Lab 3: Caching Optimizations I

After this lab, you'll be able to…

- Understand the basic training algorithm our CNN framework uses
- Apply loop re-ordering and blocking two functions.
- Optimize training of a simple neural network.
- Measure the impact of your optimizations using using performance counters.

Lab Outline: Three Stages

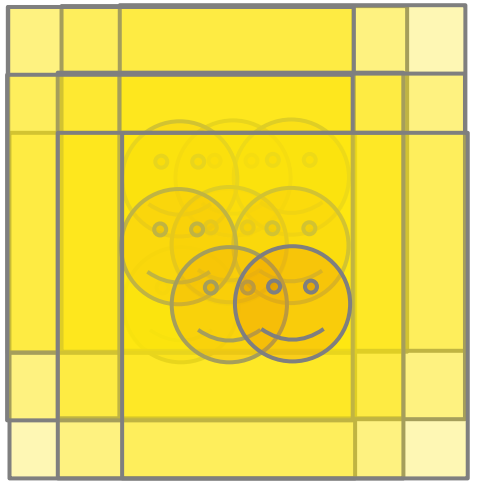
1. Follow instructions to optimize the function. (75%)
    1. Detailed instructions for you to follow (everything but writing the code).
    2. Clear expectations for results
2. Apply the same optimizations to a separate function on your own. (10%)
    1. High-level instructions for your to follow
    2. Clear expectations for results
3. Apply further optimizations (15%)
    1. Open-ended optimizations across several functions
    2. Open-ended results.

Lecture & Discussion Section Plan

- In this lecture
  - Discussion about loop reordering and loop blocking
  - Illustration of these optimizations on a different code base.
  - High level description of neural network training algorithm
- In the discussion section
  - Detailed dissection of the function you'll be optimizing
  - Walk through the non-coding parts of the lab

Example: Video Stabilization

- Video stabilization remove shaking from videos.
- Simple algorithm
  - For each pair of consecutive frames
    - ”slide” the second frame around to find the best alignment with the first frame.
    - Compute the “sum of absolute differences” between the two images at each position.
    - The position that minimizes the sum is the best match.
    - Adjust the second image by that much.
- (real image stabilization is much more complex)



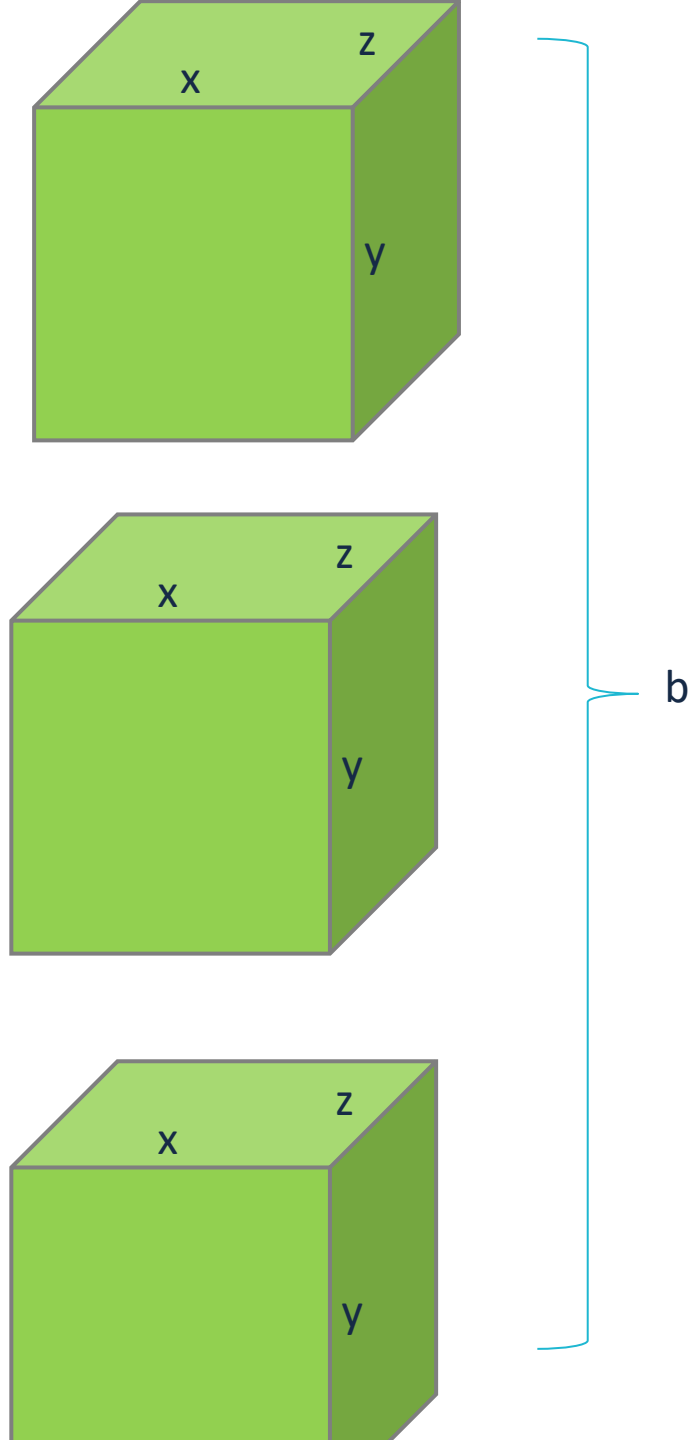
X-offset

Y-offset

1	1	1
1	0.1	1
1	1	1

Storing Images in Tensors

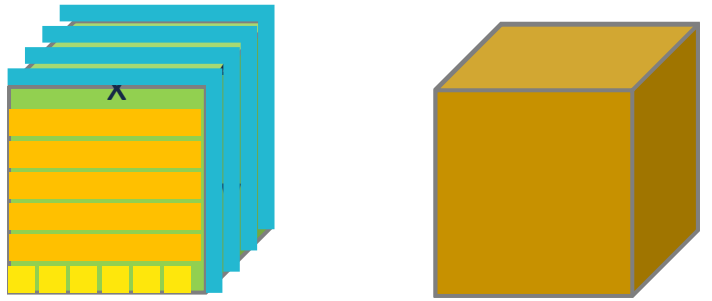
- Our tensors are 4D arrays of doubles (earlier, I said 3D, but we added a D)
  - x – horizontal dimension
  - y – vertical dimension
  - z – color planes (RGD or just gray)
  - b – images (we store a ‘batch’ of images in one tensor)



Tensor Layout

```
index = b * (size.x * size.y * size.z) +
        z * (size.x * size.y) +
        y * (size.x) +
        x;
```

- Internally, tensors are stored a linear array
  - size = x*y*z*b
- Here's the code to translate coordinates to a linear index (tensor_t.hpp)
  - Increment x moves the index by 1
  - Incrementing y moves by size.x
  - Incrementing z moves by size.x*size.y
  - Incrementing b moves by size.x*size.y*size.z



This is the layout of a tensor containing 2 (b = 2), 4(x=4) x 4(y=4), RGB (z=3) images.

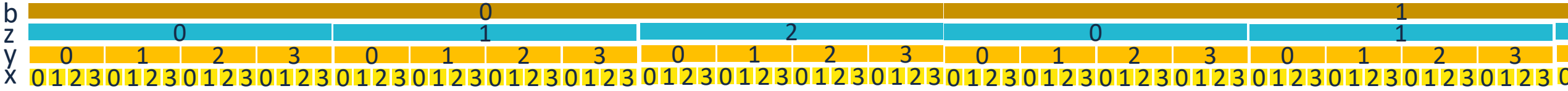
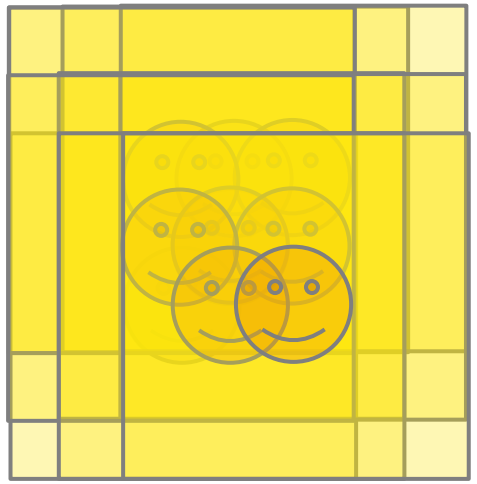


Image Alignment

- We are going to do image alignment on a batch of images (i.e., frames of video)
  - Images are 228x228 (x = y = 228) gray scale (z = 1)
  - For frame b, we will shift around frame b - 1
  - Compute the sum of absolute differences for each offset.
    - Offset will be 0-7 for and 0-7 for y
  - Store the result in an output tensor
    - X = 8 (offsets)
    - Y = 8 (offsets)
    - Z = 1
    - B = B (there will be one extra output)



