Let's add -march=broadwell to OPTFLAGS

Impacts of architecture specialization: vectorization and FMA

▪ Vectorization

- SSE instructions (SIMD 128 bits) used on a processor supporting AVX ones (SIMD 256 bits)
- => 50% efficiency loss



▪ FMA

- Fused Multiply-Add ($A + BC$)
- Intel architectures: supported on MIC/KNC and Xeon starting from Haswell

$A = A + BC$

VMULPS , <C>, %XMM0

VADDPS <A>, %XMM0, <A>

can be replaced with something like:

VFMADD312PS , <C>, <A>

Kernel1 + -march=native

```
> make clean
> make OPTFLAGS="-O3 -march=broadwell" KERNEL=1
> ./matmul 100 1000
Cycles per FMA: 0.32
> maqao cqa matmul fct-loops=kernel1 --confidence-
levels=gain,hint
```

CQA output for kernel1 (using "gain" and "hint" conf. levels)

Vectorization status

Your loop is fully vectorized, using full register length

Vector unaligned load/store...

- Use vector aligned instructions:

1) align your arrays on 32 bytes

boundaries: replace { void *p = malloc... }

2) inform your compiler that your arrays are vector aligned: if array 'foo' is 32 bytes-aligned, define a pointer 'p_foo' as

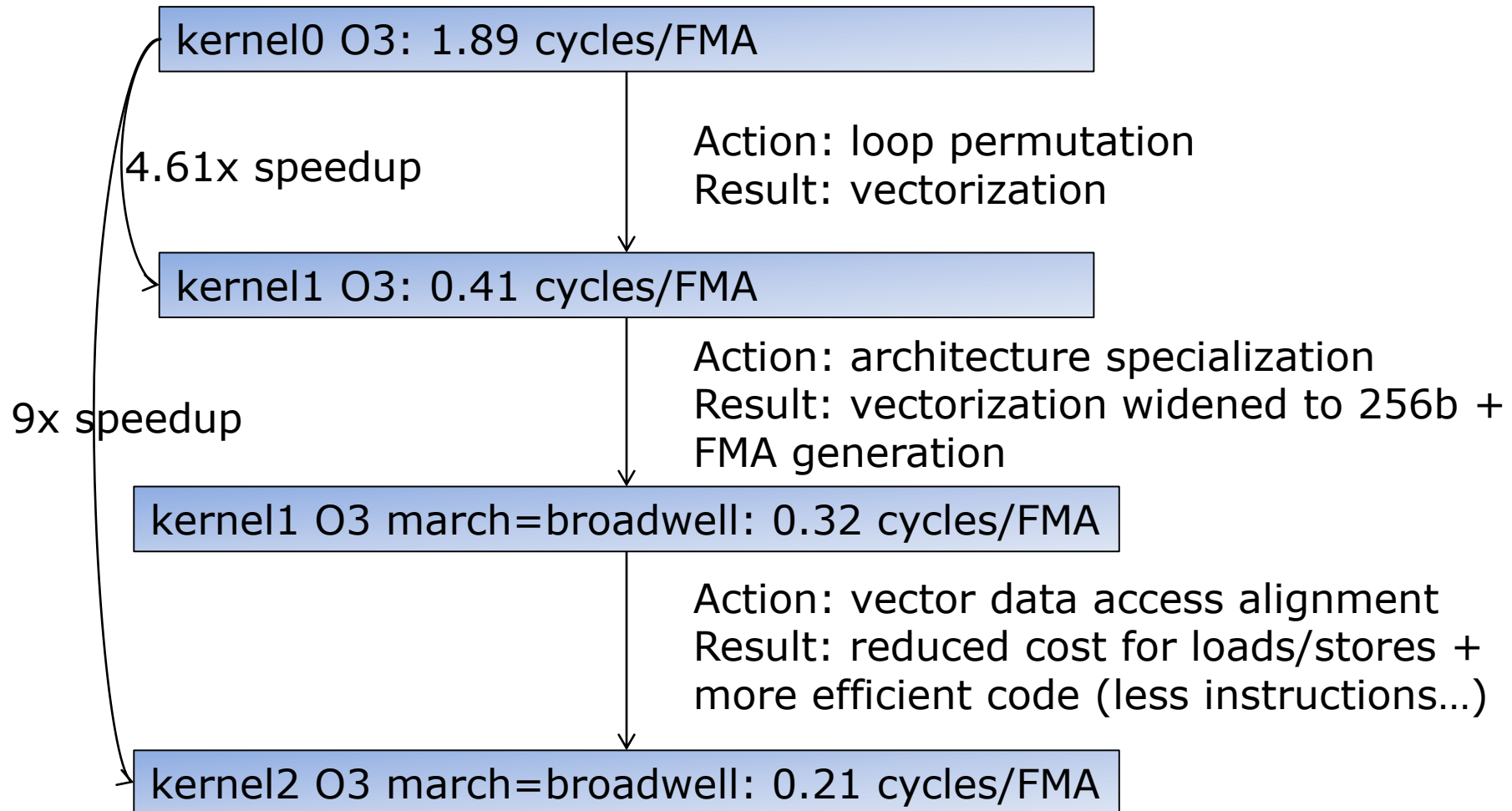
`__builtin_assume_aligned (foo,32)...`

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

kernel2: assuming aligned vector accesses

```
> make clean
> make OPTFLAGS="-O3 -march=broadwell" KERNEL=2
> ./matmul 100 1000
Cannot call kernel2 on matrices with size%8 != 0 (data non
aligned on 32B boundaries)
Aborted
> ./matmul 104 1000
Cycles per FMA: 0.21
```

Summary of optimizations and gains



Hydro example

Switch to the other CQA handson folder

```
> cd $HOME/MAQAO_HANDSON/cqa/hydro
```

