REINFORCEMENT LEARNING THROUGH ASYNCHRONOUS ADVANTAGE ACTOR-CRITIC ON A GPU

Mohammad Babaeizadeh
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
mb2@uiuc.edu

Iuri Frosio, Stephen Tyree, Jason Clemons, Jan Kautz
NVIDIA, USA
{ifrosio,styree,jclemons,jkautz}@nvidia.com

ABSTRACT

We introduce a hybrid CPU/GPU version of the Asynchronous Advantage Actor-Critic (A3C) algorithm, currently the state-of-the-art method in reinforcement learning for various gaming tasks. We analyze its computational traits and concentrate on aspects critical to leveraging the GPU's computational power. We introduce a system of queues and a dynamic scheduling strategy, potentially helpful for other asynchronous algorithms as well. Our hybrid CPU/GPU version of A3C, based on TensorFlow, achieves a significant speed up compared to a CPU implementation; we make it publicly available to other researchers at https://github.com/NVlabs/GA3C.

1 INTRODUCTION

In the past, the need for task-specific, or even hand-crafted, features limited the application of Reinforcement Learning (RL) in real world problems (Sutton & Barto, 1998). However, the introduction of Deep Q-Learning Networks (DQN) (Mnih et al., 2015) revived the use of Deep Neural Networks (DNNs) as function approximators for value and policy functions, unleashing a rapid series of advancements. Remarkable results include learning to play video games from raw pixels (Bellemare et al., 2016; Lample & Singh Chaplot, 2016) and demonstrating super-human performance on the ancient board game Go (Silver et al., 2016). Research has yielded a variety of effective training formulations and DNN architectures (van Hasselt et al., 2015; Wang et al., 2015), as well as methods to increase parallelism while decreasing the computational cost and memory footprint (Nair et al., 2015; Mnih et al., 2016). In particular, Mnih et al. (2016) achieve state-of-the-art results on many gaming tasks through a novel lightweight, parallel method called Asynchronous Advantage Actor-Critic (A3C). When the proper learning rate is used, A3C learns to play an Atari game (Brockman et al., 2016) from raw screen inputs more quickly and efficiently than previous methods: on a 16-core CPU, A3C achieves higher scores than previously published methods run for the same amount of time on a GPU.

Our study sets aside many of the learning aspects of recent work and instead delves into the computational issues of deep RL. Computational complexities are numerous, largely centering on a common factor: RL has an inherently sequential aspect, since the training data are generated while learning. The DNN model is constantly queried to guide the actions of agents whose gameplay in turn feeds DNN training. Training batches are commonly small and must be efficiently shepherded from the agents and simulator to the DNN trainer. When using a GPU, the mix of small DNN architectures, small training batch sizes, and contention for the GPU for both inference and training can lead to a severe under-utilization of the computational resources.

To systematically investigate these issues, we implement both CPU and GPU versions of A3C in TensorFlow (TF) (Abadi et al., 2015), optimizing each for efficient system utilization and to approximately replicate published scores in the Atari 2600 environment (Brockman et al., 2016). We

analyze a variety of "knobs" in the system and demonstrate effective automatic tuning of those during training. Our hybrid CPU/GPU implementation of A3C, named GA3C, generates and consumes training data substantially faster than its CPU counterpart, up to $\sim 6\times$ faster for small DNNs and $\sim 45\times$ for larger DNNs. While we focus on the A3C architecture, this analysis can be helpful for researchers and framework developers designing the next generation of deep RL methods.

2 RELATED WORK

Recent advances in deep RL have derived from both novel algorithmic approaches and related systems optimizations. Investigation of the algorithmic space seems to be the most common approach among researchers. Deep Q-Learning Networks (DQN) demonstrate a general approach to the learning problem (Mnih et al., 2015), relying heavily on the introduction of an experience replay memory to stabilize the learning procedure. This improves reliability but also increases the computational cost and memory footprint of the algorithm. Inspired by DQN, researchers have proposed more effective learning procedures, achieving faster and more stable convergence: Prioritized DQN (Schaul et al., 2015) makes better use of the replay memory by more frequently selecting frames associated with significant experiences. Double-DQN (van Hasselt et al., 2015) separates the estimate of the value function from the choice of actions (policy), thus reducing the tendency in DQN to be overly optimistic when evaluating its choices. Dueling Double DQN (Wang et al., 2015) goes a step further by explicitly splitting the computation of the value and advantage functions within the network. The presence of the replay memory makes the DQN approaches more suitable for a GPU implementation when compared to other LR methods, but state-of-the-art results are achieved by A3C (Mnih et al., 2016), which does not make use of it.

Among systems approaches, AlphaGo (Silver et al., 2016) recently achieved astonishing results through combined algorithmic and hardware specialization. The computational effort is impressive: 40 search threads, 1202 CPUs, and 176 GPUs are used in the distributed version for inference only. Supervised training took around three weeks for the policy network, using 50 GPUs, and another day using the RL approach for refinement. A similar amount of time was required to train the value network. Gorilla DQN (Nair et al., 2015) is a similarly impressive implementation of distributed RL system, achieving a significant improvement over DQN. The system requires 100 concurrent actors on 31 machines, 100 learners and a central parameter server with the network model. This work demonstrates the potential scalability of deep RL algorithms, achieving better results in less time, but with a significantly increased computational load, memory footprint, and cost.

