

## MAQAO Hands-on exercises

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

## Setup

---

Login to the cluster (using login8 for access to /dine/data)

```
> ssh <username>@login8.cosma.dur.ac.uk
```

Copy handson material to your workspace directory

```
> export TW49_SHARED=/cosma/apps/do009/shared  
Hint: copy in ~/.bash_profile or ~/.bashrc  
> export WORK=/cosma/apps/do009/$USER  
> cd $WORK  
> tar xvf $TW49_SHARED/MAQAO/MAQAO_HANDSON.tgz
```

If NPB-MZ-MPI not yet installed, CF next slide (or related slides)

Load MAQAO environment

```
> module load maqao/2026.0.0
```

## Setup (bt-mz compilation with GNU compiler and MPI & debug symbols)

---

Go to the NPB directory provided with Scalasca handsons

```
> tar xvf $TW49_SHARED/Scalasca/NPB3.4-MZ-MPI.tar.gz
> cd $WORK/NPB3.4-MZ-MPI
```

Load Intel compiler and environment

```
> module load gnu_comp openmpi
```

Compile and run

```
> make bt-mz CLASS=B
> cd bin
> cp ../jobscript/bluefield1/reference.sbatch .
> sbatch reference.sbatch
```

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

Cédric Valensi

## Setup ONE View for batch mode

---

The ONE View configuration file must contain all variables for executing the application. Retrieve the configuration file prepared for bt-mz in batch mode from the MAQAO\_HANDSON directory

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

```
"executable": "bt-mz_B.x"
...
"scripts": {
  "command": "sbatch <myscript>",
  "files": [
    {"path": "bt_maqao.sbatch", "tag": "myscript"}
  ]
}
...
"number_processes": 4
"number_processes_per_node": 2
"envv_OMP_NUM_THREADS": 6
...
"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.sbatch .
> less bt_maqao.sbatch
```

```
...
#SBATCH --ntasks-per-node=2<number_processes_per_node>
#SBATCH --cpus-per-task=6<OMP_NUM_THREADS>
...
export OMP_NUM_THREADS=6<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 -R1 --config=bt_OV_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>`.

### **WARNINGS:**

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

## Display MAQAO ONE View results

---

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

Mount `$WORK` locally:

```
> mkdir cosma_work
> sshfs <user>@login.cosma.dur.ac.uk:\
/cosma/apps/do009/<user> cosma_work
> firefox cosma_work/NPB3.4-MZ-MPI/bin/ov_sbatch\
/RESULTS/bt-mz_B.x_one_html/index.html
```

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

```
> tar czf $WORK/bt_html.tgz ov_sbatch/RESULTS/bt-mz_B.x_one_html
> scp <user>@login.cosma.dur.ac.uk:bt_html.tgz .
> tar xf bt_html.tgz
> firefox ov_sbatch/RESULTS/bt-mz_B.x_one_html/index.html
```

## sshfs & scp hints

---

- To install sshfs on Debian-based Linux distributions (like Ubuntu)

```
> sudo apt install sshfs
```

- Recommended to close a sshfs directory after use

```
> fusermount -u /path/to/sshfs/directory
```

- scp is slow to copy directories (especially when containing many small files), copy a .tgz archive of the directory

## Display MAQAO ONE View results (optional)

---

A sample result directory is in **MAQAO\_HANDSON/bt/offline.tgz**

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

```
> maqao oneview -R1 -xp=ov_sbatch --output-format=text
```

## Scalability profiling of bt-mz with MAQAO

Cédric Valensi

## Setup ONE View for scalability analysis

---

Retrieve the configuration file prepared for bt-mz 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/bt_OV_scal.json .
> less bt_OV_scal.json
```

```
"executable": "./bt-mz_B.x"
...
"run_command": "<executable>"
...
"scripts": {
  "command": "sbatch <myscript>",
  "files": [
    {"path": "bt_maqao.sbatch", "tag": "myscript"}
  ]
}
...
"number_processes": 1
...
"number_processes_per_node": 1
...
"envv_OMP_NUM_THREADS": 6
...
"mpi_command": "mpirun -n <number_processes>"
...
"multiruns_params": [
  {"name": "2x6", "number_processes": 2, "number_nodes": 1, "number_processes_per_node": 2},
  {"name": "4x6", "number_processes": 4, "number_nodes": 2, "number_processes_per_node": 2},
]
"base_run_name": "1x6"
```

## Launch MAQAO ONE View on bt-mz (scalability mode)

---

Launch ONE View (execution will be longer!)

```
> maqao oneview -R1 --with-scalability=strong \  
-c=bt_0V_scal.json -xp=ov_scal
```

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

```
> firefox cosma_work/NPB3.4-MZ-MPI/bin/ov_scal/RESULTS/  
bt-mz_B.x_one_html/index.html
```

**OR**

```
> tar czf $WORK/bt_scal.tgz \  
ov_scal/RESULTS/bt-mz_B.x_one_html
```

```
> scp <user>@login.cosma.dur.ac.uk:ov_scal.tgz .  
> tar xf ov_scal.tgz  
> firefox ov_scal/RESULTS/bt-mz_B.x_one_html/index.html
```

A sample result directory is in **MAQAO\_HANDSON/bt/offline.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

## Compile with GNU compiler

---

Go to the handson directory

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

Compile all variants

```
> module load gnu_comp/14.1.0  
> make all
```

Load MAQAO environment (if necessary)

```
> module load maqao/2026.0.0
```

## Analysing matrix multiply with MAQAO

---

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

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> srun -A do009 -p bluefield1 --exclusive -t 1 \
./matmul_orig/matmul 400 300
cycles per FMA: 3.13
```

Analyse matrix multiply with ONE View

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

## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/matmul/ov_orig/RESULTS/\
matmul_orig_one_html/index.html &
```

Global Metrics		?
Total Time (s)		17.65
Profiled Time (s)		17.64
Time in analyzed loops (%)		100
Time in analyzed innermost loops (%)		99.9
Time in user code (%)		100
Compilation Options Score (%)		50.0
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	2.51
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	8.00
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.00
	Nb Loops to get 80%	1

# 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

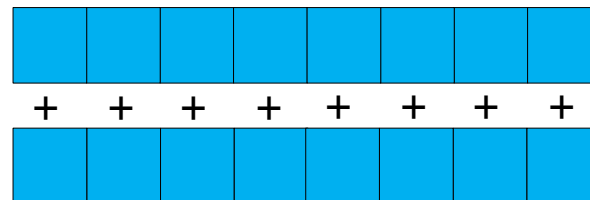
- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or ffast-math) to extend loop vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check 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):

VADDSS  
(scalar)



VADDPS  
(packed)



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

# Impact of loop permutation on data access

Logical mapping

$j=0,1\dots$

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

Efficient vectorization + prefetching

Physical mapping

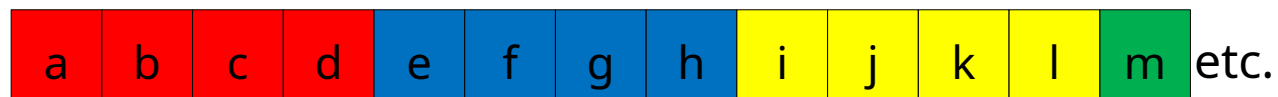
(C stor. order: row-major)



```
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 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

---

Run permuted loops version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> srun -A do009 -p bluefield1 --exclusive -t 1 \
./matmul_perm/matmul 400 300
cycles per FMA: 0.46
```