X-offset

Y-offset			
	1	1	1
	1	0.1	1
	1	1	1

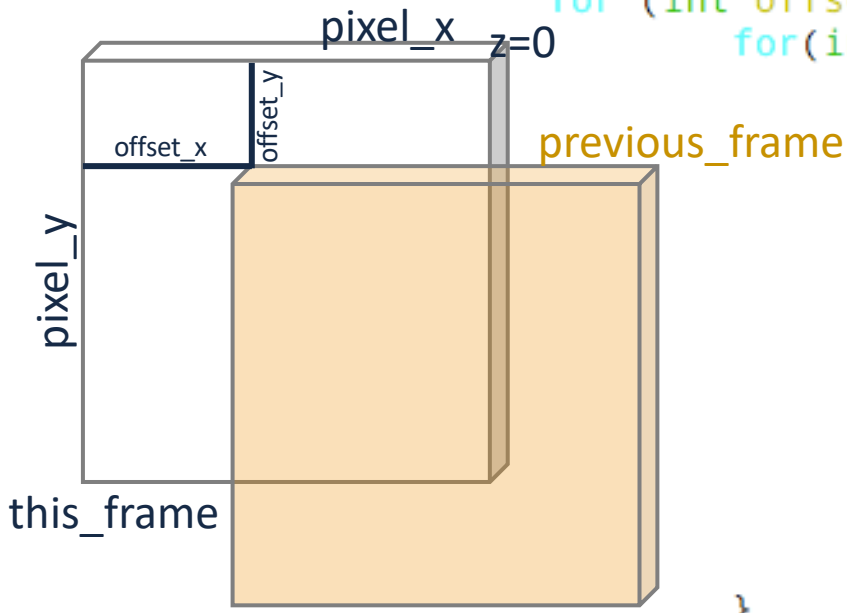
Let’s Look at The Code

- In your starter repo under ‘example’
- Take a look.

Let’s Look at The Code

- In your starter repo under ‘example’
- Take a look.

```
void do_stabilize_baseline(const tensor_t<double> & images, tensor_t<double> & output)
{

        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;
                for (int offset_x = 0; offset_x < 8; offset_x++)  {
                        for (int offset_y = 0; offset_y < 8; offset_y++)  {
                                for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {
                                        for(int pixel_y = 0; pixel_y < images.size.y; pixel_y++) {

                                                int shifted_x = pixel_x + offset_x;
                                                int shifted_y = pixel_y + offset_y;

                                                if (shifted_x > images.size.x ||
                                                    shifted_y > images.size.y)
                                                        continue;
                                                output(offset_x, offset_y, 0, this_frame) +=
                                                        fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                             images(shifted_x, shifted_y, 0, previous_frame));
                                        }
                                }
                        }
                }
        }

}
```

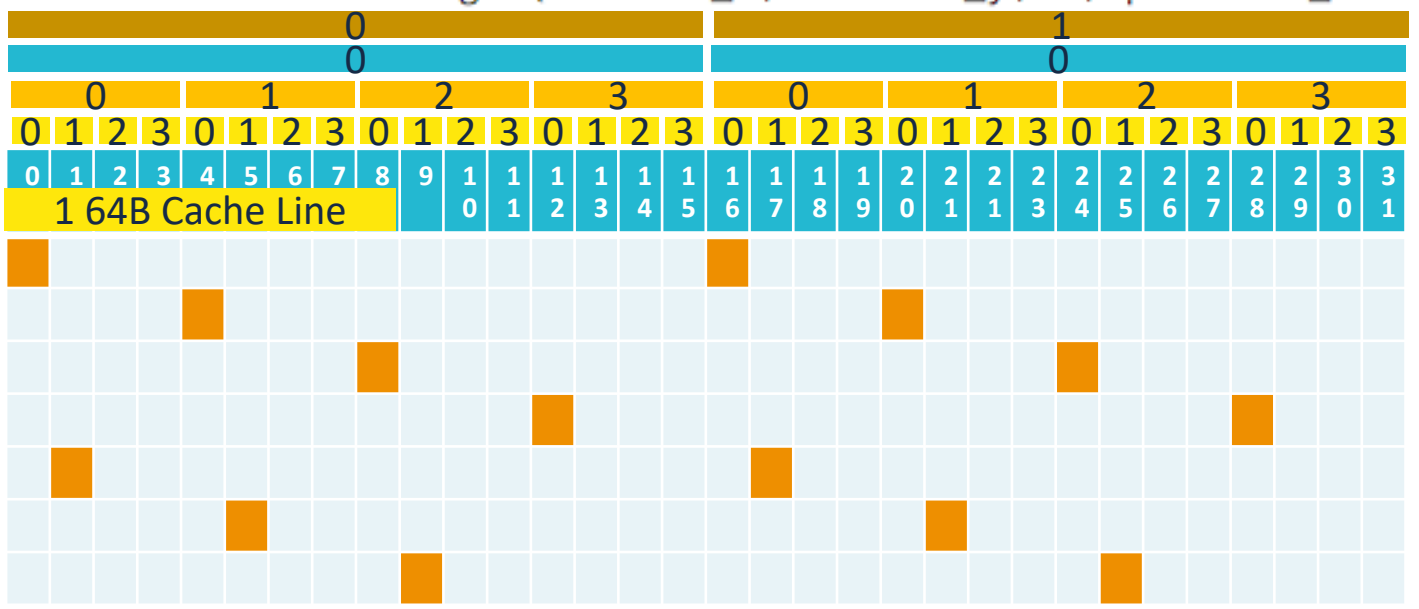


- Initial loop order (5 loops)
  - this_frame, offset_x, offset_y, pixel_x, pixel_y,

```
void do_stabilize_baseline(const tensor_t<double> & images, tensor_t<double> & output)
{

        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;
                for (int offset_x = 0; offset_x < 8; offset_x++)  {
                        for (int offset_y = 0; offset_y < 8; offset_y++)  {
                                for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {
                                        for(int pixel_y = 0; pixel_y < images.size.y; pixel_y++) {

                                                int shifted_x = pixel_x + offset_x;
                                                int shifted_y = pixel_y + offset_y;

                                                if (shifted_x > images.size.x ||
                                                    shifted_y > images.size.y)
                                                        continue;
                                                output(offset_x, offset_y, 0, this_frame) +=
                                                        fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                             images(shifted_x, shifted_y, 0, previous_frame));
                                        }
                                }
                        }
                }
        }
}
```

```
index = b * (size.x * size.y * size.z) +
        z * (size.x * size.y) +
        y * (size.x) +
        x;
```

Frame	Prev_frame	Offset_x	Offset_y	Pixel_y	Pixel_x
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	0	2	0
1	0	0	0	3	0
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	0	2	1



Loop Reordering

```
void do_stabilize_baseline(const tensor_t<double> & images, tensor_t<double> & output)
{

        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;
                for (int offset_x = 0; offset_x < 8; offset_x++)  {
                        for (int offset_y = 0; offset_y < 8; offset_y++)  {
                                for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {
                                        for(int pixel_y = 0; pixel_y < images.size.y; pixel_y++) {

                                                int shifted_x = pixel_x + offset_x;
                                                int shifted_y = pixel_y + offset_y;

                                                if (shifted_x > images.size.x ||
                                                    shifted_y > images.size.y)
                                                        continue;
                                                output(offset_x, offset_y, 0, this_frame) +=
                                                        fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                             images(shifted_x, shifted_y, 0, previous_frame));
                                        }
                                }
                        }
                }
        }
}
```

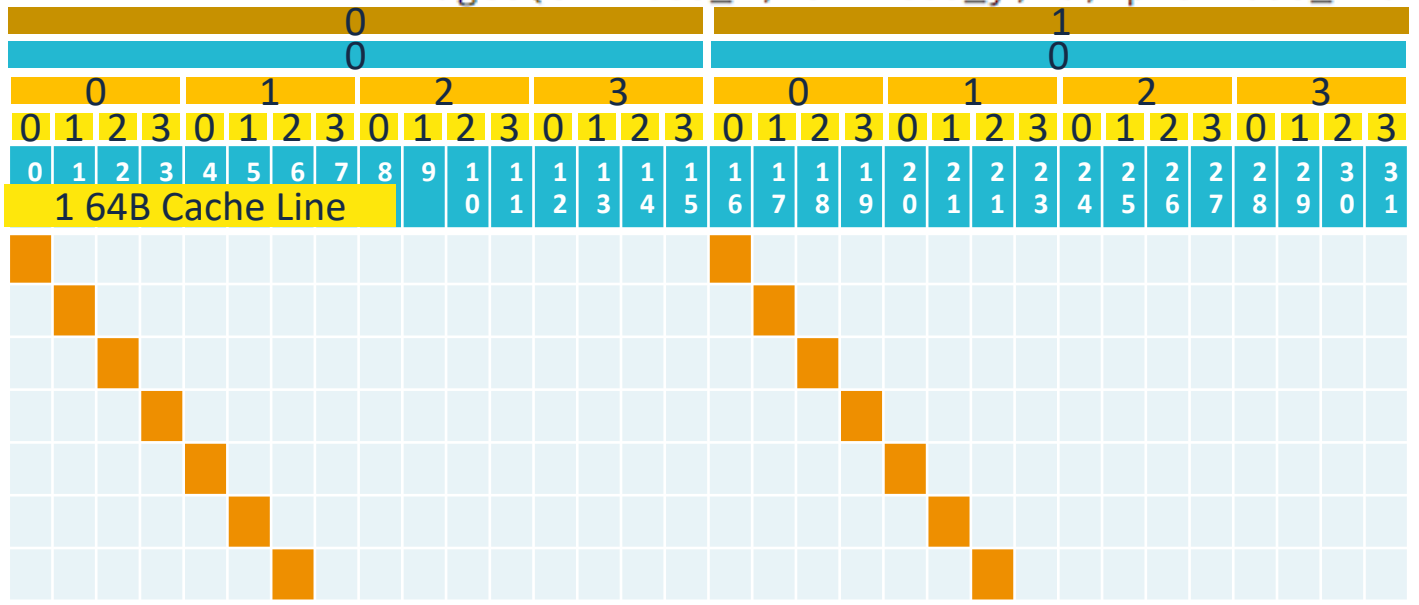
- Incrementing y in the innermost loop leads to poor spatial locality.
- Since the loops are independent, we can ”re order” or “renest” them.

```
void do_stabilize_reorder_pixelxy(const tensor_t<double> & images, tensor_t<double> & output)
{
        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;
                for (int offset_x = 0; offset_x < 8; offset_x++)  {
                        for (int offset_y = 0; offset_y < 8; offset_y++)  {
                                for(int pixel_y = 0; pixel_y < images.size.y; pixel_y++) {
                                        for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {

                                                int shifted_x = pixel_x + offset_x;
                                                int shifted_y = pixel_y + offset_y;

                                                if (shifted_x > images.size.x ||
                                                    shifted_y > images.size.y)
                                                        continue;

                                                output(offset_x, offset_y, 0, this_frame) +=
                                                        fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                             images(shifted_x, shifted_y, 0, previous_frame));
                                        }
                                }
                        }
                }
        }
}
```

```
index = b * (size.x * size.y * size.z) +
        z * (size.x * size.y) +
        y * (size.x) +
        x;
```

