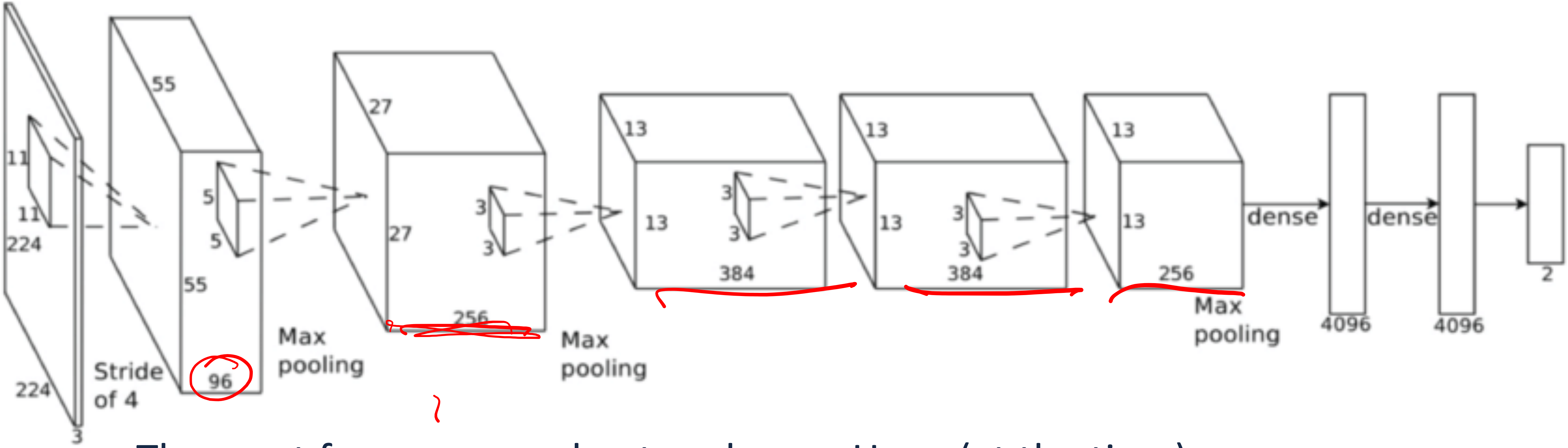
Final Project

After this lab, you'll be able to…

- If you want
  - Vectorize loops using the OpenMP `simd` directive.
  - Interpret the output of gcc to understand where it successfully vectorizes code.
  - Modify code to improve vectorizability
- Gain more practice applying other optimizations to a larger CNN model

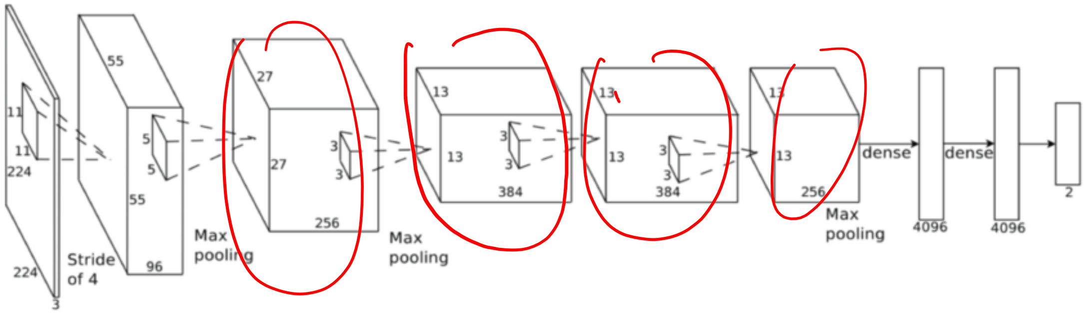
Alex Net



- The most famous neural network
- Started the current AI crazy
- Built for the imagenet competition and destroyed everyone.

