Improving Single-GPU Performance for Deep Q-Networks

15-418 Final Project

Rishub Jain (rishubj), Hariank Muthakana (hmuthaka)

May 9, 2018

1 Summary

We parallelized multiple reinforcement learning neural network tasks that have significant GPU and CPU usage on the same GPU and CPU by interleaving computation, which is currently not done efficiently in existing neural network frameworks. Our algorithm consistently outperformed various Tensorflow baselines for varying numbers of parallel models trained and varying amounts of environment latency.

2 Background

Often researchers need to train several Machine Learning (ML) models in order to evaluate different hyper-parameters, algorithms, and environments. If a researcher wants to train multiple large models and only has a single GPU, they would have to run each of their training jobs sequentially. In this scenario, using all resources available between the CPU and GPU is of utmost importance. Many models in deep reinforcement learning require full use of the GPU and one thread of the CPU, but not at the same time. These models in fact often require more time on the CPU than the GPU. If we have multiple models to train, we could interleave the GPU and CPU computations of multiple tasks, such that the GPU and all threads of the CPU are being mostly used at all times.

The deep Q-network (DQN) is a landmark method in deep reinforcement learning, allowing for superhuman-level learning in complex environments such as Atari games. A key characteristic of such networks is the heavy CPU computation required to simulate the environment alongside training. We developed and implemented a CPU/GPU computation interleaving algorithm to speed up training an arbitrary number of DQNs, and benchmarked against Tensorflow [6], the most popular neural network framework.

2.1 DQN

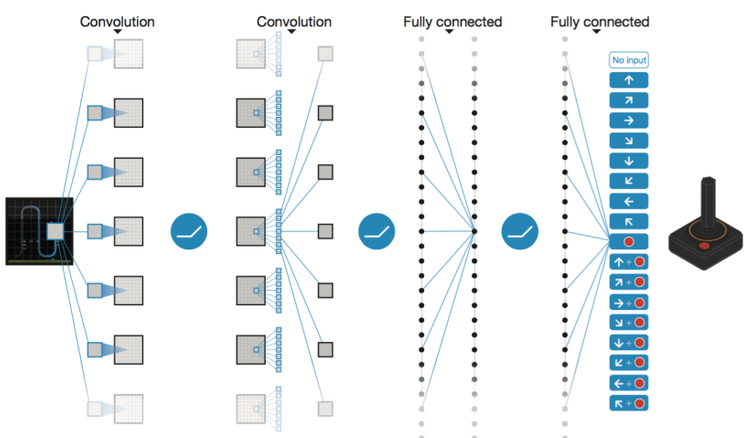

Figure 1: Deep Q-Network

In reinforcement learning problems, we seek to learn from experience rather than labeled data. The standard formulation is an environment where at each step we take an action and get a reward. DQN [1] is a reinforcement learning algorithm that combines a deep neural network and an existing RL algorithm called Q-learning to maximize reward on complex tasks. It essentially uses the network to estimate the "Q-function", a measure of how valuable taking a certain action at a certain state is, given the state. This is illustrated in Figure 1.

One major difference between DQN (and RL algorithms in general) and standard deep learning tasks is the amount of CPU computation required per iteration. In RL tasks we generally have to step through the environment once at every iteration. If the environment is complex, like if it is simulated within a physics engine, this can quickly bottleneck the entire algorithm. In the case of DQN, the environment does not have to be simulated before training takes place because of a feature called *replay memory*. When we step through the environment and receive rewards, the algorithm logs its experience in memory - and during training, it merely samples batches from that memory. This motivates our algorithm which uses message-based parallelism to train on the GPU and simulate multiple environments on the CPU simultaneously.

2.2 Tensorflow Limitations

If one wanted to train only one DQN, using Tensorflow, or a different popular Neural Network framework, would likely be the superior option. However, when training multiple models, Tensorflow has many limitations. There is an option to train multiple models on the same GPU within Tensorflow, and this is done by partitioning the GPU such that each job gets a section of the memory. Then, the GPU would share its time spent among the different jobs. This leads to the GPU often context switching between jobs, which can be inefficient. Also, since each job cannot access the entire memory of the GPU, if the network computation requires more memory, those Tensorflow jobs will simply fail. This led us to implement our own algorithm for improving performance.