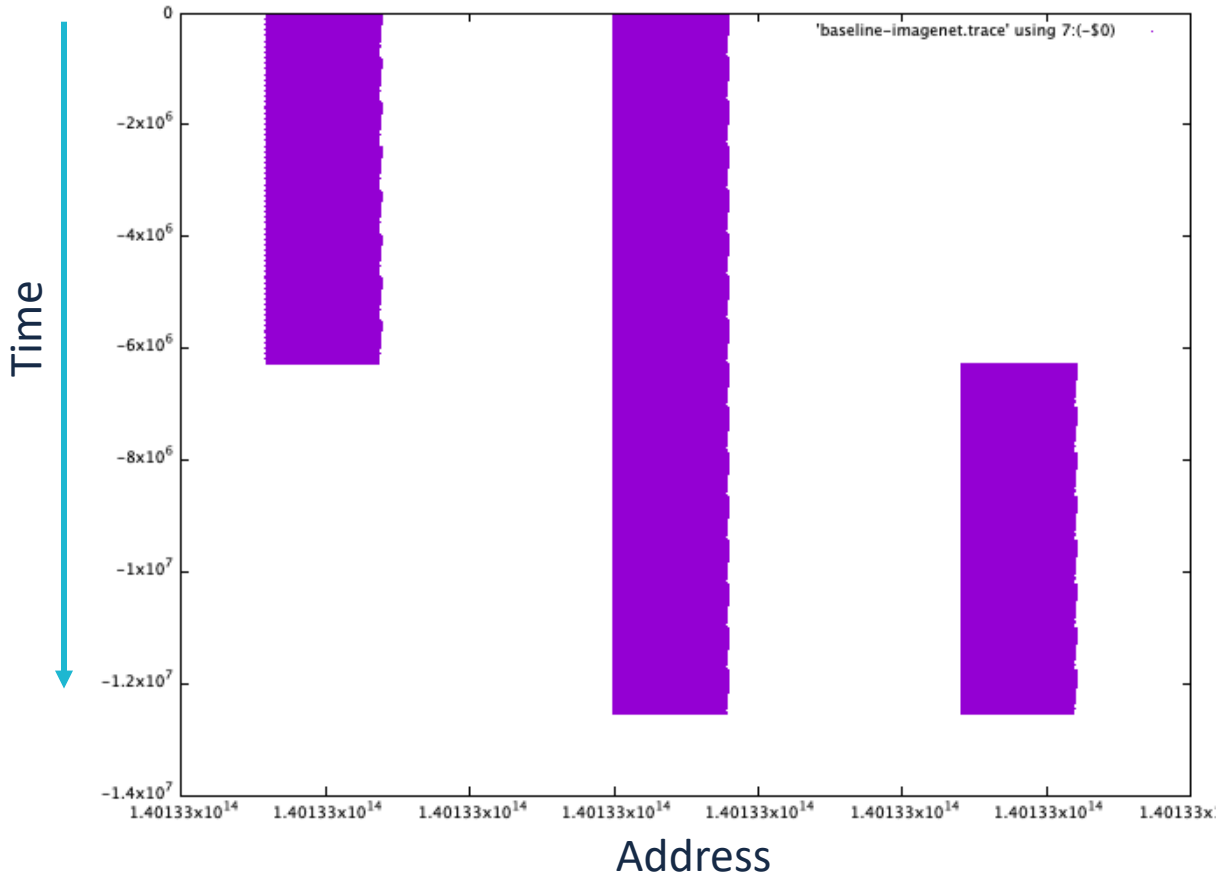
frame
z
pixel_y
pixel_x

Frame	Prev_frame	Offset_x	Offset_y	Pixel_y	Pixel_x
1	0	0	0	0	0
1	0	0	0	0	1
1	0	0	0	0	2
1	0	0	0	0	3
1	0	0	0	1	0
1	0	0	0	1	1
1	0	0	0	1	2



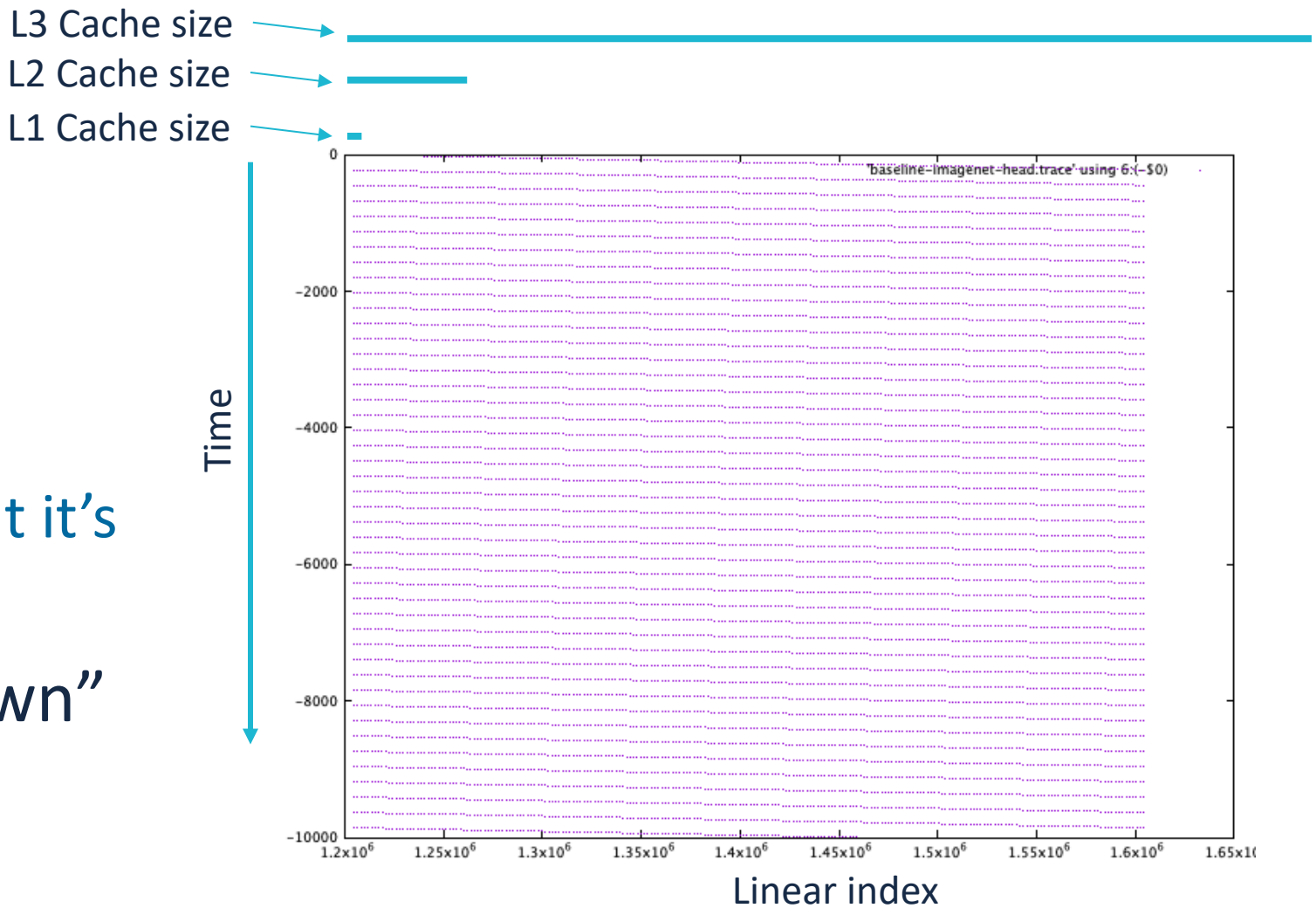
Full Size data (228x228x1x3) All accesses