3 ASYNCHRONOUS ADVANTAGE ACTOR CRITIC (A3C)

3.1 REINFORCEMENT LEARNING BACKGROUND

In standard RL, an agent interacts with an environment over a number of discrete time steps. At each time step t, the agent observes a state s_t and, in the discrete case, selects an action a_t from the set of valid actions. An agent is guided by policy π, a function mapping from states s_t to actions a_t. After each action, the agent observes the next state s_{t+1} and receives feedback in the form of a reward r_t. This process continues until the agent reaches a terminal state or time limit, after which the environment is reset and a new episode is played.

The goal of learning is to find a policy π that maximizes the expected reward. In policy-based model-free methods, a function approximator such as a neural network computes the policy $\pi(a_t|s_t; \theta)$, where θ is the set of parameters of the function. There are many methods for updating θ based on the rewards received from the environment. REINFORCE methods (Williams, 1992) use gradient ascent on $\mathbb{E}[R_t]$, where $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$ is the accumulated reward starting from time step t and increasingly discounted at each subsequent step by factor $\gamma \in (0, 1]$.

The standard REINFORCE method updates θ using the gradient $\nabla_\theta \log \pi(a_t|s_t; \theta) R_t$, which is an unbiased estimator of $\nabla_\theta \mathbb{E}[R_t]$. The variance of the estimator is reduced by subtracting a learned *baseline* (a function of the state $b_t(s_t)$) and using the gradient $\nabla_\theta \log \pi(a_t|s_t; \theta)(R_t - b_t(s_t))$ instead. One common baseline is the value function defined as $V^\pi(s_t) = \mathbb{E}[R_t|s_t]$ which is the expected return for following the policy π in state s_t. In this approach the policy π and the baseline b_t can be viewed as *actor* and *critic* in an actor-critic architecture (Sutton & Barto, 1998).

2

3.2 ASYNCHRONOUS ADVANTAGE ACTOR CRITIC (A3C)

A3C (Mnih et al., 2016), which achieves state-of-the-art results on many gaming tasks including Atari 2600, uses a single DNN to approximate both the policy and value function. The DNN has two convolutional layers with $16 \times 8 \times 8$ filters with a stride of 4, and $32 \times 4 \times 4$ filters with a stride of 2, followed by a fully connected layer with 256 units; each hidden layer is followed by a rectifier non-linearity. The two outputs are a softmax layer which approximates the policy function $\pi(a_t|s_t; \theta)$, and a linear layer to output an estimate of $V(s_t; \theta)$. Multiple agents play concurrently and optimize the DNN through asynchronous gradient descent. Similar to other asynchronous methods, the network weights are stored in a central parameter server (Figure 1a). Agents calculate gradients and send updates to the server after every $t_{max} = 5$ actions, or when a terminal state is reached. After each update, the central server propagates new weights to the agents to guarantee they share a common policy.

Two cost functions are associated with the two DNN outputs. For the policy function, this is:

$$f_\pi(\theta) = \log \pi(a_t|s_t; \theta)(R_t - V(s_t; \theta_t)) + \beta H(\pi(s_t; \theta)), \tag{1}$$

where θ_t are the values of the parameters θ at time t, $R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_t)$ is the estimated discounted reward in the time interval from t to $t + k$ and k is upper-bounded by t_{max}, while $H(\pi(s_t; \theta))$ is an entropy term, used to favor exploration during the training process. The factor β controls the strength of the entropy regularization term. The cost function for the estimated value function is:

$$f_v(\theta) = (R_t - V(s_t; \theta))^2. \tag{2}$$

Training is performed by collecting the gradients $\nabla \theta$ from both of the cost functions and using the standard non-centered RMSProp algorithm (Tieleman & Hinton, 2012) as optimization:

$$\begin{aligned} g &= \alpha g + (1 - \alpha)\Delta\theta^2 \\ \theta &\leftarrow \theta - \eta\Delta\theta/\sqrt{g + \epsilon}. \end{aligned} \tag{3}$$

The gradients g can be either shared or separated between agent threads but the shared implementation is known to be more robust (Mnih et al., 2016).

The original implementation of A3C (Mnih et al., 2016) uses 16 agents on a 16 core CPU and it takes about four days to learn how to play an Atari game (Brockman et al., 2016). The main reason for using CPU other than GPU, is the inherently sequential nature of RL in general, and A3C in particular. In RL, the training data are generated while learning, which means the training and inference batches are small and GPU is mostly idle during the training, waiting for new data to arrive. Since A3C does not utilize any replay memory, it is completely sequential and therefore a CPU implementation is as fast as a naive GPU implementation.

4 HYBRID CPU/GPU A3C (GA3C)

We propose GA3C, an alternative architecture of A3C, with emphasize on an efficient GPU utilization to increase the number of training data generated and processed per second. We demonstrate that our implementation of GA3C effectively converges significantly faster than our CPU implementation of A3C, achieving the state-of-the-art performance in a shorter time.