- Huge (at the time)
  - 60 million weights
  - 20 Canela-style layers
  - 2 weeks to train on two GPUS

In this lab: Mininet



- ¼ the size of AlexNet
- 128x128 pixel RGB images
- 590MB of weights

SIMD: Single Instruction Multiple Data

Vector

- A single instruction operates on multiple data values (I.e., a 'vector')
  - 1 instruction
  - 4 additions

10	12	8	5

+

4	7	9	2

=

14	19	17	7

UCSDCSE Computer Science and Engineering

SIMD: Single Instruction Multiple Data

- A single instruction operates on multiple data values (I.e., a 'vector')
  - 1 instruction
  - 4 additions at once!
  - CPI = 1, Cycles/addition = 0.25

10	12	8	5

+

4	7	9	2

=

14	19	17	7

UCSDCSE Computer Science and Engineering

How To Use SIMD

- Option 1: OpenMP (Open multiprocessing)
  - "#pragma omp simd"
  - Somewhat portable across compilers
- Option 2: gcc -O3
  - Rely on gcc's auto-vectorizer (turned on by default)
  - Not 'portable' because it's transparent
  - Same internal implementation as "omp simd"
  - Doesn't handle long vectors only length=2 But 4 should be possible
- Option 3.1: By hand – compiler "intrinsic"
- Option 3.2: By hand – gcc support vector data types
- Option 3.3: inline assembly
- I'll talk about option 3 in class tomorrow.

x86 Vectors (AVX instructions)

- Our processor support 128 and 256-bit vectors
- They can be divided into vectors of different sizes
  - 16 or 32 8-bit ints
  - 8 or 16 16-bit ints
  - 4 or 8 32-bit ints
  - 2 or 4 64-bit ints
  - 4 or 8 32-bit floats
  - 2 or 4 64-bit floats

Yet More x86 Registers

Integer Registers

16bit	32bit	64bit	Description
AX	EAX	RAX	The accumulator register
BX	EBX	RBX	The base register
CX	ECX	RCX	The counter
DX	EDX	RDX	The data register
SP	ESP	RSP	Stack pointer
BP	EBP	RBP	Points to the base of the stack frame
	R*n*	R*n*D	(n = 8...15) General purpose registers
SI	ESI	RSI	Source index for string operations
DI	EDI	RDI	Destination index for string operations
IP	EIP	RIP	Instruction Pointer
FLAGS			Condition codes

MMX Registers

x87 name	MMX name	Description
ST7	MM7	64 bit each. MMX: 1 double- or single-precision floating point value. x87: 32, 64, or 80bits
ST6	MM6	
ST5	MM5	
ST4	MM4	
ST3	MM3	
ST2	MM2	
ST1	MM1	
ST0	MM0	

SSE/AVX Registers

AVX-512 names 512 bits	AVX names 256 bits	SSE names 128 bits
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15

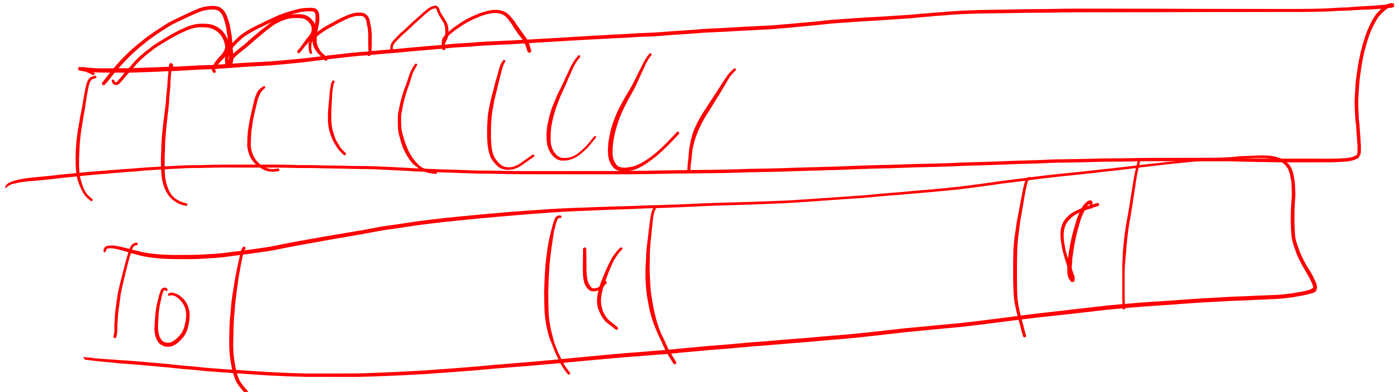
- Each XMM/YMM/ZMM register can hold
  - 8, 16, 32, 64, or 128-bit integers (totaling 128 bits)
    - e.g., 16 x 8-bit int or 4 x 32-bit int or 2 x 64-bit int.
  - 32 or 64-bit FP totaling 128 (XMM), 256 (YMM), or 512 (ZMM) bits.
    - e.g., 16 x 32-bit int in ZMM0