Analyse matrix multiply with ONE View

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

## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/matmul/ov_perm/RESULTS/\
matmul_perm_one_html/index.html &
```

Global Metrics		?
Total Time (s)		2.61
Profiled Time (s)		2.60
Time in analyzed loops (%)		99.2
Time in analyzed innermost loops (%)		89.4
Time in user code (%)		99.2
Compilation Options Score (%)		50.0
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	1.07
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.23
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	2.12
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.25
	Nb Loops to get 80%	2

Faster (was 17.57)

More efficient vectorization  
(was 8.00)

## CQA output after loop permutation

gain potential hint expert

### Vectorization

Your loop is vectorized, but using only 128 out of 256 bits (SSE/AVX-128 instructions on AVX/AVX2 processors). By fully vectorizing your loop, you can lower the cost of an iteration from 1.17 to 0.58 cycles (2.00x 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=znver2`. CQA target is `AMD_fam17h_mod31h` (2nd generation EPYC and 3rd generation EPYC based on the Zen 2 microarchitecture) but specialization flags are `-march=x86-64`
- 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.

Let's try this

# Viewing results (HTML)

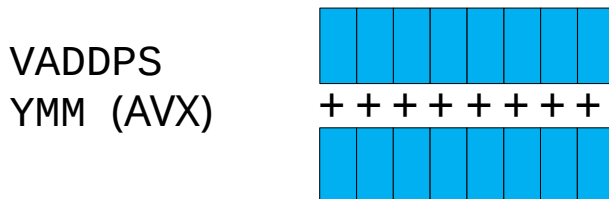
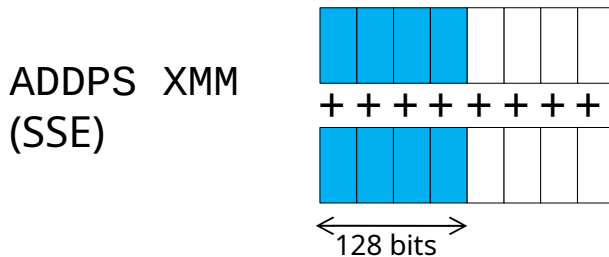
Global Metrics <span>?</span>		Compilation Options <span>↶</span>							
Total Time (s)	1.65	<table border="1"> <thead> <tr> <th>Source Object</th> <th>Issue</th> </tr> </thead> <tbody> <tr> <td>▼ matmul_align</td> <td></td> </tr> <tr> <td>○ kernel_align.c</td> <td>-funroll-loops is missing.</td> </tr> </tbody> </table>		Source Object	Issue	▼ matmul_align		○ kernel_align.c	-funroll-loops is missing.
Source Object	Issue								
▼ matmul_align									
○ kernel_align.c	-funroll-loops is missing.								
Profiled Time (s)	1.64								
Time in analyzed loops (%)	100.0								
Time in analyzed innermost loops (%)	98.8								
Time in user code (%)	100								
Compilation Options Score (%)	75.0								
Perfect Flow Complexity	1.00								
Array Access Efficiency (%)	100								
Perfect OpenMP + MPI + Pthread	1.00								
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00								
	Potential Speedup	1.00							
No Scalar Integer	Nb Loops to get 80%	1							
	Potential Speedup	1.00							
FP Vectorised	Nb Loops to get 80%	1							
	Potential Speedup	1.01							
Fully Vectorised	Nb Loops to get 80%	2							
	Potential Speedup	1.01							
FP Arithmetic Only	Nb Loops to get 80%	2							

Let's try this

## Impacts of architecture specialization: vectorization

---

- Vectorization
  - SSE instructions (SIMD 128 bits) used on a processor supporting AVX256 ones (SIMD 256 bits)
  - => 50% efficiency loss



## Analyse matrix multiply with compiler optimizations (microarchitecture-specialization and loop unrolling)

---

Run array-aligned version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> srun -A do009 -p bluefield1 --exclusive -t 1 \
./matmul_comp_opt/matmul 400 300 # remark: size%8 has to equal 0
cycles per FMA: 0.32
```

Analyse matrix multiply with ONE View

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

## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/matmul/ov_comp_opt/RESULTS/\  
matmul_comp_opt_one_html/index.html &
```

The screenshot displays a web interface with a 'Global Metrics' section and a 'Vectorization' section. The 'Global Metrics' section shows 'Total Time (s)' as 1.65 and 'Profiled Time (s)' as 1.64. The 'Vectorization' section indicates that the loop is fully vectorized. Below this, a 'Details' section provides a table of vectorization options and their performance metrics.

Configuration	Metric	Value
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.01
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.01
	Nb Loops to get 80%	2

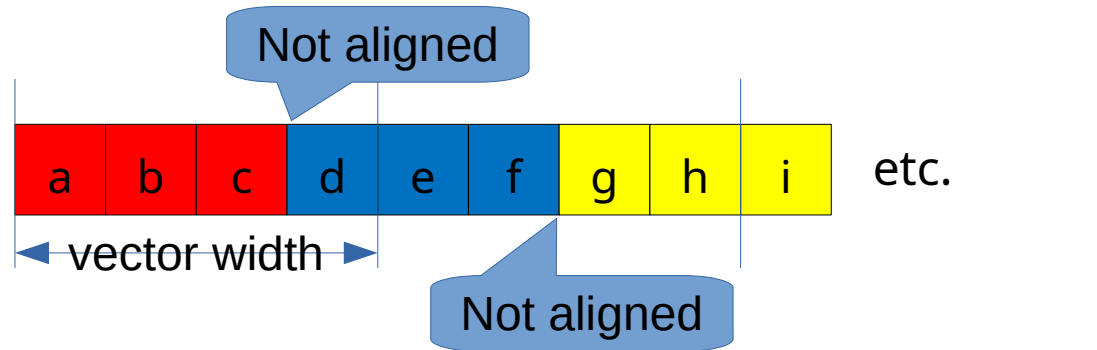
A green callout box points to the '1.01' value for 'Fully Vectorised' Potential Speedup, stating: 'Now optimal vectorization (was 2.14)'.

## Multidimensional array alignment

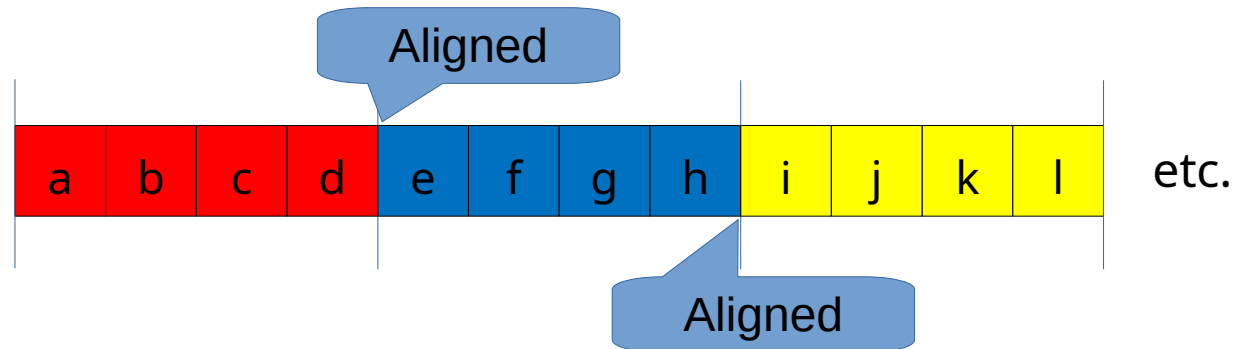
Data organized as a 2D array:  $n$  lines of 3 columns  
Each vector can hold 4 consecutive elements

a[0]: line 0    a[1]: line 1    a[2]: line 2

$a[n][3]$ , only 1st  
element is aligned



$a[n][4]$ , 1st element of  
each line are aligned



## Analyse matrix multiply with array alignment

---

Run unrolled version of matrix multiply

```
> cd $WORK/MAQAO_HANDSON/matmul #if cur. directory has changed
> srun -A do009 -p bluefield1 --exclusive -t 1 \
./matmul_align/matmul 400 300
driver.c: Using posix_memalign instead of malloc
cycles per FMA: 0.25
```

Analyse matrix multiply with ONE View

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

## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/matmul/ov_align/RESULTS/\  
matmul_align_one_html/index.html &
```

Global Metrics		?
Total Time (s)		1.36
Profiled Time (s)		1.35
Time in analyzed loops (%)		99.3
Time in analyzed innermost loops (%)		89.6
Time in user code (%)		99.3
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	1.02
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.03
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	2.02
	Nb Loops to get 80%	2

Small gain (was 1.68)

# Using comparison mode: global level

```
> maqao oneview --compare-reports -xp=ov_matmul_cmp \
-inputs=ov_orig,ov_perm,ov_comp_opt,ov_align
```