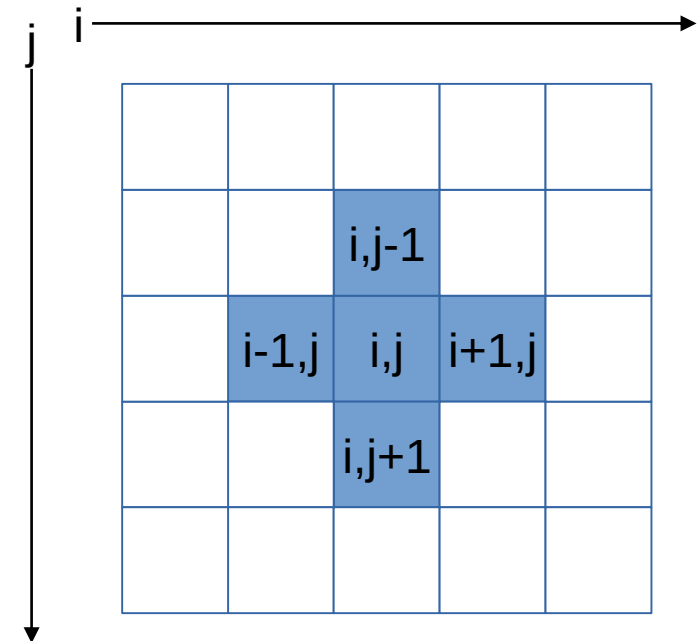
Hydro code

```
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



Compiling, running and analyzing kernel0 (icc -O3 -xHost)

```
> make KERNEL=0
> ./hydro 250 100 # 1st param: grid_size and 2nd param: repet nb
Cycles per element for solvers: 1582.3
> maqao oneview --create-report=one xp=ov_k0 c=ov_cfg.lua
--format=html
> firefox ov_k0/RESULTS/hydro_one_html/index.html &
```

ONEVIEW: report one

file:///home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/ov_k0/RESULTS/hydr

MAQAO Global Application Functions **Loops** Help

Loops Index

Double click on a loop to display its analysis details.

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 131	kernel.c:104-106	binary:kernel.c:104-106	project	25.95
Loop 84	kernel.c:104-106	binary:kernel.c:104-106	c_velocitySolver	22.9
Loop 52	kernel.c:104-106	binary:kernel.c:104-106	c_densitySolver	22.14
Loop 91	kernel.c:104-106	binary:kernel.c:104-106	c_velocitySolver	20.99
Loop 133	kernel.c:368-369	binary:kernel.c:368-369	project	2.29
Loop 43	kernel.c:15-284	binary:kernel.c:15-284	c_densitySolver	1.15
Loop 72	kernel.c:15-284	binary:kernel.c:15-284	c_velocitySolver	0.76
Loop 70	kernel.c:15-284	binary:kernel.c:15-284	c_velocitySolver	0.76
Loop 133	kernel.c:372-375	binary:kernel.c:372-375	project	0.76
Loop 118	kernel.c:44-46	binary:kernel.c:44-46	c_velocitySolver	0.38
Loop 106	kernel.c:202-310	binary:kernel.c:202-310	c_velocitySolver	0.38
Loop 57	kernel.c:28-32	binary:kernel.c:28-32	c_densitySolver	0.38
Loop 120	kernel.c:15-334	binary:kernel.c:15-334	c_velocitySolver	0.38
Loop 61	kernel.c:44-46	binary:kernel.c:44-46	c_densitySolver	0.38
Loop 105	kernel.c:231-233	binary:kernel.c:231-233	c_velocitySolver	0.38

ONEVIEW: report one

ONEVIEW: report one - Loop 1...

file:///home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/ov_k0/RESULTS/hydr

MAQAO Global Application Functions **Loops** Help

Loop 131

Coverage 25.95 %
Function project
Source file and lines kernel.c:104-106
Module binary

Source Code

```
/home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/ /kernel.c: 104 - 106  
  
