Lab 4: Multicore Parallelism

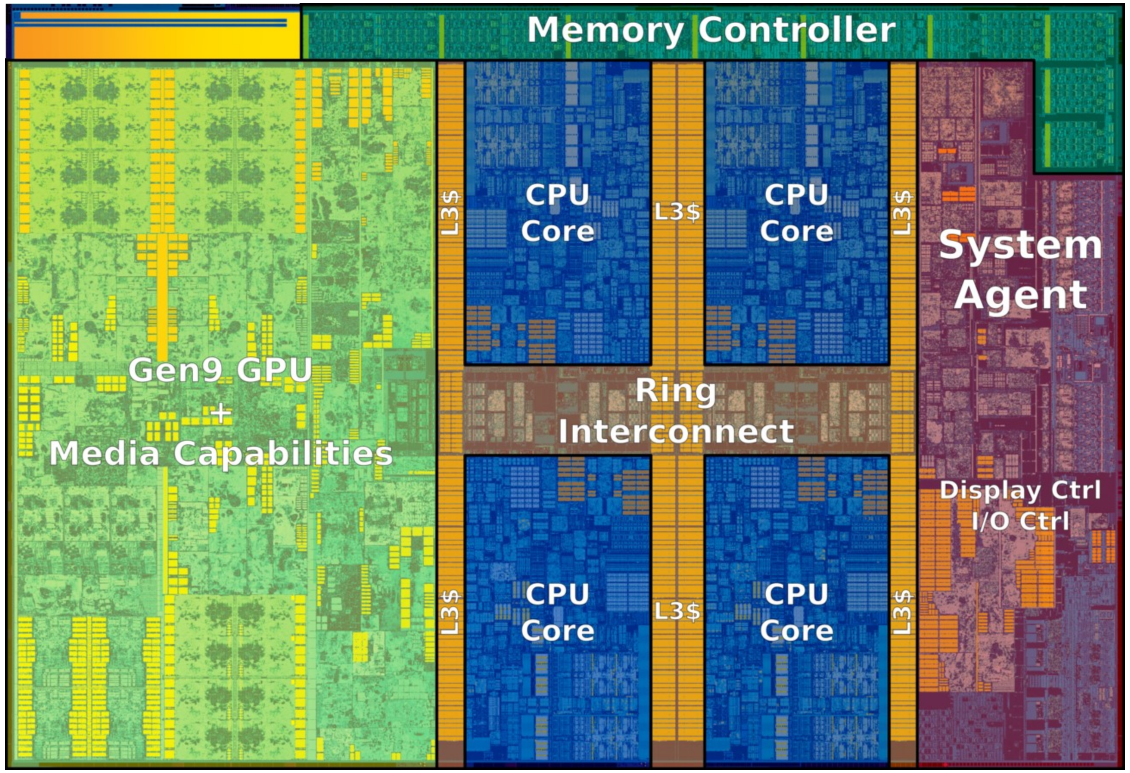
After this lab, you'll be able to…

- Understand the basic training algorithm our CNN framework uses
- Optimize the performance of loops using OpenMP
- Measure the impact of your optimizations
- Understand how convolution, pool, and RELU layers work in a CNN.

Multiple Processors

- We have four cores, but we've only been using one!
- We can share work across the cores to go faster and/or for lower energy.

- Take CSE 160 for more extensive treatment



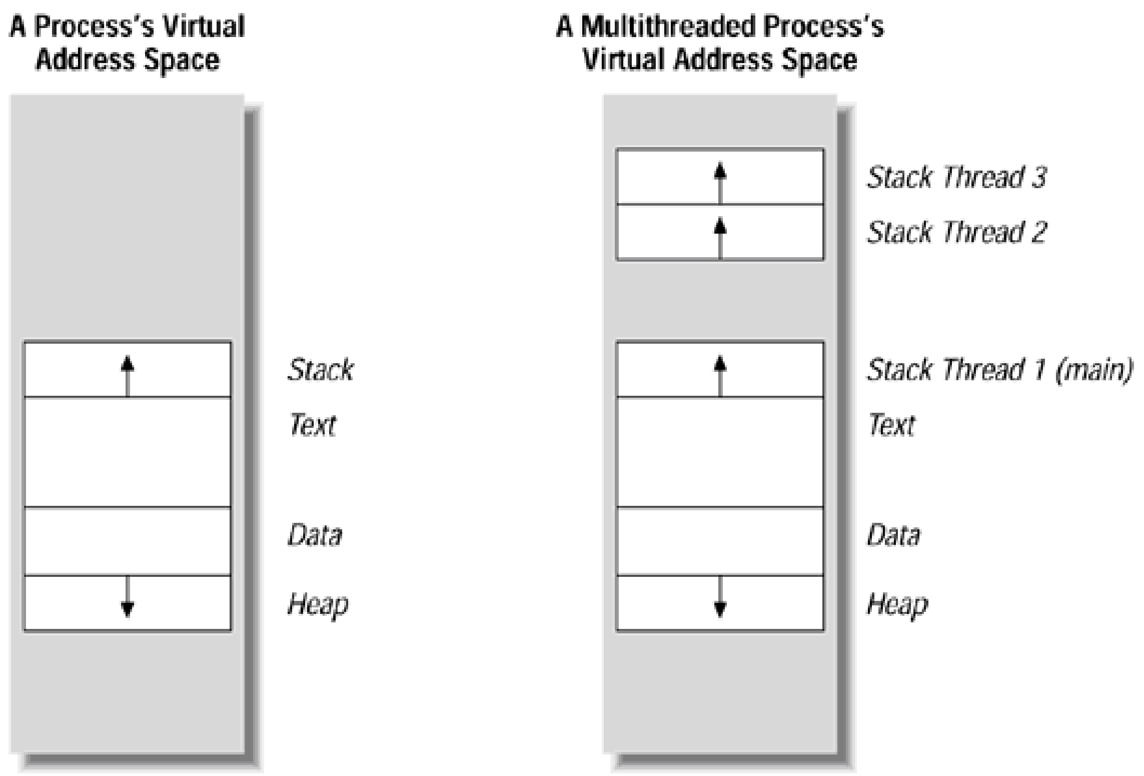
Concurrency

- Different pieces of work are being done by different threads
- If they are running at the same time, we call this parallel execution.

