Optimizing Your TensorFlow Installation

•••

Jason Zaman
blog.perfinion.com
TensorFlow and Deep Learning Singapore
19th July 2018

Overview

1. Who Am I?
2. Warning!
3. Benchmarks
4. History
5. CPU Extensions
6. SIMD
7. Optimizations
8. Bazel
9. Configure
10. Build
11. GPU

Who am I?

Jason “perfinion” Zaman

Gentoo Linux Developer - SELinux and Gentoo Hardened projects.

Maintain TensorFlow and Android Studio on Gentoo.

jason@perfinion.com GPG keyid: `0x7EF137EC935B0EAF`

Blog: https://blog.perfinion.com/ Twitter: @perfinion
Github: github.com/perfinion

```
$ python cifar10_main.py
2018-07-16 22:32:09.542796: I
tensorflow/core/platform/cpu_feature_guard.cc:1
41] Your CPU supports instructions
that this TensorFlow binary was not
compiled to use: AVX2 FMA
```

Let’s see if this makes a difference

Benchmark 1:

TensorFlow official models repo:
https://github.com/tensorflow/models

official/resnet/cifar10_main.py

ResNet-32 on CIFAR-10
Batch size: 512

Time per 100 steps.

Benchmark 2:

tf.matmul() for different sized matrices.

256x256 → 16384x16384.

Median of 5 repetitions.

Testing setup

TensorFlow v1.9.0 in different configs:

Installed with pip

vs

Compiled from source

vs

GPU

Workstation:

- AMD Threadripper 1950x
  - 3.4 GHz boost to 3.7 GHz
  - 16 cores / 32 threads, 40MB Cache
- 32 GB RAM
- Nvidia GTX 1080 Ti 11GB
  - ~10 TFLOPS

Laptop:

- Intel Haswell Core i7-4600U
  - 2.1 GHz, 2 core / 4 thread, 4MB Cache
- 12 GB RAM
- No GPU

Results - ResNet-32 on a CPU

Steps	100	200	300	400	500	600	700	800	900	AVG
PIP	816.756	811.481	808.753	817.629	817.948	812.814	810.611	812.374	811.009	813.263
SRC	632.772	630.486	625.173	625.161	617.068	611.707	616.836	614.719	618.393	621.368
GPU	8.789	8.681	8.471	9.173	8.114	8.148	8.387	8.194	8.607	8.507

Units = Seconds per 100 steps.

PIP / Compiled = **1.31x speedup!**

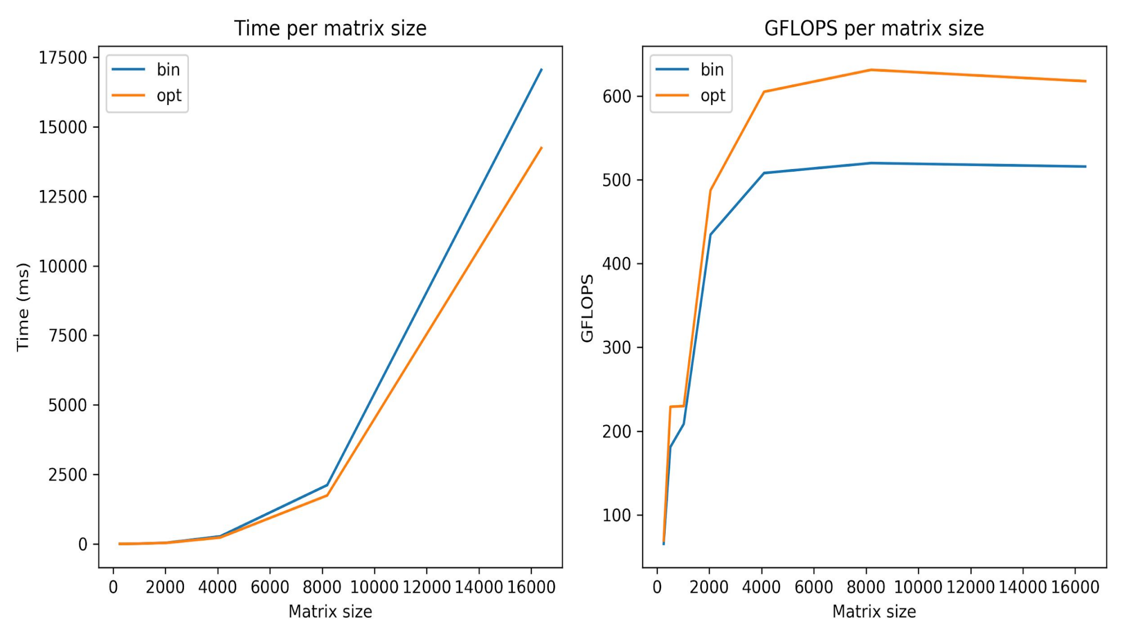
Results - Matrix multiplication on Threadripper