- Top half:
  - Frame = 1
  - Previous_frame = 0
- Bottom half
  - Frame = 2
  - Previous_frame = 1

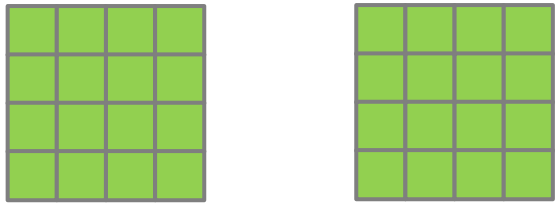


Full Size data (228x228x1x100): first 10000 accesses; (current frame only)

- One frame only.
- No spatial locality
- To temporal locality
  - We each cache line, but it's was evicted long ago.
- The cache is "blown own"
- Execution time: 2.22s

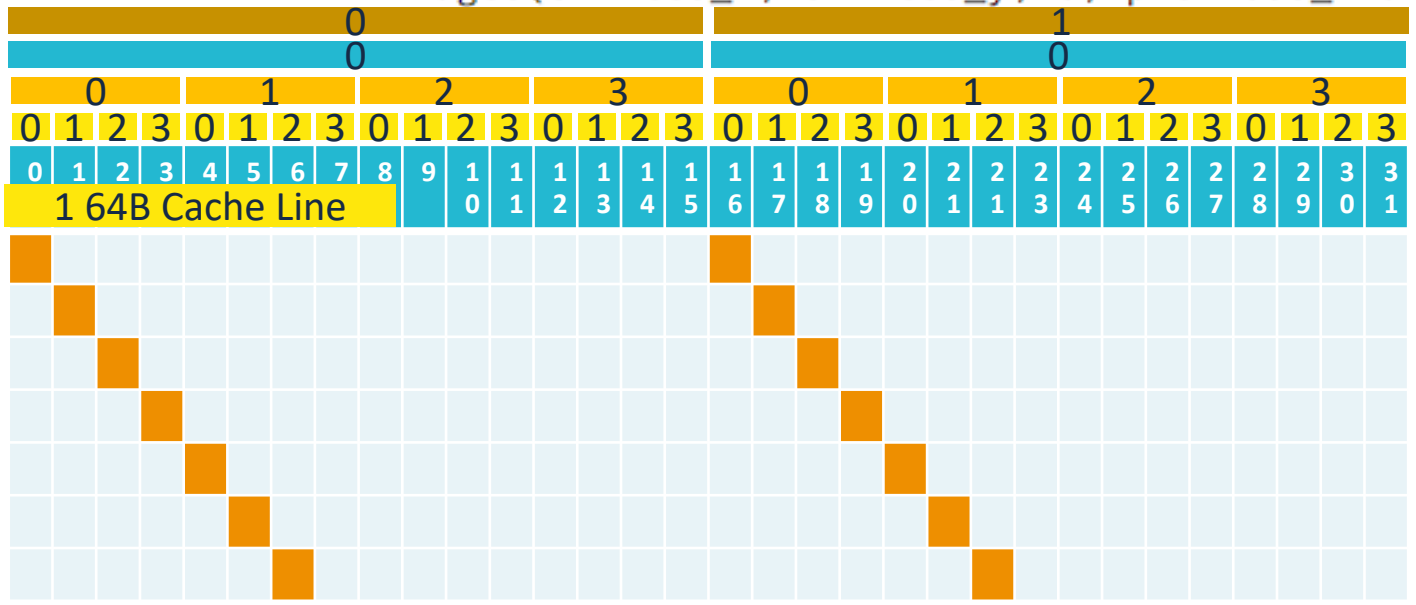


```
void do_stabilize_reorder_pixelxy(const tensor_t<double> & images, tensor_t<double> & output)
{
        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;
                for (int offset_x = 0; offset_x < 8; offset_x++)  {
                        for (int offset_y = 0; offset_y < 8; offset_y++)  {
                                for(int pixel_y = 0; pixel_y < images.size.y; pixel_y++) {
                                        for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {

                                                int shifted_x = pixel_x + offset_x;
                                                int shifted_y = pixel_y + offset_y;

                                                if (shifted_x > images.size.x ||
                                                    shifted_y > images.size.y)
                                                        continue;

                                                output(offset_x, offset_y, 0, this_frame) +=
                                                        fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                             images(shifted_x, shifted_y, 0, previous_frame));
                                }
                        }
                }
        }
}
```



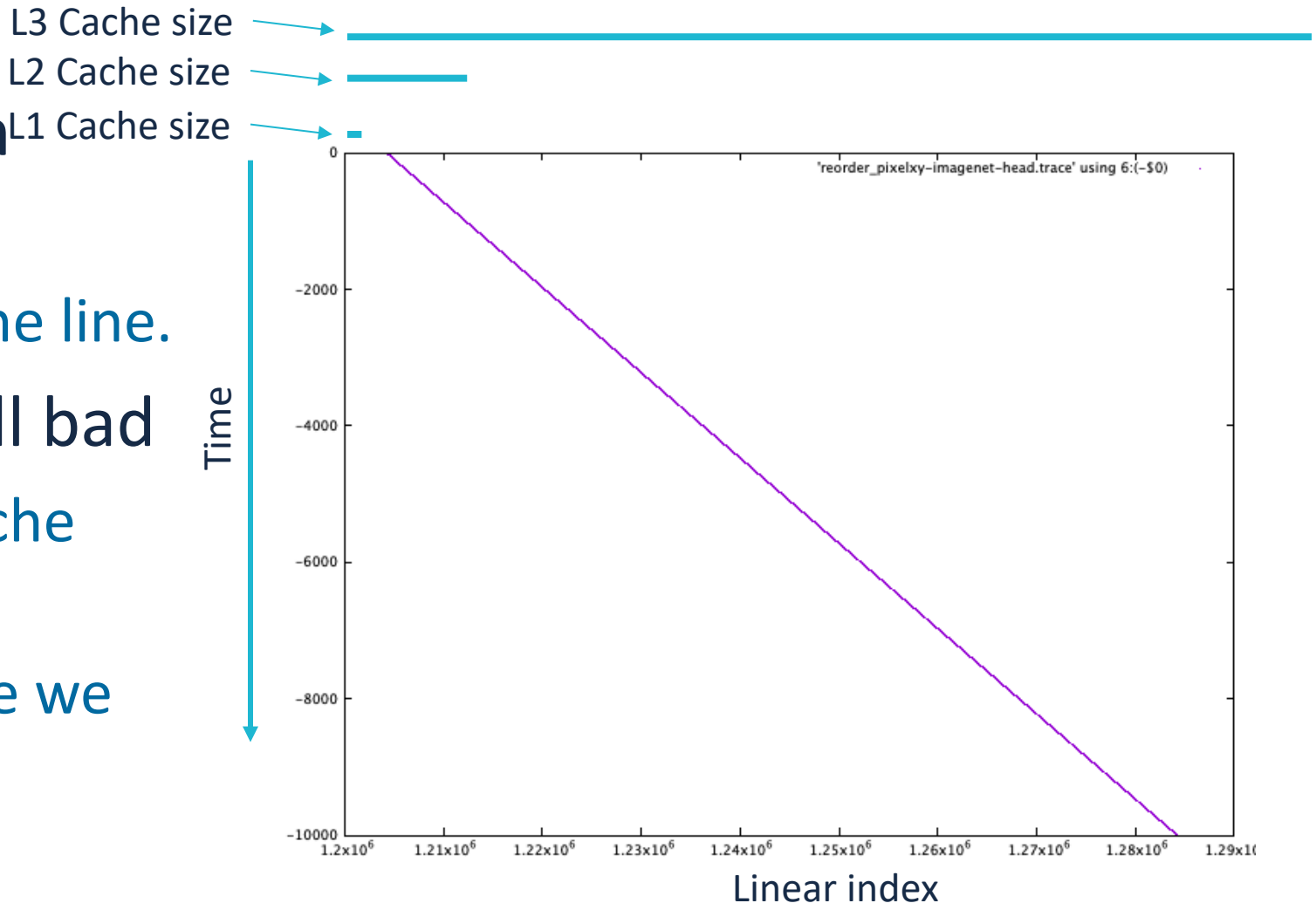
frame
z
pixel_y
pixel_x

Frame	Prev_frame	Offset_x	Offset_y	Pixel_y	Pixel_x
1	0	0	0	0	0
1	0	0	0	0	1
1	0	0	0	0	2
1	0	0	0	0	3
1	0	0	0	1	0
1	0	0	0	1	1
1	0	0	0	1	2