- Multi-processing vs. multi-threading

Threads

- Shared Memory Model
  - All threads see the same view of memory (except locals)
  - Each thread has its own stack space and PC
- OS runs threads in any order
- For correctness – you may need to use synchronization



PThreads Programming by Nichols et al. (1996)

How To Use Threads (CSE 141L)

- Option 1: OpenMP (Open multiprocessing)
  - High-level API for parallelizing C and C++ code
  - Relatively easy
  - Only applicable to relatively simple loops
- Option 2: pthreads
  - Low-level API
  - Harder to use
  - You can use it for many more things
  - Take CSE120

Hello World in openmp

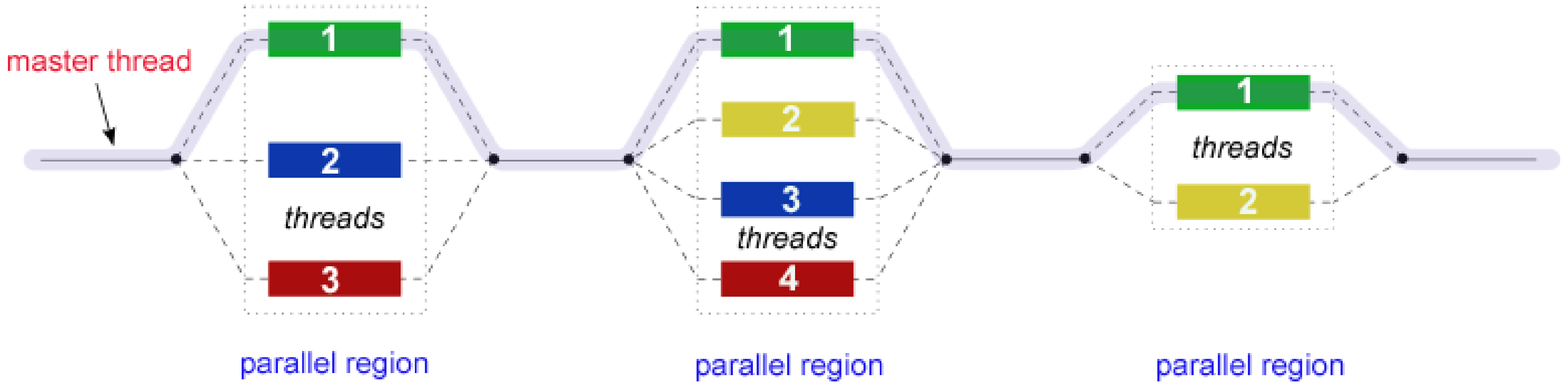
- Need to use –fopen and <omp.h>
- Options all start with “#pragma omp”
  - “#pragma ...” is a way to add arbitrary extension to C/C++
- “#pragma omp parallel”
  - Means give me a bunch of threads (system default)
  - compiler uses pthreads under the hood
- “omp_get_thread_num()”
  - Gets you the number of your thread (0…n)

Synchronization

- Things can go terribly wrong working with shared variables
  - Race conditions – output depends on which thread comes first
  - BUT, your program must be correct for all orderings
- We can avoid this by using synchronization
  - E.g., #pragma omp critical
    - Under the hood, these are done with mutexes or locks (CSE120)
  - But synchronization means serialization
    - Do it too much, you lose all performance benefits