256x256 → 16384x16384

Optimized: 631.2 GFLOPS
Pip: 519.9 GFLOPS
GPU: 11657 GFLOPS

Optimized Speedup:

1.22x



Results - Matrix multiplication on Haswell laptop

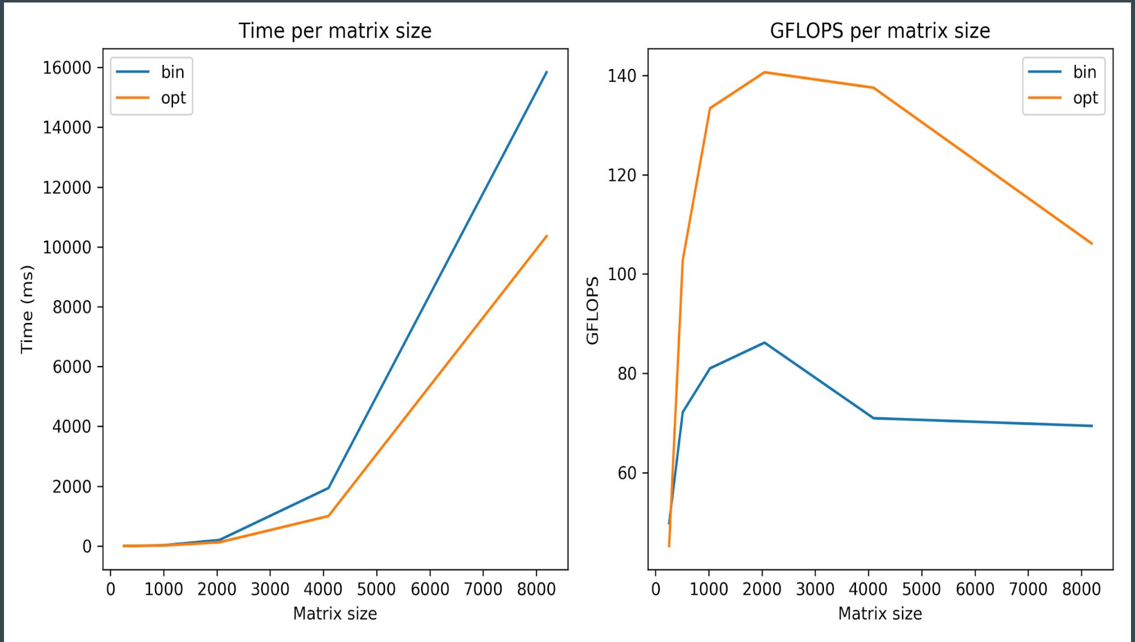
256x256 ➔ 8192x8192

Optimized: 140.6 GFLOPS

Pip: 86.2 GFLOPS

Optimized Speedup:

1.63x



Single Instruction, Single Data

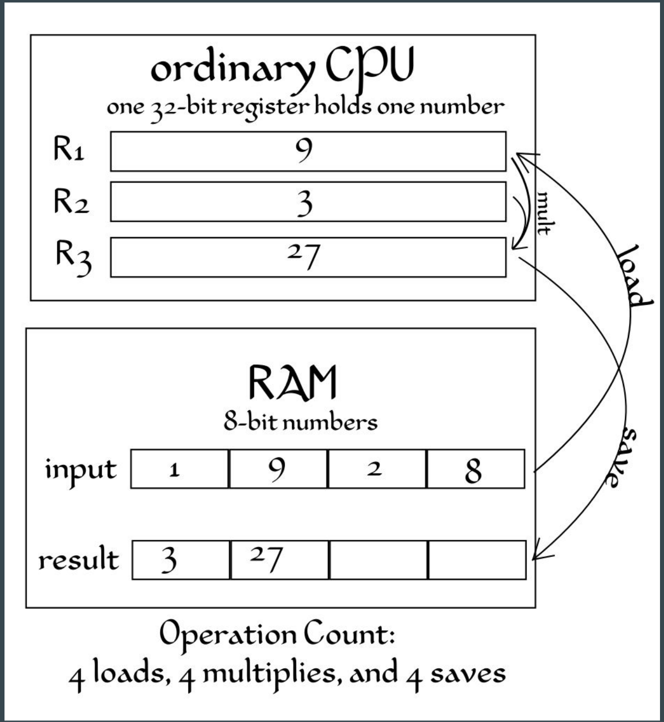
Originally CPUs would compute on data one at a time.