4.1 GA3C ARCHITECTURE

The primary components of GA3C (Figure 1b) are a DNN with training and prediction on a GPU, as well as a multi-process, multi-thread CPU architecture with the following components:

- **Agent** is a process interacting with the simulation environment: choosing actions according to the learned policy and gathering experiences for further training. Similar to A3C, multiple concurrent agents run independent instances of the environment. Unlike the original, each agent does not have its own copy of the model. Instead it queues policy requests in a *Prediction Queue* before each action, and periodically submits a batch of input/reward experiences to a *Training Queue*; the size of each training batch is typically equal to t_{max} experiences, though it is sometimes smaller for experiences collected at the end of an episode.
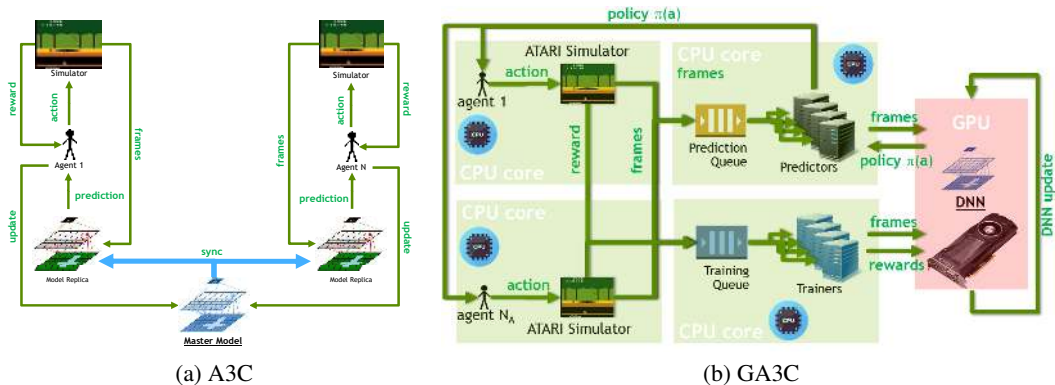
(a) A3C

(b) GA3C

Figure 1: Comparison of A3C and GA3C architectures. Agents act concurrently both in A3C and GA3C. In A3C, however, each agent has a replica of the model, whereas in GA3C there is only one GPU instance of the model. In GA3C, agents utilize predictors to query the network for policies while trainers gather experiences for network updates.

- **Predictor** is a thread which dequeues as many prediction requests as are immediately available and batches them into a single inference query to the DNN model on the GPU. When predictions are completed, the predictor returns the requested policy to each respective waiting agent. To hide latency, one or more predictors can act concurrently.

- **Trainer** is a thread which dequeues training batches submitted by agents and submits them to the GPU for model updates. GPU utilization can be increased by grouping training batches among several agents; we found that this generally leads to a more stable convergence, but the convergence speed is reduced when the merged training batches are too large; a compromise is explored in Section 5.3. Multiple trainers may run in parallel to hide latency.

Unlike A3C, GA3C maintains only one copy of the DNN model (Fig. 1a and 1b), centralizing predictions and training updates and removing the need for synchronization. Also, in comparison with A3C, agents in GA3C do not compute the gradients themselves. Instead, they send experiences to trainers that update the network on the GPU accordingly. This introduces a potential lag between the generation and consumption of experiences, which we analyze in detail in Sections 4.4 and 5.3.

4.2 Performance Metrics and Trade-offs

The GA3C architecture exposes numerous tradeoffs for tuning its computational efficiency. In general, it is most efficient to transfer data to a GPU in large enough blocks to maximize the usage of the bandwidth between the GPU and CPU. Application performance on the GPU is optimized when the application has large amounts of parallel computations that can hide the latency of fetching data from memory. Thus, we want to maximize the parallel computations the GPU is performing, maximize the size of data transfer to the GPU, and minimize the number of transfers to the GPU. Increasing the number of predictors, N_P, allows faster fetching prediction queries, but leads to smaller prediction batches, resulting in multiple data transfers and overall lower GPU utilization. A larger number of trainers, N_T, potentially leads to more frequent updates to the model, but an overhead is paid when too many trainers occupy the GPU while predictors cannot access it. Lastly, increasing the number of agents, N_A, ideally generates more training experiences while hiding prediction latency. However, we would expect diminishing returns from unnecessary context switching overheads after exceeding some threshold depending on the number of CPU cores.

These aspects are well captured by a metric like the *Trainings Per Second* (TPS), which is the rate at which we remove batches from the training queue. It corresponds to the rate of model updates and it is approximately proportional to the overall learning speed, given a fixed learning rate and training batch size. Another metric is the *Predictions Per Second* (PPS), the rate of issuing prediction queries from prediction queue, which maps to the combined rate of gameplay among all agents. Notice that in A3C a model update occurs every time an agent plays $t_{max} = 5$ actions (Mnih et al., 2016).
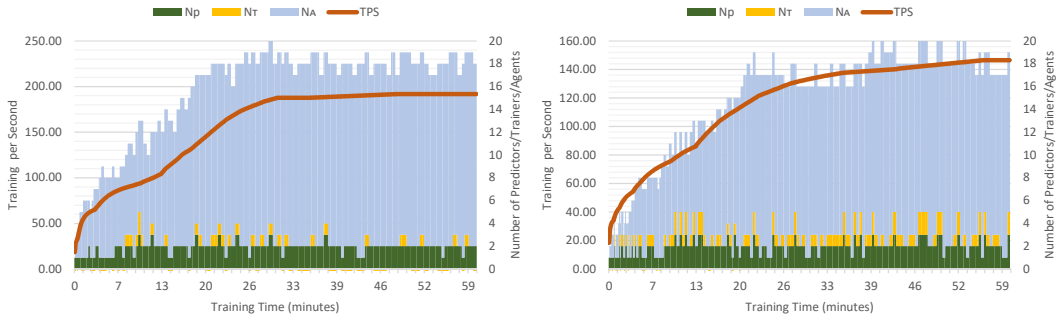
Figure 2: Automatic dynamic adjustment of N_T, N_P, and N_A, to maximize TPS for BOXING (left) and PONG (right), starting from a sub-optimal configuration ($N_A = N_T = N_P = 1$)

Hence, in a balanced configuration, PPS $\approx$ TPS $\times t_{max}$. Since each action is repeated four times as in (Mnih et al., 2016), the number of frames per second is $4 \times$ PPS.