Fork-Join Parallelism

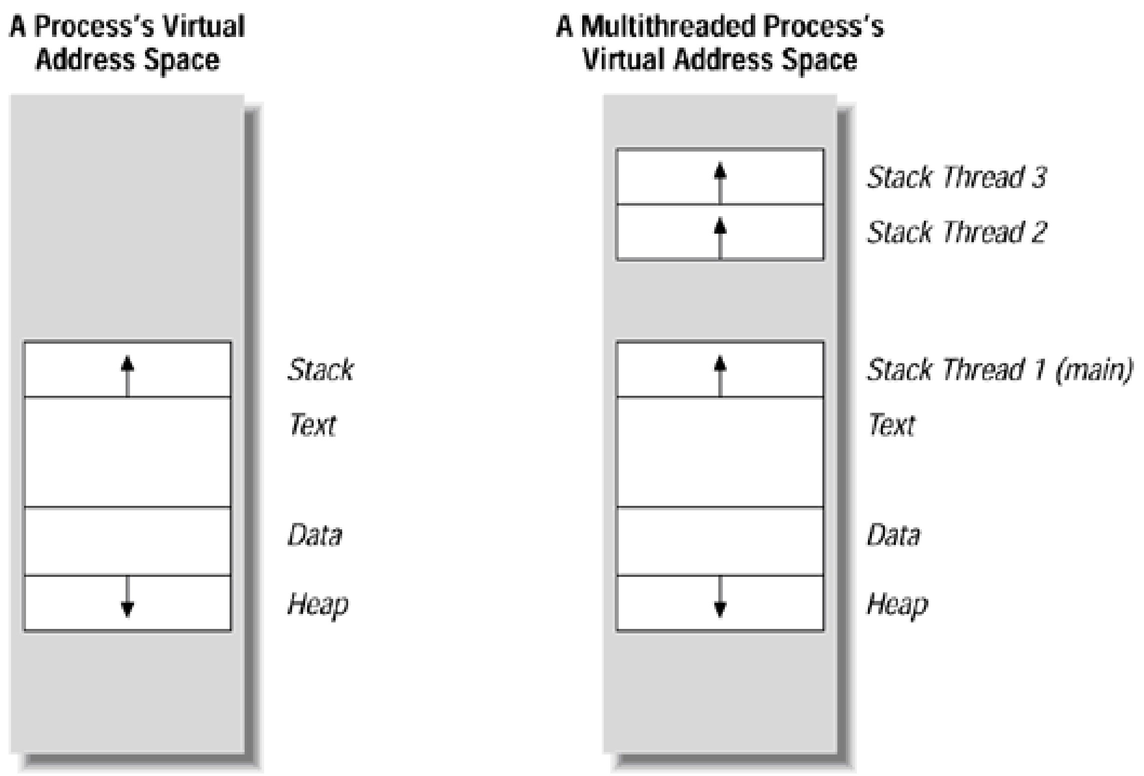
- Do some work in serial, do a bunch in parallel, join the results, repeat.



Managing Data

- Data allocated outside a pragma is **shared**
- Data allocated inside a pragma is **private**

- **Be careful – this is absolutely essential**



PThreads Programming by Nichols et al. (1996)

Calculating pi

- Classic approach, solve:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- **This has some really nice loop independence we can exploit**

```
static long num_steps = 100000;
double step;
int main ()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i=0;i< num_steps; i++){
     x = (i+0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Calculating pi – in parallel

- From openmp, only use:
  - **#pragma omp parallel, omp_get_num_threads(), omp_get_thread_num()**
  - **Careful declaration of local vs. global vars**
  - **(Hint, an array can be global but each thread only need access each part)**

Synchronization Revisited

- Barriers
  - Automatic at the end of a parallel pragma, but can also do:
  - “#pragma omp barrier” within a pragma
- Mutual Exclusion
  - #pragma omp critical

Synchronization Revisited

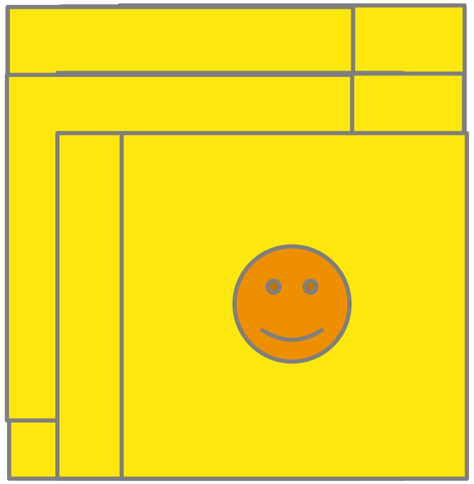
- Barriers
  - Automatic at the end of a parallel pragma, but can also do:
  - "#pragma omp barrier" within a pragma
- Mutual Exclusion
  - #pragma omp critical
  - #pragma omp atomic
    - Only applies to a single statement, can be faster than omp critical by leveraging some atomic hardware instructions

OpenMP Primitives: Loops

- #pragma omp parallel for
  - Run the following for loop with multiple threads.
  - The loop needs to be pretty simple.
  - Something like “for(int i = C; i < K; i+=B)”
    - K and B need to fixed for the execution of the loop
    - Otherwise, nothing will happen.
  - It uses all the cores by default.

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

Tiled Example Code

```
void do_stabilize_tile_y_1(const tensor_t<double> & images, tensor_t<double> & output, int TILE_SIZE)
{

        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 < MAX_OFFSET; offset_x++)  {
                                for (int offset_y = 0; offset_y < MAX_OFFSET; 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));
                                                }
                                        }
                                }
                        }
                }
        }
}
```

- **Iterations of the outer loop don't write-share anything**
  - Each iteration writes to output(…, this_frame) and nothing else
  - They can run in parallel
- Execution time: 0.516s

Parallelized Version

```
void do_stabilize_tile_y_1_omp_simple(const tensor_t<double> & images, tensor_t<double> & output, int TILE_SIZE)
{
	OPEN_TRACE("trace.out");
	// parallizing across results in no sharing in `output` since
	// each frame output it's result to one element of `output`.
#pragma omp parallel for
	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 < MAX_OFFSET; offset_x++)  {
				for (int offset_y = 0; offset_y < MAX_OFFSET; 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;
							//DUMP_ACCESSES();
							double t = fabs(images(pixel_x, pixel_y, 0, this_frame) -
									images(shifted_x, shifted_y, 0, previous_frame));
							output(offset_x, offset_y, 0, this_frame) += t;
						}
					}
				}
			}
		}
	}
}
```

- The #pragma parallelizes the next loop.
- Iterations of that loop will run in parallel.
  - Our tiling and loop re-nesting optimizations work as they did before.
- Execution time: 0.173 (2.99x)

Another Approach

```
void do_stabilize_tile_y_1(const tensor_t<double> & images, tensor_t<double> & output, int TILE_SIZE)
{