Multiplying two arrays:

1. Load A[0].
2. Load B[0].
3. Multiply A[0] · B[0].
4. Save result to RAM.
5. Repeat for [1], [2], [3] ...

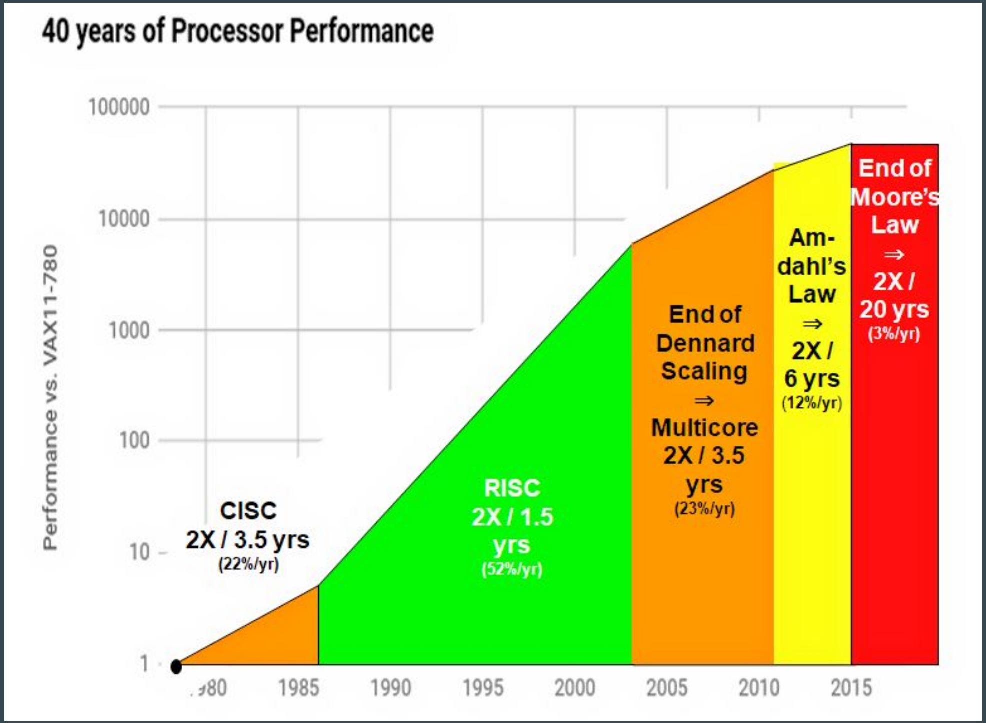
This is pretty slow. Loading and saving takes a long time.

Computers aren’t getting faster.



Computers used to get faster, FASTER

- Moore’s Law
  - “The number of transistors in a dense integrated circuit doubles about every two years.”
- Dennard Scaling
  - “As transistors get smaller their power density stays constant, so that the power use stays in proportion with area.”
- Amdahl’s Law
  - The theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.



https://www.eejournal.com/article/fifty-or-sixty-years-of-processor-developmentfor-this/

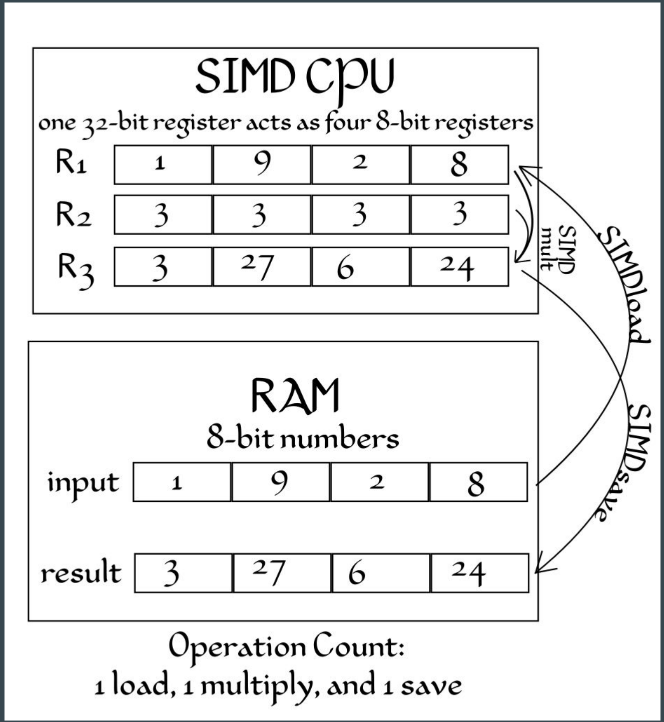
SIMD - Single Instruction, Multiple Data

Since loading and saving takes a long time, why not do it all at once?