Computational aspects are not disconnected from the convergence of the learning algorithm. For instance, employing too many agents will tend to fill the training queue, introducing a significant time delay between agent experiences (a_t, s_t and R_t in Eq. (1)) and the corresponding model updates, possibly threatening model convergence (see Section 4.4). Another example is batching of training data: larger batches improves GPU occupancy by increasing the parallelism. They also decrease the TPS (*i.e.*, the number of model updates per second), increasing the chance that the DNN model used in prediction and to compute the gradient in Eq. (4) are indeed the same model. The consequence (experimentally observed, see Section 5.3) is an increased stability of the learning process but, beyond a certain training batch size, this leads to a reduction in the convergence speed. In short, N_T, N_P, and N_A encapsulate many complex dynamics relating both computational and convergence aspects of the learning procedure. Their effect on the convergence of the learning process has to be measured by analyzing not only TPS but also the learning curves.

4.3 DYNAMIC ADJUSTMENT OF TRADE-OFFS

The setting of N_P, N_T and N_A that maximizes the TPS depends on many aspects such as the computational load of the simulation environment, the size of the DNN, and the available hardware. As a rule of thumb, we found that the number of agents N_A should at least match the available CPU cores, with two predictors and two trainers $N_P = N_T = 2$. However, this rule hardly generalizes to a large variety of different situations and only occasionally corresponds to the computationally most efficient configuration. Therefore, we propose an annealing process to configure the system dynamically. Every minute, we randomly change N_P, N_T, or N_A by ± 1, monitoring alterations in TPS to accept or reject the new setting. The optimal configuration is then automatically identified in a reasonable time, for different environments or systems. Figure 2 shows the automatic adjustment procedure finding two different optimal settings for two different games, on the same real system.

4.4 POLICY LAG IN GA3C

At a first sight, GA3C and A3C are different implementations of the same algorithm, but GA3C has a subtle difference which affects the stability of the algorithm. This problem is caused by the latency between the time $t - k$, when a training example has been generated, and when it is consumed for training, t, essentially changing the gradients to:

$$\nabla_\theta \big[\log \pi \left(a_{t-k} | s_{t-k}; \theta \right) \left(R_{t-k} - V \left(s_{t-k}; \theta_t \right) \right) + \beta H \left(\pi \left(s_{t-k}; \theta \right) \right) \big]. \tag{4}$$

Since the Training Queue is not blocking, the states it contains can be old. The value of the delay k is bounded by the maximum size of the queue and influenced by how the system configuration balances training and prediction rates. In other words, the DNN controller selects the action a_{t-k} at time $t - k$; the corresponding experience lies in a training queue until time t, when a trainer thread pops the element out of the queue to compute the gradient as in Eq. (4). The DNN controller at time t generally differs from the one at time $t - k$, since trainers can modify the DNN weights at any time. Therefore, the policy and value function π and V used to compute the gradient at time t will

5

differ from those used at time $t - k$ to collect the experience, whereas the action used to compute the gradient in Eq. (4) remains a_{t-k}.

This delay can lead to instabilities for two reasons. The first one is the possible generation of very large values in $\log \pi \left(a_{t-k} | s_{t-k}; \theta_t \right)$. In fact, $\pi \left(a_{t-k} | s_{t-k}; \theta_{t-k} \right)$ is generally large, since it is the probability of sampled action a_{t-k}, but over the course of lag k new parameters θ_t can make $\pi \left(a_{t-k} | s_{t-k}; \theta_t \right)$ very small. In the worst case, the updated probability is zero, generating infinite values in the $\log$ and causing optimization to fail. To avoid this, we add a small term $\epsilon > 0$:

$$\nabla_\theta \left[\log \left(\pi \left(a_{t-k} | s_{t-k}; \theta \right) + \epsilon \right) \left(R_{t-k} - V \left(s_{t-k}; \theta_t \right) \right) + \beta H \left(\pi \left(s_{t-k}; \theta \right) + \epsilon \right) \right]. \tag{5}$$

Beyond fixing the error in the case $\pi = 0$, this fix also improves the stability of the algorithm and removes the necessity of gradient clipping. In fact, as $\partial \log(\pi + \epsilon)/\partial\theta = (\partial\pi/\partial\theta)/(\pi + \epsilon)$, ϵ establishes an upper bound for the multiplicative factor in front of $\partial\pi/\partial\theta$. A similar term is also added in the entropy computation to avoid a similar explosion. It is important to remember that even A3C suffers from a similar issue. In fact, as the action a_t in Eq. (1) is selected by sampling the output softmax, there is a chance that $\pi(a_t)$ is very small and therefore $\partial \log(\pi)/\partial\theta$ is large. However, gradient clipping prevents the usage of a gradient with large magnitude in A3C.

The second reason for introducing instabilities in GA3C is a generalization of the first one. Since training and predictions are computed with potentially different DNN parameters, the resulting gradient is noisy, and can therefore lead to unreliable updates of the DNN weights. This is different from A3C, where every agent has its own copy of the model and uses it to compute both π and $\partial \log(\pi)/\partial\theta$, before synchronizing the DNN model with the other agents.

