

---

# **High Performance Computing and aspects of Computing in Lattice Gauge Theories**

---

\\"

Giannis Koutsou

Computation-based Science and Technology Research Centre (CaSToRC)  
The Cyprus Institute

\\"

EuroPLEx School, 12<sup>th</sup> November 2020

# Lecture 3:

## Practical exercises

- Data layout transformations for improving performance
  1. Exercise 1: Simple Array-of-Structures to Structure-of-Array transformation
  2. Exercise 2: 2-dimensional stencil operation; "Gauged" laplacian operation

# Lecture 3:

## Practical exercises

- Data layout transformations for improving performance
  1. Exercise 1: Simple Array-of-Structures to Structure-of-Array transformation
  2. Exercise 2: 2-dimensional stencil operation; "Gauged" laplacian operation
- Checkout branch `exercises` of repository `EuroPЛЕx2020` on Github:

```
[user@localhost EuroPЛЕx2020]$ git fetch  
[user@localhost EuroPЛЕx2020]$ git checkout exercises
```

# Data layout optimization

Most modern processors have some vectorization capabilities

- SSE, AVX, Altivec, QPX, NEON
- Single Instruction Multiple Data -- SIMD

```
for(int i=0; i<L; i++)  
    y[i] = a*x[i] + y[i];
```

```
float4 va = {a,a,a,a};  
for(int i=0; i<L; i+=4) {  
    float4 vy;  
    float4 vx;  
    load4(vx, &x[i]);  
    load4(vy, &y[i]);  
    vy = va*vx + vy;  
    store4(&y[i], vy);  
}
```

```
for(int i=0; i<L; i+=4) {  
    y[i] = a*x[i] + y[i];  
    y[i+1] = a*x[i+1] + y[i+1];  
    y[i+2] = a*x[i+2] + y[i+2];  
    y[i+3] = a*x[i+3] + y[i+3];  
}
```

**Note:** the above is pseudo-code, namely `load4`, `float4`, etc. are simplifications.

- In many cases, the compiler will generate vector instructions (auto-vectorization)
- However this usually requires an appropriate data layout
- The above is already optimal, but in general this may not be true

# Data layout transformations

## Facilitating vectorization

- Order the data in arrays such that the same operation can be applied to consecutive elements
  - Assists the compiler in detecting auto-vectorization opportunities
  - Optimized memory accesses
  - Assists the programmer in using *vector intrinsics*

# Best practices for optimal data layout

## AoS to SoA

### AoS: Array of structures

Arrays of structures arise when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {
    float x;
    float y;
    float z;
} coords;
```

# Best practices for optimal data layout

## AoS to SoA

### AoS: Array of structures

Arrays of structures arise when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {  
    float x;  
    float y;  
    float z;  
} coords;
```

You can now allocate an array of coords:

```
size_t L = 1000;  
coords *arr = malloc(sizeof(coords)*L);
```

# Best practices for optimal data layout

## AoS to SoA

### AoS: Array of structures

Arrays of structures arise when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {
    float x;
    float y;
    float z;
} coords;
```

You can now allocate an array of coords:

```
size_t L = 1000;
coords *arr = malloc(sizeof(coords)*L);
```

Then operations over the coordinates, such as the following, make auto-vectorization difficult.

```
for(int i=0; i<L; i++)
r[i] = arr[i].x*arr[i].x +
       arr[i].y*arr[i].y +
       arr[i].z*arr[i].z);
```

# Best practices for optimal data layout

## AoS to SoA

### AoS: Array of structures

Arrays of structures arise when mapping physical problems to code. E.g. say you want to define an array of coordinates:

```
typedef struct {
    float x;
    float y;
    float z;
} coords;
```

You can now allocate an array of coords:

```
size_t L = 1000;
coords *arr = malloc(sizeof(coords)*L);
```

Then operations over the coordinates, such as the following, make auto-vectorization difficult.

```
for(int i=0; i<L; i++)
r[i] = arr[i].x*arr[i].x +
       arr[i].y*arr[i].y +
       arr[i].z*arr[i].z;
```

A common way around this is to use a Structure of Arrays (SoA) rather than an Array of Structures.

# Array of Structures to Structure of Arrays

Define a structure of arrays, this is similar to how one programs for GPUs:

```
typedef struct {
    float *x;
    float *y;
    float *z;
} coords;
```

# Array of Structures to Structure of Arrays

Define a structure of arrays, this is similar to how one programs for GPUs:

```
typedef struct {  
    float *x;  
    float *y;  
    float *z;  
} coords;
```

And now allocate each element of `coords` separately:

```
coords arr;  
arr.x = malloc(sizeof(float)*L);  
arr.y = malloc(sizeof(float)*L);  
arr.z = malloc(sizeof(float)*L);
```

# Array of Structures to Structure of Arrays

Define a structure of arrays, this is similar to how one programs for GPUs:

```
typedef struct {  
    float *x;  
    float *y;  
    float *z;  
} coords;
```

And now allocate each element of `coords` separately:

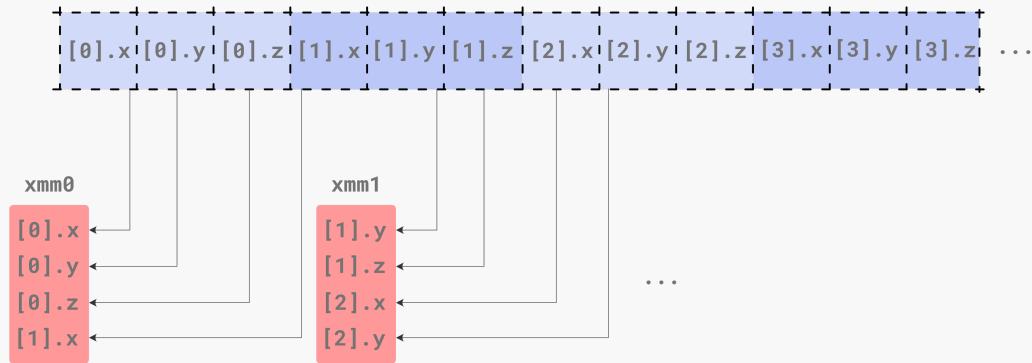
```
coords arr;  
arr.x = malloc(sizeof(float)*L);  
arr.y = malloc(sizeof(float)*L);  
arr.z = malloc(sizeof(float)*L);
```

You can see how it is more obvious vectorize the distance calculation after unrolling:

```
for(int i=0; i<L; i+=4) {  
    r[i] = arr.x[i]*arr.x[i]+arr.y[i]*arr.y[i]+arr.z[i]*arr.z[i];  
    r[i+1] = arr.x[i+1]*arr.x[i+1]+arr.y[i+1]*arr.y[i+1]+arr.z[i+1]*arr.z[i+1];  
    r[i+2] = arr.x[i+2]*arr.x[i+2]+arr.y[i+2]*arr.y[i+2]+arr.z[i+2]*arr.z[i+2];  
    r[i+3] = arr.x[i+3]*arr.x[i+3]+arr.y[i+3]*arr.y[i+3]+arr.z[i+3]*arr.z[i+3];  
}
```

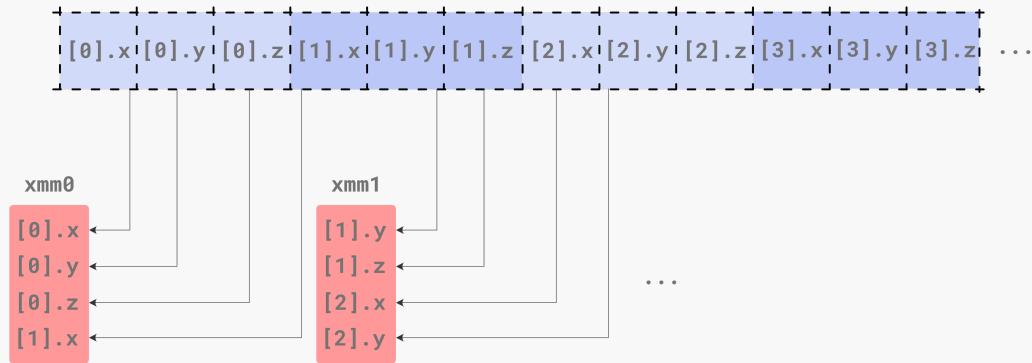
# AoS to SoA

First elements in AoS, loaded into 4-element vector registers



# AoS to SoA

First elements in AoS, loaded into 4-element vector registers



# AoS to SoA

In the first exercise we will modify a program that uses an Array-of-Structures to use a Structure-of-Arrays. The main computational kernel in the program does the following:

- Initializes an array of 4-vectors:  $a_i = (ct_i, x_i, y_i, z_i)$
- Computes the squared "norm":  $s_i = (ct_i)^2 - (x_i^2 + y_i^2 + z_i^2)$  multiple times
- Reports the average time to compute all elements  $s_i$ .

To obtain the exercises, pull the `exercises` branch of the Git repository:

- `cd` into the repository on your local system, fetch and checkout the `exercises` branch, and `cd` into the `AoS-to-SoA` directory:

```
[user@localhost ~]$ cd EuroPLEx2020/
[user@localhost EuroPLEx2020]$ git fetch
[user@localhost EuroPLEx2020]$ git checkout exercises
[user@localhost EuroPLEx2020]$ cd AoS-to-SoA/
[user@localhost AoS-to-SoA]$ ls -1
Makefile
plot-A.py
plot-B.py
plot-C.py
run-A.sh
run-B.sh
run-C.sh
space-time-aos.c
space-time-soa.c
```

# AoS to SoA

First, look into `space-time-aos.c`

```
while(1) {
    double t0 = stop_watch(0);
    for(int i=0; i<NREP; i++)
        comp_s(arr, L);
    t0 = stop_watch(t0)/(double)NREP;
    t0acc += t0;
    t1acc += t0*t0;
    if(n > 2) {
        double ave = t0acc/n;
        double err = sqrt(t1acc/n -
                           ave*ave)/sqrt(n);
        if(err/ave < 0.1) {
            t0acc = ave;
            t1acc = err;
            break;
        }
    }
    n++;
}
```

# AoS to SoA

First, look into `space-time-aos.c`

```
while(1) {
    double t0 = stop_watch(0);
    for(int i=0; i<NREP; i++)
        comp_s(arr, L);
    t0 = stop_watch(t0)/(double)NREP;
    t0acc += t0;
    t1acc += t0*t0;
    if(n > 2) {
        double ave = t0acc/n;
        double err = sqrt(t1acc/n -
                           ave*ave)/sqrt(n);
        if(err/ave < 0.1) {
            t0acc = ave;
            t1acc = err;
            break;
        }
    }
    n++;
}
```

- The kernel is (`comp_s()`)
- The kernel is called `NREP` times
- Does this repeatedly. Stops when variation in time to call kernel `NREP` times is smaller than 10%
- `t0acc` holds the average time per `comp_s()` call

# AoS to SoA

First, look into `space-time-aos.c`

```
while(1) {
    double t0 = stop_watch(0);
    for(int i=0; i<NREP; i++)
        comp_s(arr, L);
    t0 = stop_watch(t0)/(double)NREP;
    t0acc += t0;
    t1acc += t0*t0;
    if(n > 2) {
        double ave = t0acc/n;
        double err = sqrt(t1acc/n -
                           ave*ave)/sqrt(n);
        if(err/ave < 0.1) {
            t0acc = ave;
            t1acc = err;
            break;
        }
    }
    n++;
}
```

- The kernel is (`comp_s()`)
- The kernel is called `NREP` times
- Does this repeatedly. Stops when variation in time to call kernel `NREP` times is smaller than 10%
- `t0acc` holds the average time per `comp_s()` call

`stop_watch()` is just a system timer:

```
double
stop_watch(double t0) {
    struct timeval tp;
    gettimeofday(&tp, NULL);
    double t1 = tp.tv_sec + tp.tv_usec*1e-6;
    return t1-t0;
}
```

# AoS to SoA

x, y, z, t, and s are elements of a struct

```
typedef struct {
    float x, y, z, t;
    float s;
} st_coords;
```

# AoS to SoA

x, y, z, t, and s are elements of a struct

```
typedef struct {
    float x, y, z, t;
    float s;
} st_coords;
```

The computation of s is a loop over an array of st\_coords:

```
void
comp_s(st_coords *arr, int L)
{
#pragma omp parallel for
for(int i=0; i<L; i++) {
    float x = arr[i].x;
    float y = arr[i].y;
    float z = arr[i].z;
    float t = arr[i].t;
    float s = t*t - (x*x + y*y + z*z);
    arr[i].s = s;
}
return;
}
```

Note that we are using OpenMP to parallelize the loop

# AoS to SoA

For the first task you will modify `space-time-aos.c`. You have instructions tagged with `_TODO_A_`.

```
void
comp_s(st_coords *arr, int L)
{
#pragma omp parallel for
for(int i=0; i<L; i++) {
    float x = arr[i].x;
    float y = arr[i].y;
    float z = arr[i].z;
    float t = arr[i].t;
    float s = t*t - (x*x + y*y + z*z);
    arr[i].s = s;
}
return;
}
...
double beta_fp = 0 /* _TODO_A_: insert number of */
/* Gflop/sec based on timing */;
double beta_io = 0 /* _TODO_A_: insert number of */
/* Gbyte/sec based on timing */;
```

Correct `beta_fp` and `beta_io`.

# AoS to SoA

After correcting for `beta_fp` and `beta_io`, you need to compile. `make A` should do this for you:

```
[user@localhost AoS-To-SoA]$ make A  
cc -O3 -Xpreprocessor -fopenmp -std=gnu99 space-time-aos.c -o space-time-aos -lomp -lm
```

and run the prepared shell script:

```
[user@localhost AoS-To-SoA]$ sh run-A.sh
```

The shell script sets the number of OpenMP threads to **four**. Modify this if you are on a system that you know has more cores.

# AoS to SoA

After correcting for `beta_fp` and `beta_io`, you need to compile. `make A` should do this for you:

```
[user@localhost AoS-To-SoA]$ make A  
cc -O3 -Xpreprocessor -fopenmp -std=gnu99 space-time-aos.c -o space-time-aos -lomp -lm
```

and run the prepared shell script:

```
[user@localhost AoS-To-SoA]$ sh run-A.sh
```

The shell script sets the number of OpenMP threads to **four**. Modify this if you are on a system that you know has more cores.

Running the shell script will run the code for various lengths `L`. The output is in `aos.txt`. There is a Python script so that you can visualize the output, by running:

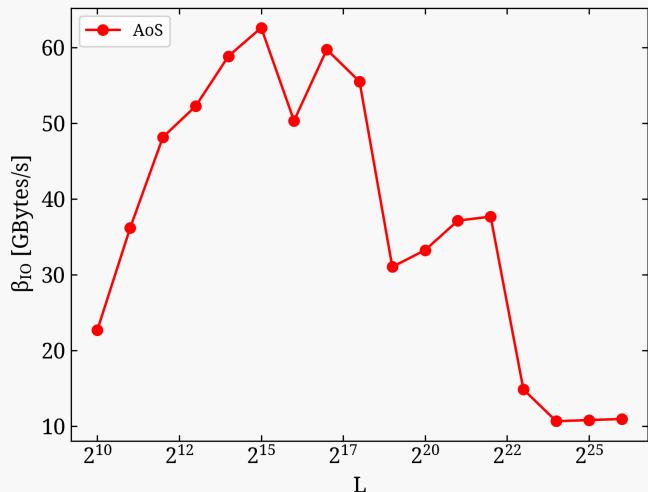
```
[user@localhost AoS-To-SoA]$ python3 plot-A.py
```

which will produce `A.png`.

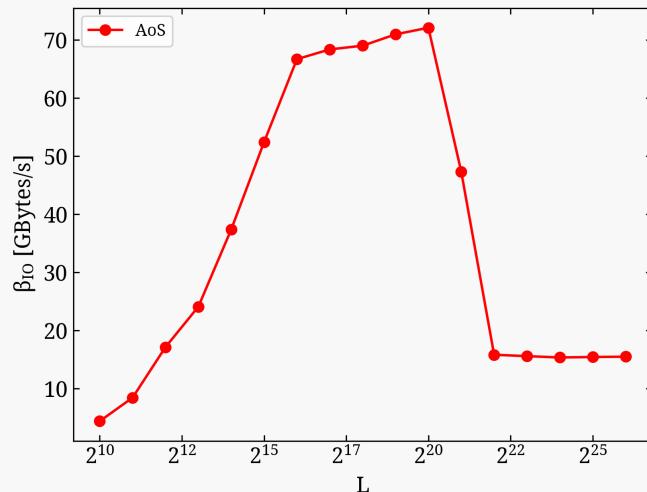
# AoS to SoA

## Task A

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)

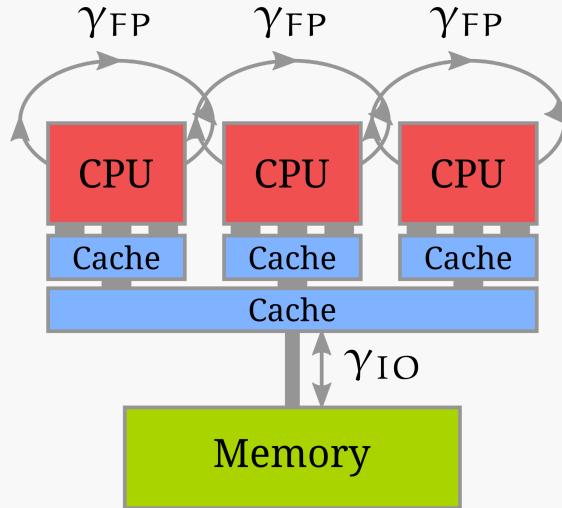


Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)



# AoS to SoA

## Task A



- When  $L \ll$ , all arrays fit in cache. I/O to cache is at least an order of magnitude faster than I/O to memory
- When  $L \gg$  arrays do not fit into cache. Repeated calls to `comp_s()` require repeatedly fetching the arrays from memory. This is more representative of a typical application.

# AoS to SoA

In Task B you need to modify `space-time-soa.c`. First transfer your modifications from `space-time-aos.c`, tagged with `_TODO_A_`.

# AoS to SoA

In Task B you need to modify `space-time-soa.c`. First transfer your modifications from `space-time-aos.c`, tagged with `_TODO_A_`.

`space-time-soa.c` implements a structure of arrays of length `VL` rather than an array of structures:

```
typedef struct {
    float x[VL];
    float y[VL];
    float z[VL];
    float t[VL];
    float s[VL];
} st_coords;
```

E.g.: `arr[i].x` -> `arr[i0].x[i1]`, with:

- `i0 = i/VL = 0, ..., N/VL-1` and
- `i1 = i%VL = 0, ..., VL-1`

# AoS to SoA

In Task B you need to modify `space-time-soa.c`. First transfer your modifications from `space-time-aos.c`, tagged with `_TODO_A_`.

`space-time-soa.c` implements a structure of arrays of length `VL` rather than an array of structures:

```
typedef struct {
    float x[VL];
    float y[VL];
    float z[VL];
    float t[VL];
    float s[VL];
} st_coords;
```

E.g.: `arr[i].x` -> `arr[i0].x[i1]`, with:

- `i0 = i/VL = 0, ..., N/VL-1` and
- `i1 = i%VL = 0, ..., VL-1`

For the `_TODO_B_` pieces, you need to implement `comp_s()` to operate on this new data layout:

```
/*
 * compute: s = t**2 - x**2 - y**2 - z**2,
 * with s, t, x, y, z member variables of
 * length VL of the st_coords structure
 */
void
comp_s(st_coords *arr, int L)
{
    /* _TODO_B_: complete the kernel */
    #pragma omp parallel for
    ...
}
```

# AoS to SoA

In Task B you need to modify `space-time-soa.c`. First transfer your modifications from `space-time-aos.c`, tagged with `_TODO_A_`.

`space-time-soa.c` implements a structure of arrays of length `VL` rather than an array of structures:

```
typedef struct {
    float x[VL];
    float y[VL];
    float z[VL];
    float t[VL];
    float s[VL];
} st_coords;
```

E.g.: `arr[i].x` -> `arr[i0].x[i1]`, with:

- `i0 = i/VL = 0, ..., N/VL-1` and
- `i1 = i%VL = 0, ..., VL-1`

For the `_TODO_B_` pieces, you need to implement `comp_s()` to operate on this new data layout:

```
/*
 * compute: s = t**2 - x**2 - y**2 - z**2,
 * with s, t, x, y, z member variables of
 * length VL of the st_coords structure
 */
void
comp_s(st_coords *arr, int L)
{
    /* _TODO_B_: complete the kernel */
    #pragma omp parallel for
    ...
}
```

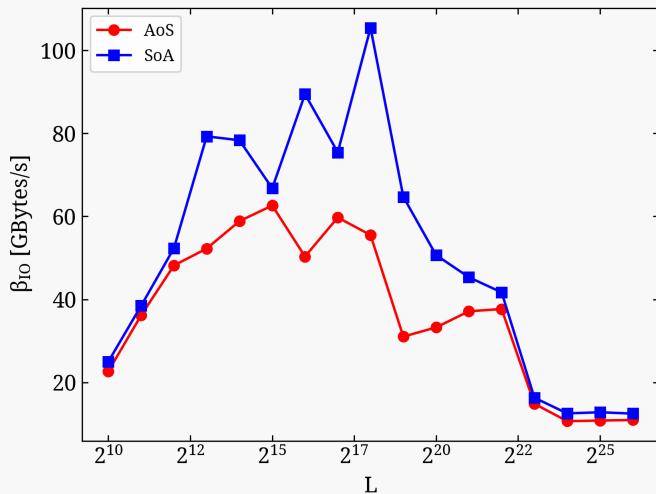
Once done, as in Task A, compile with `make B` and run using the script `run-B.sh`:

```
[user@localhost AoS-To-SoA]$ make B
cc -O3 -Xpreprocessor -fopenmp -std=gnu99 space-time-soa.c -o space-time-soa -lomp -lm
[user@localhost AoS-To-SoA]$ sh run-B.sh
```

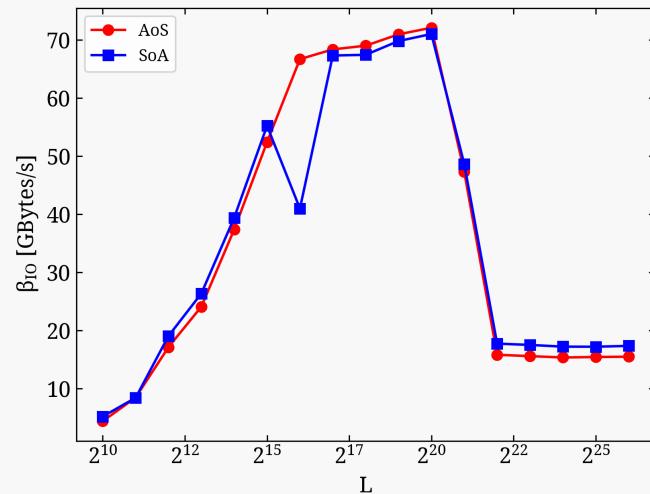
# AoS to SoA

## Task B

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)



Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)



# AoS to SoA

## Task C

In Task C, you will use the same program, `space-time-soa.c`. But you will run it multiple times, varying `VL`. You are **not required** to modify code. Look at `run-C.sh`:

```
export OMP_NUM_THREADS=4
for VL in 2 4 8 16 32 64 128 256; do
    make -B VL=${VL} B
    for ((L=1024; L<=$((64*1024*1024)); L*=2)) ; do
        ./space-time-soa $L
        done | tee soa-vl${VL}.txt
done
```

This will compile and run the program for all values of `VL` in the list. Run with:

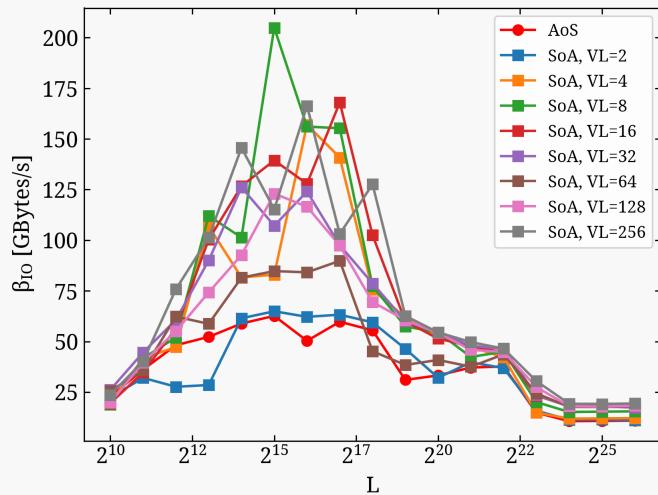
```
[user@localhost AoS-To-SoA]$ sh run-C.sh
```

and plot using: `python3 plot-C.py`. This will produce `C.png`.

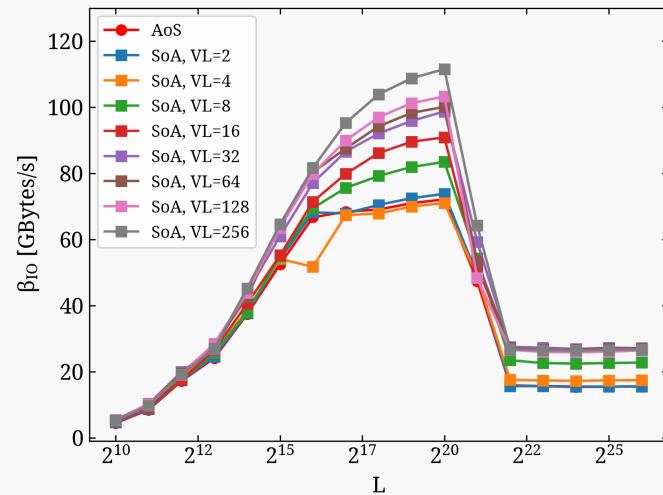
# AoS to SoA

## Task C

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)



Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)

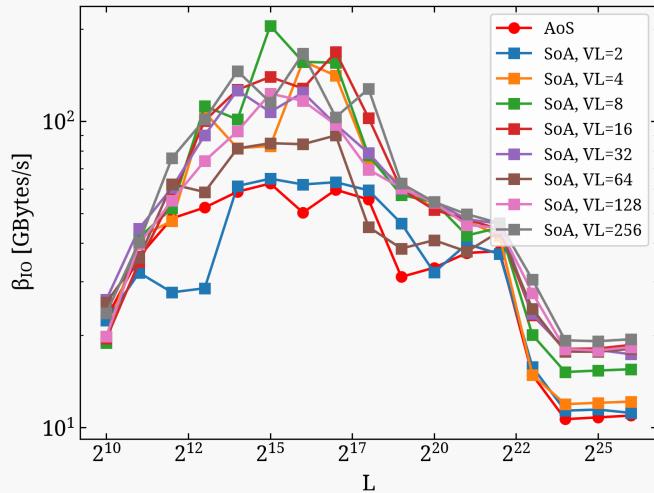


The optimal value of  $VL$  is in general architecture-dependent. For which value of  $VL$  do you see best performance?

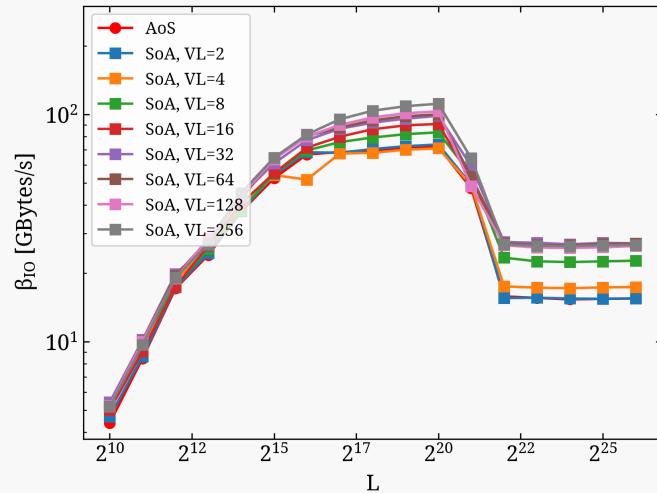
# AoS to SoA

## Task C

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)



Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)



The optimal value of  $VL$  is in general architecture-dependent. For which value of  $VL$  do you see best performance?

# Data layout in stencil codes

Another application of the data re-ordering is in stencil codes. Consider a 2-D stencils operation:

$$\Phi_{x,y} \leftarrow 4\psi_{x,y} - (\psi_{x+1,y} + \psi_{x-1,y} + \psi_{x,y+1} + \psi_{x,y-1})$$

```
for(int y=0; y<L; y++)  
    for(int x=0; x<L; x++) {  
        B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
    }
```

# Data layout in stencil codes

Another application of the data re-ordering is in stencil codes. Consider a 2-D stencils operation:

$$\Phi_{x,y} \leftarrow 4\psi_{x,y} - (\psi_{x+1,y} + \psi_{x-1,y} + \psi_{x,y+1} + \psi_{x,y-1})$$

```
for(int y=0; y<L; y++)  
    for(int x=0; x<L; x++) {  
        B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
    }
```

Consider an unrolled implementation of this, with `x` running fastest and `y` slowest:

```
for(int y=0; y<L; y++)  
    for(int x=0; x<L; x+=4) {  
        B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
        B[y][x+1] = 4*A[y][x+1] - (A[y][x+2] + A[y][x] + A[y+1][x+1] + A[y-1][x+1]);  
        B[y][x+2] = 4*A[y][x+2] - (A[y][x+3] + A[y][x+1] + A[y+1][x+2] + A[y-1][x+2]);  
        B[y][x+3] = 4*A[y][x+3] - (A[y][x+4] + A[y][x+2] + A[y+1][x+3] + A[y-1][x+3]);  
    }
```

# Data layout in stencil codes

Another application of the data re-ordering is in stencil codes. Consider a 2-D stencils operation:

$$\Phi_{x,y} \leftarrow 4\psi_{x,y} - (\psi_{x+1,y} + \psi_{x-1,y} + \psi_{x,y+1} + \psi_{x,y-1})$$

```
for(int y=0; y<L; y++)  
    for(int x=0; x<L; x++) {  
        B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
    }
```

Consider an unrolled implementation of this, with `x` running fastest and `y` slowest:

```
for(int y=0; y<L; y++)  
    for(int x=0; x<L; x+=4) {  
        B[y][x] = 4*A[y][x] - (A[y][x+1] + A[y][x-1] + A[y+1][x] + A[y-1][x]);  
        B[y][x+1] = 4*A[y][x+1] - (A[y][x+2] + A[y][x] + A[y+1][x+1] + A[y-1][x+1]);  
        B[y][x+2] = 4*A[y][x+2] - (A[y][x+3] + A[y][x+1] + A[y+1][x+2] + A[y-1][x+2]);  
        B[y][x+3] = 4*A[y][x+3] - (A[y][x+4] + A[y][x+2] + A[y+1][x+3] + A[y-1][x+3]);  
    }
```

- Operations involve `A[y][x:x+4]`, `A[y][x+1:x+5]`, and `A[y][x-1:x+3]`
- They are badly aligned
- Hard to take advantage of vectorization without inserting multiple *shuffle* and *shift* operations

# Data layout in stencil codes

One transformation: change the data layout so that **distant** elements run **fastest**

- If original array is, e.g.

$A[y][x]$ , with  $x=0, \dots, L-1$  and  $y=0, \dots, L-1$

- Change to:

$vA[y_0][x][y_1]$ , with  $y_0=0, \dots, L/4-1$ ,  $y_1=0, 1, 2, 3$ , and  $y=y_1*(L/4) + y_0$

# Data layout in stencil codes

One transformation: change the data layout so that **distant** elements run **fastest**

- If original array is, e.g.

$A[y][x]$ , with  $x=0, \dots, L-1$  and  $y=0, \dots, L-1$

- Change to:

$vA[y_0][x][y_1]$ , with  $y_0=0, \dots, L/4-1$ ,  $y_1=0, 1, 2, 3$ , and  $y=y_1*(L/4) + y_0$

- The stencil operation loop will then be:

```
for(int y0=0; y0<L/4; y0++)  
    for(int x=0; x<L; x++) {  
        vB[y0][x][0] = 4*vA[y0][x][0] - (vA[y0][x+1][0] + vA[y0][x-1][0] + vA[y0+1][x][0] + vA[y0-1][x][0]);  
        vB[y0][x][1] = 4*vA[y0][x][1] - (vA[y0][x+1][1] + vA[y0][x-1][1] + vA[y0+1][x][1] + vA[y0-1][x][1]);  
        vB[y0][x][2] = 4*vA[y0][x][2] - (vA[y0][x+1][2] + vA[y0][x-1][2] + vA[y0+1][x][2] + vA[y0-1][x][2]);  
        vB[y0][x][3] = 4*vA[y0][x][3] - (vA[y0][x+1][3] + vA[y0][x-1][3] + vA[y0+1][x][3] + vA[y0-1][x][3]);  
    }
```

# Data layout in stencil codes

One transformation: change the data layout so that **distant** elements run **fastest**

- If original array is, e.g.

$A[y][x]$ , with  $x=0, \dots, L-1$  and  $y=0, \dots, L-1$

- Change to:

$vA[y_0][x][y_1]$ , with  $y_0=0, \dots, L/4-1$ ,  $y_1=0, 1, 2, 3$ , and  $y=y_1*(L/4) + y_0$

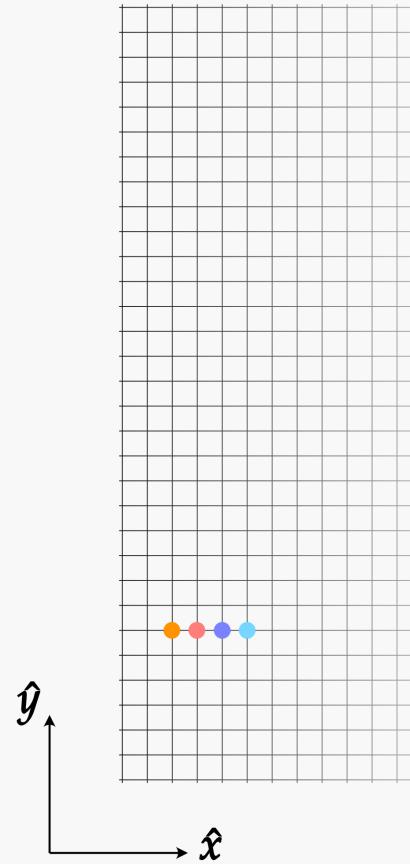
- The stencil operation loop will then be:

```
for(int y0=0; y0<L/4; y0++)  
    for(int x=0; x<L; x++) {  
        vB[y0][x][0] = 4*vA[y0][x][0] - (vA[y0][x+1][0] + vA[y0][x-1][0] + vA[y0+1][x][0] + vA[y0-1][x][0]);  
        vB[y0][x][1] = 4*vA[y0][x][1] - (vA[y0][x+1][1] + vA[y0][x-1][1] + vA[y0+1][x][1] + vA[y0-1][x][1]);  
        vB[y0][x][2] = 4*vA[y0][x][2] - (vA[y0][x+1][2] + vA[y0][x-1][2] + vA[y0+1][x][2] + vA[y0-1][x][2]);  
        vB[y0][x][3] = 4*vA[y0][x][3] - (vA[y0][x+1][3] + vA[y0][x-1][3] + vA[y0+1][x][3] + vA[y0-1][x][3]);  
    }
```

- As long as we keep the data structures ordered this way, the stencil operation will be nicely aligned

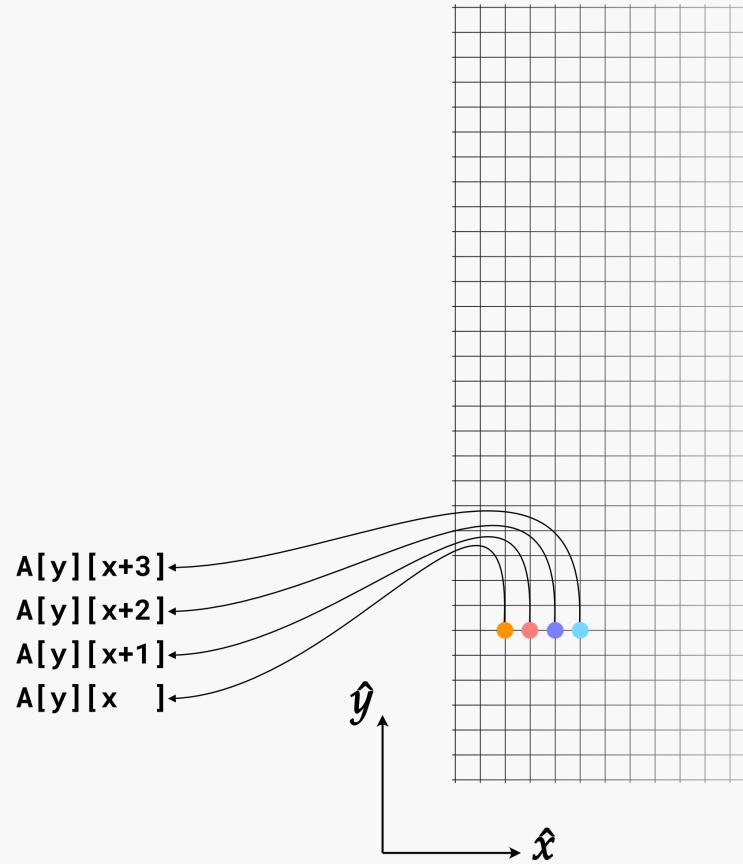
# Data layout in stencil codes

Change in data layout so that **distant** elements run **fastest**



# Data layout in stencil codes

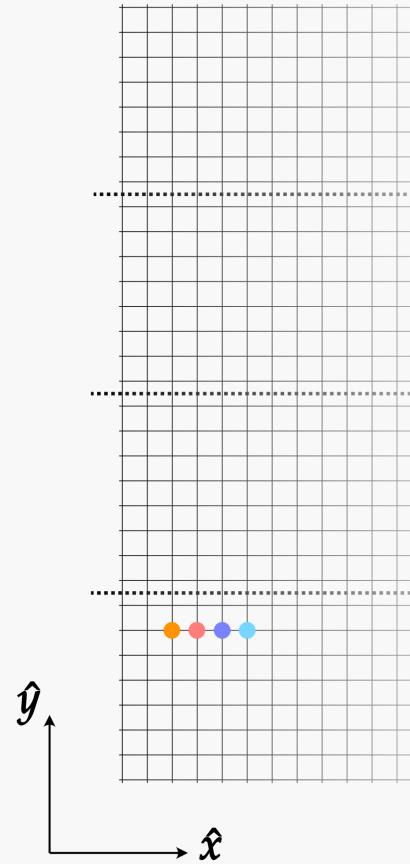
Change in data layout so that **distant** elements run **fastest**



# Data layout in stencil codes

Change in data layout so that **distant** elements run **fastest**

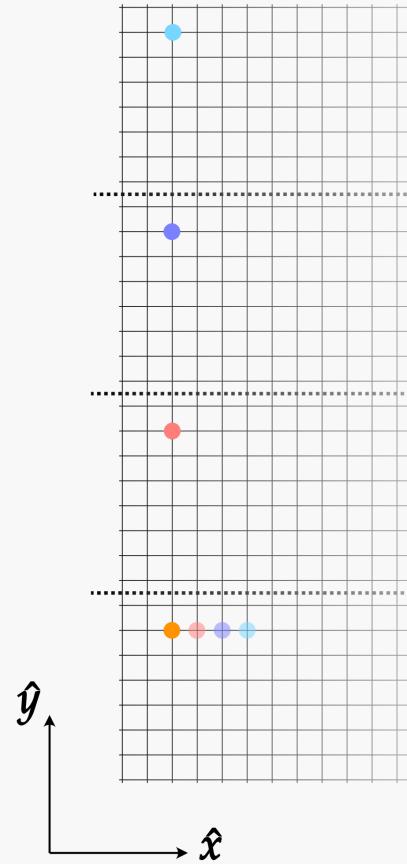
$A[y][x+3]$   
 $A[y][x+2]$   
 $A[y][x+1]$   
 $A[y][x]$



# Data layout in stencil codes

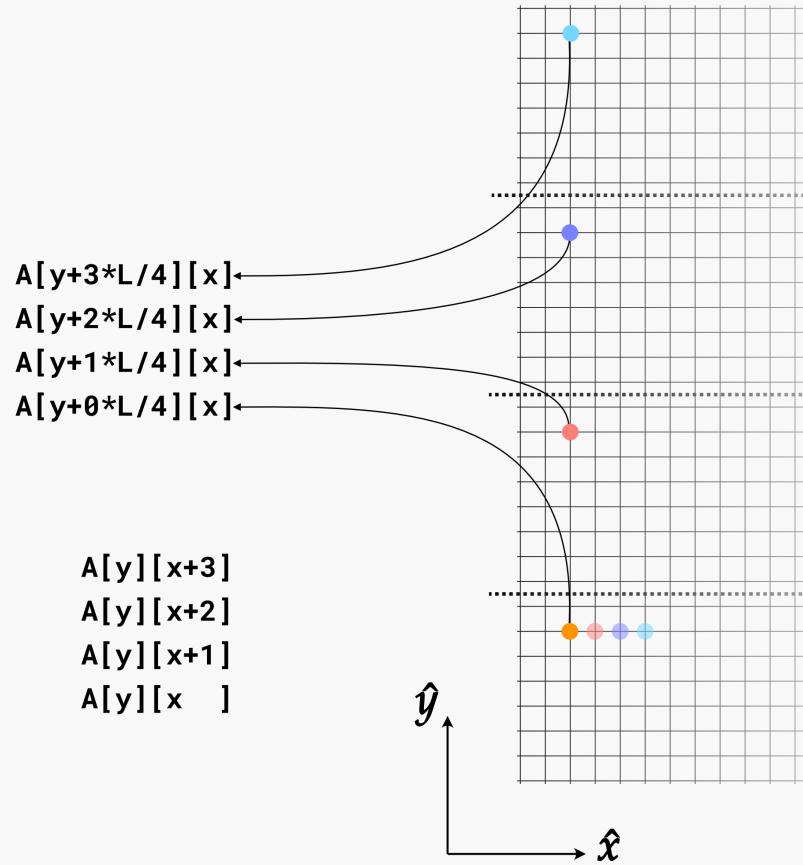
Change in data layout so that **distant** elements run **fastest**

$A[y][x+3]$   
 $A[y][x+2]$   
 $A[y][x+1]$   
 $A[y][x]$



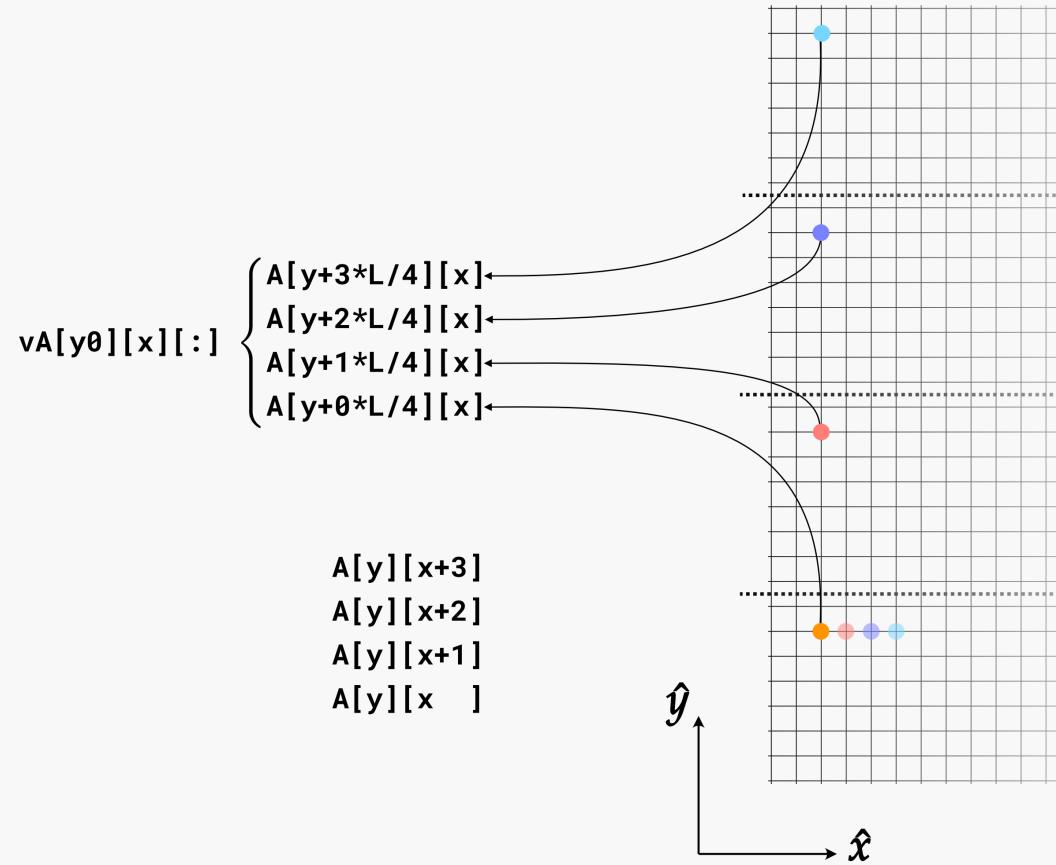
# Data layout in stencil codes

Change in data layout so that **distant** elements run **fastest**



# Data layout in stencil codes

Change in data layout so that **distant** elements run **fastest**



# Data layout in stencil codes

- To summarize, reorder the data layout as so (previous slides assumed VL=4):