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)] + x[build_index(i+1, j, grid_size)] +  
107:         )  
108:     }  
109:     setBoundary(b, x, grid_size);
```

Assembly Code

Static Reports

QQA Report

The loop is defined in /home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/kernel.c:104-106

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

Path 1

3% of peak computational performance is used (1.00 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

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 5.00 to 4.00 cycles (1.25x 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. 8 data elements could be processed at once in vector registers. By vectorizing your loop, you can lower the cost of an iteration from 5.00 to 0.62 cycles (8.00x speedup).

Workaround

- Try another compiler or update/tune your current one:

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

Path 1

3% of peak computational performance is used (1.00 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

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

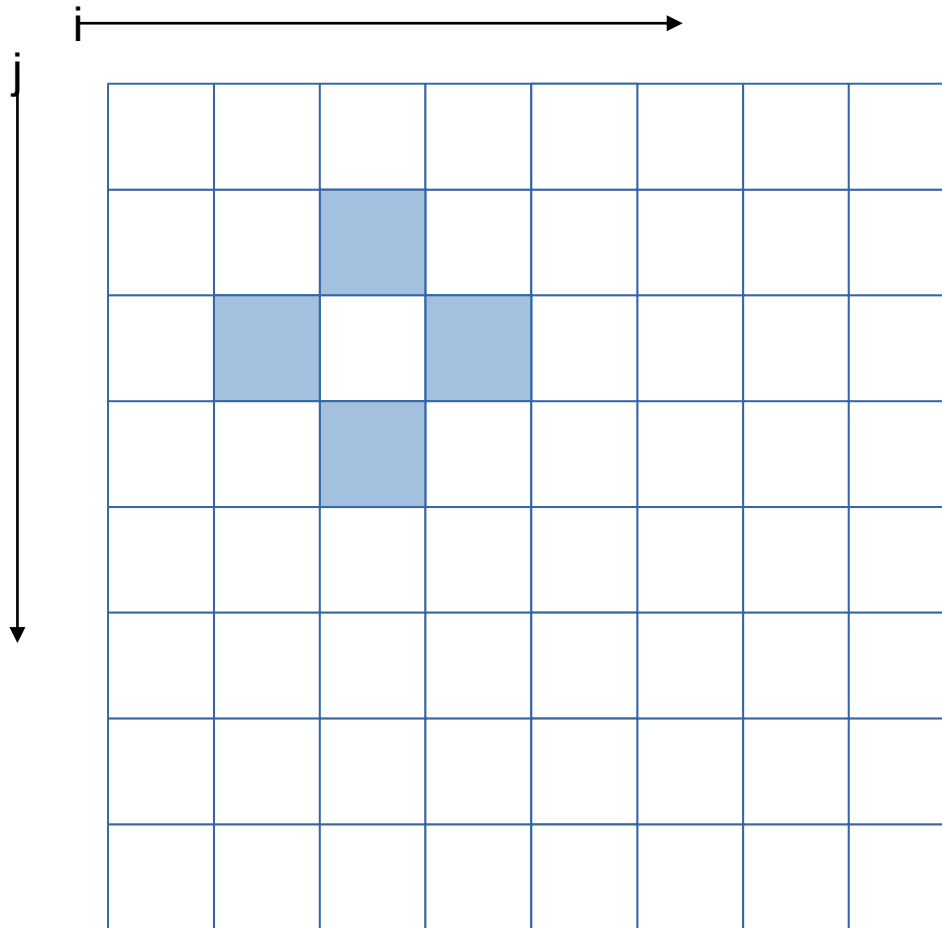
Loop is potentially data access bound.

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).

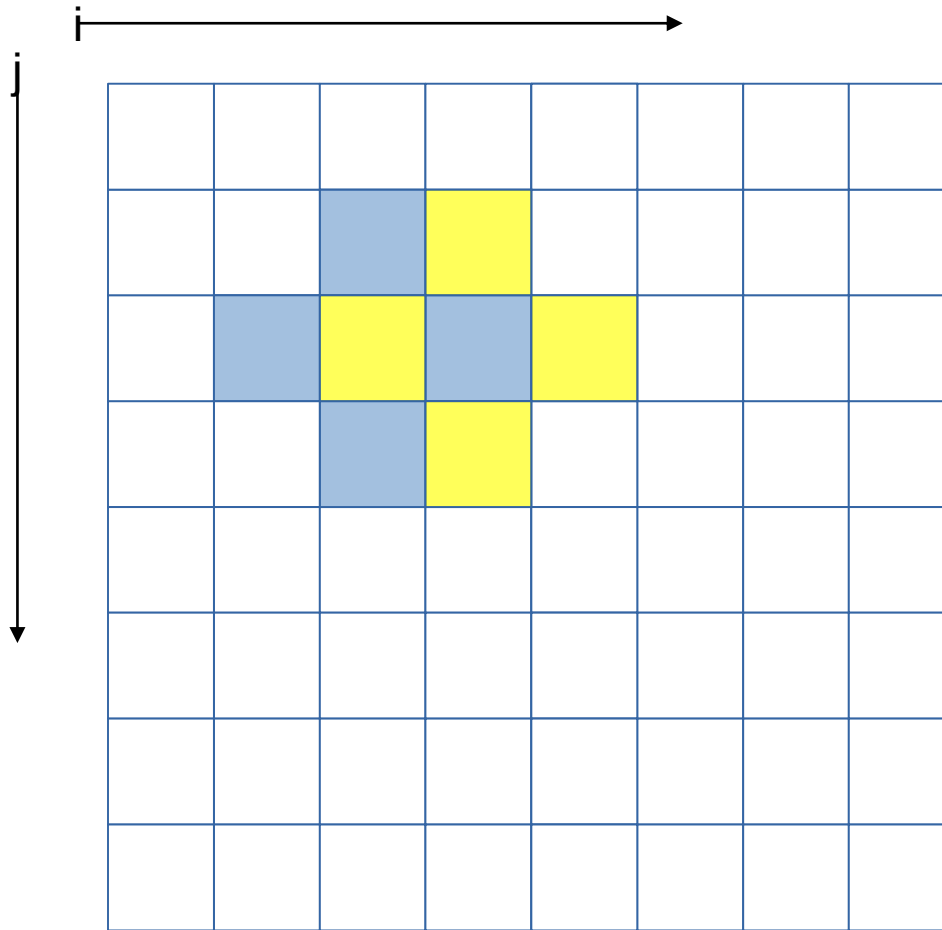
Unrolling is generally a good deal: fast to apply and often provides gain. Let's try to reuse data references through unrolling

Memory references reuse : 4x4 unroll footprint on loads



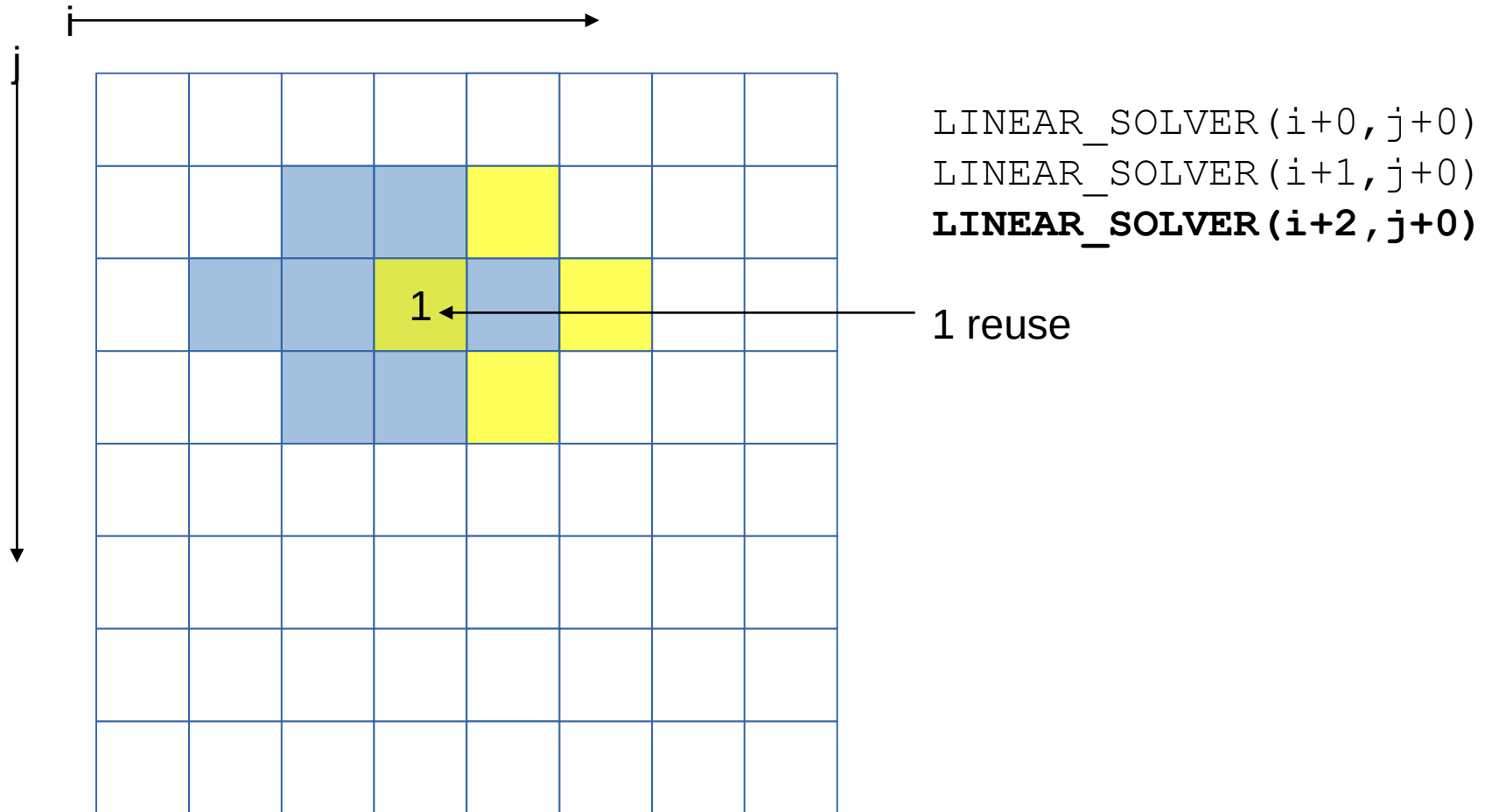
`LINEAR_SOLVER(i+0, j+0)`

Memory references reuse : 4x4 unroll footprint on loads

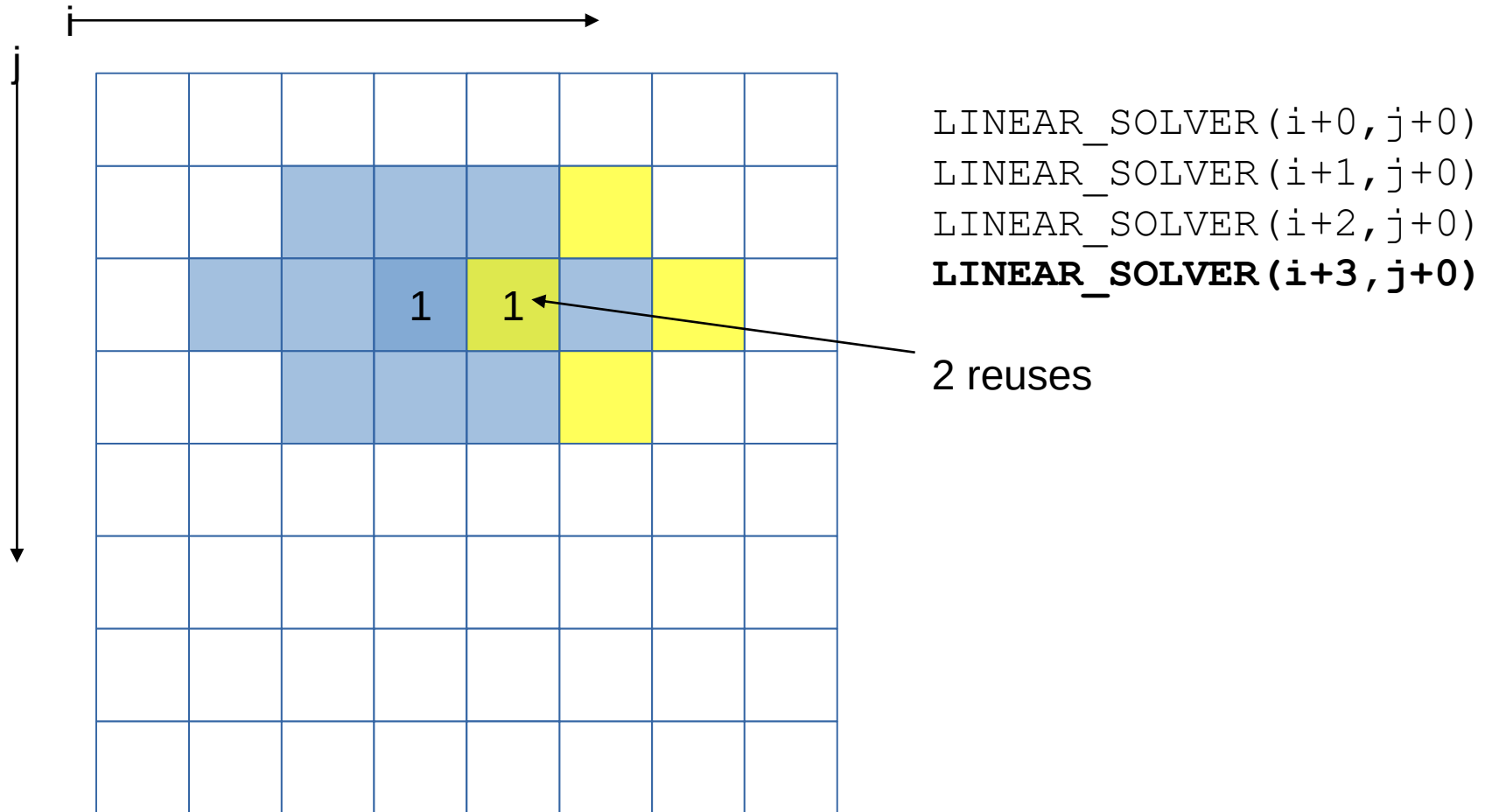


`LINEAR_SOLVER(i+0,j+0)`
`LINEAR_SOLVER(i+1,j+0)`

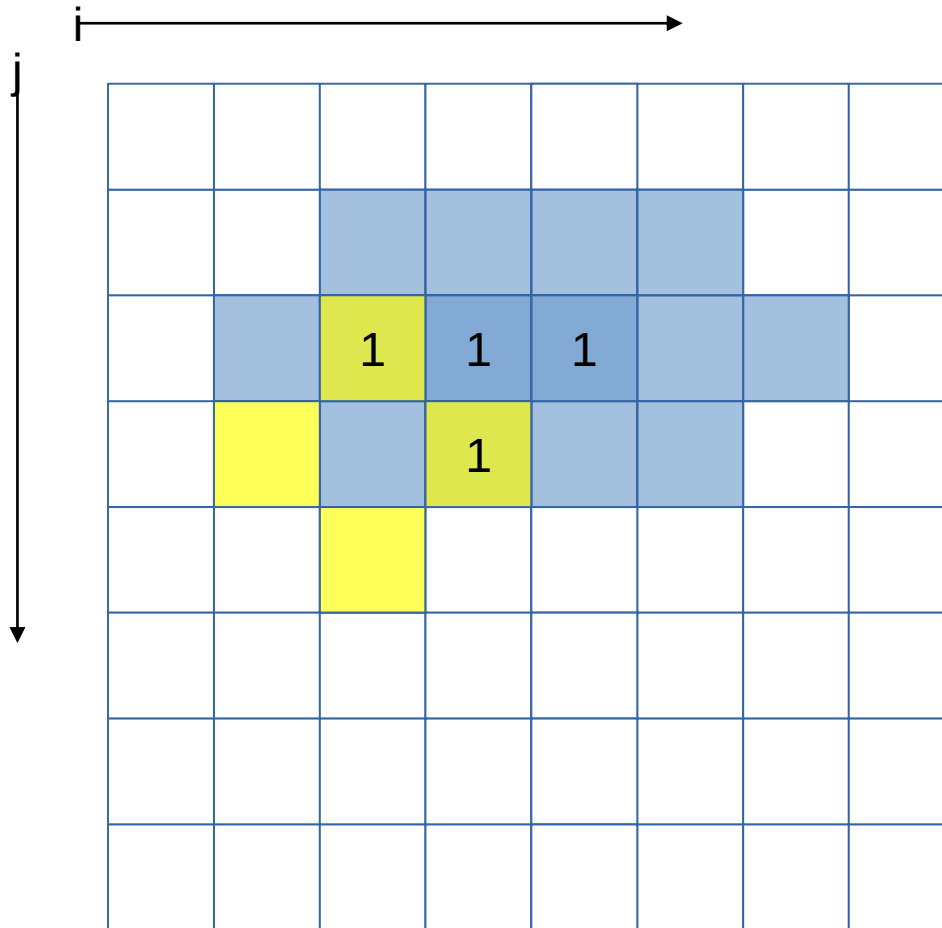
Memory references reuse : 4x4 unroll footprint on loads



Memory references reuse : 4x4 unroll footprint on loads



Memory references reuse : 4x4 unroll footprint on loads

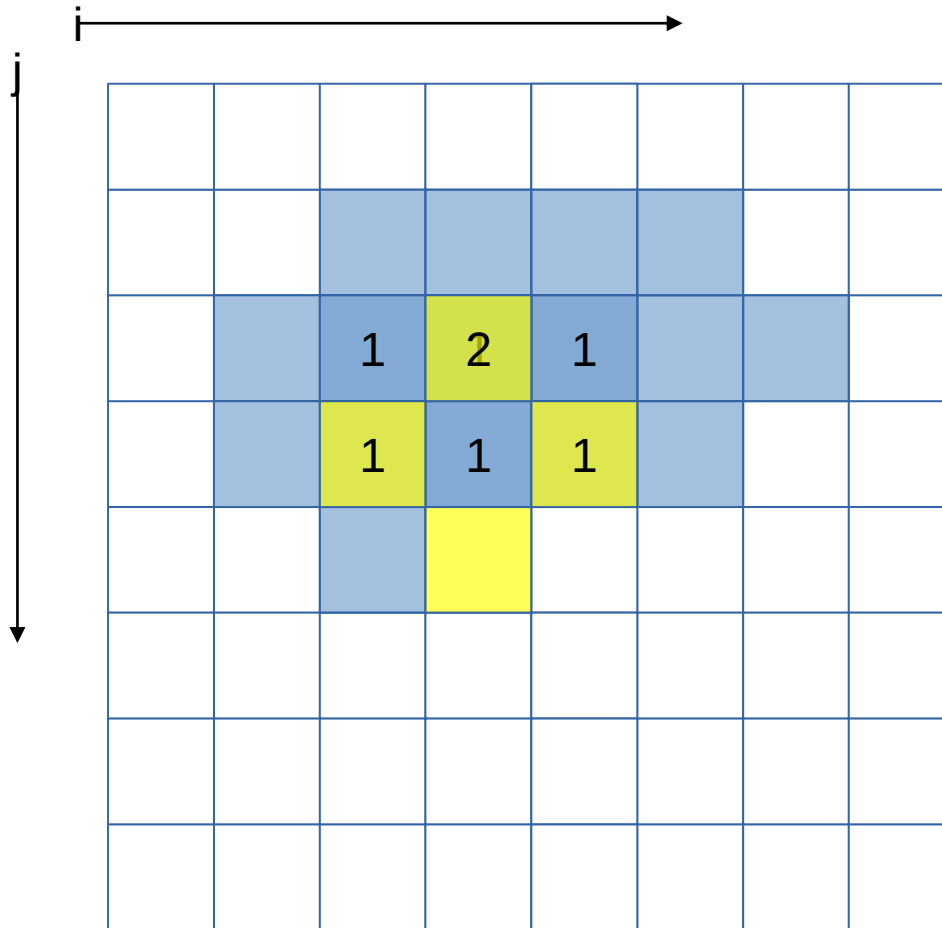