        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 < MAX_OFFSET; offset_x++)  {
                                for (int offset_y = 0; offset_y < MAX_OFFSET; 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));
                                                }
                                        }
```

- Let’s try parallelizing pixel_yy instead.
- It might put less pressure on the L3 since we only need to have 2 frames in memory instead of images.size.b

Another Approach:

```
void do_stabilize_tile_y_1_omp_critical(const tensor_t<doubl
{
        //OPEN_TRACE("trace.out");
        for (int this_frame = 1; this_frame < images.size.b;
                int previous_frame = this_frame - 1;

                // Parallelizing on pixel_yy, results in sha
#pragma omp parallel for
                for(int pixel_yy = 0; pixel_yy < images.size

                        for (int offset_x = 0; offset_x < MA
                                for (int offset_y = 0; offse

                                        for(int pixel_y = pi                                                                    ixel_y++) {
                                                for(int pixe

                                                        int
                                                        int

                                                        if (

                                                        doub
                                                                images(shifted_x, shifted_y, 0, previous_frame));

                                                        output(offset_x, offset_y, 0, this_frame) += t;
                                                }
```

```
b = 0:
z = 0:
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
       0       0       0       0       0       0       0       0
b = 1:
z = 0:
    92.2     143     144     136     126    98.9    93.3     110
    80.4     128     138     127     113    98.4    94.1     121
    91.7     134     128     112    91.4    87.7     108     124
    95.4     137     108    96.1    69.8    84.2     103     120
    93.5     122    94.9    76.7    58.7      98     110     122
     105     123      84    61.3    60.2     107     124     128
     136     121    83.7    57.6    67.6     119     130     130
     130     108    78.8    88.1     101     121     126     126


b = 0:
z = 0:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
b = 1:
z = 0:
-61 -12 -11 -15 -15 -26 -24 -18
-70 -24 -14 -15 -13 -13 -22 -16
-55 -13 -16 -15 -16 -13 -12 -18
-45 -2 -20 -9.4 -17 -14 -22 -24
-44 -8.1 -16 -8.4 -19 -5.5 -23 -25
-40 -4.8 -18 -17 -28 -10 -18 -23
-14 -6.8 -15 -28 -41 -13 -16 -18
-7.2 -2.1 -7.5 -5.1 -15 -13 -14 -11

stabilize.exe: build/stabilize.cpp:532: void stabilize(const string&, const dataset_t&, int): Assertion `0' failed.
Aborted (core dumped)
```

- Just move the #pragma
- Execution time: ???

The threads are working on different "slices" of pixel_yy, but they are updating same output array.

Critical section!

- Because multiple threads are working with the same data at the same time, we need:
  - #pragma omp critical

Another Approach: Parallelize inner loop

```
void do_stabilize_tile_y_1_omp_critical(const tensor_t<double> & images, tensor_t<double> & output, int TILE_SIZE)
{
        OPEN_TRACE("trace.out");
        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;

                // Parallelizing on pixel_yy, results in sharing in output.
#pragma omp parallel for
                for(int pixel_yy = 0; pixel_yy < images.size.y; pixel_yy +=  TILE_SIZE) {

                        for (int offset_x = 0; offset_x < MAX_OFFSET; offset_x++)  {
                                for (int offset_y = 0; offset_y < MAX_OFFSET; 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;
                                                        double t = fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                                        images(shifted_x, shifted_y, 0, previous_frame));

                                                        #pragma omp critical
                                                        {
                                                                output(offset_x, offset_y, 0, this_frame) += t;
                                                        }
                                                }
```

- This will ensure that the threads access output one at a time.
- Execution time: 23s (0.02x speedup)
- Critical sections incur overhead, and we are using too many of them.

```
void do_stabilize_tile_y_1_omp_critical_fast(const tensor_t<double> & images, tensor_t<double> & output, int TILE_SIZE)
{
        OPEN_TRACE("trace.out");
        for (int this_frame = 1; this_frame < images.size.b; this_frame++) {
                int previous_frame = this_frame - 1;
                // same thing: need to protect
#pragma omp parallel for
                for(int pixel_yy = 0; pixel_yy < images.size.y; pixel_yy +=  TILE_SIZE) {
                        tensor_t<double>_output(output.size);
                        _output.clear();
                        for (int offset_x = 0; offset_x < MAX_OFFSET; offset_x++)  {
                                for (int offset_y = 0; offset_y < MAX_OFFSET; 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;

                                                        double t = fabs(images(pixel_x, pixel_y, 0, this_frame) -
                                                                        images(shifted_x, shifted_y, 0, previous_frame));
                                                        // accumulate the updates locally
                                                        _output(offset_x, offset_y, 0, this_frame) += t;
                                                }
                                        }
                                }
                        }
#pragma omp critical // Apply them en masse this is reasonably fast because it's small, so the serialization isn't a big deal.
                        {
                                for (int offset_y = 0; offset_y < MAX_OFFSET; offset_y++)  {
                                        for (int offset_x = 0; offset_x < MAX_OFFSET; offset_x++)  {
                                                output(offset_x, offset_y, 0, this_frame) += _output(offset_x, offset_y, 0, this_frame);
                                        }
                                }

