Installing TensorFlow from the recommended `pip` .whl is quick and painless. But upon testing my installation, I got these "warnings":

```
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
library wasn't compiled to use SSE3 instructions, but these are
available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
library wasn't compiled to use SSE4.1 instructions, but these are
available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
library wasn't compiled to use SSE4.2 instructions, but these are
available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
library wasn't compiled to use AVX instructions, but these are
available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
library wasn't compiled to use AVX2 instructions, but these are
available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
library wasn't compiled to use FMA instructions, but these are
available on your machine and could speed up CPU computations.
>>> print(sess.run(hello))
b'Hello, TensorFlow!'
```

What is SSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA? And why would they make Tensorflow faster?

SSE stands for **Streaming SIMD Extensions**, and SIMD stands for **Single Instruction Multiple Data**. In short, it sounds like these instructions allow your CPU to behave a little more like a GPU, doing the same operation on multiple data objects, instead of processing them one at a time. Handy for processing 3D graphics or — you guessed it — training layers of a neural net.

The AVX (Advanced Vector Extensions) and FMA (Fused Multiply Add) instructions are also extensions to SIMD operations.

So how do I build Tensorflow with support for SSE/AVX/FMA?

First, read all of TensorFlow's instructions on building from source for your OS.

Depending on your specific configuration, you may want to consult this thread and pick your own particular set of flags for the build process. Here's what I used when I got to the `bazel build` portion of the instructions:

```
bazel build -c opt --copt=-mavx --copt=-mavx2 --copt=-mfma --copt=-
mfpmath=both -k //tensorflow/tools/pip_package:build_pip_package
```

I did not use `--config=cuda` for this particular build as I am doing a CPU-only build. A fun next step would be to try to optimize for the fastest CUDA-enabled Tensorflow build using an NVIDIA GPU!

The build process took much longer than expected — 6718.2s or **almost 2 hours** on my laptop. Plan accordingly and make sure you'll have access to power.

```
# run training step many times!
    start = time.time()
    for _ in range(2000):
        batch_xs, batch_ys = mnist.train.next_batch(100)
        sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
    end = time.time()


# .... [some lines not shown]


    correct_prediction = tf.equal(tf.argmax(y, 1), y_)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))


    print(sess.run(accuracy, feed_dict={x: mnist.test.images,
                                        y_: mnist.test.labels}))
    print("Training time: {} sec".format(end - start))
```

File: mnist_cnn.py

Training loops	Test accuracy	Time (sec) - source build	Time (sec) - default .whl	% speed improvement
300	92.33%	32.81	56.90	42.33%
400	94.10%	42.63	75.62	43.63%
500	94.37%	53.58	94.41	43.25%
1000	96.29%	106.25	193.57	45.11%

40% faster training time on MNIST CNN after building Tensorflow from source.

- Ubuntu 16.04
- conda 4.5.4
- Python 3.6.5
- Intel(R) Core(TM) i5–5200U
- 8GB DDR3

```
OCV_OPTION:

ENABLE_SSE
ENABLE_SSE2
ENABLE_SSE3
ENABLE_SSSE3
ENABLE_SSE41
ENABLE_SSE42
ENABLE_POPCNT
ENABLE_AVX
ENABLE_AVX2
ENABLE_FMA3
```

Loop Example

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

Source code

```
Loop:   L.D      F0, 0(R1)      ; F0 = array element
        ADD.D    F4, F0, F2     ; add scalar
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1, R1,# -8    ; decrement address pointer
        BNE      R1, R2, Loop   ; branch if R1 != R2
        NOP
```

Assembly code

```
Loop:   L.D      F0, 0(R1)      ; F0 = array element
        stall
        ADD.D    F4, F0, F2     ; add scalar
        stall
        stall
        S.D      F4, 0(R1)      ; store result
        DADDUI   R1, R1,# -8    ; decrement address pointer
        stall
        BNE      R1, R2, Loop   ; branch if R1 != R2
        stall
```

10-cycle schedule

Smart Schedule

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall

```
Loop:   L.D      F0, 0(R1)
        stall
        ADD.D    F4, F0, F2
        stall
        stall
        S.D      F4, 0(R1)
        DADDUI   R1, R1,# -8
        stall
        BNE      R1, R2, Loop
        stall
```

```
Loop:   L.D      F0, 0(R1)
        DADDUI   R1, R1,# -8
        ADD.D    F4, F0, F2
        stall
        BNE      R1, R2, Loop
        S.D      F4, 8(R1)
```

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10
- We were able to violate an anti-dependence easily because an immediate was involved
- Loop overhead (instrs that do book-keeping for the loop): 2
  Actual work (the ld, add.d, and s.d): 3 instrs
  Can we somehow get execution time to be 3 cycles per iteration?