Remark: open `ov_matmul_cmp/RESULTS/ov_matmul_cmp/index.html`

**Compared Reports**

- r0: ov\_orig
- r1: ov\_perm
- r2: ov\_align
- r3: ov\_unroll

Global Metrics						Application Categorization	
Metric	r0	r1	r2	r3	Time		
Total Time (s)	17.63	3.44	1.69	1.38			
Profiled Time (s)	17.62	3.43	1.68	1.37			
Time in analyzed loops (%)	100	99.9	100	99.6			
Time in analyzed innermost loops (%)	99.7	91.0	96.4	91.2			
Time in user code (%)	100	99.8	100	99.6			
Compilation Options Score (%)	50.0	50.0	75.0	100			
Perfect Flow Complexity	1.00	1.00	1.00	1.00			
Array Access Efficiency (%)	83.3	100	100	75.0			
Perfect OpenMP + MPI + Pthread	1.00	1.00	1.00	1.00			
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00	1.00	1.00	1.00			
No Scalar Integer	Potential Speedup	1.00	1.03	1.02			
	Nb Loops to get 80%	1	1	1			
FP Vectorised	Potential Speedup	2.50	1.16	1.02			
	Nb Loops to get 80%	1	1	1			
Fully Vectorised	Potential Speedup	8.00	2.13	1.03			
	Nb Loops to get 80%	1	2	1			
Only FP Arithmetic	Potential Speedup	1.00	1.23	1.03			
	Nb Loops to get 80%	1	2	1			

Report	Binary (s)	Memory (s)
r0	17.63	~0.1
r1	3.44	~0.1
r2	1.69	~0.1
r3	1.38	~0.1

## Using comparison mode: experiment summaries

Experiment Summaries <span style="float: right;">?</span>				
	r0	r1	r2	r3
<b>Application</b>	./matmul_orig	./matmul_perm	./matmul_align	./matmul_unroll
<b>Timestamp</b>	2023-04-17 09:11:34	2023-04-17 09:12:05	2023-04-17 09:12:16	2023-04-17 09:12:27
<b>Experiment Type</b>	MPI;	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Machine</b>	b116.pri.cosma7.alces.network	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Architecture</b>	x86_64	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Micro Architecture</b>	ZEN_V2	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Model Name</b>	AMD EPYC 7302 16-Core Processor	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Cache Size</b>	512 KB	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Number of Cores</b>	16	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>
<b>Maximal Frequency</b>	0 GHz	<i>same as r0</i>	<i>same as r0</i>	<i>same as r0</i>

# Using comparison mode: function & loop level

## Functions

Name	Module	Coverage (%)				Time (s)				Nb Threads			
		ov_orig	ov_perm	ov_align	ov_unroll	ov_orig	ov_perm	ov_align	ov_unroll	ov_orig	ov_perm	ov_align	ov_unroll
kernel	matmul	100	99.56	99.4	100	17.6	3.37	1.66	1.36	1	1	1	1
__GI_memset	libc-2.17.so	NA	0.44	0.6	NA	NA	0.01	0.01	NA	NA	1	1	NA

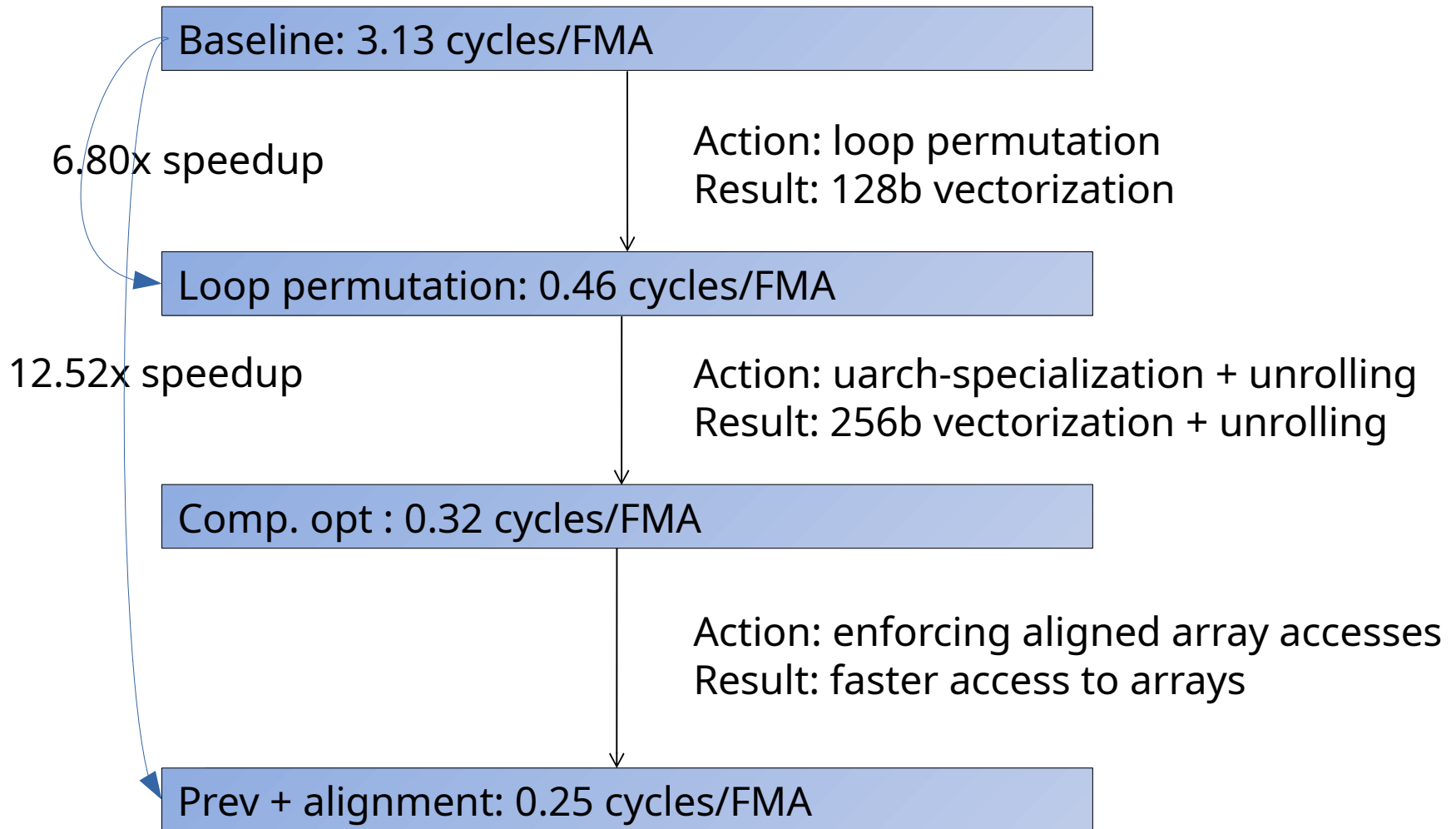
## Loops



### ▼ kernel.c: 24 - 376.29%

Run ov_orig						Run ov_perm					Run ov_align					Run ov_unroll							
Loop Source Regions	<ul style="list-style-type: none"> <li>• /cosma5/data/do008/dc-vale1/2023_Modif/MAQAO_HANDSON/matmul/matmul_orig/kernel.c: 24-25</li> </ul>					Loop Source Regions	<ul style="list-style-type: none"> <li>• /cosma5/data/do008/dc-vale1/2023_Modif/MAQAO_HANDSON/matmul/matmul_perm/kernel.c: 24-25</li> </ul>				Loop Source Regions	<ul style="list-style-type: none"> <li>• /cosma5/data/do008/dc-vale1/2023_Modif/MAQAO_HANDSON/matmul/matmul_align/kernel.c: 24-25</li> </ul>				Loop Source Regions	<ul style="list-style-type: none"> <li>• /cosma5/data/do008/dc-vale1/2023_Modif/MAQAO_HANDSON/matmul/matmul_align/kernel.c: 24-25</li> </ul>						
ASM Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)	Assembly Loop ID	Max Time Over Threads (s)	Time w.r.t. Wall Time (s)	Cov (%)	Vect. Ratio (%)	Vector Length Use (%)
1	17.58	17.58	99.91	0	12.5	4	3.01	3.01	88.79	100	50	4	1.61	1.61	96.41	100	100	4	1.24	1.24	91.18	100	100