                        }
                }
        }
}
```

- _output is local to each thread, no critical section needed
- At the end of each iteration, update outputs all at once.
  - Fewer critical sections
- Execution time: 0.115 (4.48x)

What do you need to use?

- Just these two primitives are enough for our solution:
  - #pragma omp parallel for
  - #pragma omp critical
- But you'll need to be careful where to put these.
- <Warning> – gprof doesn't work on multithreaded programs

There’s a Lot More in OpenMP

- You don’t need it for the lab, but you are welcome use it.
- There are lots of documents online.
  - Many of them are not very useful because they are too detailed
- This blog is pretty good
  - http://jakascorner.com/blog/
  - Especially these
    - http://jakascorner.com/blog/2016/04/omp-introduction.html
    - http://jakascorner.com/blog/2016/05/omp-for.html
    - http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
    - http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html
    - http://jakascorner.com/blog/2016/07/omp-default-none-and-const.html

CSE141L - Lab 4 - Threads

Discussion Session

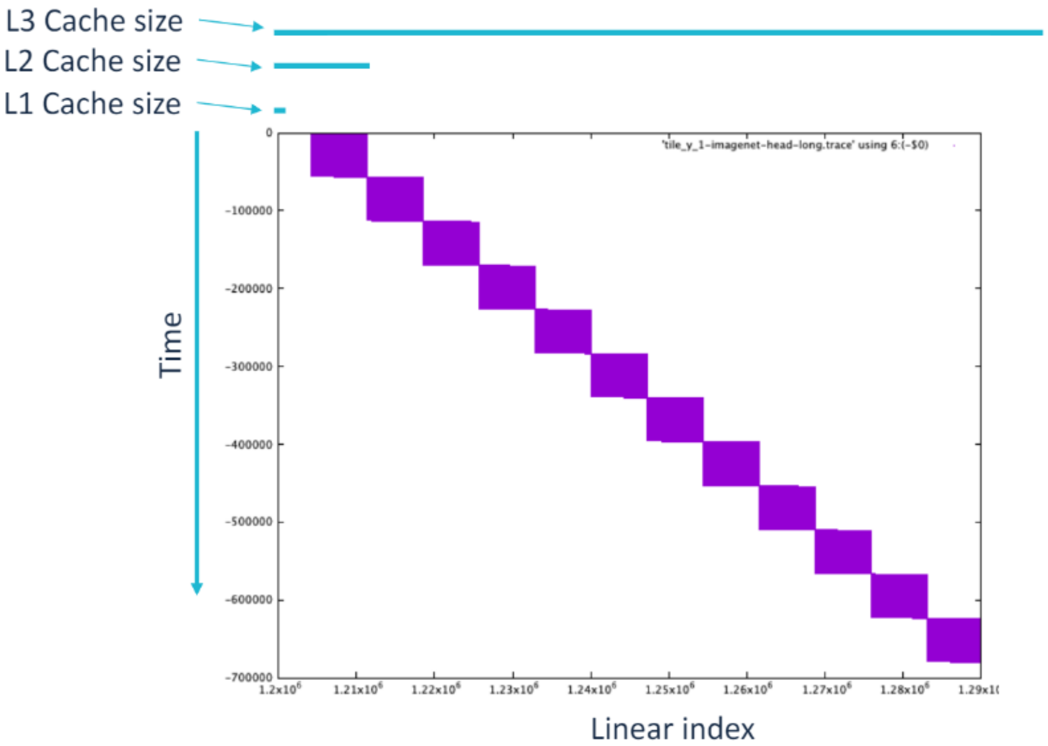
Overview

- Questions about multithreading in stabilize
- Understanding the Memory Access Visualization Tool
- Walkthrough of OpenMP on fc_layer_t::calc_grads()
  - #pragma omp parallel for
  - #pragma omp critical
- Brief overview of fc_layer_t::fix_weights
- Questions

Questions about multithreading in stabilize?

```
#pragma omp parallel for
for(int pixel_yy = 0; pixel_yy < images.size.y; pixel_yy +=  TILE_SIZE) {
    tensor_t<double> _output(output.size);
    _output.clear();
    for (int offset_x = 0; offset_x < MAX_OFFSET; offset_x++)  {
        for (int offset_y = 0; offset_y < MAX_OFFSET; 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;

                    double t = fabs(images(pixel_x, pixel_y, 0, this_frame) -
                            images(shifted_x, shifted_y, 0, previous_frame));
                    // accumulate the updates locally
                    _output(offset_x, offset_y, 0, this_frame) += t;
                }
            }
        }
    }
    #pragma omp critical // Apply them en masse this is reasonably fast because it's small, so the serialization isn't a big deal.
    {
        for (int offset_y = 0; offset_y < MAX_OFFSET; offset_y++)  {
            for (int offset_x = 0; offset_x < MAX_OFFSET; offset_x++)  {
                output(offset_x, offset_y, 0, this_frame) += _output(offset_x, offset_y, 0, this_frame);
            }
        }

    }
```

Memory Access Visualization Tool



Original - 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;
}
```

Original - 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;
}
```

- Calculate derivative of activator function
- Backpropagate Error from Next Layer

Original - 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;
}
```

Find error of each input proportional to its weight and sum it up

Baseline - fc_calc_grads (Tier 1)