Vector Operations

- Math
  - +, -, *, /, etc.
  - Example: “vaddps (%rsi, %rax,4), %ymm0, %ymm1”
    - ‘v’ == AVX prefix
    - ‘p’ == packed (i.e., vector)
    - ‘s’ == single-precision (32-bit float)
    - ‘ymm0’ and ’ymm1’ name 256-bit registers, so it adds vectors of 8 32-bit floats
    - 3 arguments!!! The source is not overwritten!!! It’s a miracle!!!
- Load/store
  - Versions of ‘mov’ can load whole vectors from memory
  - Example: `vmovups %ymm0 (%rdx, %rax, 4)`
    - ‘v’ == AVX prefix
    - ‘u’ == unaligned (i.e, address can be anything)
    - ‘p’ == packed (i.e., vector)
    - ‘s’ == single-precision (32-bit float)
    - ‘ymm0’ names a 256-bit register, so it stores 8 floats

Vectorizing a loop

- "Vectorize" means
  - Try to execute multiple iterations at once using a vector operations.
- Many corner cases
  - What if the loop bounds aren't a multiple of the vector size?
  - What if there's control in the loop?
  - What if the access's aren't "unit stride" (to consecutive memory locations).



Auto-vectorization

- "#pragma omp simd"
  - Vectorize the following loop.
- gcc -O3
  - Gcc has auto-vectorization support turned on with -O3
  - Seems to be the same as 'omp simd`
- Many things will prevent auto-vectorization
  - Control in the loop body (different 'slots' undergo different operations)
  - If consecutive iteration don't access consecutive memory locations.
- Manual vectorization can be more effective, but it's hard.

CSE 141L - Final Project

Discussion Section

Overview

- Understanding GCC Auto Vectorization Module
- Walk through Tier 1
- Tier 2: Single Instruction Multiple Data (SIMD)
- Hints for Tier 3 (a brief review)
- Convolution & activate
- Convolution & calc_grads

GCC Auto-Vectorization Module

- What is vectorization?
  - Perform one operation on multiple elements of a vector
  - Chunk-wise processing instead of element wise
  - Can improve computing time
- Motivation
  - Utilize the CPU’s vectorization features (vector registers and operations)
  - Produce fast and small binaries

GCC Auto-Vectorization Module

- Compiler does the vectorization for you! But,
  - Must verify that the optimization is legal
  - Optimization is beneficial
- -ftree-vectorize -fopt-info-vec-all in config.env
- Alternatively: AUTO_VEC = yes
- Enabled by -O3 and greater optimizations by default
  - Only -fopt-info-vec-all needed for feedback

GCC Auto-Vectorization Module

- Limitations
  - Countable loops
  - No backward loop-carried dependencies
  - Straight-line code (only one control flow: no if else)
  - Loop to be vectorized must be the innermost loop if nested

Demonstration: Auto-Vectorization Module

```
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;
}
```

Demonstration: Auto-Vectorization Module

```
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) );
                        //std::cout << ad;
                        double ng = grad_next_layer(n, 0, 0, b);
                        //std::cout << ng;
                        act_grad(n, 0, 0, b) = ad * ng;
                }
        }


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

        grads_out.size = in.size;
}
```