3 Approach

3.1 Algorithm Description

Our approach to decrease total training time of these Deep Reinforcement Learning problems that involve high latency environments was to eliminate times when the CPU or GPU are idle. During the training of a single DQN, the CPU is idle while the GPU is performing training operations on the network, and the GPU is idle when the CPU is stepping through the environment. While these resources are idle, we thought of performing the respective operations of other ML models we need to train

Initially, we had thought of only training two models, and interleaving their CPU and GPU computations. However, with more thought we had realized that each computation on the CPU could be done in parallel, since computer setups often have multiple cores but only 1 GPU. Also, we had considered trying to run the training steps in parallel on the GPU using NVIDIA's Multi-Process Service, which we could use to assign blocks of GPU cores to different tasks. However, the GPU's memory would be partitioned among the different training jobs, which could lead to issues. For example, if someone is running 3 models in parallel on a 6 GB CPU, each model would only have access to 2 GB, which could be a problem when training large networks.

Thus, we developed our final algorithm to train k DQN's, pictured in Figure 2. For each training step, we first have to run a forward pass through each DQN to obtain the next action the agent in each environment will take. Then, there is 1 thread sequentially running the training steps for each DQN. There are k other threads, each computing a step in the environment.

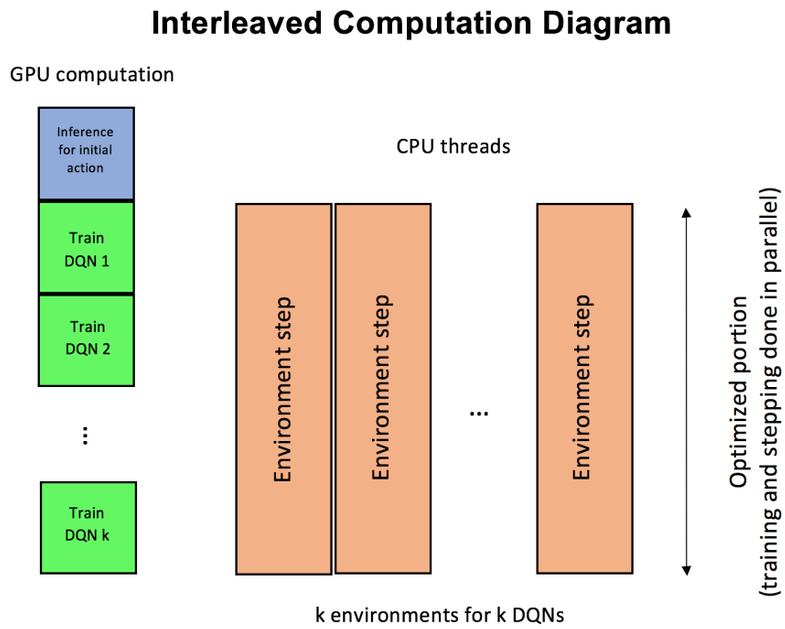

Figure 2: Interleaved Environment Stepping and Training

3.2 Hardware/Libraries Description

For all experiments we used a NVIDIA GTX 1060 6GB GPU card. The machine we trained on had an AMD Ryzen 3.2GHz 6-Core processor and 16GB of RAM. We used Tensorflow 1.5.0, CUDA 8.0, and cuDNN 6.0.

3.3 DQN Implementation

We found an existing Tensorflow implementation of DQN [7] and modified it to suit our needs. This served as both our baseline as well as scaffolding code for our interleaving algorithm.

This implementation learns how to play Atari games, specifically the game "Ms. Pacman" pictured in Figure 3. To simulate the Atari game, we used the OpenAI Gym [8] which is an open-source RL library. It provides useful abstractions that allow us to the game as an environment with states, actions, and rewards.

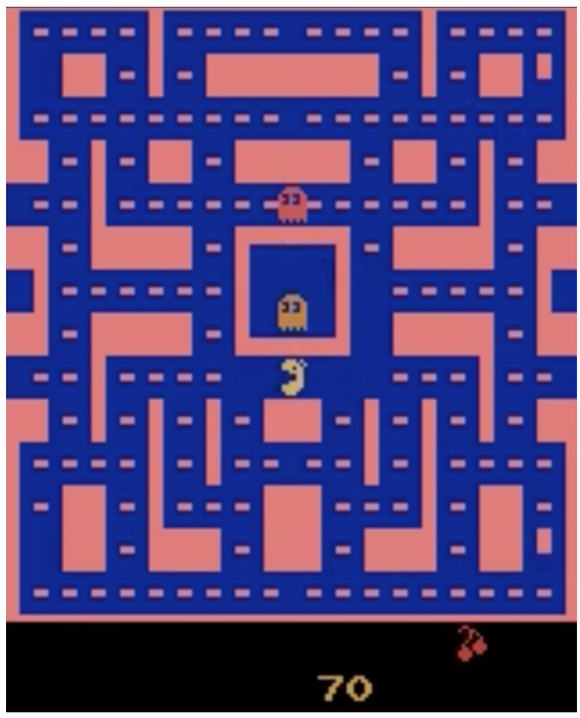

Figure 3: MsPacman Environment

3.4 Initial Benchmarking

Our first experiment was to benchmark the DQN implementation to investigate how Tensorflow handles parallel training. We trained 2 DQNs in parallel and compared it to sequential training. We used two different environments (the only major differnece being Pong having a step time of 0.47ms and MsPacman 1.17ms) and calculated speedup:

Environment	Steps	Sequential (s)	Parallel (s)	Speedup
MsPacman	10k	489.154	263.199	1.858X
MsPacman	100k	4371.261	2373.803	1.841X
Pong	10k	409.898	232.933	1.76X
Pong	100k	3721.811	2129.885	1.745X

We found that in general Tensorflow is able to parallelize some of the training. It helped that the networks we used were small enough such that two would easily fit on the GPU. There seemed to be overhead from Tensorflow and running the environments, which caused less than 2X speedup.

Next, we tried restricting the GPU memory for each training session, but we found that the speedup did not get worse as memory decreased:

Sequential (s)	Parallel (s)	Speedup	GPU Mem Used (Gb)	# models trained
1076.45952	Out of Memory	N/A	0.15	4
572.737	318.611	1.797X	0.3	2
536.579	313.426	1.711X	0.6	2
543.126	315.316	1.722X	1.2	2
542.393	326.742	1.660X	2.4	2

This also showed that Tensorflow did in fact run out of memory when running the models in parallel, but not running each sequentially. This again was motivation for our current algorithm that only allows one training task at a time on the GPU.

3.5 Neural Network Construction

We needed to be able to have specific functionality over the neural networks so that we can adhere to our algorithmic description. This includes being able to allocate and free memory on the GPU, so that forward and backward passes can be run on the GPU sequentially without keeping temporary memory. Also, we need to be able to run CPU and GPU processes simultaneously that communicate shared information. Tensorflow lacks much of this functionality, as described in the sections above. Also, once Tensorflow allocates memory, there is no way to users to free that memory, which is a feature they maintain to reduce memory fragmentation. Thus, we chose to implement our network from the base CUDA primitives. This also resulted in the added bonus of avoiding much of the overhead involved with generating the Tensorflow graphs. Since we also needed to use the Open AI environment which only has a Python API we were constrained to use Python for our Neural Network Construction.

For the construction of the Neural Network, we used several libraries that are only wrappers that directly use the CUDA primitives. We used the library "cudnn-python-wrappers" [3] to use the cuDNN primitives [9] for the convolutional layers of the DQN, and the scikit-cuda [10] library to use the cuBLAS primitives [11] for the activation functions and fully connected layers of the DQN. We also used PyCUDA [4] for GPU memory management.

3.6 Interleaving with mpi4py

In order to interleave CPU and GPU computation, and interface with both the Python RL environment and the cuDNN wrappers, we needed a way to do multithreading in Python. Our first attempt used Python's built-in multiprocessing library. However, we found that we needed to communicate between threads and the provided primitives were difficult to use. mpi4py [5] was an easier solution to our problems. We initially tried to use point-to-point communication to send GPU memory pointers, but quickly discovered that Python processes have their own address spaces in the GPU. We ended up using MPI's shared memory functionality to avoid sending large arrays.

Our algorithm proceeds as follows. Before training, we spawn $k + 1$ MPI processes given that we are training k DQN, and set up shared memory arrays with sufficient space. One process does all the GPU work, and this one makes the k networks. The other processes step through k environments. At each training step, the environment threads fill shared memory with the current states. In parallel, the GPU thread runs the states through the networks to produce the next actions while the environment threads sample from their replay buffers and fill shared memory with the next training batches. Finally, in parallel the environment threads step through the next actions while the GPU thread trains the networks sequentially. Figure 2 illustrates this process, with the bulk of the optimization happening when the environments are stepping while training occurs.

4 Results

We ran two different types of experiments: the speedup of multiple DQNs on a fixed environment, and the speedup of 2 DQNs vs. step latency, which we artificially increase by adding a small delay after each step. Our algorithm achieves a consistent speedup over sequential Tensorflow training for an arbitrary number of DQNs, and a consistent speedup over parallel Tensorflow training for increasing step delays.