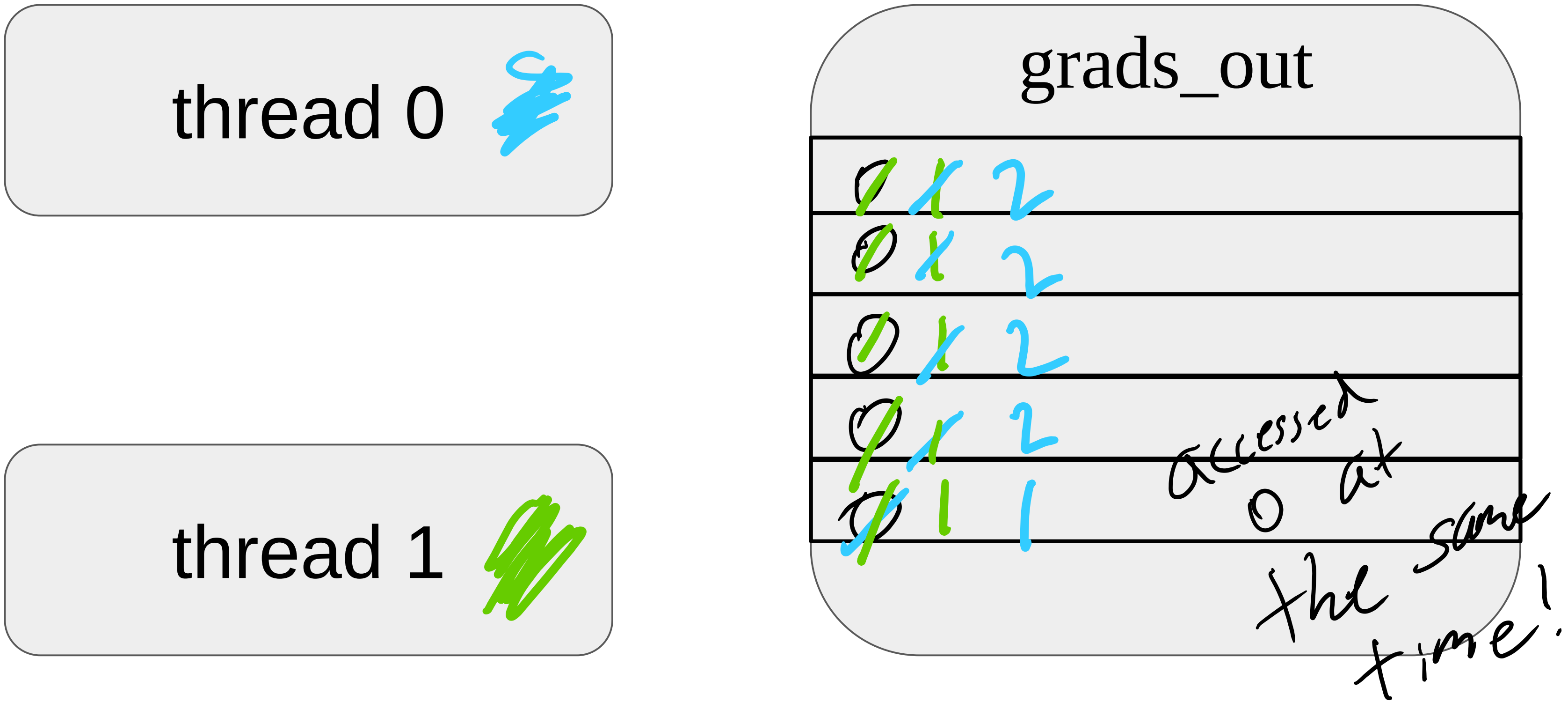
```
void calc_grads( const tensor_t<double>& grad_next_layer ) {
    memset( grads_out.data, 0, grads_out.size.x * grads_out.size.y * grads_out.size.z * sizeof( double ) );
    grads_out.size.x = grads_out.size.x * grads_out.size.y * grads_out.size.z;
    grads_out.size.y = 1;
    grads_out.size.z = 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;
        }
    }
    // Reorder loops and  tile on n
    for ( int nn = 0; nn < out.size.x; nn+=BLOCK_SIZE ) {
        for ( int b = 0; b < out.size.b; b++ ) {
            for ( int n = nn; n < nn + BLOCK_SIZE && n < out.size.x; n++ ) {
                for ( int i = 0; i < grads_out.size.x; i++ ) {
                    grads_out(i, 0, 0, b) += act_grad(n, 0, 0, b) * weights( i, n, 0);
                }
            }
        }
    }
    grads_out.size = in.size;
}
```