5   ANALYSIS

We profile the performance of GA3C and in the process seek to better understand the system dynamics of deep RL training on hybrid CPU/GPU systems. Experiments are conducted on the GPU-enabled systems described in Table 1 and monitored with CUDA profilers and custom profiling code based on performance counter timing within Python. We present profiling and convergence experiments both with and without automatic adjustment of the number of agents N_A, trainers N_T, and predictors N_P, and without constraints on the size of the prediction and training queues.
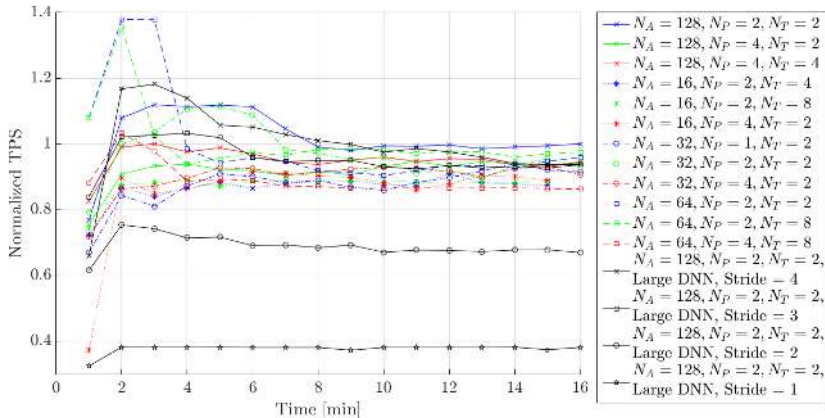

Figure 3: TPS of the top three configurations of predictors N_P and trainers N_T for several settings of agents N_A, while learning PONG on System I from Table 1. TPS is normalized by best performance after 16 minutes. Larger DNN models are also shown, as described in the text.

5.1   EFFECT OF RESOURCE UTILIZATION ON TPS

Maximizing training speed.   To begin, consider raw training speed as expressed in model update frequency, or trainings per second (TPS). Figure 3 shows TPS on System I in Table 1 for the first 16 minutes of training on PONG. We consider numbers of agents $N_A \in \{16, 32, 64, 128\}$ and plot the top 3 combinations of $N_P, N_T \in \{1, 2, 4, 8, 16\}$. On this system, increasing N_A yields a higher TPS
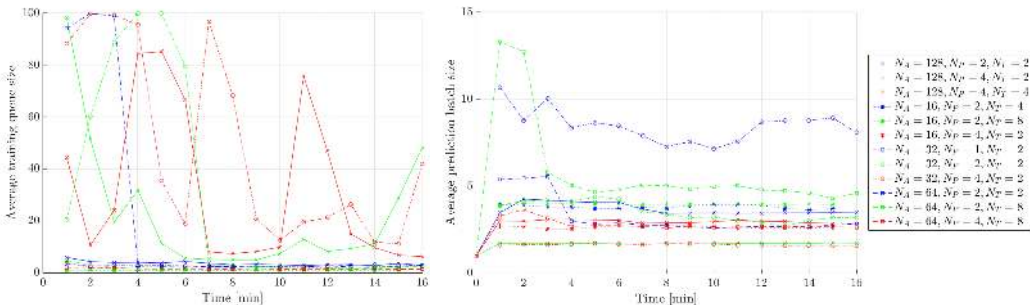
Figure 4: The average training queue size (left) and prediction batch size (right) of the top 3 performing configurations of N_P and N_T, for each N_A, with PONG and the System I in Table 1.

up to $N_A = 128$ where diminishing returns are observed, likely due to additional process overhead. The highest consistent TPS on this system is observed with $N_A = 128$ and $N_P = N_T = 2$ with a speed-up of $\sim 4\times$ relative to the CPU-only implementation (see Table 2).

	System I	System II	System III	System IV
Processor (Intel)	Xeon E5-2640v3 2.60 GHz 16 cores, dual socket	Core i-3820 3.60 GHz 8 cores	Haswell E5-2698v3 2.30 GHz 16 cores	Xeon E5-2680v2 2.80 GHz 10 cores
GPU (NVIDIA)	Geforce Titan X (Maxwell)	GeForce 980 (Maxwell)	Tesla K80 (Kepler)	Quadro M6000 (Maxwell)
Software / Profilers	Python 3.5, CUDA 7.5 (I-III)/CUDA 8 (IV), CUDNN v5.1, TensorFlow r0.11 nvprof, nvvp			

Table 1: Systems used for profiling and testing.

GPU utilization and DNN size. The fastest configuration ($N_A = 128$, $N_P = N_T = 2$) for System I in Table 1 has an average GPU utilization time of only 56%, with average and peak occupancy of 76% and 98%, respectively.[1] This suggests there is computational capacity for a larger network model. Therefore we profile GA3C on a series of deeper DNN architectures[2] to evaluate this hypothesis. Figure 3 shows that TPS drops by only 7% with a one-layer deeper DNN controller; at the same time, the

		PPS			Utilization (%)	
System	DNN	A3C	GA3C	Speed up	CPU	GPU
System I	small	352	1361	4×	32	56
	large, stride 4	113	1271	11×	27	68
	large, stride 3	97	1206	12×	27	77
	large, stride 2	43	874	20×	26	82
	large, stride 1	11	490	45×	17	90
System II	small	116	728	6×	62	33
	large, stride 1	12	336	28×	49	78
System III	small	300	1248	4×	31	60
	large, stride 1	38	256	6×	17	82

Table 2: PPS on different systems (Table 1), for small and large DNNs, with CPU and GPU utilization for GA3C.

average GPU utilization and occupancy increase by approximately 12% and 0.5%, respectively. The 7% drop in TPS is the consequence of the increased depth which forces an additional serial computational on the GPU (and therefore a 12% increase in its utilization). The negligible 0.5% increase in occupancy is likely explained by an efficient management of the computational resources by cuDNN; there is still room available to run additional parallel tasks (or, in other words, a wider DNN) at minimal cost.