4.1 Multiple DQNs

To test how well our algorithm parallelizes across DQN training, we trained increasing numbers of DQNs using the full 6GB of available memory for 3000 iterations each. Our baseline was training of the DQNs in Tensorflow. Figure 4 shows that we achieved a consistent speedup against sequential training with Tensorflow, increasing until 3 DQNs and tapering off from there. Figure 5 shows our speedup compared to parallel training with Tensorflow (with multiple processes). We are able to achieve a speedup when compared against Tensorflow only when training 1 and 2 DQNs. This is likely because of Tensorflow's internal optimizations that might scale better with increased amounts of computation and load on the CPU, GPU, and memory. Also our algorithm, as shown in Figure 2, depends on the time it takes to step through the environment to be roughly equal to the time it takes to train the multiple DQNs. As the number of DQNs increase, this time is skewed more towards the training computation, and the other threads are left waiting for it to finish.

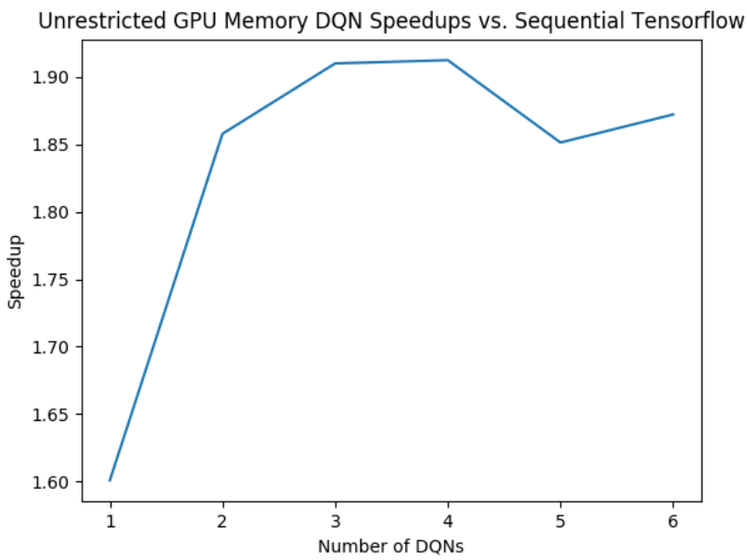

Figure 4: Sequential Speedup with Unrestricted Memory

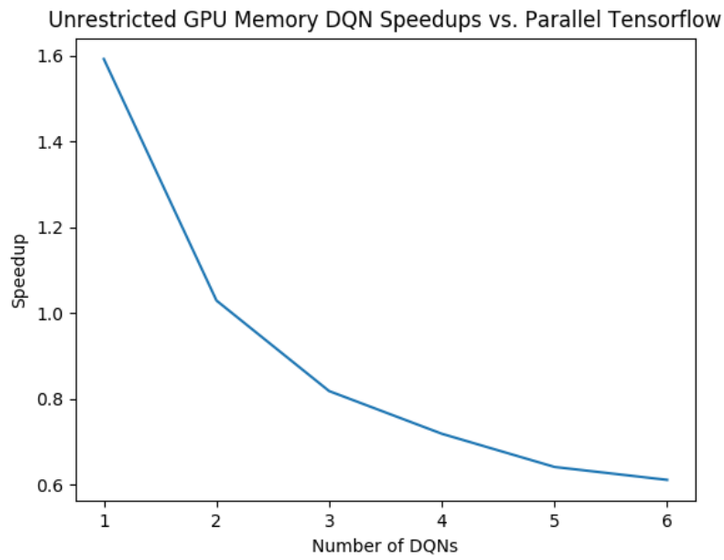

Figure 5: Parallel Speedup with Unrestricted Memory

4.2 GPU memory experiments

In order to see if our method provided any memory optimization, we restricted the GPU memory to a 0.025 fraction of the full 6GB per DQN. This is sufficient to hold the network parameters, all gradients, and other space required for training, since our cuDNN training code was able to run for an arbitrary number of DQNs. However, parallel Tensorflow training failed with any more than 2 DQNs, because Tensorflow consistently allocates more space per process (probably to hold its training graph). For example, with 3 DQNs Tensorflow requests around 289Mb per process which is more than 0.025 of 6GB. Also, the majority of a train step's memory usage comes from the temporary storing of gradients to be used during backpropagation. Our algorithm runs each train step for each DQN sequentially, where Tensorflow divides the total memory into chunks and every process can only access memory in its chunk. So, our method is able to run in parallel even in the presence of strict GPU memory requirements.