## Summary of optimizations and gains



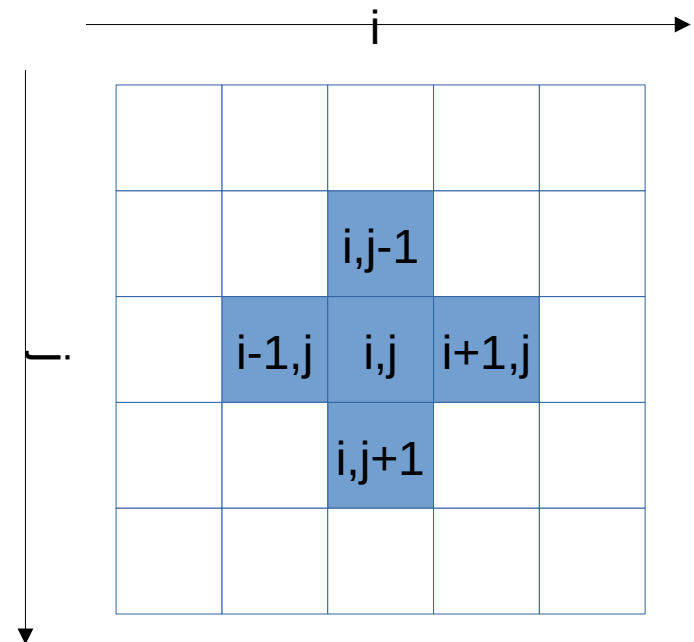
## Hydro code

```
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



## Compile and run with AMD compiler

---

Switch to the hydro handson folder

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

Load MAQAO 2026.0.0b (if no more loaded)

```
> module load maqao/2026.0.0
```

Load latest AMD Compiler (aocc 4.0)

```
> module load llvm/21.1.8
```

Compile

```
> make
```

## Running and analyzing original kernel

---

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

```
> cd $WORK/MAQAO_HANDSON/hydro #if cur. directory has changed
> less ov_orig.json
```

```
"executable": "./hydro_orig"
"run_command": "<executable> 300 200" -- <size of matrix>
<number of repetitions>
...
"number_processes_per_node": 1
"mpi_command": "srun -A do009 -p bluefield1 --exclusive -t 1"
...
```

## Running and analyzing original kernel

---

Run

```
> srun -A do009 -p bluefield1 --exclusive -t 1 \  
./hydro_orig 300 200 # 300x300 mesh, 200 repetitions  
Cycles per element for solvers: 2817.87
```

Profile with MAQAO

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

## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/hydro/ov_orig/RESULTS/\
hydro_orig_one_html/index.html &
```

Global Metrics <span>?</span>	
Total Time (s)	16.56
Profiled Time (s)	16.56
Time in analyzed loops (%)	100.0
Time in analyzed innermost loops (%)	99.9
Time in user code (%)	100.0
Compilation Options Score (%)	100
Array Access Efficiency (%)	42.3
Potential Speedups	
Perfect Flow Complexity	1.01
Perfect OpenMP + MPI + Pthread	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00
No Scalar Integer	Potential Speedup
	Nb Loops to get 80%
	1
FP Vectorised	Potential Speedup
	Nb Loops to get 80%
	1
Fully Vectorised	Potential Speedup
	Nb Loops to get 80%
	2
FP Arithmetic Only	Potential Speedup
	Nb Loops to get 80%
	3

## CQA output for original kernel

### Workaround

- Try another compiler or update/tune your current one:
  - recompile with fassociative-math (included in Ofast or fast-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)

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

## Running and analyzing kernel with loop permutation

---

Run

```
> srun -A do009 -p bluefield1 --exclusive -t 1 \  
./hydro_perm 300 200 # 300x300 mesh, 200 repetitions  
Cycles per element for solvers: 2992.95
```