1. Load 4 from A all at once.
2. Load 4 from B all at once.
3. Multiply 4 at once.
4. Save 4 at once.

CPUs added special-purpose instructions, originally for multimedia. First widely used was MMX in 1996 .

ARM has extensions called NEON.



CPU instruction set extensions

- Extra capabilities added to processors.
- TensorFlow checks for:
  - SSE, SSE2, SSE3, SSE4.1, SSE4.2, AVX, AVX2, FMA, AVX512F
  - There are other extensions that TF does not check for.

- Not every processor supports all extensions.
- Since TF 1.6, pip pre-built packages use up to AVX.
- My machines also support AVX2 and FMA.

AVX - Advanced Vector Extensions

`All 8 multiplies done at once:`

```
a[0] = a[0] · b[0];
a[1] = a[1] · b[1];
a[2] = a[2] · b[2];
a[3] = a[3] · b[3];
a[4] = a[4] · b[4];
a[5] = a[5] · b[5];
a[6] = a[6] · b[6];
a[7] = a[7] · b[7];
```

Enabled with `gcc -mavx` or `gcc -mavx2`.

AVX: 128-bit or 256-bit wide (4-8 fp32)

- In processors since Intel Sandybridge and AMD Bulldozer 2011.

AVX2: 256-bit wide (8 fp32)

- In processors since Intel Haswell and AMD Piledriver 2012 / 2013.

FMA - Fused Multiply-Add

For 3 * 8 floats all at once:

$a = a \cdot b + c$ a, b, c = 256-bits wide (8 fp32).

- In processors since Intel Haswell and AMD Piledriver 2012 / 2013.
- Significant help for matrix multiplication.
- 16 SP FLOPs/cycle with 8-wide FMA instruction.

Enabled with `gcc -mfma`.

https://en.wikipedia.org/wiki/FLOPS#FLOPs_per_cycle_for_various_processors

Native optimization flags

In addition to AVX, FMA etc, compilers have tons of tuning parameters.
Luckily gcc can just “figure it out”.

```
$ gcc -march=native -E -v - </dev/null 2>&1 | grep cc1
/usr/libexec/gcc/x86_64-pc-linux-gnu/7.3.0/cc1 -E -quiet -v - -march=znver1 -mmmx -mno-3dnow -msse
-msse2 -msse3 -mssse3 -msse4a -mcx16 -msahf -mmovbe -maes -msha -mpclmul -mpopcnt -mabm -mno-lwp
-mfma -mno-fma4 -mno-xop -mbmi -mno-sgx -mbmi2 -mno-tbm -mavx -mavx2 -msse4.2 -msse4.1 -mlzcnt
-mno-rtm -mno-hle -mrdrnd -mf16c -mfsgsbase -mrdseed -mprfchw -madx -mfxsr -mxsave -mxsaveopt
-mno-avx512f -mno-avx512er -mno-avx512cd -mno-avx512pf -mno-prefetchwt1 -mclflushopt -mxsavec
-mxsaves -mno-avx512dq -mno-avx512bw -mno-avx512vl -mno-avx512ifma -mno-avx512vbmi
-mno-avx5124fmaps -mno-avx5124vnniw -mno-clwb -mmwaitx -mclzero -mno-pku -mno-rdpid --param
l1-cache-size=32 --param l1-cache-line-size=64 --param l2-cache-size=512 -mtune=znver1
```

GCC optimization levels

`gcc -O<number>` (O the letter, not the number).

`-O0:` Turns off optimization entirely. Fast compile, good for debugging. **Default**.

`-O1:` Basic optimization level.

`-O2:` Recommended for most things. SSE / AVX *may* be used, but not fully.

`-O3:` Highest optimization possible. Also vectorizes loops, can use all AVX registers.

`-Os:` Small size. Basically enables -O2 options which do not increase size. Can be useful for machines that have limited storage and/or CPUs with small cache sizes.

What flags to use?

If you are building on the same machine that will be running:

```
-O3 -march=native
```

If you are building on a different machine: find out which flags, then set those manually:

```
-O3 -march=skylake -msse -msse2 -msse3 -mssse3 -msse4.1
-msse4.2 -mfma -mavx -mavx2 -mno-avx512f
```

Okay, so how is it built?