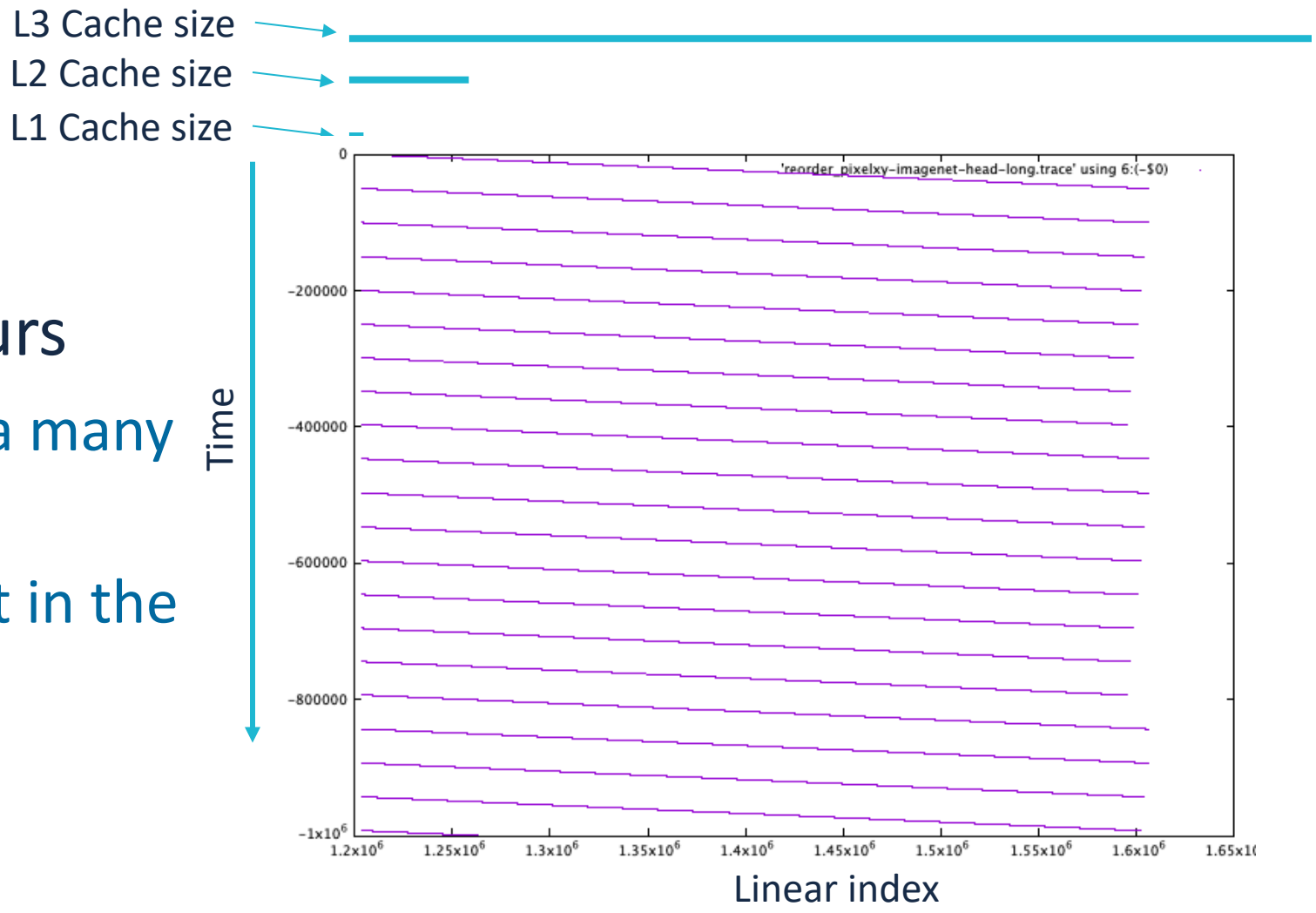
Full Size data (228x228x1x100). Reordered pixel loops; first 10000 accesses; (current frame only)

- Spatial locality is much better
  - We use all of each cache line.
- Temporal locality is still bad
  - We don't *reuse* any cache lines
  - They are evicted before we have the chance.
- Execution time: 2.05s



Full Size data (228x228x1x100). Reordered pixel loops; 100000 accesses; (current frame only)

- Reuse of *memory* occurs
  - We read the same data many times
  - We just need to keep it in the cache



Cache "Tiling" Optimization: Two Steps

- Step 1: Break one loop into two nested loops
  - The outer loop takes big steps from one "tile" to another
  - The inner loop takes little steps
- Step 2:
  - Reorder the loops so that the code works one tile at a time.

Step 1: Split loop

```
#define TILE_SIZE 4
void do_stabilize_pretile_y(const tensor_t<double> & images, tensor_t<double> & output)
{
        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;

                for(int pixel_yy = 0; pixel_yy < images.size.y; pixel_yy += TILE_SIZE) {
                        for(int pixel_y = pixel_yy; pixel_y < pixel_yy + TILE_SIZE && pixel_y < images.size.y; pixel_y++) {
                                for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {

                                        for (int offset_x = 0; offset_x < 8; offset_x++)  {
                                                for (int offset_y = 0; offset_y < 8; offset_y++)  {


                                                        int shifted_x = pixel_x + offset_x;
                                                        int shifted_y = pixel_y + offset_y;

                                                        if (shifted_x > images.size.x ||
                                                            shifted_y > images.size.y)
                                                                continue;
                                                        output(offset_x, offset_y, 0, this_frame) +=
                                                                fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                                     images(shifted_x, shifted_y, 0, previous_frame));

                                                }
                                        }
                                }
                        }
                }
        }
}
```

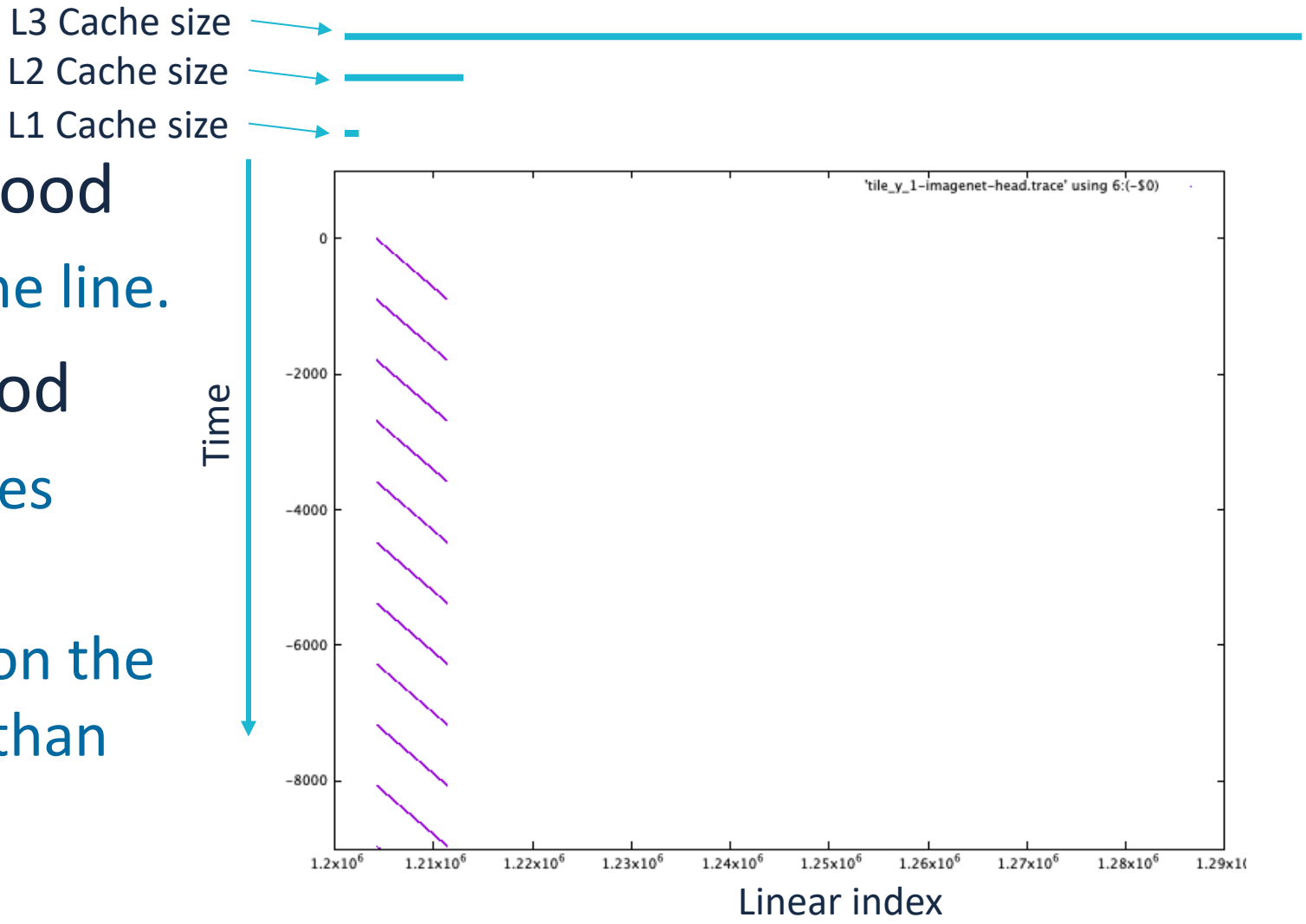
- pixel_yy moves from one TILE_SIZE tile to another
  - Loop bound checks it against image.size.y
- pixel_y counts within one tile.
  - Loop both checks against TILE_SIZE *and* image.size.y
  - This handles the case where TILE_SIZE does not evenly divide image.size.y
- Nothing has really changed yet. The access stream is identical.

