Tuning Deep Learning Episode 1: DeepMind's A3C in Torch

In February, a new paper from Google's DeepMind team appeared on arxiv. This one was interesting – they showed dramatically improved performance and training time of their Atari-playing Deep Q-Learning network. The training speedup was so great that 16 CPU cores outperformed their previous GPU results by an order of magnitude.

Having access to a range of machines with large numbers of CPUs I was, of course, intrigued by the possibilities. The Torch machine learning framework has already added an implementation of the agents described in the new paper and I wanted to see how well it performed and what kind of hardware would suit it best.

Part 1: What Does Good Look Like?

Downloading and installing Torch and Kaixhin's implementation of the paper's Asynchronous Advantage Actor-Critic algorithm was straightforward and my 8-core laptop learned to play the Catch demo at 1,590 simulation steps/second.

Is 1,590 steps/second good? I don't know! The CPU was pegged out, so maybe yes? For comparison I ran the same test on one of our 24-core Intel Xeon test machines. The results, as expected, were better:
But wait, a 25% speedup for going from an 8-core laptop to a 24-core server doesn't sound right! What's going on here?

Time for a Performance Report. Allinea’s tuning and analysis tool shows the breakdown of how an application uses its time without requiring instrumentation.

On each system I put "perf-report" in front of Torch’s "luajit" command (Torch is written in Lua with a lot of supporting C libraries) and compared the results:

**Summary:** luajit is **Compute-bound** in this configuration

**CPU**
A breakdown of the 100.0% CPU time:
- Scalar numeric ops: 42.7%
- Vector numeric ops: 4.0%
- Memory accesses: 53.1%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

**MPI**
A breakdown of the 0.0% MPI time:
- Time in collective calls: 0.0%
- Time in point-to-point calls: 0.0%
- Effective process collective rate: 0.00 bytes/s
- Effective process point-to-point rate: 0.00 bytes/s

No time is spent in MPI operations. There’s nothing to optimize here!

**I/O**
A breakdown of the 0.0% I/O time:
- Time in reads: 100.0%
- Time in writes: 0.0%
- Effective process read rate: 540 MB/s
- Effective process write rate: 0.00 MB/s

No time is spent in I/O operations. There’s nothing to optimize here!

**Threads**
A breakdown of how multiple threads were used:
- Computation: 50.0%
- Synchronization: 1.0%
- Physical core utilization: 100.0%
- System load: 32.0%

The system load is high. Check that other jobs or system processes are not running on the same nodes.

**Memory**

**Energy**
The above is the result from my laptop. Sure enough it spends all its time computing and uses all 8 hyperthreaded cores to the max.

Look at the CPU section more closely - only 4.8% of the time is spent in efficient AVX vector instructions and the rest is all doing scalar arithmetic! This leaves a lot of performance on the table. As most neural networks operations are essentially implemented by a series of matrix multiplications I expected this to be much higher.

The server’s results are enlighten us further:
Here the default Torch options have failed to vectorize any loops whatsoever! And in both cases over half the time is spent waiting for memory accesses. The higher clock speed and increased core count of the server won't help if it's waiting for slow DRAM accesses.

To see if I can improve matters I took a look at a MAP profile of one of the runs (replacing 'perf-report' with 'map --profile' in front of the 'luajit' command):

The heavy lifting (AddMM) is being performed by a custom sgemm implementation instead of a high-performance hardware-tuned version. So why did the server perform worse on a core-for-core basis? Why couldn't it make any use of the AVX vector instructions required to approach peak CPU performance?
Here the Torch install process has used a different BLAS implementation – and an inferior one at that!

Part 2: BLASted Libraries!

There are many different high-performance matrix multiplication libraries available. Each has its own quirks and benefits. I've had good results with Intel's MKL library on Xeon CPUs before and decided to try it again here.

Telling torch to use the Intel MKL library for matrix multiplication was straightforward and the results on one of our 24-core test nodes were much improved:

Now we're talking! That's why it's always worth remembering all of Donald Knuth's
famous quote:

"[about 97% of the time] premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%”.