TensorFlow is built using Bazel, a build system developed by Google that is fast, scalable and correct.

The problem is, it's mostly unknown outside of Google.

`WORKSPACE` in the root of the tree. `BUILD` files contain the rules. Supports lots of languages. Extensions written in Skylark (similar to python).

https://bazel.build/

```
$ bazel build //main:helloworld
```

Configure TensorFlow

```
$ git clone https://github.com/tensorflow/tensorflow.git
$ cd tensorflow; git checkout v1.9.0
$ ./configure
Do you wish to build TensorFlow with CUDA support? [y/N]: n
No CUDA support will be enabled for TensorFlow.

Please specify optimization flags to use during compilation when
bazel option "--config=opt" is specified [Default is
-march=native]: -O3 -march=native
```

Build and install TensorFlow

```
$ bazel build --config=opt \
    //tensorflow/tools/pip_package:build_pip_package \
    //tensorflow:libtensorflow_framework.so \
    //tensorflow:libtensorflow.so

$ bazel-bin/tensorflow/tools/pip_package/build_pip_package /tmp/tf/
$ pip install /tmp/tf/tensorflow-*.whl
```

On Gentoo, I’ve already done all the work:

```
# emerge tensorflow
```

But I have a GPU, why bother?

Nvidia CUDA has different compute capabilities, defaults to “3.5,5.2”.

If you are using a GPU you’d still want the most optimized version.

Not everything can run on the GPU.

Input pipeline is still on the CPU.

<table>
<tr><th rowspan="2">Feature support (unlisted features are supported for all compute abilities)</th><th colspan="11">Compute capability (version)</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.2</th><th>3.5, 3.7, 5.0, 5.2</th><th>5.3</th><th>6.x</th><th>7.x</th></tr>
<tr><td>Integer atomic functions operating on 32-bit words in global memory</td><td rowspan="2">No</td><td colspan="10" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory</td></tr>
<tr><td>Integer atomic functions operating on 32-bit words in shared memory</td><td colspan="2" rowspan="4">No</td><td colspan="9" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory</td></tr>
<tr><td>Integer atomic functions operating on 64-bit words in global memory</td></tr>
<tr><td>Warp vote functions</td></tr>
<tr><td>Double-precision floating-point operations</td><td colspan="3">No</td><td colspan="8">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory</td><td colspan="4" rowspan="7">No</td><td colspan="7" rowspan="7">Yes</td></tr>
<tr><td>Floating-point atomic addition operating on 32-bit words in global and shared memory</td></tr>
<tr><td>_ballot()</td></tr>
<tr><td>_threadfence_system()</td></tr>
<tr><td>_syncthreads_count(), _syncthreads_and(), _syncthreads_or()</td></tr>
<tr><td>Surface functions</td></tr>
<tr><td>3D grid of thread block</td></tr>
<tr><td>Warp shuffle functions</td><td colspan="5">No</td><td colspan="6">Yes</td></tr>
<tr><td>Funnel shift</td><td colspan="6">No</td><td colspan="5">Yes</td></tr>
<tr><td>Dynamic parallelism</td><td colspan="7">No</td><td colspan="4">Yes</td></tr>
<tr><td>Half-precision floating-point operations:
addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="8">No</td><td colspan="3">Yes</td></tr>
<tr><td>Atomic addition operating on 64-bit floating point values in global memory and shared memory</td><td colspan="9">No</td><td colspan="2">Yes</td></tr>
<tr><td>Tensor core</td><td colspan="10">No</td><td>Yes</td></tr>
</table>

More info

Slides will be on my blog: https://blog.perfinion.com/

Bazel: https://bazel.build/

TF Install docs: https://www.tensorflow.org/install/install_sources

Official models: https://github.com/tensorflow/models/tree/master/official/resnet

Gentoo package: https://packages.gentoo.org/packages/sci-libs/tensorflow

Sneak peak at future TF build changes

I submitted PR 20284 to use system libraries instead of rebuilding everything statically. Makes building easier and faster.

```
# apt-get install libjpeg-turbo8 libjpeg-turbo8-dev zlib1g zlib1g-dev libsnappy1v5 \
    libsnappy-dev libre2-4 libre2-dev

$ bazel build --verbose_failures --config=opt \
    --action_env TF_SYSTEM_LIBS=”com_googlesource_code_re2,jpeg,snappy,zlib_archive”
    //tensorflow/tools/pip_package:build_pip_package \
    //tensorflow:libtensorflow_framework.so \
    //tensorflow:libtensorflow.so
```

On Gentoo: enable the `system-libs` USE-flag.