```
LINEAR_SOLVER(i+0,j+0)  
LINEAR_SOLVER(i+1,j+0)  
LINEAR_SOLVER(i+2,j+0)  
LINEAR_SOLVER(i+3,j+0)
```

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

4 reuses

Memory references reuse : 4x4 unroll footprint on loads

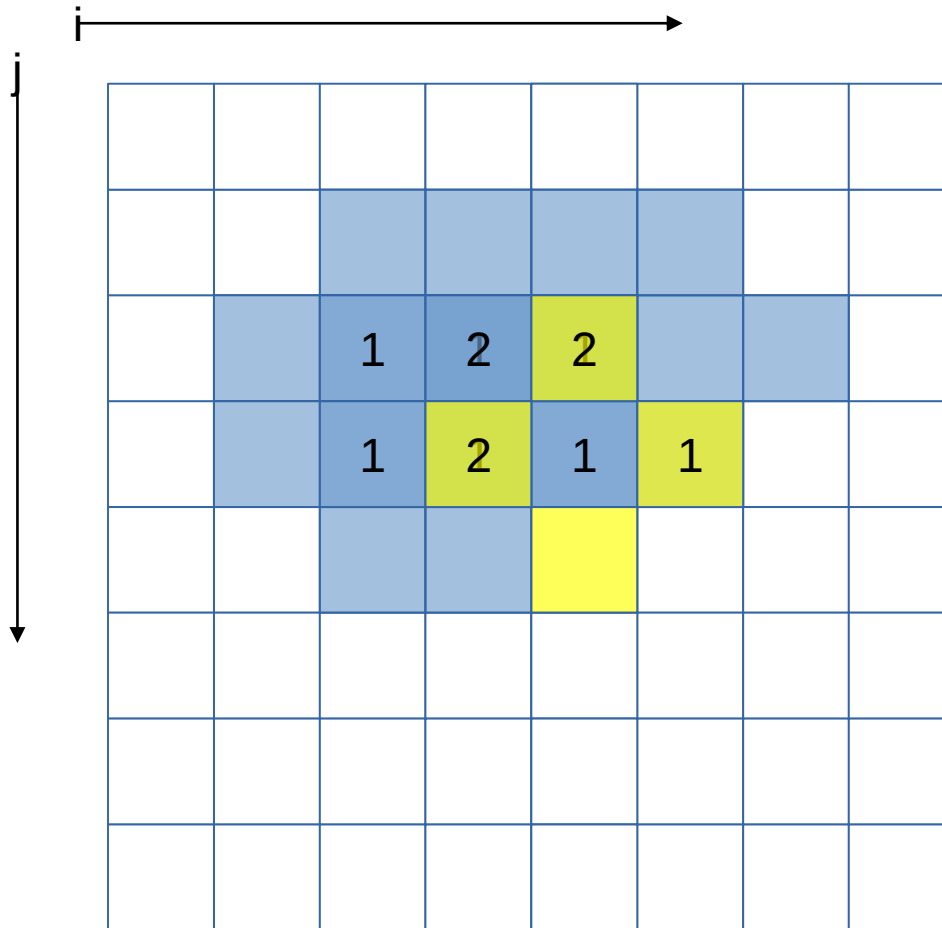