Remark: small performance regression but makes room for further optimizations

Profile with MAQAO

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

## Viewing results (HTML)

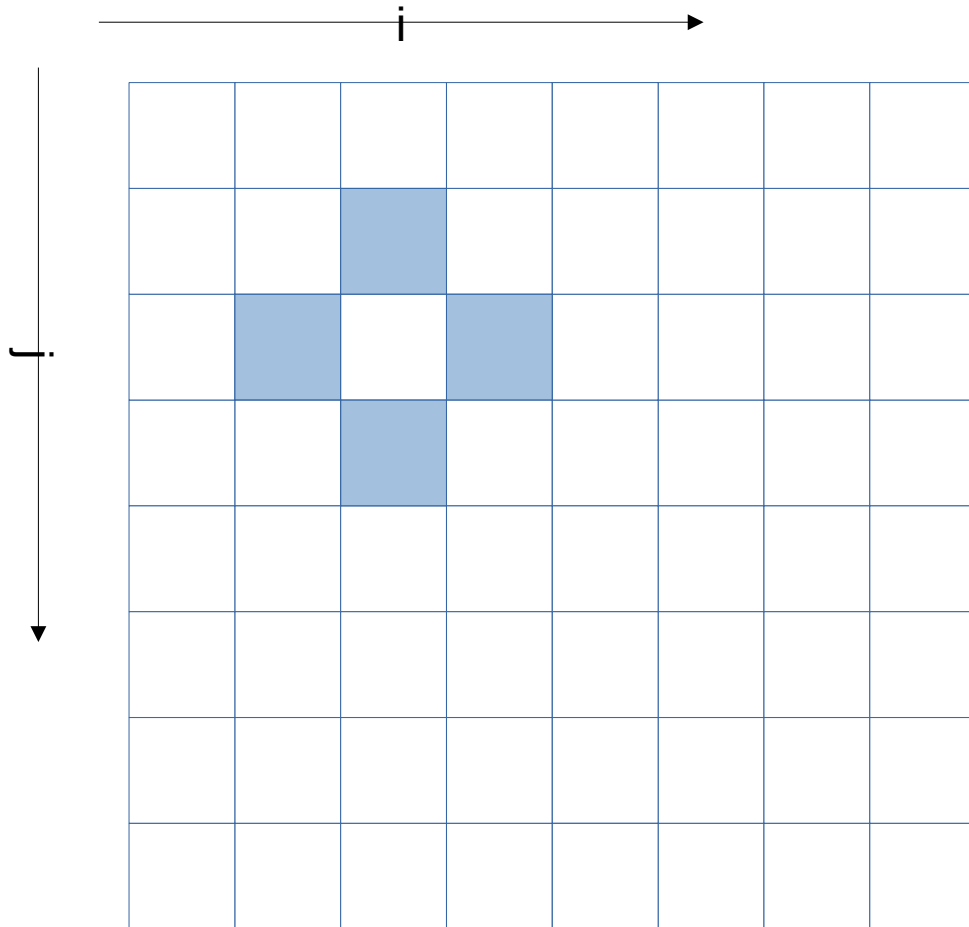
On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/hydro/ov_perm/RESULTS/\
hydro_perm_one_html/index.html &
```

Global Metrics		?
Total Time (s)		18.12
Profiled Time (s)		18.11
Time in analyzed loops (%)		100.0
Time in analyzed innermost loops (%)		100.0
Time in user code (%)		100
Compilation Options Score (%)		100
Array Access Efficiency (%)		88.9
Potential Speedups		
Perfect Flow Complexity		1.01
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	1.96
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.74
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.01
	Nb Loops to get 80%	4

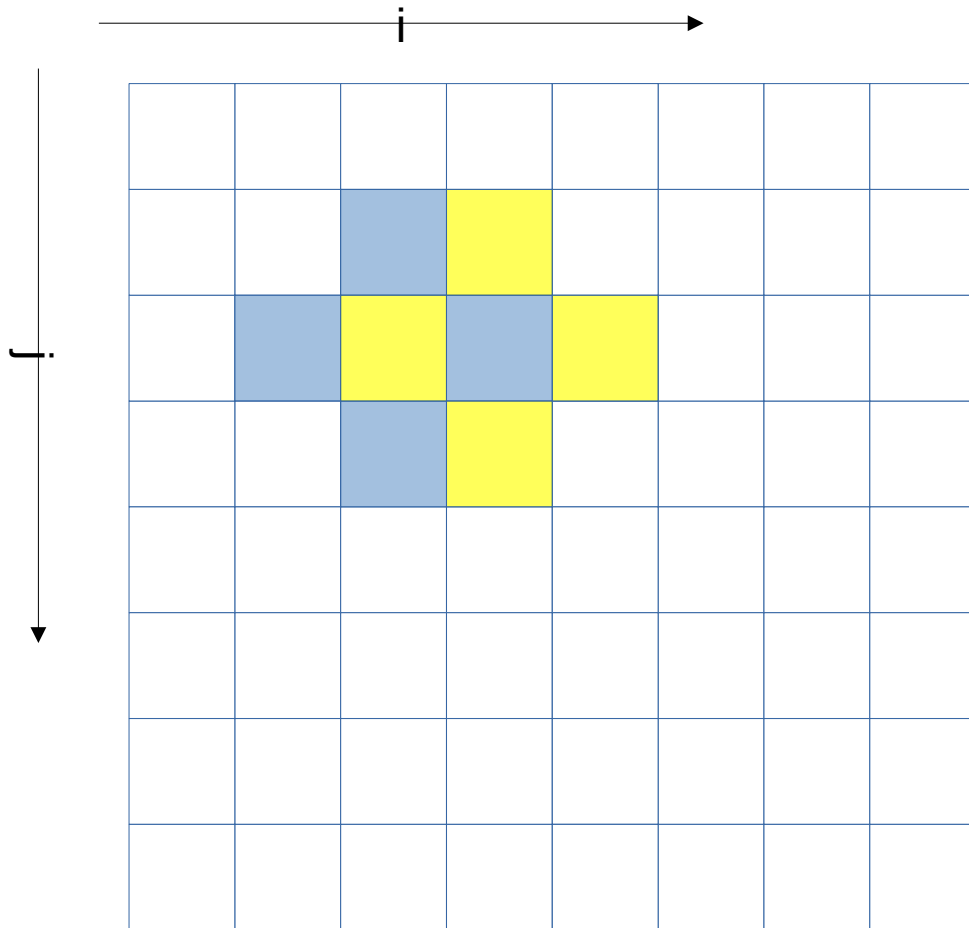
Higher (better),  
formerly 42.3