---

[1] Occupancy reflects the parallelism of the computation by capturing the proportion of GPU threads which are active while the GPU is in use. For example, occupancy of 76% with utilization of 56% implies roughly 43% overall thread utilization.

[2] We increase the number of filters in the first layer of the DNN from 16 to 32, and add a third convolutional layer with 64 4×4 filters and stride 2. To generate four large DNNs with varying computational cost, we reduce the stride on the first layer from 4 pixels to 1 pixel, with a step of 1 pixel. Note the DNN with stride 1 in the first layer also has $4\times$ larger fully-connected layer.
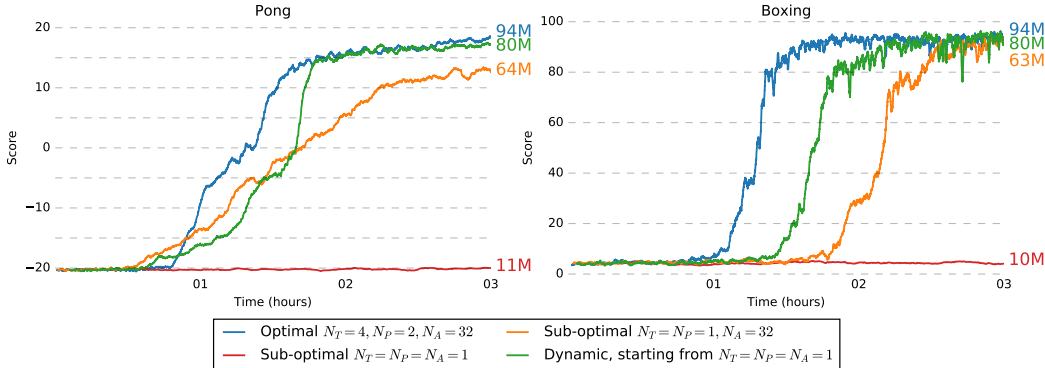
Figure 5: Effect of PPS on convergence speed. For each game, four different settings of GA3C are shown, all starting from the same DNN initialization. Numbers on the right show the cumulative number of frames played among all agents for each setting over the course of 3 hours. Configurations playing more frames converge faster. The dynamic configuration method is capable of catching up with the optimal configuration despite starting with a sub-optimal setting, $N_T = N_P = N_A = 1$.

By reducing the stride of the first layer of the DNN (in addition to adding a convolutional layer), we scale DNN size with finer granularity, and we compare FPS between GA3C and our CPU implementation of A3C. Table 2 shows the speed up provided by GA3C increases as the DNN grows. This is mainly due to increasing GPU utilization, as reported in Table 2. With the largest DNN, our CPU implementation achieves TPS ≈ 11, which is approximately $45\times$ slower than GA3C. This behavior is consistent across different systems, as shown in Table 2, where the CPU implementation of A3C using the largest DNN with stride 1 is $7\times$ (System III) to $32\times$ (System I) slower than the small network. Scaling with DNN size is more favorable on a GPU, with a slow down factor of $4.9\times$ in the worst case (System III) and $2.2\times$ in the best case (System II). Further, more recent GPUs (Maxwell architecture) scale better ($2.2\times$ and $2.7\times$ slow down for Systems I and II) than older GPUs ($4.8\times$ slow down for the Kepler architecture, System III).

Generally speaking, for large DNNs, maximum TPS and FPS are achieved by intensively using the GPU for prediction and training, while the CPU runs the simulation environment and remains mostly idle. In practice, this allows experimenting with larger architectures, which may be particularly important for real world problems, e.g. robotics or autonomous driving (Lillicrap et al., 2015). Moreover, the idle CPU represents an additional computational resource, but such investigation is beyond the scope of this paper.

Significant latency. Profiling on System I in Table 1 reveals that the average time spent by an agent waiting for a prediction call to be completed is 108ms, only 10% of which is taken by the GPU inference. The remaining 90% is overhead spent accumulating the batch and calling the prediction function in Python. Similarly, for training we find that of the average 11.1ms spent performing a DNN update, 59% is overhead. This seems to suggest that a more optimized implementation (possibly based on a low level language like C++) may reduce these overheads, but this investigation remains for future work.

Manually balancing components. Agents, predictors, and trainers all share the GPU as a resource; thus balance is important. Figure 3 shows the top three performing configurations of N_P and N_T for different numbers of agents, N_A, with System I in Table 1. A 14% drop in TPS is exhibited between the best and worst depicted configuration, despite the exclusion of all but the top three performers for each number of agents. The best results have 4 or fewer predictor threads, seemingly preventing batches from becoming too small. The $N_P : N_T$ ratios for top performers tend to be $1:2$, $1:1$, or $2:1$, whereas higher ratios such as $1:8$ and $1:4$ are rarely successful, likely due to the implicit dependence of training on prediction speed. However, if the training queue is too full, training calls take more GPU time, thereby throttling prediction speed. This is further confirmed by our experimental finding that TPS and PPS plots track closely. Figure 4 shows training queue size and prediction batch size for the top configurations. In all cases, the training queue stabilizes well below its maximum capacity. Additionally, the fastest configuration has one of the largest average prediction batch sizes, yielding higher GPU utilization.