Multiple threads on nn loop

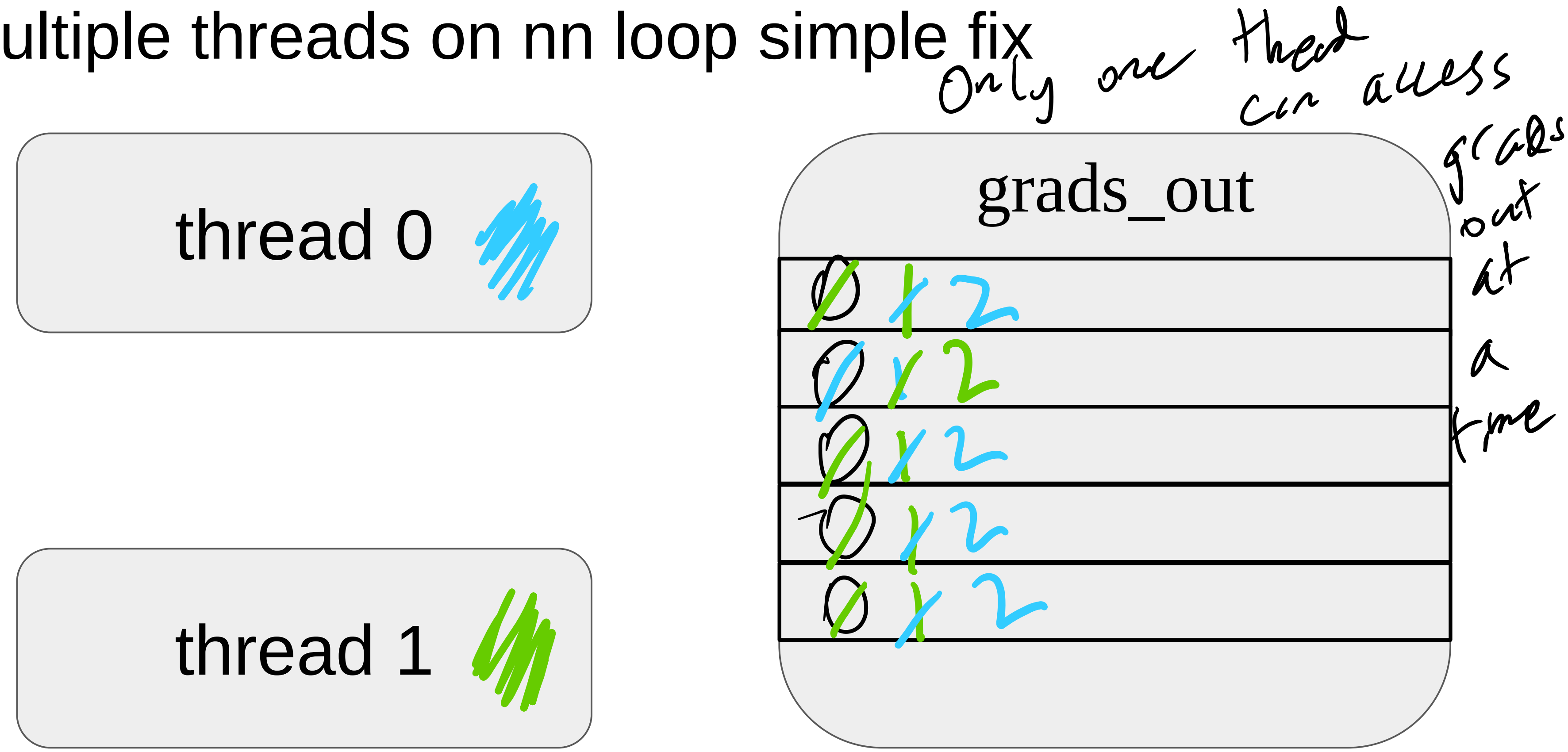
```
for ( int nn = 0; nn < out.size.x; nn+=BLOCK_SIZE ) {
    for ( int b = 0; b < out.size.b; b++ ) {
        for ( int n = nn; n < nn + BLOCK_SIZE && n < out.size.x; n++ ) {
            for ( int i = 0; i < grads_out.size.x; i++ ) {
                grads_out(i, 0, 0, b) += act_grad(n, 0, 0, b) * weights( i, n, 0);
            }
        }
    }
}
```