```
for(int y1=0; y1<VL; y1++)  
    for(int y0=0; y0<L/VL; y0++)  
        for(int x=0; x<L; x++) {  
            vA[y0][x][y1] = A[y1*(L/VL) + y0][x];  
        }
```

- The stencil operation will then be:

```
for(int y0=0; y0<L/VL; y0++)  
    for(int x=0; x<L; x++) {  
        vB[y0][x][0] = 4*vA[y0][x][0] - (vA[y0][x+1][0] + vA[y0][x-1][0] + vA[y0+1][x][0] + vA[y0-1][x][0]);  
        vB[y0][x][1] = 4*vA[y0][x][1] - (vA[y0][x+1][1] + vA[y0][x-1][1] + vA[y0+1][x][1] + vA[y0-1][x][1]);  
        vB[y0][x][2] = 4*vA[y0][x][2] - (vA[y0][x+1][2] + vA[y0][x-1][2] + vA[y0+1][x][2] + vA[y0-1][x][2]);  
        vB[y0][x][3] = 4*vA[y0][x][3] - (vA[y0][x+1][3] + vA[y0][x-1][3] + vA[y0+1][x][3] + vA[y0-1][x][3]);  
    }
```

# Data layout in stencil codes

- To summarize, reorder the data layout as so (previous slides assumed  $VL=4$ ):

```
for(int y1=0; y1<VL; y1++)  
    for(int y0=0; y0<L/VL; y0++)  
        for(int x=0; x<L; x++) {  
            vA[y0][x][y1] = A[y1*(L/VL) + y0][x];  
        }
```

- The stencil operation will then be:

```
for(int y0=0; y0<L/VL; y0++)  
    for(int x=0; x<L; x++) {  
        vB[y0][x][0] = 4*vA[y0][x][0] - (vA[y0][x+1][0] + vA[y0][x-1][0] + vA[y0+1][x][0] + vA[y0-1][x][0]);  
        vB[y0][x][1] = 4*vA[y0][x][1] - (vA[y0][x+1][1] + vA[y0][x-1][1] + vA[y0+1][x][1] + vA[y0-1][x][1]);  
        vB[y0][x][2] = 4*vA[y0][x][2] - (vA[y0][x+1][2] + vA[y0][x-1][2] + vA[y0+1][x][2] + vA[y0-1][x][2]);  
        vB[y0][x][3] = 4*vA[y0][x][3] - (vA[y0][x+1][3] + vA[y0][x-1][3] + vA[y0+1][x][3] + vA[y0-1][x][3]);  
    }
```

- But this is **wrong** on the boundaries  $y0=0$  and  $y0=L/4-1$ !

# Data layout in stencil codes

## Caution on the boundaries

- In the original data layout:

```
y = 0;
for(int x=0; x<L; x++) {
    B[0][x] = 4*A[0][x] - (A[0][x+1] + A[0][x-1] + A[1][x] + A[L-1][x]);
}
```

# Data layout in stencil codes

## Caution on the boundaries

- In the original data layout:

```
y0 = 0;
for(int x=0; x<L; x++) {
    B[0][x] = 4*A[0][x] - (A[0][x+1] + A[0][x-1] + A[1][x] + A[L-1][x]);
}
```

- In the new data layout:

```
y0 = 0;
for(int x=0; x<L; x++) {
    vB[0][x][0]=4*vA[0][x][0] - (vA[0][x+1][0] + vA[0][x-1][0] + vA[1][x][0] + vA[L/VL-1][x][3]);
    vB[0][x][1]=4*vA[0][x][1] - (vA[0][x+1][1] + vA[0][x-1][1] + vA[1][x][1] + vA[L/VL-1][x][0]);
    vB[0][x][2]=4*vA[0][x][2] - (vA[0][x+1][2] + vA[0][x-1][2] + vA[1][x][2] + vA[L/VL-1][x][1]);
    vB[0][x][3]=4*vA[0][x][3] - (vA[0][x+1][3] + vA[0][x-1][3] + vA[1][x][3] + vA[L/VL-1][x][2]);
}
```

# Data layout in stencil codes

## Caution on the boundaries

- In the original data layout:

```
y0 = 0;
for(int x=0; x<L; x++) {
    B[0][x] = 4*A[0][x] - (A[0][x+1] + A[0][x-1] + A[1][x] + A[L-1][x]);
}
```

- In the new data layout:

```
y0 = 0;
for(int x=0; x<L; x++) {
    vB[0][x][0]=4*vA[0][x][0] - (vA[0][x+1][0] + vA[0][x-1][0] + vA[1][x][0] + vA[L/VL-1][x][3]);
    vB[0][x][1]=4*vA[0][x][1] - (vA[0][x+1][1] + vA[0][x-1][1] + vA[1][x][1] + vA[L/VL-1][x][0]);
    vB[0][x][2]=4*vA[0][x][2] - (vA[0][x+1][2] + vA[0][x-1][2] + vA[1][x][2] + vA[L/VL-1][x][1]);
    vB[0][x][3]=4*vA[0][x][3] - (vA[0][x+1][3] + vA[0][x-1][3] + vA[1][x][3] + vA[L/VL-1][x][2]);
}
```

- And for the other boundary

```
y0 = L/VL-1;
for(int x=0; x<L; x++) {
    vB[L/VL-1][x][0]=4*vA[L/VL-1][x][0] - (vA[L/VL-1][x+1][0] + vA[L/VL-1][x-1][0] + vA[0][x][1] + vA[L/VL-2][x][0]);
    vB[L/VL-1][x][1]=4*vA[L/VL-1][x][1] - (vA[L/VL-1][x+1][1] + vA[L/VL-1][x-1][1] + vA[0][x][2] + vA[L/VL-2][x][1]);
    vB[L/VL-1][x][2]=4*vA[L/VL-1][x][2] - (vA[L/VL-1][x+1][2] + vA[L/VL-1][x-1][2] + vA[0][x][3] + vA[L/VL-2][x][2]);
    vB[L/VL-1][x][3]=4*vA[L/VL-1][x][3] - (vA[L/VL-1][x+1][3] + vA[L/VL-1][x-1][3] + vA[0][x][0] + vA[L/VL-2][x][3]);
}
```

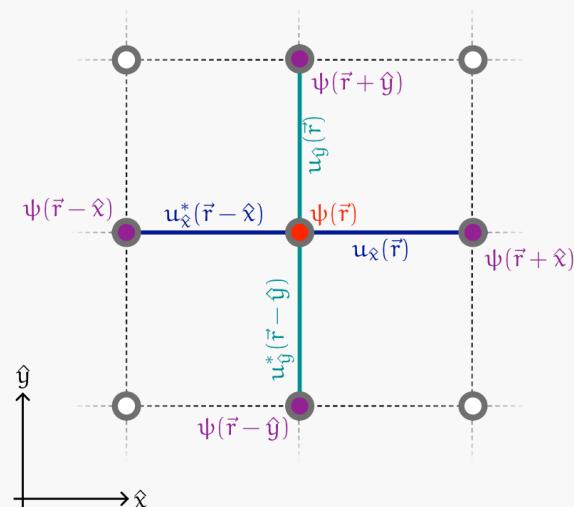
# Data layout in stencil codes: Exercise

- Consider the 2D "gauged" Laplacian ( $N_d = 2$ )

$$\phi(\vec{r}) = \nabla \psi(\vec{r}) = 4\psi(\vec{r}) - \sum_{i=0}^{N_d-1} [u_i^{\wedge}(\vec{r})\psi(\vec{r} + \hat{i}) + u_i^*(\vec{r} - \hat{i})\psi(\vec{r} - \hat{i})]$$

$$u_i^{\wedge}(\vec{r}) \in U(1), \psi(\vec{r}) \in \mathbb{C}$$

- $2 \cdot N_d$  "u"s at each site  $\vec{r}$ , but only  $N_d$  unique "u"s
- Conjugation flips the direction of  $u$ , i.e.  
 $u_{-\hat{i}}(\vec{r}) = u_i^*(\vec{r})$
- $\hat{i}$  unit vector in direction  $i$
- Convention:  $\hat{x} = \hat{1}, \hat{y} = \hat{0}$



# Data layout in stencil codes: Exercise

- The code in the exercise uses Conjugate gradient (CG) to solve

$$\nabla x = b$$

initial guess  $x$

$$r_0 = b - \nabla x$$

$$p = r_0$$

loop until  $\|r_0\|$  small enough

$$\alpha = \frac{r_0^\top r_0}{p^\top \nabla p}$$

$$x = \alpha p + x$$

$$r_1 = -\alpha \nabla p + r_0$$

$$\beta = \frac{r_1^\top r_1}{r_0^\top r_0}$$

$$p = \beta p + r_1$$

$$r_0 = r_1$$

Linear algebra already implemented  
in exercise: `xdotx()`, `xdoty()`,  
`axpy()`, `aypx()`, `xmy()`

```
[koutsov@localhost Lapl]$ ./lapl 128
  0, res = +1.000000e+00
  1, res = +2.546227e-01
  2, res = +8.557546e-02
  3, res = +2.866554e-02
  4, res = +9.553013e-03
  5, res = +3.151271e-03
  6, res = +1.062331e-03
  7, res = +3.664755e-04
  8, res = +1.269510e-04
  9, res = +4.541719e-05
 10, res = +1.590056e-05
 11, res = +5.618028e-06
 12, res = +1.970783e-06
 13, res = +7.037988e-07
 14, res = +2.475690e-07
 15, res = +8.810894e-08
 16, res = +3.164535e-08
 17, res = +1.140112e-08
 18, res = +4.032213e-09
 19, res = +1.443336e-09
 20, res = +5.135969e-10
...
 38, res = +5.274840e-18
 39, res = +1.950025e-18
 40, res = +7.164136e-19
Converged after      40 iterations, res = +1.055927e-13
Time in lapl(): +1.206e-04 sec/call   4.62e+00 Gflop/s   5.43e+00 GB/s
```

# Data layout in stencil codes: Exercise

- Navigate to directory `./Lap1/`
- [`lap1.c`](#) implements a CG algorithm which solves  $\nabla x = b$  for a random  $b$  and gauge  $u$
- The `_TODO_A_` pieces need completion
- This includes the bulk of the Laplacian operation in function `lap1()` at [`lap1.c:146`](#)
- After you're done compile with `make A` and run for varying problem size `L` by running the script `run-A.sh`
- You can then plot the performance with `python3 plot-A.py`

# Data layout in stencil codes: Exercise

## Some implementation details

- Complex fields don't use the C `_Complex` data type

```
typedef struct {
    float phi_re;
    float phi_im;
} field;
```

```
typedef struct {
    float u_re[ND];
    float u_im[ND];
} link;
```

the reason will become apparent in the next part of the exercise

# Data layout in stencil codes: Exercise

## Some implementation details

- Complex fields don't use the C `_Complex` data type

```
typedef struct {
    float phi_re;
    float phi_im;
} field;
```

```
typedef struct {
    float u_re[ND];
    float u_im[ND];
} link;
```

the reason will become apparent in the next part of the exercise

- Periodic boundary conditions are implemented via modulo operations:

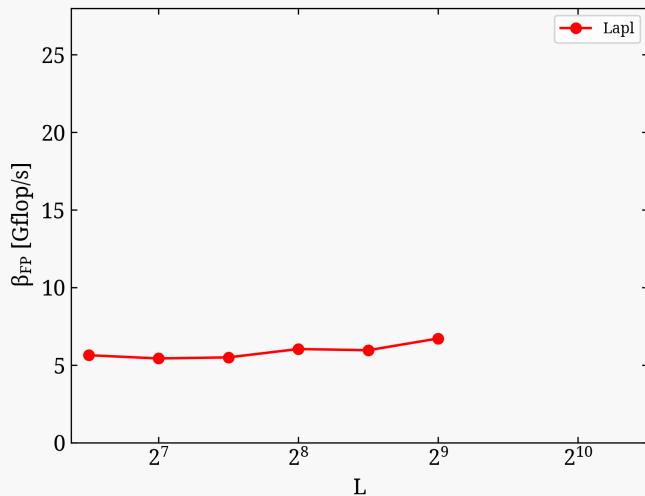
```
int y0 = y*L;
int yp = ((y + 1)%L)*L;
int ym = ((y + L - 1)%L)*L;
```