## Memory references reuse : 4x4 unroll footprint on loads



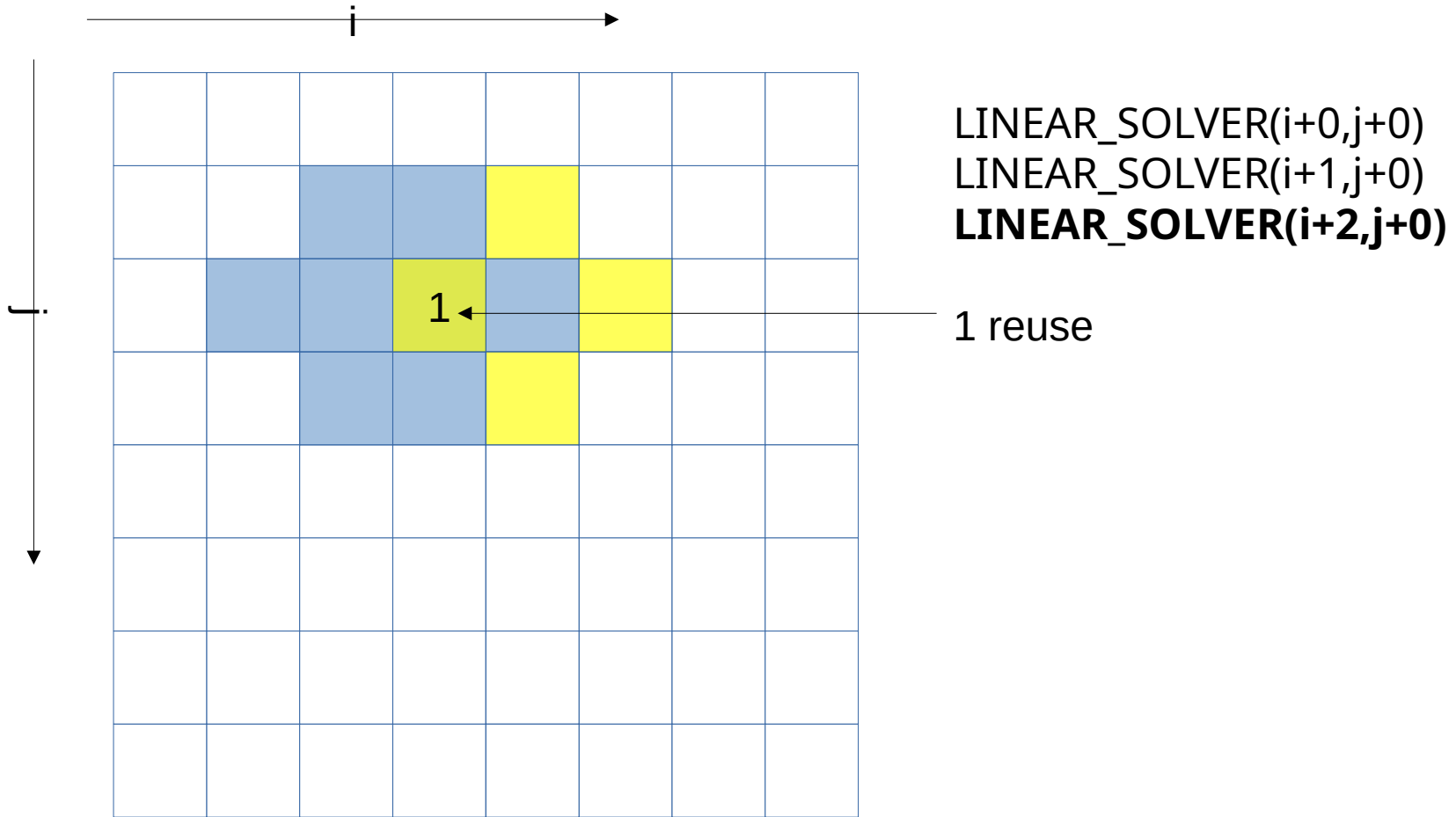
**LINEAR\_SOLVER(i+0,j+0)**

## Memory references reuse : 4x4 unroll footprint on loads

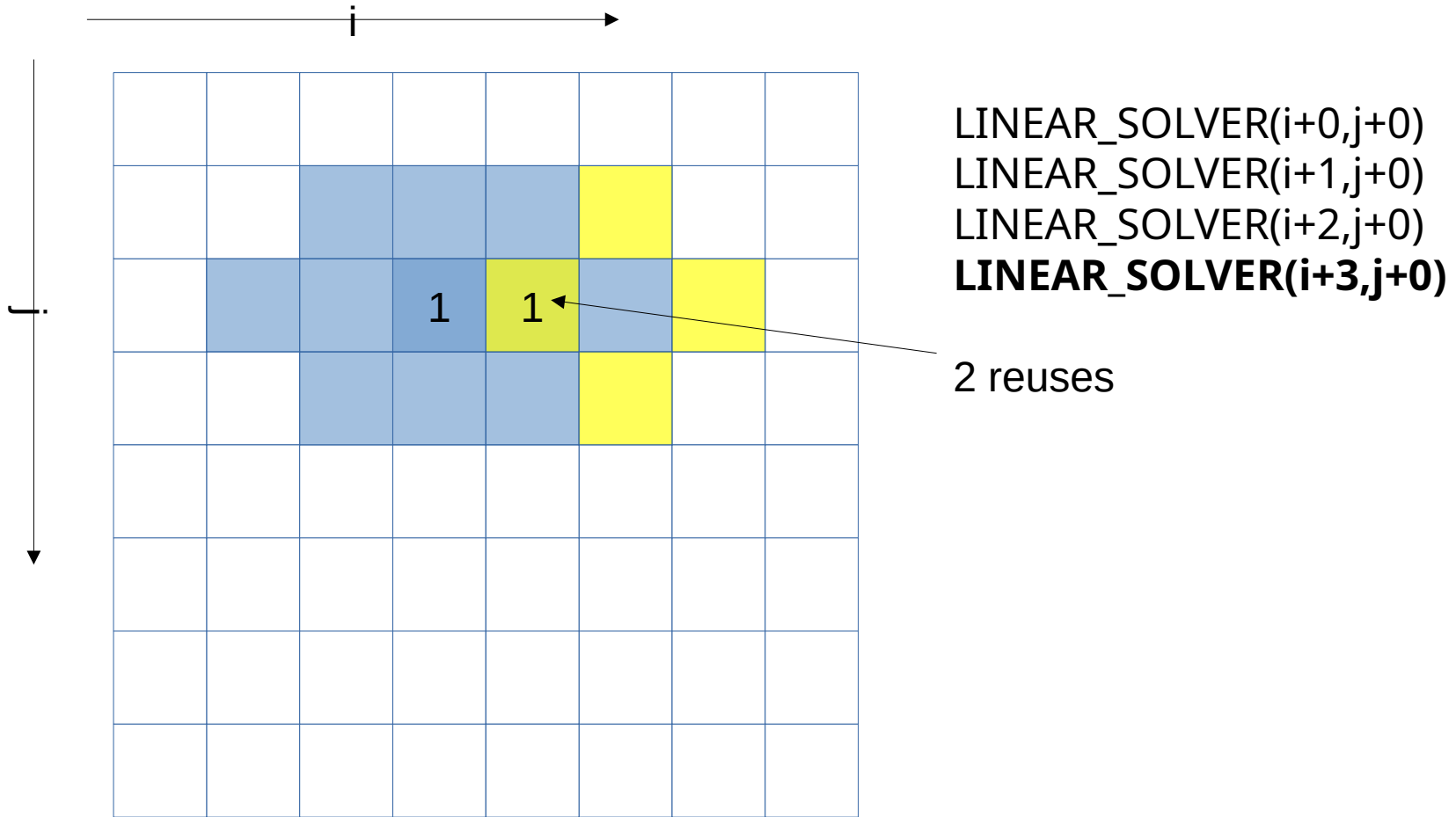


LINEAR\_SOLVER(i+0,j+0)  
**LINEAR\_SOLVER(i+1,j+0)**

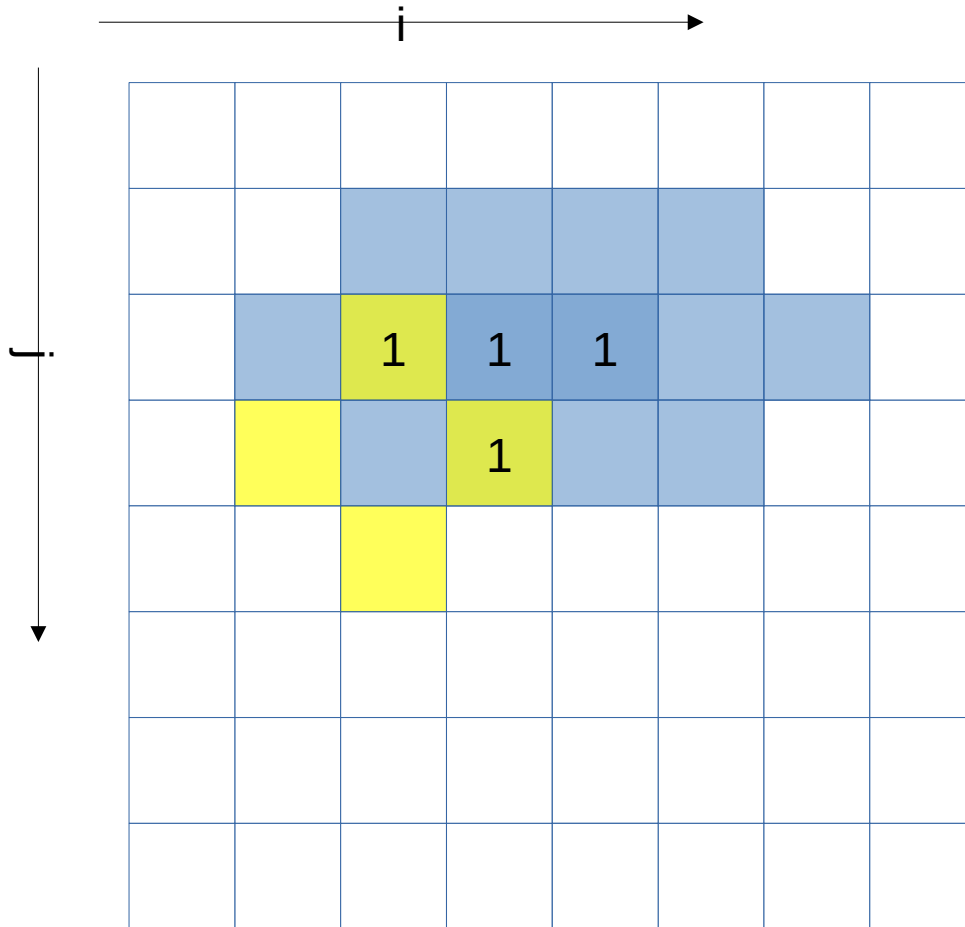
# Memory references reuse : 4x4 unroll footprint on loads



# Memory references reuse : 4x4 unroll footprint on loads



# Memory references reuse : 4x4 unroll footprint on loads

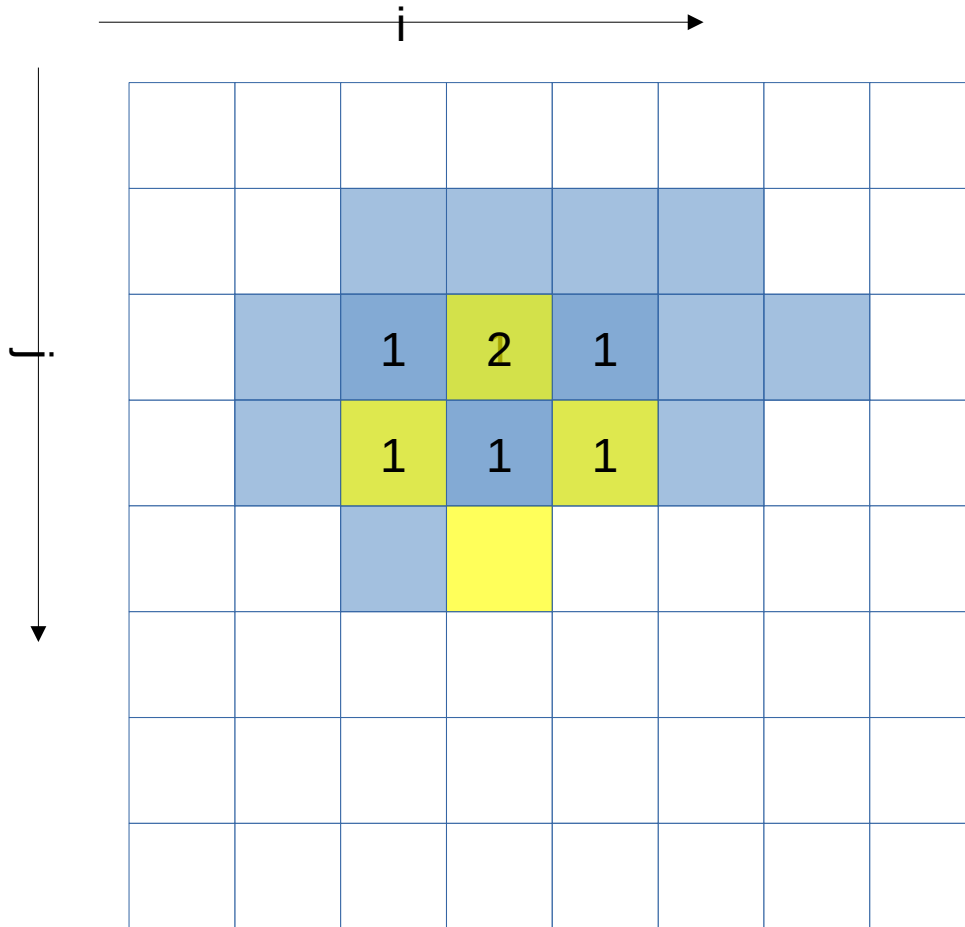


LINEAR\_SOLVER(i+0,j+0)  
 LINEAR\_SOLVER(i+1,j+0)  
 LINEAR\_SOLVER(i+2,j+0)  
 LINEAR\_SOLVER(i+3,j+0)

**LINEAR\_SOLVER(i+0,j+1)**

4 reuses

# Memory references reuse : 4x4 unroll footprint on loads

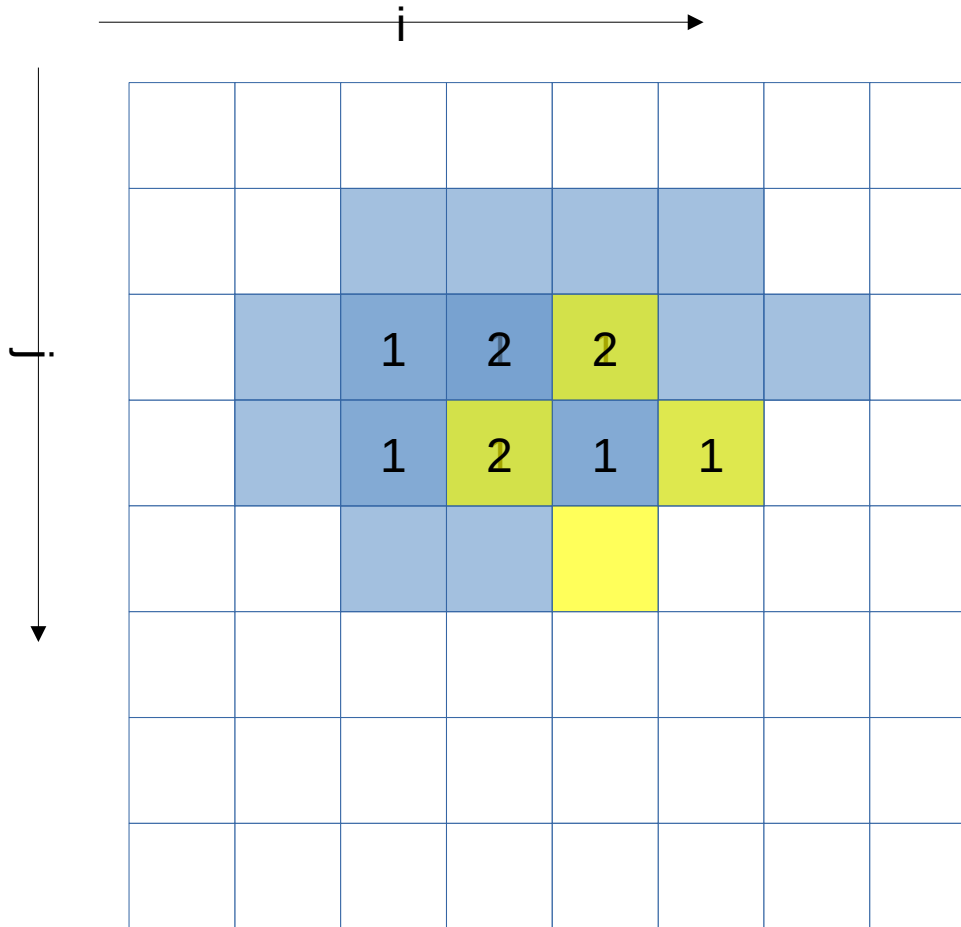


LINEAR\_SOLVER(i+0,j+0)  
 LINEAR\_SOLVER(i+1,j+0)  
 LINEAR\_SOLVER(i+2,j+0)  
 LINEAR\_SOLVER(i+3,j+0)