Step 1: Reorder loops

```
void do_stabilize_tile_y_1(const tensor_t<double> & images, tensor_t<double> & output)
{
        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;

                for(int pixel_yy = 0; pixel_yy < images.size.y; pixel_yy +=  TILE_SIZE) {

                        for (int offset_x = 0; offset_x < 8; offset_x++)  {
                                for (int offset_y = 0; offset_y < 8; offset_y++)  {

                                        for(int pixel_y = pixel_yy; pixel_y < pixel_yy + TILE_SIZE && pixel_y < images.size.y; pixel_y++) {
                                                for(int pixel_x = 0; pixel_x < images.size.x; pixel_x++) {

                                                        int shifted_x = pixel_x + offset_x;
                                                        int shifted_y = pixel_y + offset_y;

                                                        if (shifted_x > images.size.x ||
                                                            shifted_y > images.size.y)
                                                                continue;
                                                        output(offset_x, offset_y, 0, this_frame) +=
                                                                fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                                     images(shifted_x, shifted_y, 0, previous_frame));
                                                }
                                        }
                                }
                        }
                }
        }
}
```

- The loop is now split
- We will work on one tile at a time.

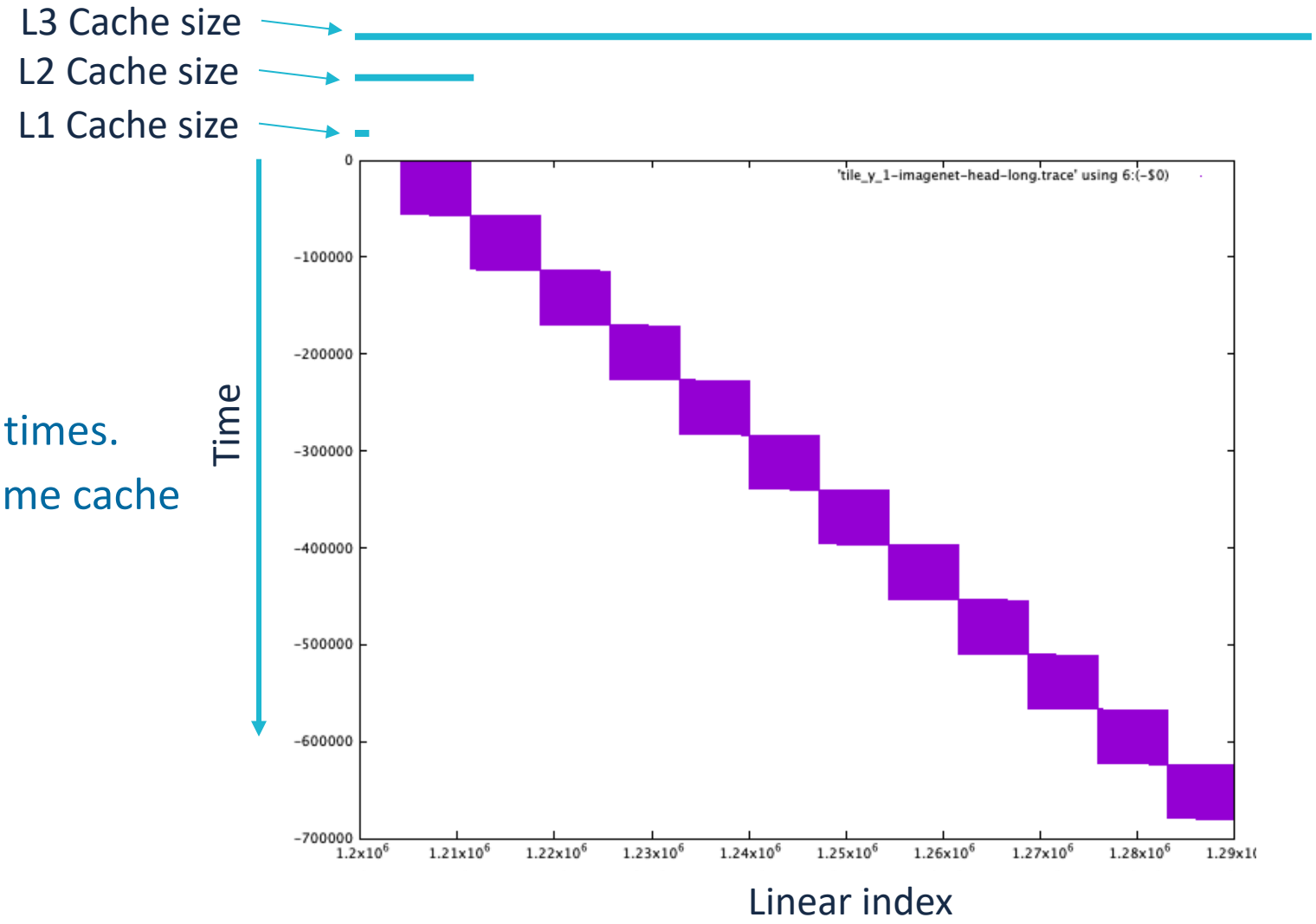
Full Size data (228x228x1x100). Reordered pixel loops; first 10000 accesses (current frame only)



- Spatial locality is still good
  - We use all of each cache line.
- Temporal locality is good
  - We reuse the cache lines many times.
  - In fact, we never miss on the same cache line more than once per frame.

Full Size data (228x228x1x100). Reordered pixel loops; first 100000 accesses

- Spatial locality is still good
  - We use all of each cache line.
- Temporal locality is good
  - We reuse the cache lines many times.
  - In fact, we never miss on the same cache line more than once per frame.
- Execution time: 1.98s
- Removed 97% of L2 misses.
- Removes 99.4% of L1 misses



Common Problems and Tasks

- Want to Measuring performance?
  - Submit to gradescope
  - Use –run-git-remotely if feeling brave
  - Performance on ieng6 is meaningless.
- Need to debugging?
  - Use ieng6
- Regressions don’t pass
  - Debug on ieng6
- Gradescope times out
  - Does it complete eventually on ieng6?
  - Then your code is too slow.
  - Are you printing anything to cout?
- Speedup is not high enough on Tier 1 or 2
  - You probably didn’t implement the optimizations correctly.
  - Check example code in ‘example’
  - Debug on your own or with TA
- Speedup not high enough on Tier 3
  - Think carefully about memory access patterns.
  - Checkout tracing code at the top of opt_cnn.hpp
- Regressions pass locally but not on gradescope
  - Did you commit? Push?
  - Are you sure?
  - Post on piazza with links to gradescope submission.
- --run-git-remotely is behaving strangely
  - Post to piazza. Use gradescope instead.
- Canary keeps failing
  - Post on piazza with link to gradescope submissions

CSE141L Lab 4
Discussion Slides

Overview

- Old to new coordinate mapping in fc_layer_t::activate
- Working of fc_layer_t::activate
- Loop Reordering
- fc_layer_t::activate after loop reordering
- Cache Tiling
- fc_layer_t::activate after Cache Tiling
- Brief overview of fc_layer_t::calc_grads

fc_layer_t::activate - Mapping old coordinates to new coordinates

```
copy_input(in);
tdsize old_size = in.size;
tdsize old_out_size = out.size;
// cast to correct shape
in.size.x = old_size.x * old_size.y * old_size.z;
in.size.y = old_size.b;
in.size.z = 1;
in.size.b = 1;
```

```
for ( int b = 0; b < in.size.y; b++ ) {
    for ( int i = 0; i < in.size.x; i++ ) {
        for ( int n = 0; n < out.size.x; n++ ) {
            double in_val = in(i, b, 0);
            double weight_val = weights( i, n, 0 );
            double mul_val = in_val * weight_val;
            double acc_val = activator_input(n, 0, 0, b) + mul_val;
            activator_input(n, 0, 0, b) = acc_val;
        }
    }
}
```

```
index = b * (size.x * size.y * size.z) +
        z * (size.x * size.y) +
        y * (size.x) +
        x;
```