```
int v0p = (x+1)%L + y0;
int v0m = (L+x-1)%L + y0;
```

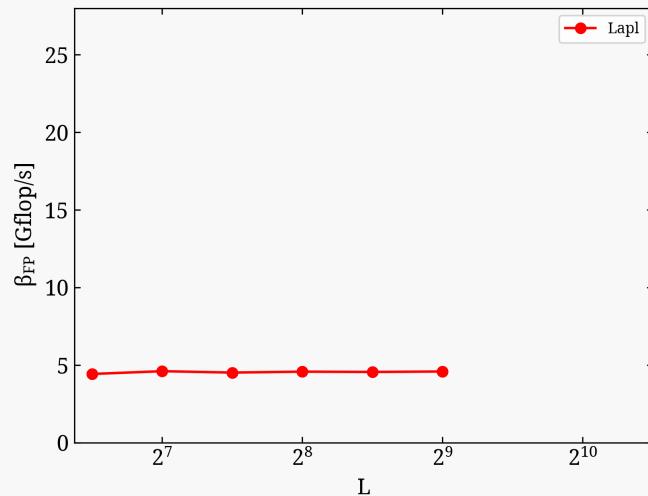
# Data layout in stencil codes: Exercise

## Task A

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)



Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)



# Data layout in stencil codes: Exercise

- Task B involves implementing a version of `lap1()` which is easier for vectorization
- The file is `lap1v.c` and can be compiled with `make B`

# Data layout in stencil codes: Exercise

- Task B involves implementing a version of `lap1()` which is easier for vectorization
- The file is `lap1v.c` and can be compiled with `make B`
- The new data structures reflect the new data layout, with `VL` being the "vectorization length":

```
typedef struct {
    float phi_re[VL];
    float phi_im[VL];
} field;
```

```
typedef struct {
    float u_re[ND][VL];
    float u_im[ND][VL];
} link;
```

# Data layout in stencil codes: Exercise

- Task B involves implementing a version of `lap1()` which is easier for vectorization
- The file is `lap1v.c` and can be compiled with `make B`
- The new data structures reflect the new data layout, with `VL` being the "vectorization length":

```
typedef struct {
    float phi_re[VL];
    float phi_im[VL];
} field;
```

```
typedef struct {
    float u_re[ND][VL];
    float u_im[ND][VL];
} link;
```

- The functions `rand_links()` and `rand_field()`, used to initialize with random numbers, are already setup to initialize the arrays in the **same** order as in `lap1.c`

# Data layout in stencil codes: Exercise

- Task B involves implementing a version of `lap1()` which is easier for vectorization
- The file is `lap1v.c` and can be compiled with `make B`
- The new data structures reflect the new data layout, with `VL` being the "vectorization length":

```
typedef struct {
    float phi_re[VL];
    float phi_im[VL];
} field;
```

```
typedef struct {
    float u_re[ND][VL];
    float u_im[ND][VL];
} link;
```

- The functions `rand_links()` and `rand_field()`, used to initialize with random numbers, are already setup to initialize the arrays in the **same** order as in `lap1.c`
- This allows you to check the code: the residuals for each iteration should be **exactly** the same between `lap1` and `lap1v` if you have completed the code correctly

# Data layout in stencil codes: Exercise

- Complete the code tagged with `_TODO_B_`. Also, transfer from `lap1` the tasks tagged `_TODO_A_`
- The `_TODO_B_` tasks include:
  - Completing the laplacian main kernel, as before, but now using the new data layout
  - Completing the shift operations required on the boundaries

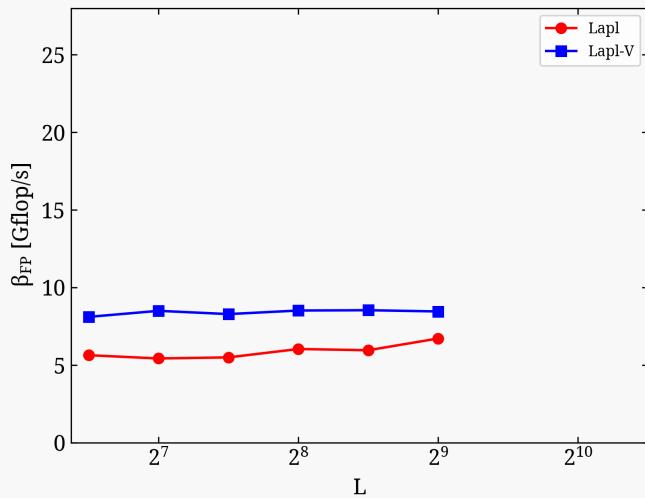
# Data layout in stencil codes: Exercise

- Complete the code tagged with `_TODO_B_`. Also, transfer from `lap1` the tasks tagged `_TODO_A_`
- The `_TODO_B_` tasks include:
  - Completing the laplacian main kernel, as before, but now using the new data layout
  - Completing the shift operations required on the boundaries
- Once done, compile with `make B`, run `run-B.sh` and plot with `plot-B.py`

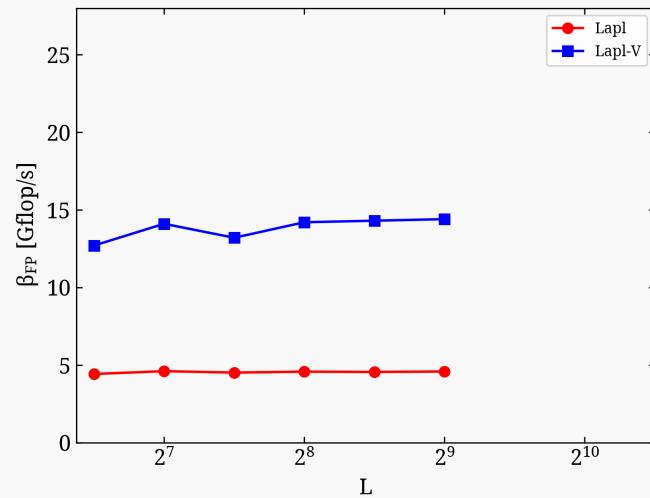
# Data layout in stencil codes: Exercise

## Task B

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)



Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)



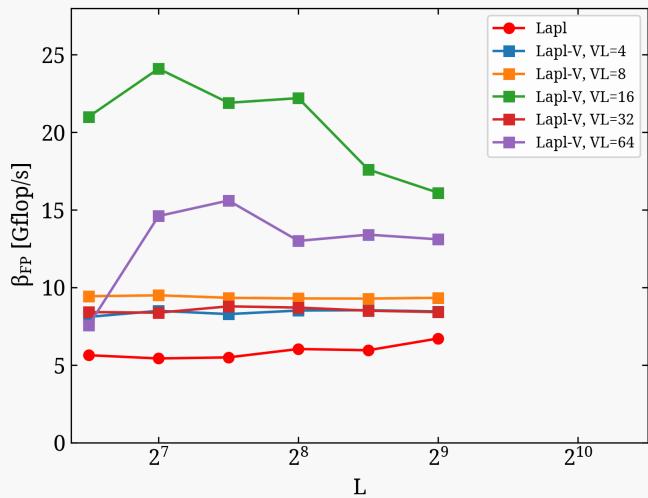
# Data layout in stencil codes: Exercise

- Part C of this exercise only requires you to run `run-C.sh`
- As in `AoS-to-SoA`, this runs `laplv` for various values of `VL`
- With `plot-C.py` you can plot and compare with `lapl`

# Data layout in stencil codes: Exercise

## Task C

Intel Core i5-8259U @ 2.3 GHz  
(this 4-core laptop)



Intel Xeon E5-2650 0 @ 2 GHz  
(8 cores, cluster node)