Figure 5 shows our speedup compared to sequential Tensorflow training, and we achieved very similar results to the unrestricted memory experiments. This mirrored our results from initial Tensorflow benchmarking, where we found that speedup did not get worse as memory decreased, as long as there was enough memory per process.

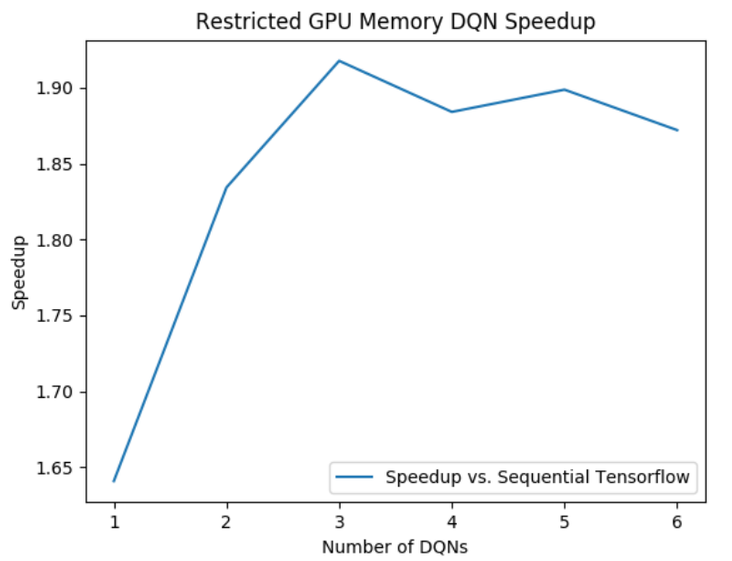

Figure 6: Sequential Speedup, Restricted

4.3 Increasing Environment Simulation Time

Finally, we measured speedup as a function of environment step time. This was crucial in determining if our algorithm was useful for DQN training in complex environments and if it could be extended to CPU-heavy deep learning tasks. Since the MsPacman environment has a fairly fast step time (1ms), we artificially added a sleep time to each environment step and trained 2 DQNs. Our algorithm was able to achieve a speedup over parallel Tensorflow training for all step delay amounts.

Figure 7 shows the results of our experiment. The speedup increases up until an added delay of 0.01 seconds, and then decreases. This is likely because there is a "sweet spot" of the time it takes for an environment to step in our algorithm. For low latency environments, the training steps take longer than the environment steps, and the CPU remains idle for some time each iteration. For high latency environments, each environment step takes longer than the combined training steps on the GPU, leaving the GPU idle for some time. Thus, there is a balance between these two times, and our algorithm performs the best when they are equal.

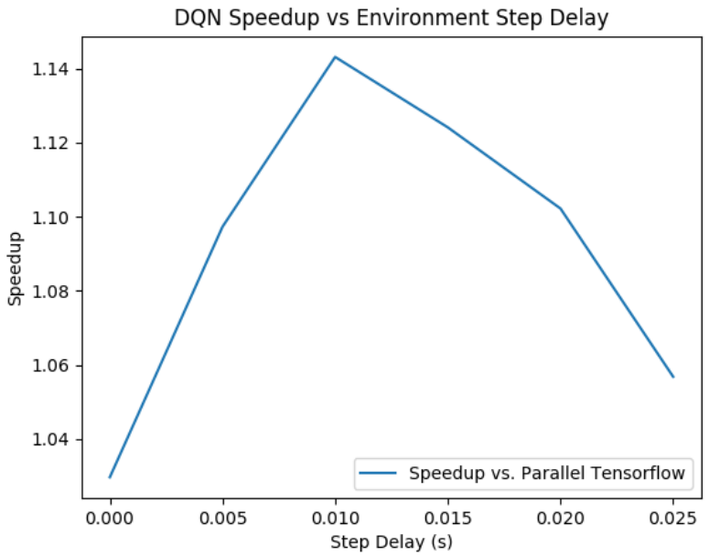

Figure 7: Environment Delay Speedup

5 References

1. https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf
2. https://github.com/tbennun/cudnn-training
3. https://github.com/hannes-brt/cudnn-python-wrappers
4. https://mathema.tician.de/software/pycuda
5. http://mpi4py.scipy.org/
6. https://www.tensorflow.org/
7. https://github.com/ageron/tiny-dqn
8. https://gym.openai.com/
9. https://developer.nvidia.com/cudnn
10. https://github.com/lebedov/scikit-cuda/tree/master/skcuda
11. https://developer.nvidia.com/cublas

6 Divison of Work

Equal work was performed by both project members.