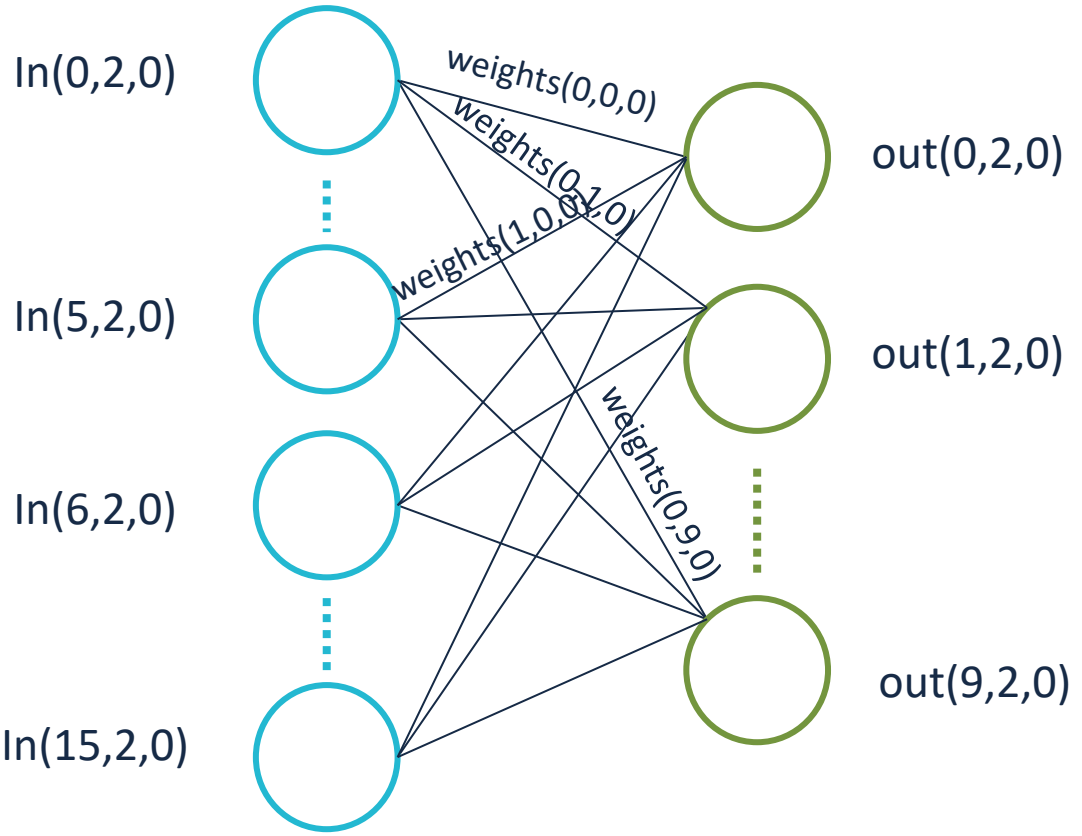
x y

```
in_val = in(i,b,0)
```

```
new index = 0 +
            0 +
            b * (old_size.x * old_size.y *
            old_size.z)+
            i
```

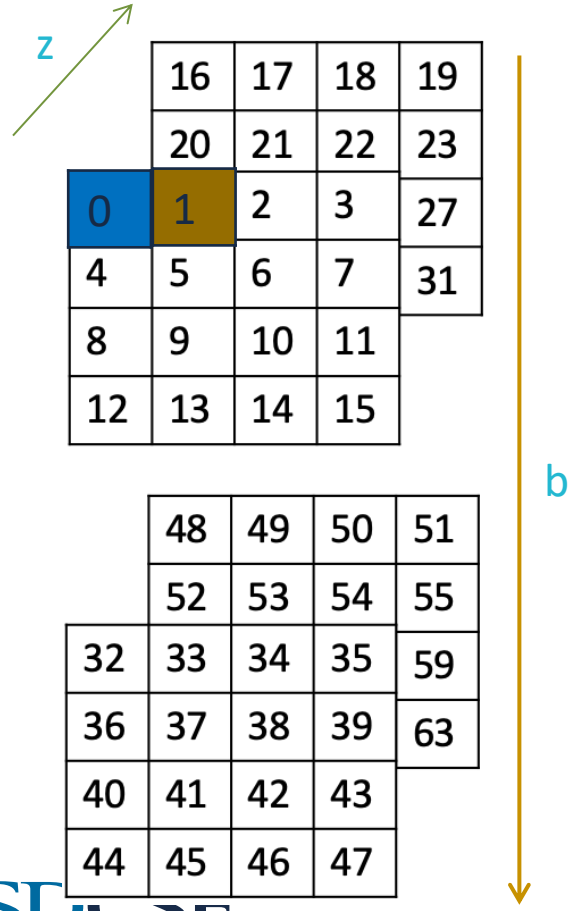
Working of fc_layer_t::activate

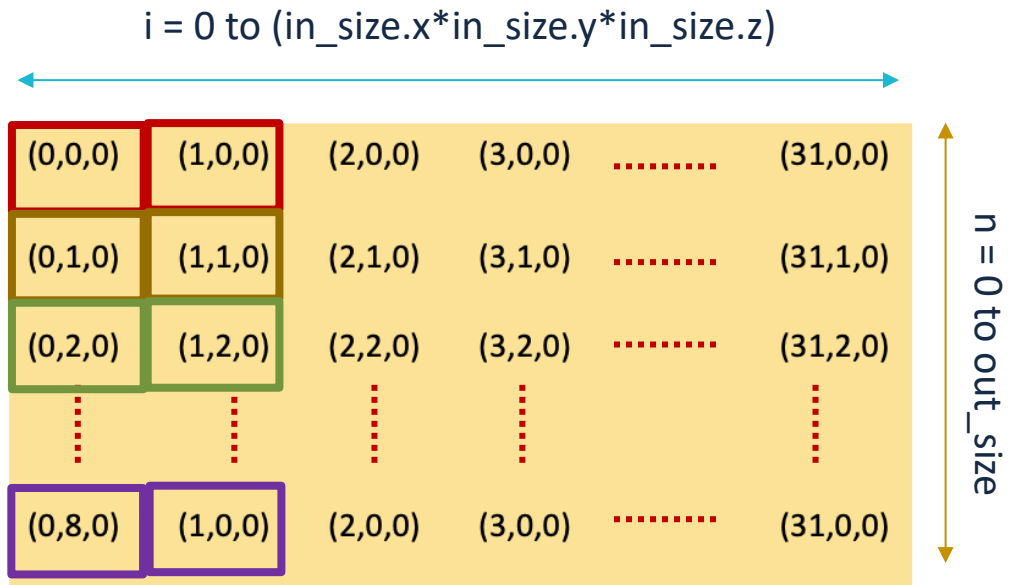
```
for ( int b = 0; b < in.size.y; b++ ) {
    for ( int i = 0; i < in.size.x; i++ ) {
        for ( int n = 0; n < out.size.x; n++ ) {
            double in_val = in(i, b, 0);
            double weight_val = weights( i, n, 0 );
            double mul_val = in_val * weight_val;
            double acc_val = activator_input(n, 0, 0, b) + mul_val;
            activator_input(n, 0, 0, b) = acc_val;
        }
    }
}
// finally, apply the activator function.
for ( unsigned int n = 0; n < activator_input.element_count(); n++ ) {
    out.data[n] = activator_function( activator_input.data[n] );
}
```

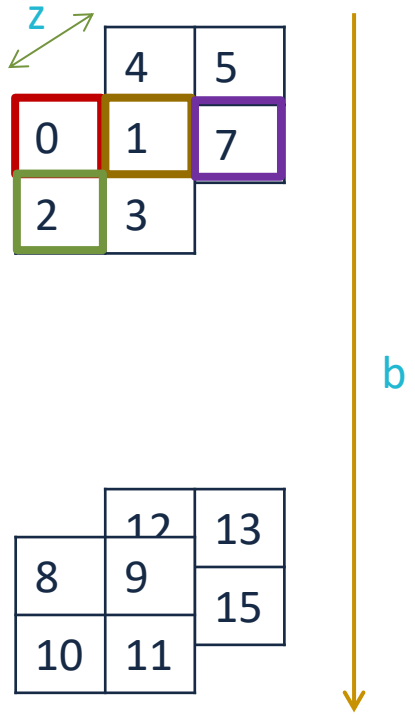


- Loop b iterates through batches of the input/output
- Loop i iterates through all elements of linearized input within a batch
- Loop n iterates through all elements of linearized output within a batch
- Input and weight product for the corresponding output n in batch b accumulates in activator_input(n,0,0,b)