```
LINEAR_SOLVER(i+0,j+0)  
LINEAR_SOLVER(i+1,j+0)  
LINEAR_SOLVER(i+2,j+0)  
LINEAR_SOLVER(i+3,j+0)
```

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

7 reuses

Memory references reuse : 4x4 unroll footprint on loads

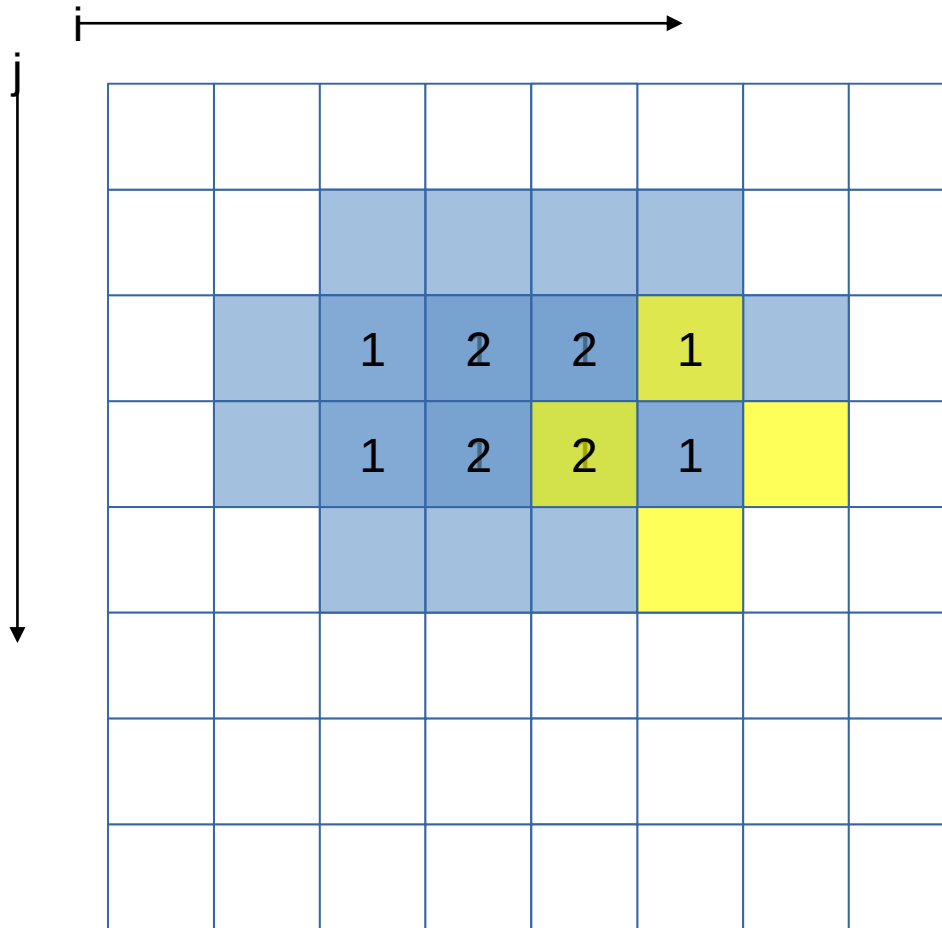


LINEAR_SOLVER($i+0, j+0$)
LINEAR_SOLVER($i+1, j+0$)
LINEAR_SOLVER($i+2, j+0$)
LINEAR_SOLVER($i+3, j+0$)

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

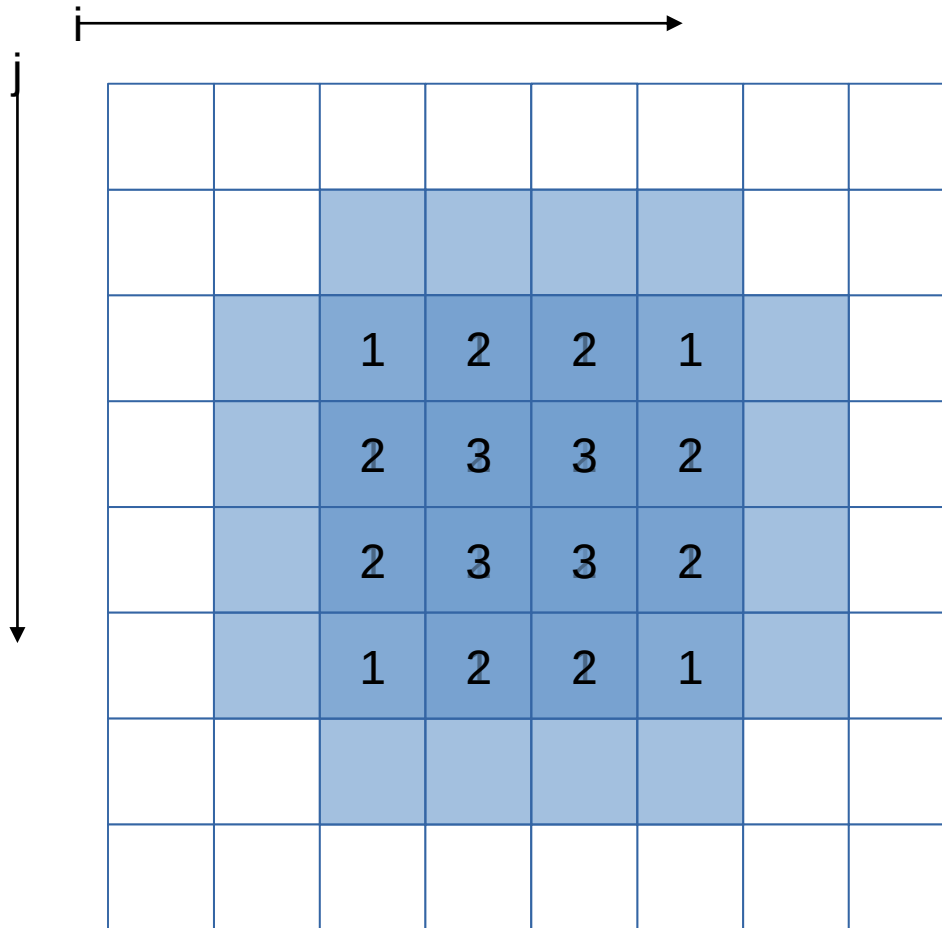