LINEAR\_SOLVER(i+0,j+1)  
**LINEAR\_SOLVER(i+1,j+1)**

7 reuses

# Memory references reuse : 4x4 unroll footprint on loads

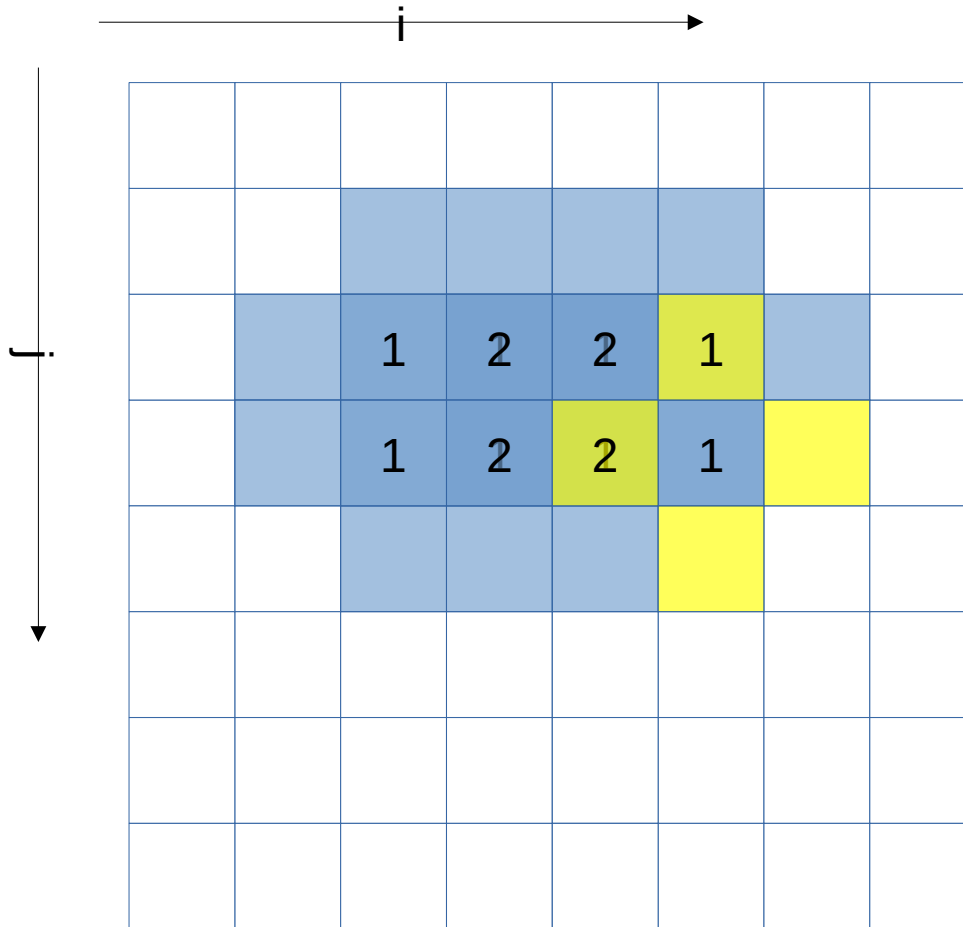


LINEAR\_SOLVER( $i+0, j+0$ )  
 LINEAR\_SOLVER( $i+1, j+0$ )  
 LINEAR\_SOLVER( $i+2, j+0$ )  
 LINEAR\_SOLVER( $i+3, j+0$ )

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

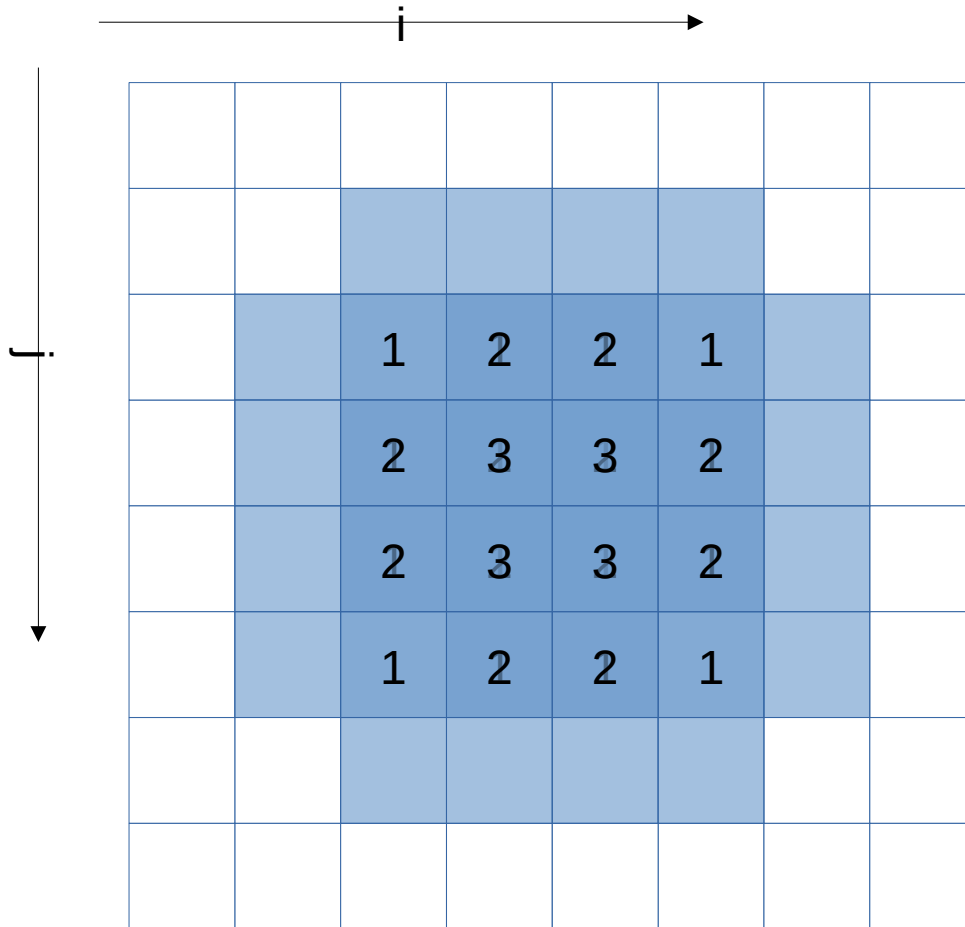


LINEAR\_SOLVER(i+0,j+0)  
 LINEAR\_SOLVER(i+1,j+0)  
 LINEAR\_SOLVER(i+2,j+0)  
 LINEAR\_SOLVER(i+3,j+0)

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

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

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

                LINEARSOLVER (... , i+0, j+3);
                LINEARSOLVER (... , i+1, j+3);
                LINEARSOLVER (... , i+2, j+3);
                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 kernel with manual 4x4 unroll

---

Run