So are we done now? Let's look at the Performance Report of the new optimized version:

<table>
<thead>
<tr>
<th>Summary: luajit is Compute-bound in this configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute: 89.0% Time spent running application code. High values are usually good. This is high, check the CPU performance section for advice.</td>
</tr>
<tr>
<td>MPI: 0.0% Time spent in MPI calls. High values are usually bad. This is very low, this code may benefit from a higher process count.</td>
</tr>
<tr>
<td>I/O: 11.0% Time spent in filesystem I/O. High values are usually bad. This is low, check the I/O breakdown section for optimization advice.</td>
</tr>
</tbody>
</table>

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below. As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

**CPU**

A breakdown of the 89.0% CPU time:

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-core code</td>
<td>99.7%</td>
</tr>
<tr>
<td>OpenMP regions</td>
<td>0.3%</td>
</tr>
<tr>
<td>Scalar numeric ops</td>
<td>10.0%</td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td>5.7%</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>73.3%</td>
</tr>
</tbody>
</table>

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

**MPI**

A breakdown of the 0.0% MPI time:

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in collective calls</td>
<td>0.0%</td>
</tr>
<tr>
<td>Time in point-to-point calls</td>
<td>0.0%</td>
</tr>
<tr>
<td>Effective process collective rate</td>
<td>0.00 bytes/s</td>
</tr>
<tr>
<td>Effective process point-to-point rate</td>
<td>0.00 bytes/s</td>
</tr>
</tbody>
</table>

No time is spent in MPI operations. There's nothing to optimize here!

**I/O**

A breakdown of the 11.0% I/O time:

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in reads</td>
<td>78.0%</td>
</tr>
<tr>
<td>Time in writes</td>
<td>21.9%</td>
</tr>
<tr>
<td>Effective process read rate</td>
<td>31.4 MB/s</td>
</tr>
<tr>
<td>Effective process write rate</td>
<td>117 KB/s</td>
</tr>
</tbody>
</table>

Most of the time is spent in read operations with a low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

**OpenMP**

A breakdown of the 0.9% time in OpenMP regions:

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>95.5%</td>
</tr>
<tr>
<td>Synchronization</td>
<td>4.5%</td>
</tr>
<tr>
<td>Physical core utilization</td>
<td>100.0%</td>
</tr>
<tr>
<td>System load</td>
<td>149.0%</td>
</tr>
</tbody>
</table>

Significant time is spent synchronizing threads in parallel regions. Check the affected regions with a profiler.

**Memory**

Per-process memory usage may also affect scaling:

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean process memory usage</td>
<td>361 MB</td>
</tr>
<tr>
<td>Peak process memory usage</td>
<td>629 MB</td>
</tr>
<tr>
<td>Peak node memory usage</td>
<td>19.0%</td>
</tr>
</tbody>
</table>

There is significant variation between peak and mean memory usage. This may be a sign of workloads imbalance or a memory leak. The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

**Energy**

A breakdown of how the 0.446 Wh was used:

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>not supported%</td>
</tr>
<tr>
<td>System</td>
<td>100.0%</td>
</tr>
<tr>
<td>Mean node power</td>
<td>107 W</td>
</tr>
<tr>
<td>Peak node power</td>
<td>112 W</td>
</tr>
</tbody>
</table>

CPU metrics are not supported (no intel_mpl module)

The MKL multiplication kernels are accomplishing the same amount of work in much less time – the power of using the CPU's wide vector units. Now the time spent in memory accesses really dominates the computation! This has become a
memory-bound problem. Does MAP give us any hints about how to improve this?

Indeed, the vectorized matrix operations are so fast that a significant amount of time is spent now inside memory-heavy Lua script code, such as the luaT_toudata, luaT_checkudata and lua_call functions in the screenshot above.

We could try to optimize the Lua code that glues the matrix multiplications together, but at that stage you're usually better off rewriting them in C in the first place.

The whole point of a scripting language is that it's easy to write and maintain and the overhead is ideally negligible because there's so much work to do in the deep math routines.

In this case the real question is: does the ratio of time spent inside the math routines increase when we start running a more interestingly-sized model? Let's move away from the toy Catch example and use the same code to training a larger network to play Atari Pong from raw pixel data.