	Atari Game Scores									Attributes	
	AMIDAR	BOXING	CENTIPEDE	NAME THIS GAME	PACMAN	PONG	QBERT	SEAQUEST	UP-DOWN	Time	System
Human	1676	10	10322	6796	15375	16	12085	40426	9896	–	–
Random	6	−2	1926	198	1748	−18	272	216	533	–	–
A3C	264	60	3756	10476	654	6	15149	2355	74706	4 days	CPU
GA3C	218	92	7386	5643	1978	18	14966	1706	8623	1 day	GPU

Table 3: Average scores on a subset of Atari games achieved by: a random player (Mnih et al., 2015); a human player (Mnih et al., 2015); A3C after four days of training on a CPU (Mnih et al., 2016); and GA3C after one day of training. For GA3C, we measured the average score on 30 games, each initialized with a random seed.

5.2 EFFECT OF TPS ON LEARNING SPEED

The beneficial effect of an efficient configuration on the training speed is shown in Figure 5. Training with a suboptimal configuration (e.g. $N_P = N_T = N_A = 1$ or $N_P = N_T = 1$, $N_A = 16$) leads to a severe underutilization of the GPU, a low TPS, and a slow training process. Using the optimal configuration achieves a much higher score in a shorter period of time, mainly driven by playing more frames, i.e. collecting more experiences, in the same amount of time.

Mnih et al. (2016) note that asynchronous methods generally achieve significant speedups from using a greater number of agents, and even report superlinear speedups for asynchronous one-step Q-learning. It is worth noting that optimal configurations for GA3C generally employ a much higher number of agents compared to the CPU counterpart, e.g. the optimal configuration for System I in Table 1 uses 128 agents. This suggests that GPU implementations of asynchronous learning methods may benefit from both a higher TPS and from collecting experiences from a wider number of agents.

The learning curve for GA3C with dynamic configuration (Figure 5) tracks closely with the learning curve of the optimal configuration. The total number of frames played is generally slightly lower over the same time due to the search procedure overhead: the configuration is changed once every minute, tending to oscillate around the optimal configuration. Notice also that, in Figure 5, the starting point of the dynamic configuration is $N_T = N_P = N_A = 1$, which is much slower than the optimal configuration. But scoring performance is nearly identical, indicating that the dynamic method may ease the burden of configuring GA3C on a new system.

Table 3 compares scores achieved by A3C on the CPU (as reported in (Mnih et al., 2016)) with the best agent trained by our TensorFlow implementation of GA3C. Unfortunately, a direct speed comparison is infeasible without either the original source code or the average number of frames or training updates per second. However, results in this table do show that after one day of training our open-source implementation can achieve similar scores to A3C after four days of training.

Figure 6 shows typical training curves for GA3C on several Atari games as a function of wall-clock time. When compared to the training curves reported in Mnih et al. (2016), GA3C shows faster convergence toward the maximum score in a shorter time for certain games such as PONG, convergence towards a better score in a larger amount of time (e.g. QBERT) or, for other games, a slower convergence rate (e.g. BREAKOUT). It has to be noted, however, that data reported by Mnih et al. (2016) are the average learning curves of the top five learners in a set of fifty learners, each with a different learning rate. On the other hand, in Figure 6, we are reporting three different runs for two (not essentially optimal) learning rates, fixed for all the games. This demonstrates some robustness of GA3C with respect to the choice of the learning rate, whereas it is also likely that better learning curves can be obtained using optimized learning rates. A deeper investigation on a large amount of data, potentially facilitated by our release of the GA3C code, may also reveal how peculiarities of each game differently affect the convergence of A3C and GA3C, but this goes beyond the scope of this paper.

5.3 POLICY LAG, LEARNING STABILITY AND CONVERGENCE SPEED

One of the main differences between GA3C and A3C is the asynchronous computation of the forward step (policy π) and the gradients (Eq. (4)) used to update the DNN. Delays between these
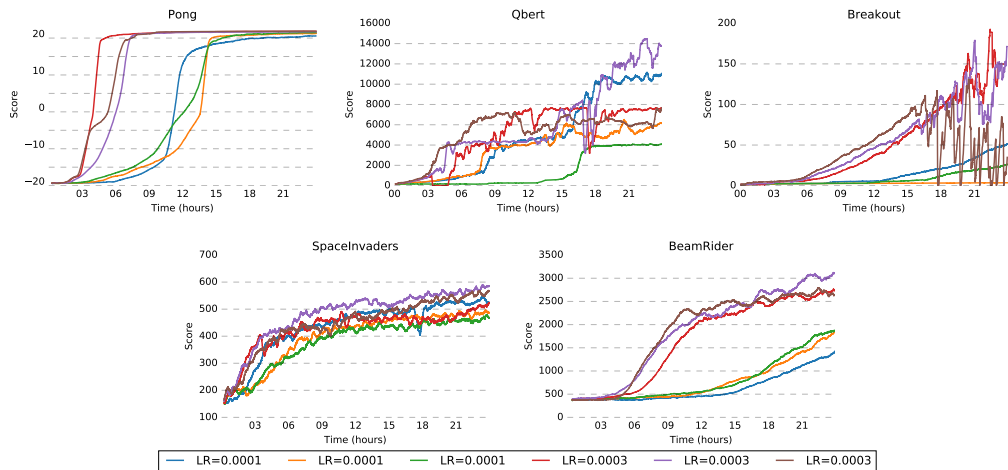
Figure 6: Training curves for GA3C on five Atari games. Each training has been performed three times for each of two learning rates (0.0003 and 0.0001) on System IV in Table 1.
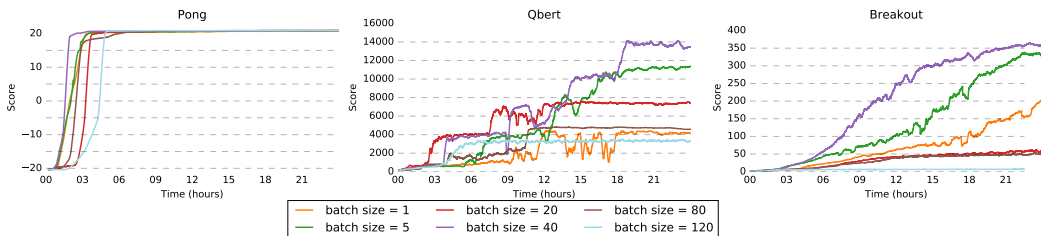