```
> srun -A do009 -p bluefield1 --exclusive -t 1 \  
./hydro_unroll 300 200 # 300x300 mesh, 200 repetitions  
Cycles per element for solvers: 805.34
```

Profile with MAQAO

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

## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/hydro/ov_unroll/RESULTS/\
hydro_unroll_one_html/index.html &
```

Global Metrics		?
Total Time (s)		4.90
Profiled Time (s)		4.89
Time in analyzed loops (%)		100.0
Time in analyzed innermost loops (%)		100.0
Time in user code (%)		100
Compilation Options Score (%)		100
Perfect Flow Complexity		1.04
Array Access Efficiency (%)		62.9
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	2.17
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	7.10
	Nb Loops to get 80%	4
FP Arithmetic Only	Potential Speedup	1.05
	Nb Loops to get 80%	3

## 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)
- 16: multiply (all inside FMA instructions)
- 16: divide

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

4x4 Unrolling were applied

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

### 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 80.00 to 40.00 cycles (2.00x speedup).

### Workaround

Reduce the number of division or square root instructions:

- If denominator is constant over iterations, use reciprocal (replace  $x/y$  with  $x*(1/y)$ ). Check precision impact.

Let's try this

## Running and analyzing kernel with divides hoisting

---

Run

```
> srun -A do009 -p bluefield1 --exclusive -t 1 \  
./hydro_div 300 200 # 300x300 mesh, 200 repetitions  
Cycles per element for solvers: 599.55
```

Profile with MAQAO

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

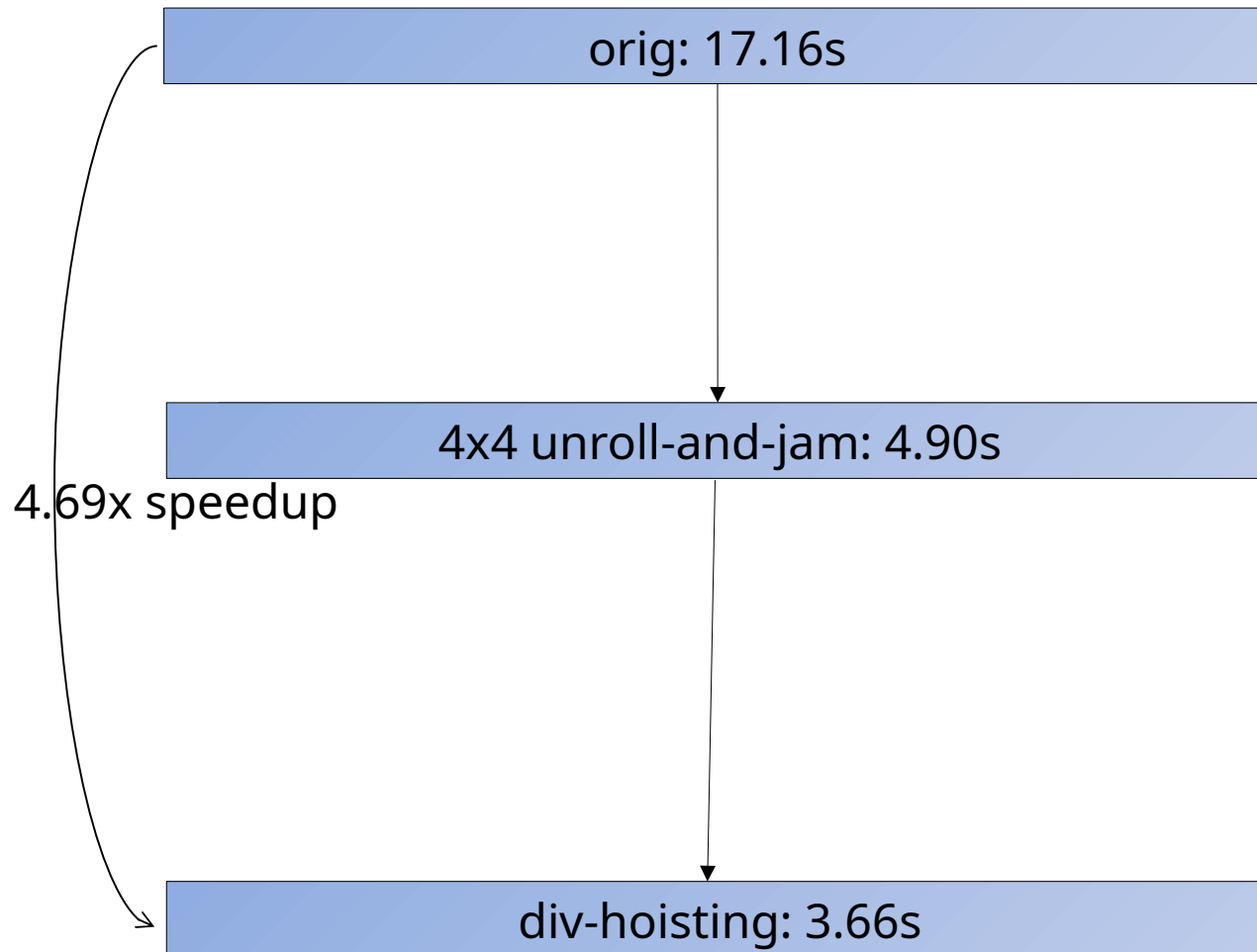
## Viewing results (HTML)

On your local machine (sshfs):

```
> firefox cosma_work/MAQAO_HANDSON/hydro/ov_div/RESULTS/\
hydro_div_one_html/index.html &
```

Global Metrics		?
Total Time (s)		3.66
Profiled Time (s)		3.66
Time in analyzed loops (%)		99.9
Time in analyzed innermost loops (%)		99.9
Time in user code (%)		100
Compilation Options Score (%)		100
Perfect Flow Complexity		1.05
Array Access Efficiency (%)		63.0
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	1.03
	Nb Loops to get 80%	2
Fully Vectorised	Potential Speedup	6.53
	Nb Loops to get 80%	5
FP Arithmetic Only	Potential Speedup	1.40
	Nb Loops to get 80%	2

## Summary of optimizations and gains



## More sample codes

---

More codes to study with MAQAO in

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