- Multiple threads writing to the same memory location of grads_out

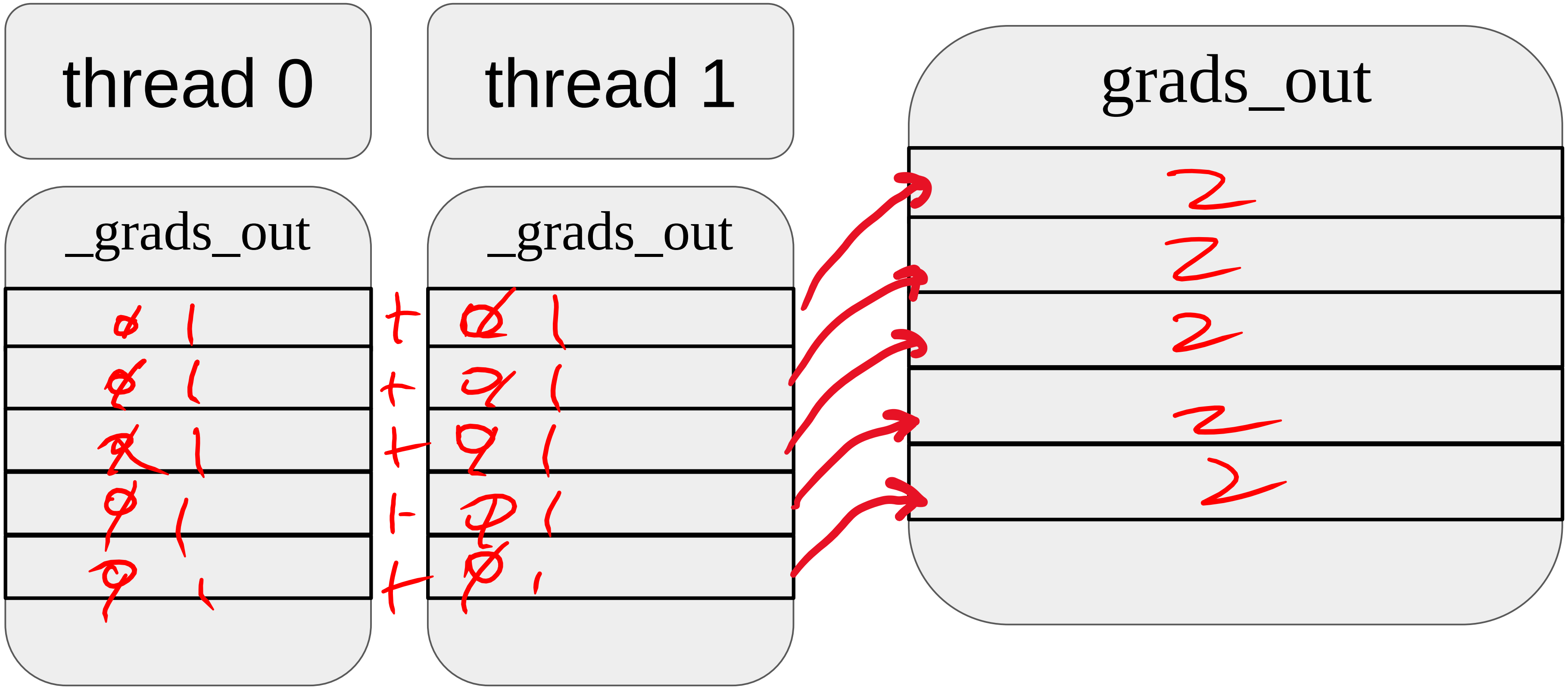
Multiple threads on nn loop issue



Multiple threads on nn loop simple fix



Multiple threads on nn loop better fix



Multiple threads on b loop

```
for ( int nn = 0; nn < out.size.x; nn+=BLOCK_SIZE ) {
    for ( int b = 0; b < out.size.b; b++ ) {
        for ( int n = nn; n < nn + BLOCK_SIZE && n < out.size.x; n++ ) {
            for ( int i = 0; i < grads_out.size.x; i++ ) {
                grads_out(i, 0, 0, b) += act_grad(n, 0, 0, b) * weights( i, n, 0);
            }
        }
    }
}
```

- Each thread is writing to different locations of grads_out
    - More specifically, each thread handles writing to a different batch output
- No race conditions!

Multiple threads on n loop

```
for ( int nn = 0; nn < out.size.x; nn+=BLOCK_SIZE ) {
    for ( int b = 0; b < out.size.b; b++ ) {
        for ( int n = nn; n < nn + BLOCK_SIZE && n < out.size.x; n++ ) {
            for ( int i = 0; i < grads_out.size.x; i++ ) {
                grads_out(i, 0, 0, b) += act_grad(n, 0, 0, b) * weights( i, n, 0);
            }
        }
    }
}
```

- Is there a problem here?
- How can we fix it?
- Create local version of grads_out at the beginning of the loop and then sum results at the end of the loop!

Multiple threads on i loop

```
for ( int nn = 0; nn < out.size.x; nn+=BLOCK_SIZE ) {
    for ( int b = 0; b < out.size.b; b++ ) {
        for ( int n = nn; n < nn + BLOCK_SIZE && n < out.size.x; n++ ) {
            for ( int i = 0; i < grads_out.size.x; i++ ) {
                grads_out(i, 0, 0, b) += act_grad(n, 0, 0, b) * weights( i, n, 0);
            }
        }
    }
}
```

- Is there a problem here?
- No! We don’t have to fix anything here. Each thread writes to distinct memory locations in grads_out.

Baseline - fc_fix_weights (Tier 2)

```
void fix_weights() {
    tdsize old_in_size = in.size;
    in.size.x = in.size.x * in.size.y * in.size.z;
    in.size.y = 1;
    in.size.z = 1;

    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++ ) {
                double& w = weights( i, n, 0 );
                double m = (act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM);
                double g_weight = w - (LEARNING_RATE * m * in(i, 0, 0, b) + LEARNING_RATE * WEIGHT_DECAY * w);
                w = g_weight;
            }
            old_act_grad(n, 0, 0, b) = act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM;
        }
    }
    in.size = old_in_size;
}
```

Multiple threads on b loop

```
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++ ) {
            double& w = weights( i, n, 0 );
            double m = (act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM);
            double g_weight = w - (LEARNING_RATE * m * in(i, 0, 0, b) + LEARNING_RATE * WEIGHT_DECAY * w);
            w = g_weight;
        }
        old_act_grad(n, 0, 0, b) = act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM;
    }
}
```

- Is there a problem here?
- How can we fix it?
- Use omp critical around the accumulation.

Multiple threads on n loop

```
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++ ) {
            double& w = weights( i, n, 0 );
            double m = (act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM);
            double g_weight = w - (LEARNING_RATE * m * in(i, 0, 0, b) + LEARNING_RATE * WEIGHT_DECAY * w);
            w = g_weight;
        }
        old_act_grad(n, 0, 0, b) = act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM;
    }
}
```

- Is there a problem here?
- No! We don’t have to fix anything here. Each thread writes to distinct memory locations in weights and old_act_grad.

Multiple threads on i loop

```
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++ ) {
            double& w = weights( i, n, 0 );
            double m = (act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM);
            double g_weight = w - (LEARNING_RATE * m * in(i, 0, 0, b) + LEARNING_RATE * WEIGHT_DECAY * w);
            w = g_weight;
        }
        old_act_grad(n, 0, 0, b) = act_grad(n, 0, 0, b) + old_act_grad(n, 0, 0, b) * MOMENTUM;
    }
}
```

- Is there a problem here?
- No! We don’t have to fix anything here. Each thread writes to distinct memory locations in weights.

Tier 3 Hints

- Gprof does not work with multithreading
  - Comment out OMP=yes in config
  - Can’t iteratively check gprof
- Don’t be alarmed if you get slowdowns for some loops
- You’ll need to do more than just adding multithreading to achieve the speedup of 6x
  - Try combining multithreading with previous optimizations like tiling and loop reordering
  - This is demonstrated in Tier 1

Questions?