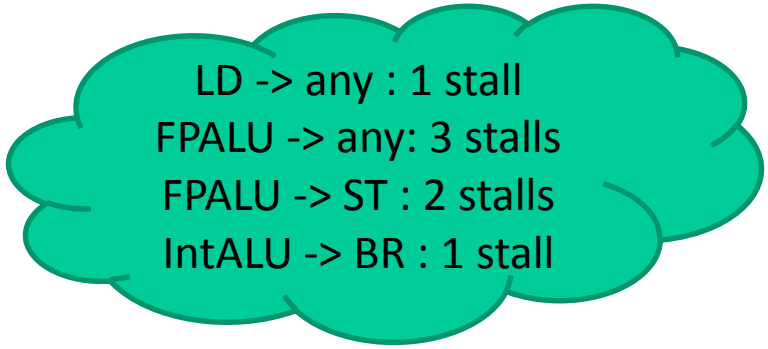
Loop Unrolling

```
Loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      F4, 0(R1)
        L.D      F6, -8(R1)
        ADD.D    F8, F6, F2
        S.D      F8, -8(R1)
        L.D      F10,-16(R1)
        ADD.D    F12, F10, F2
        S.D      F12, -16(R1)
        L.D      F14, -24(R1)
        ADD.D    F16, F14, F2
        S.D      F16, -24(R1)
        DADDUI   R1, R1, #-32
        BNE      R1,R2, Loop
```

- Loop overhead: 2 instrs; Work: 12 instrs
- How long will the above schedule take to complete?

Scheduled and Unrolled Loop

```
Loop:   L.D      F0, 0(R1)
        L.D      F6, -8(R1)
        L.D      F10,-16(R1)
        L.D      F14, -24(R1)
        ADD.D    F4, F0, F2
        ADD.D    F8, F6, F2
        ADD.D    F12, F10, F2
        ADD.D    F16, F14, F2
        S.D      F4, 0(R1)
        S.D      F8, -8(R1)
        DADDUI   R1, R1, # -32
        S.D      F12, 16(R1)
        BNE      R1,R2, Loop
        S.D      F16, 8(R1)
```



- Execution time: 14 cycles or 3.5 cycles per original iteration

Loop Unrolling

- Increases program size
- Requires more registers
- To unroll an n-iteration loop by degree k, we will need (n/k) iterations of the larger loop, followed by (n mod k) iterations of the original loop

Automating Loop Unrolling

- Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered
- Determine if unrolling will help – possible only if iterations are independent
- Determine address offsets for different loads/stores
- Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers

Superscalar Pipelines

Integer pipeline	FP pipeline
Handles L.D, S.D, ADDUI, BNE	Handles ADD.D

- What is the schedule with an unroll degree of 4?

Superscalar Pipelines

	Integer pipeline		FP pipeline	
Loop:	L.D	F0,0(R1)		
	L.D	F6,-8(R1)		
	L.D	F10,-16(R1)	ADD.D	F4,F0,F2
	L.D	F14,-24(R1)	ADD.D	F8,F6,F2
	L.D	F18,-32(R1)	ADD.D	F12,F10,F2
	S.D	F4,0(R1)	ADD.D	F16,F14,F2
	S.D	F8,-8(R1)	ADD.D	F20,F18,F2
	S.D	F12,-16(R1)		
	DADDUI	R1,R1,# -40		
	S.D	F16,16(R1)		
	BNE	R1,R2,Loop		
	S.D	F20,8(R1)		

- Need unroll by degree 5 to eliminate stalls
- The compiler may specify instructions that can be issued as one packet
- The compiler may specify a fixed number of instructions in each packet: Very Large Instruction Word (VLIW)