Working of fc_layer_t::activate



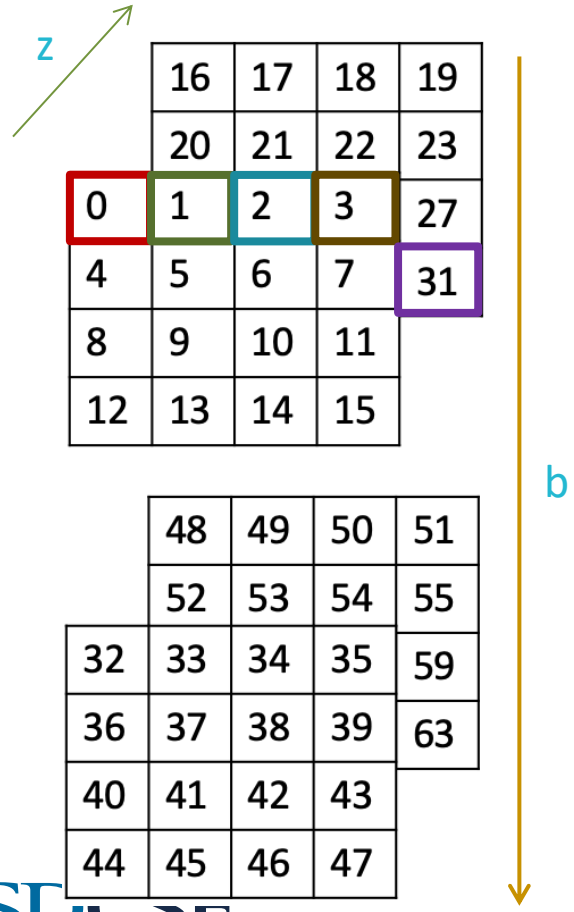


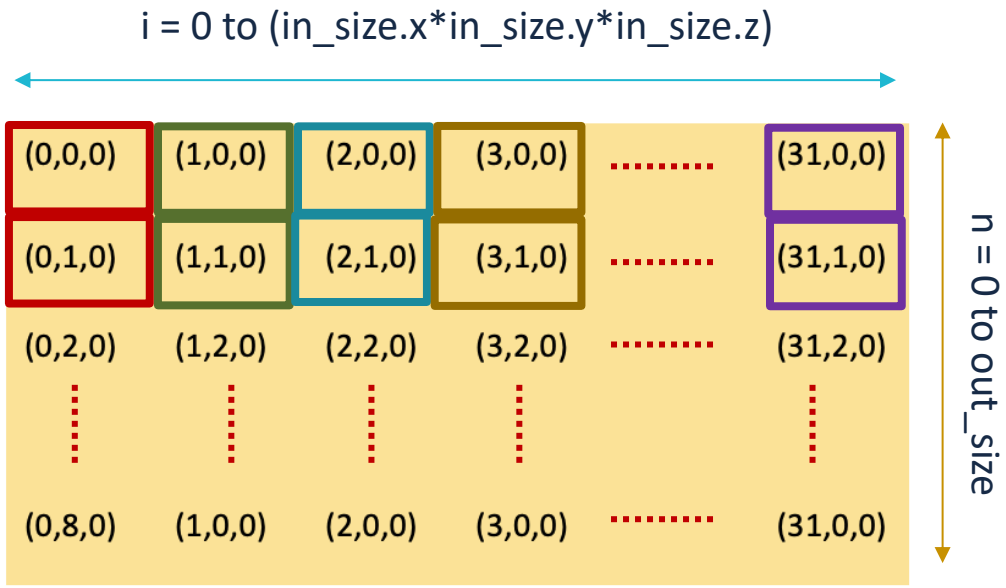


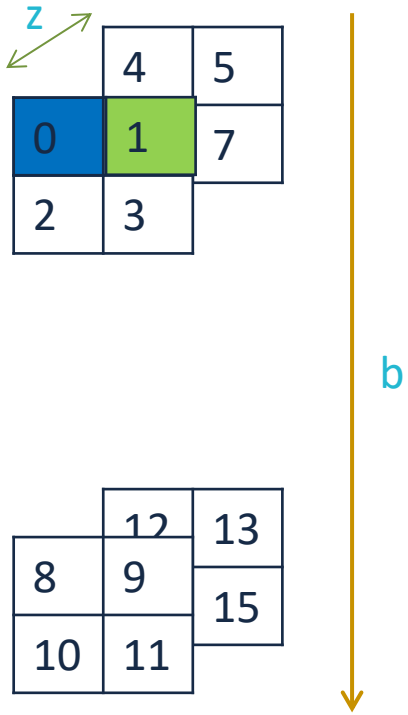
Loop Reordering

- Reordering loops which are independent doesn’t affect overall functionality
- Loop order can be changed to create access patterns that can enhance spatial locality
- Look at the example code mentioned in lecture slides for loop reordering

fc_layer_t::activate after loop reordering







Cache Tiling

- In the source code, we iterate through all n (outputs) for each input i
- Input is streamed from memory each time. Assuming in.size.x is large, we would have no reuse in cache

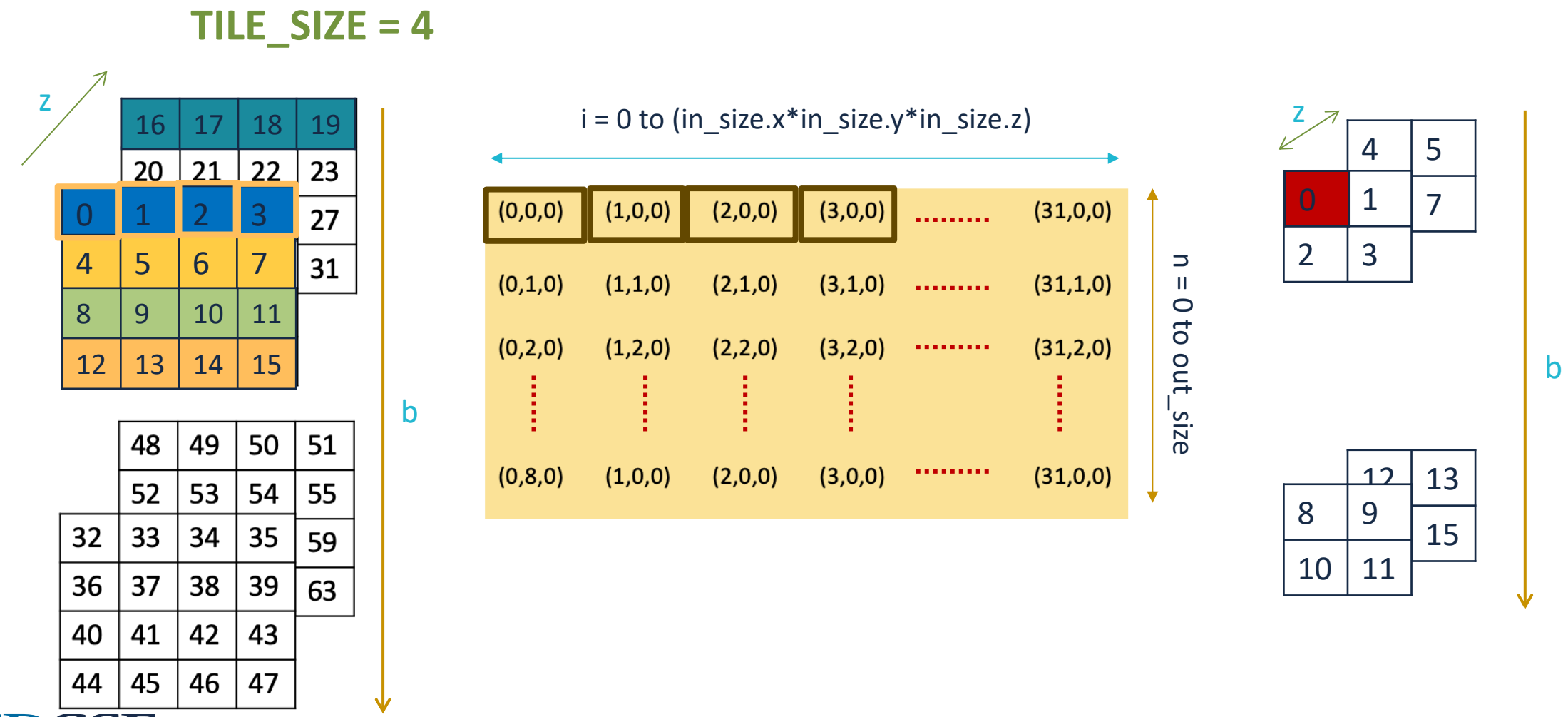
- The blocked code reuses a set of TILE_SIZE input values across multiple iterations of the loop
- If TILE_SIZE is chosen such that this set of values fits in cache, memory traffic is brought down by a factor of TILE

- Look at the Cache tiling optimization example given in Lecture slides

Source code

```
for ( int b = 0; b < in.size.y; b++ ) {
    for ( int i = 0; i < in.size.x; i++ ) {
        for ( int n = 0; n < out.size.x; n++ ) {
            double in_val = in(i, b, 0);
            double weight_val = weights( i, n, 0 );
            double mul_val = in_val * weight_val;
            double acc_val = activator_input(n, 0, 0, b) + mul_val;
            activator_input(n, 0, 0, b) = acc_val;
        }
    }
}
```

fc_layer_t::activate after Cache Tiling



fc_layer_t::calc_grads

```
void calc_grads( const tensor_t<double>& grad_next_layer ) {
    memset( grads_out.data, 0, grads_out.size.x * grads_out.size.y * grads_out.size.z * sizeof( double ) );
    grads_out.size.x = grads_out.size.x * grads_out.size.y * grads_out.size.z;
    grads_out.size.y = 1;
    grads_out.size.z = 1;
    for ( int b = 0; b < out.size.b; b++ ) {
        for ( int n = 0; n < activator_input.size.x; n++ ){
            double ad = activator_derivative( activator_input(n, 0, 0, b) );
            double ng = grad_next_layer(n, 0, 0, b);
            act_grad(n, 0, 0, b) = ad * ng;
        }
    }
    for ( int b = 0; b < out.size.b; b++ ) {
        for ( int n = 0; n < weights.size.y; n++ ) {
            for ( int i = 0; i < weights.size.x; i++ ) {
                    grads_out(i, 0, 0, b) += act_grad(n, 0, 0, b) * weights( i, n, 0);
            }
        }
    }
    grads_out.size = in.size;
}
```

- First, the activator_derivative of an output n in layer ‘i’ is calculated
- We backpropagate the error from layer ‘i+1’
- Finally, we calculate how much each input contributed to the error which is proportional to it’s

Questions