```
LINEAR_SOLVER(i+0,j+0)  
LINEAR_SOLVER(i+1,j+0)  
LINEAR_SOLVER(i+2,j+0)  
LINEAR_SOLVER(i+3,j+0)
```

```
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

Kernel1

```
> make clean
> make KERNEL=1
> ./hydro 250 100
Cycles per element for solvers: 571.65
> maqao oneview --create-report=one xp=ov_k1 c=ov_cfg.lua
  --format=html
> firefox ov_k1/RESULTS/hydro_one_html/index.html
```


ONEVIEW: report one

file:///home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/ov_k1/RESULTS/hydr

MAQAO Global Application Functions **Loops** Help

Loops Index

Double click on a loop to display its analysis details.

Loop id	Source Lines	Source File	Source Function	Coverage (%)
Loop 137	kernel.c:15-168	binary:kernel.c:15-168	linearSolver1	59.52
Loop 53	kernel.c:15-168	binary:kernel.c:15-168	c_densitySolver	17.80
Loop 123	kernel.c:15-334	binary:kernel.c:15-334	c_velocitySolver	4.76
Loop 87	kernel.c:372-375	binary:kernel.c:372-375	c_velocitySolver	3.57
Loop 89	kernel.c:360-363	binary:kernel.c:360-363	c_velocitySolver	3.57
Loop 73	kernel.c:15-284	binary:kernel.c:15-284	c_velocitySolver	2.38
Loop 80	kernel.c:28-32	binary:kernel.c:28-32	c_velocitySolver	1.19
Loop 117	kernel.c:44-46	binary:kernel.c:44-46	c_velocitySolver	1.19
Loop 109	kernel.c:202-310	binary:kernel.c:202-310	c_velocitySolver	1.19
Loop 44	kernel.c:15-284	binary:kernel.c:15-284	c_densitySolver	1.19
Loop 75	kernel.c:360-363	binary:kernel.c:360-363	c_velocitySolver	1.19
Loop 71	kernel.c:15-284	binary:kernel.c:15-284	c_velocitySolver	1.19
Loop 62	kernel.c:44-46	binary:kernel.c:44-46	c_densitySolver	1.19