Part 3: A Larger Network

We used the same code to train a larger a network on Atari Pong. This was clearly going to take a rather long time on my laptop, managing just 128 simulation steps per
second. Here's how the two machines stacked up:

We can see that the Xeon maintains a strong performance advantage, as expected – note that in this and later results the MKL library is always used where available. Previously the Lua script was becoming a bottleneck. Has this changed with the larger run?
Yes – it's not the Lua code that's the bottleneck now. It's a range of hand-coded matrix max and sqrt routines (THFloatTensor_max in the screenshot above and THFloatTensor_sqrt below):

The common factor in these macros is inefficient memory access patterns coupled with zero vectorization, seen in the time breakdown for the highlighted line in the right-hand panel.

Additionally some spend significant time deallocating memory inside such macros, here seen by the THFree time in the Stacks display below:
Let's have a look at one of these macros in more detail:

No wonder the compiler can't vectorize this loop with all this branching going on. This is a very inefficient way to apply a max or sqrt operation to a matrix, assuming that is intended. We can also see that the 5% time spent in the free call is entirely unnecessary:
The highlighted line above allocates a small amount from the heap and frees it again at the end of the macro. Allocating from the heap requires thread locking! There’s no need to allocate this small array of counters on the heap unless the number of dimensions is unusually large.

There are clearly some excellent optimization targets in the Torch library. But why hasn’t Facebook – or the wider ML community – found them before?

The answer is that machine learning frameworks are becoming little operating systems in their own right, with scheduling, cross-platform optimization and many other complexities.

Here we have exactly the same Lua code driving two differently-sized neural networks and we’re seeing a completely different set of performance constraints between the models.

If you want to have good performance, you need to be ready to look under the covers and see where the bottlenecks for your model are.

**Part 4: A Hardware Solution**

Ok, so we’ve taken the low-hanging fruit from the computational optimization already. We have a pretty well-parallelized algorithm which, unfortunately, is memory bound. Is there different hardware that would run this model better, without rewriting parts of Torch?

Well, the Intel Xeon Phi Knight's Landing has 16GB high-bandwidth memory close to the cores. The memory sections in our Performance Reports tell us that our dataset would fit entirely within that. As thread synchronization is not appearing as a bottleneck we can expect to scale performance to use its larger number of cores too.

This is a valuable prediction, so let's run the experiment. Can we get better
Yes we can! Each core in the new-generation Intel Xeon Phi is less powerful than a normal Xeon, but that's made up for in three ways:

1. **Memory bandwidth** – the high-bandwidth memory close to the CPU can improve a lot of day-to-day application workloads.

2. **Wider vector units** – the new AVX 512 instruction set offers much higher peak performance for well-tuned codes.

3. **Quantity** – with 72 cores per chip (68 on our engineering sample) and 4-way hyperthreading the old Russian proverb “Quantity has a quality all of its own” has never been more appropriate!

**Conclusions**

We analyzed the performance of a top machine learning library, Torch, on a new CPU-friendly training method published by Google's DeepMind team.

Performance Reports showed us that the default Torch install was badly tuned for numerical computation and MAP pinpointed the SGEMM routines. Replacing those with hardware-optimized ones boosted performance by 3x.
Allinea MAP now showed that more time was spent running Lua script code than performing numerical computation, a sign that the problem size was too small to run efficiently on Torch. We moved up to a larger scale problem (training a network to play Atari Pong) and saw that changing the layout of the neural networks changed the bottlenecks hit in the implementation. Now several inefficient hand-coded matrix routines inside macros were memory-bound.

We now had a choice – rewrite core Torch code or run on a system with faster memory performance. The Performance Reports results suggested the data would fit inside the high-bandwidth memory on an Intel Xeon Phi Knight's Landing. When we ran on the new architecture we achieved a further 2.33x speedup.

Our conclusion is that Performance Reports and Allinea MAP will help data scientists and developers determine whether their deep learning algorithms are running efficiently or are hitting limitations in your current framework.

Their results will help you choose better libraries, frameworks and hardware or instances to match the needs of your current network topology, giving order-of-magnitude speedups on cutting-edge algorithms.

A key problem facing researchers and companies is that data scientists are domain experts, not low-level performance engineers. Allinea Performance Reports is the tool they need to get the most out of their machine learning frameworks.