Figure 7: Training GA3C with a range of minimum training batch sizes. Increasing the minimum training batch size from 1 to 40 reduces the effect of the policy lag (delay k in Eq. (4)), leading to convergence that is faster and more stable. GA3C achieved the overall best results with a minimum batch size between 20 and 40. Increasing beyond this threshold dramatically reduces convergence speed for some games, especially those inclined to unstable learning curves.

two operations may introduce noise in the gradients, making the learning process unstable. We experimentally investigated the impact of this asynchrony on the learning process to determine if a synchronization mechanism, which may negatively impact both PPS and TPS, can increase the stability of the algorithm.

In GA3C, each agent generally pushes t_{max} experiences in the training queue. By default, trainers collect a batch of experiences from a single agent from the training queue and send the batch to the GPU to compute gradients, as in Eq. (5). Each time a trainer updates the DNN weights, the remaining experiences in the training queue are no longer in sync with the DNN model. This situation becomes worse when the average length of the training queue is large.

By allowing larger training batch sizes, we reduce the number of DNN updates per second (TPS), and consequently diminish the effect of the delay k in Eq. (5). In this way we increase the chance that the collected experiences and the computed gradients are in sync, which improves the stability. Notice that, even if the TPS is lower, the average magnitude of the updates is indeed larger, since we sum the gradients computed over the training batch.

In this setting, the optimal training batch size compromises among TPS, the average gradient step magnitude and the training stability. Another factor to be considered is that batching training data potentially leverages the GPU computational capability better by reducing the time devoted to compute the DNN updates while increasing the GPU occupancy during this phase. This gives more GPU time to the predictors, potentially increasing the PPS. However, this advantage tends to disappear when the training batch size is too large and predictors stay idle while the DNN update is computed.

Figure 7 compares convergence curves when no minimum size for training batch is compulsory (the default GA3C implementation where gradient updates are computed on a single agent's batch) and when a minimum training batch size is enforced (combining multiple agent batches into a single gradient update). In the latter case, trainers collect experiences from multiple agents at the same time from the training queue and send them to the GPU for computation of gradients as in Eq. (5). Up to a certain batch size (between 20 and 40, in our experiments), increasing the training batch size stabilizes the learning procedure and generally leads to faster convergence. Some games such as PONG indeed do not suffer from this instability, and the effect of the minimum batch size is less evident in this case. We speculate that a careful selection of the learning rate combined with the proper minimum training batch size may lead to even faster convergence.

6 CONCLUSION

By investigating the computational aspects of our hybrid CPU/GPU implementation of GA3C, we achieve a significant speed up with respect to its CPU counter part. This comes as a result of a flexible system capable of finding a reasonable allocation of the available computational resources. Our approach allows producing and consuming training data at the maximum pace on different systems, or to adapt to temporal changes of the computational load on one system. Despite the fact that we analyze A3C only, most of our findings can be applied to similar RL asynchronous algorithms.

We believe that the analysis of the computational aspects of RL algorithms may be a consistent theme in RL in the future, motivating further studies such as this one. The potential benefits of such investigation goes well beyond the computational aspects. For instance, we demonstrate that GA3C scales with the size of the DNN much more efficiently than our CPU implementation of A3C, thus opening the possibility to explore the use of large DNN controllers to solve real world RL problems.

By open sourcing GA3C (see `https://github.com/NVlabs/GA3C`), we allow other researchers to further explore this space, investigate in detail the computational aspects of deep RL algorithms, and test new algorithmic solutions, including strategies for the combined utilization of the CPU and GPU computational resources.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying Count-Based Exploration and Intrinsic Motivation. *ArXiv e-prints*, June 2016.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

G. Lample and D. Singh Chaplot. Playing FPS Games with Deep Reinforcement Learning. *ArXiv e-prints*, September 2016.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015. URL http://arxiv.org/abs/1509.02971.

V. Mnih, A. Puigdomenech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *ArXiv preprint arXiv:1602.01783*, 2016.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. URL http://dx.doi.org/10.1038/nature14236.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively parallel methods for deep reinforcement learning. *CoRR*, abs/1507.04296, 2015. URL http://arxiv.org/abs/1507.04296.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. URL http://arxiv.org/abs/1511.05952.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.

Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015. URL http://arxiv.org/abs/1511.06581.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.