ONEVIEW: report one

ONEVIEW: report one - Loop 1...

file:///home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/ov_k1/RESULTS/hydr

MAQAO Global Application Functions **Loops** Help

```

148:   for (j = 1; j <= grid_size-3; j+=4)
149:   {
150:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+0);
151:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+1);
152:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+2);
153:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+0, j+3);
154:
155:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+0);
156:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+1);
157:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+2);
158:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+1, j+3);
159:
160:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+0);
161:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+1);
162:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+2);
163:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+2, j+3);
164:
165:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+0);
166:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+1);
167:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+2);
168:     LINEARSOLVER (x, x0, a, inv_c, grid_size, i+3, j+3);
169:   }
170: }
171: setBoundry(b, x, grid_size);

```

Assembly Code

Static Reports

CQA Report

The loop is defined in /home/hpc/a2c06/lu23bim/MAQAO_HANDSON/cqa/hydro/kernel.c:15-168

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

Path 1

6% of peak computational performance is used (2.00 out of 32.00 FLOP per cycle (GFLOPS @ 1GHz))

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 48.00 to 6.00 cycles (8.00x speedup).

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 dependencies", try the IVDEP directive. If, using IVDEP, "vectorization possible but seems inefficient", try the VECTOR ALWAYS directive.
- Remove inter-iterations dependencies 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[i][j] = b[j][i]; (slow, non stride 1) => for(i) for(j) a[i][j] = b[j][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)

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

CQA output for kernel1

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

The binary loop is composed of 96 FP arithmetical operations:

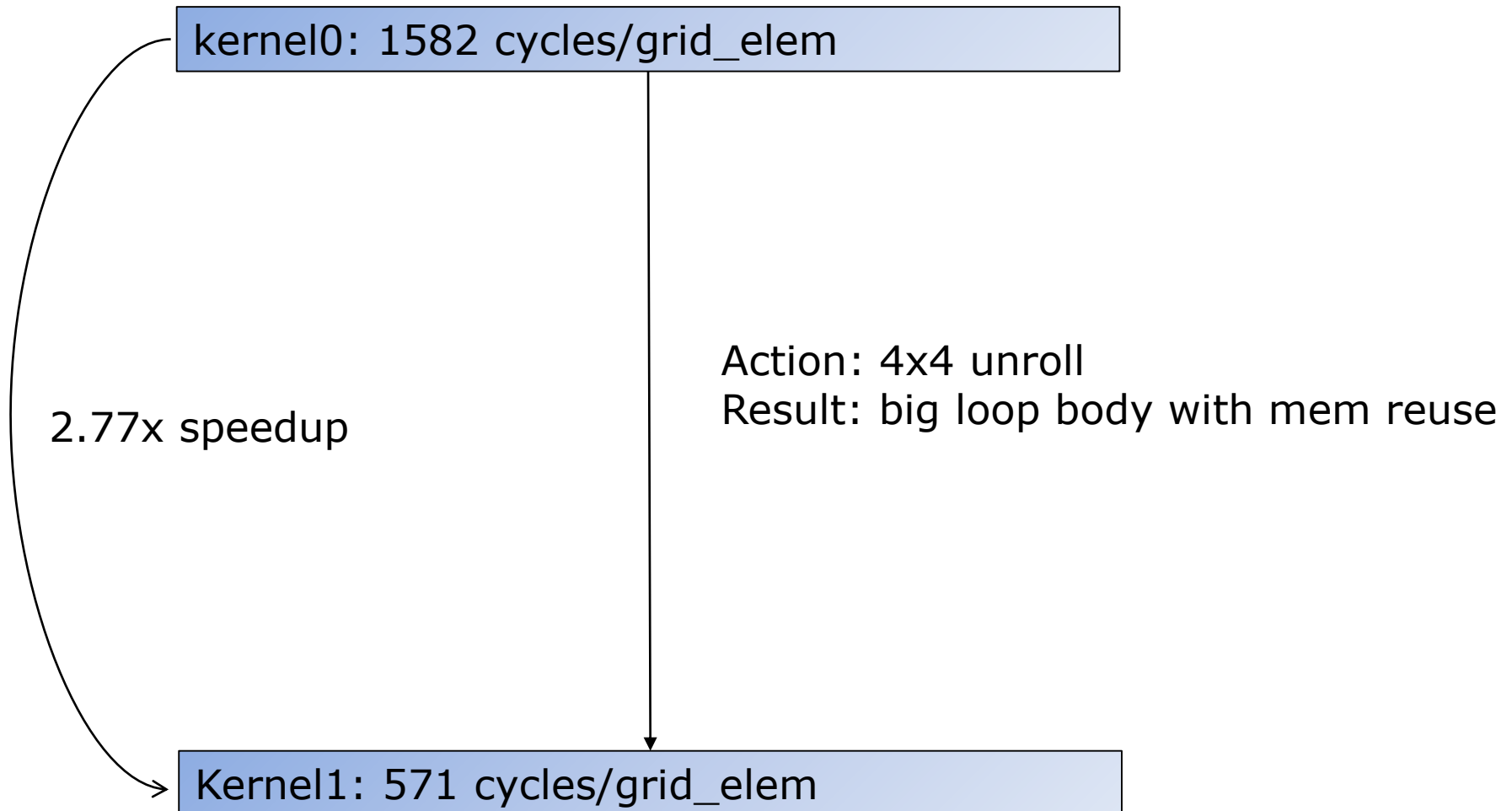
- 64: addition or subtraction
- 32: multiply

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



Kernel1: KNL

```
> salloc --nodes=1 --tasks-per-node=1
> mpiexec -n 1 ./hydro 250 100
Cycles per element for solvers: 1249.20
> maqao onview --create-report=one xp=ov_KNL c=ov_cfg_KNL.lua
  --format=html
> exit
> firefox ov_KNL/RESULTS/hydro_one_html/index.html
```

More sample codes

More codes to study with MAQAO in

`MAQAO_DIR/MAQAO_sample_loops.tgz`