Demonstration: Auto-Vectorization Module

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

            grads_out.size = in.size;
        }
```

General Flow for Tier 1

- Loop Reordering on Baseline Implementation to enable auto-vectorization
- Loop Tiling for further performance benefit
- Loop Reordering on tiled code to enable auto-vectorization
- Analysis of speedup due to vectorization

Tier 2: single instruction multiple data (simd)

```
#pragma omp parallel
{
        #pragma omp for simd
        for ( int nn = 0; nn < out.size.x; nn+=BLOCK_SIZE ) {
                for ( int b = 0; b < in.size.y; b++ ) {
                        for ( int n = nn; n < nn + BLOCK_SIZE && n < out.size.x; n++ ) {
                                for ( int i = 0; i < in.size.x; i++ ) {
                                        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() with loop reordering/tiling (cache lab) & multithreading (threads lab) & vectorization (tutorials)

Tier 3: Some Hints

- Optimization techniques
  - Cache lab: Loop reordering & loop tiling
  - Threads lab: Multithreading
  - Final Project: Tutorial for Vectorization
- Combine multiple optimizations to achieve the speedup
- Some examples from previous labs
  - Multithreading+other optimizations -> tier 1 in threads lab
  - Vectorization -> tier 2 (tutorials)

conv_layer_t::activate

```
void activate( tensor_t<double>& in ) {
	copy_input(in);
	for ( int b = 0; b < out.size.b; b++ ) {
		for ( uint filter = 0; filter < filters.size(); filter++ ) {
			tensor_t<double>& filter_data = filters[filter];
			for ( int x = 0; x < out.size.x; x++ ) {
				for ( int y = 0; y < out.size.y; y++ ) {
					point_t mapped(x*stride, y*stride, 0);
					double sum = 0;
					for ( int i = 0; i < kernel_size; i++ )
						for ( int j = 0; j < kernel_size; j++ )
							for ( int z = 0; z < in.size.z; z++ ) {
								double f = filter_data( i, j, z );

								double v;
								if (mapped.x + i >= in.size.x ||
								mapped.y + j >= in.size.y) {
									v = pad;
								} else {
									v = in( mapped.x + i, mapped.y + j, z, b );
								}
								sum += f*v;
							}
					out( x, y, filter, b ) = sum;
				}
			}
		}
	}
}
```

conv_layer_t::calc_grads

```
void calc_grads(const tensor_t<double>& grad_next_layer ) {
	throw_assert(grad_next_layer.size == out.size, "mismatch input size for calc_grads");
	for ( int b = 0; b < in.size.b; b++ )
		for ( uint k = 0; k < filter_grads.size(); k++ )
			for ( int i = 0; i < kernel_size; i++ )
				for ( int j = 0; j < kernel_size; j++ )
					for ( int z = 0; z < in.size.z; z++ )
						filter_grads[k].get( i, j, z, b ).grad = 0;

	for ( int b = 0; b < in.size.b; b++ ) {
		for ( int x = 0; x < in.size.x; x++ ) {
			for ( int y = 0; y < in.size.y; y++ ) {
				range_t rn = map_to_output( x, y );
				for ( int z = 0; z < in.size.z; z++ ) {
					double sum_error = 0;
					for ( int i = rn.min_x; i <= rn.max_x; i++ ) {
						int minx = i * stride;
						for ( int j = rn.min_y; j <= rn.max_y; j++ ) {
							int miny = j * stride;
							for ( int k = rn.min_z; k <= rn.max_z; k++ ) {
								int w_applied = filters[k].get( x - minx, y - miny, z );
								sum_error += w_applied * grad_next_layer( i, j, k, b );
								filter_grads[k].get( x - minx, y - miny, z, b ).grad += in( x, y, z, b ) * grad_next_layer( i, j, k, b );
							}
						}
					}
					grads_out( x, y, z, b ) = sum_error;
				}
			}
		}
	}
}
```