GA3C: GPU-based A3C for Deep Reinforcement Learning

Mohammad Babaeizadeh
University of Illinois at Urbana-Champaign
mb2@uiuc.edu

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

Abstract

We introduce and analyze the computational aspects of a hybrid CPU/GPU implementation of the Asynchronous Advantage Actor-Critic (A3C) algorithm, currently the state-of-the-art method in reinforcement learning for various gaming tasks. Our analysis concentrates on the critical aspects to leverage the GPU's computational power, including the introduction of a system of queues and a dynamic scheduling strategy, potentially helpful for other asynchronous algorithms as well. We also show the potential for the use of larger DNN models on a GPU. Our TensorFlow implementation achieves a significant speed up compared to our CPU-only implementation, and it will be made publicly available to other researchers.

1 Introduction

The need for task-specific features previously limited the application of Reinforcement Learning (RL) [1], but the introduction of Deep Q-Learning Networks (DQN) [2] 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 [3, 4] and demonstrating super-human performance on ancient board games [5]. Research has yielded a variety of effective training formulations and DNN architectures [6, 7], as well as methods to increase parallelism and decrease computational resources [8, 9]. In particular, Mnih *et al.* [9] achieve state-of-the-art results on many gaming tasks through the Asynchronous Advantage Actor-Critic (A3C) algorithm. A3C dispenses with the large replay memory of previous approaches [10, 2] by stabilizing learning with updates from several concurrently playing agents. For the task of learning to play an Atari game [11] from raw screen inputs, with a proper learning rate, A3C learns better strategies on a 16-core CPU than previous methods trained for the same amount of time on a GPU.

Our study sets aside the learning aspects of recent work and instead delves into computational issues of deep RL. Computational complexities are numerous, but largely center on a common factor: RL has an inherently sequential aspect since the training data is generated during learning. The DNN is constantly queried to guide 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. While DQN methods maintain a large set of past experiences to smooth over some of these issues, A3C feeds agent experiences directly into training with little delay. When using a GPU, the mix of small DNN architectures and small batch sizes can severely underutilize computational resources.

To systematically investigate these issues, we implement[2] both CPU and GPU versions of A3C in TensorFlow (TF) [12], optimizing each for efficient system utilization and to match published

*corresponding author

[2]Our implementation is open-source at `https://github.com/NVlabs/GA3C`

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

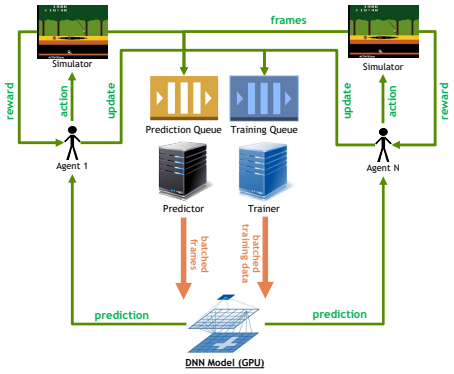

Figure 1: GA3C architecture: agents act concurrently aided by predictors to query the network for policies, while trainers gather experiences for network updates.

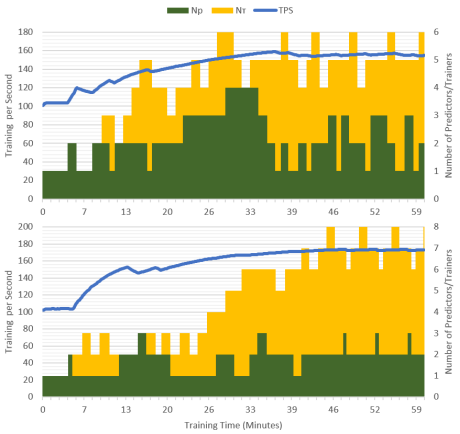

Figure 2: Automatic dynamic adjustment of N_T and N_P to maximize TPS for PONG (top) and PACMAN (bottom), starting from a sub-optimal configuration ($N_A = 32$, $N_T = N_P = 1$).

scores in the Atari 2600 environment [11]. We analyze a variety of "knobs" in the system and demonstrate effective automatic tuning during training. The GPU implementation (GA3C) learns substantially faster (up to $\sim 90\times$ for large DNNs) than its CPU counterpart. While we focus on the A3C architecture, we hope this analysis will be helpful for researchers and framework developers working on the next generation of deep RL methods.

2 Hybrid CPU/GPU A3C (GA3C)

In Asynchronous Advantage Actor-Critic (A3C) [9], multiple agents play concurrently and optimize a DNN controller using asynchronous gradient descent. Similar to other asynchronous methods, the DNN weights are stored in a central parameter server. Agents calculate gradients and send updates to the server, which propagates new weights to the agents between updates to maintain a shared policy. The original implementation of A3C [9] uses 16 agents on a 16 core CPU and takes $1-4$ days to learn how to play an Atari game [11].

For our performance profiling, we implement a hybrid GPU-CPU architecture (GA3C) as depicted in Figure 1. The primary components are a traditional DNN model with training and prediction on a GPU, as well as a multi-process, multi-threaded CPU architecture with the following components:

- **Agent** is a process interacting with the simulation environment by actions chosen according to the learned policy and gathering experiences for further training. Similar to A3C, multiple concurrent agents run independent instances of the environment in GA3C. 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*.

- **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 may act concurrently.

- **Trainer** is a thread which dequeues training batches submitted by agents and submits them to the GPU for model updates. While GPU utilization can be increased by grouping training batches among several agents, we found this can lead to slower convergence because fewer updates are applied to the network. Multiple trainers may run in parallel to hide latency.

This architecture exposes numerous tradeoffs for efficiency tuning. Increasing the number of predictors, N_P, allows faster fetching of 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,

Processor	Dual Socket Intel Xeon E5-2640 v3 @ 2.60GHz; *Cores:* 32 logical / 16 physical; *RAM:* 512 GB
GPU	NVIDIA GeForce Titan X (Maxwell); *PCIE:* Gen 3, 16x
Software	Python 3.5; CUDA 7.5 (CUDNN v5.1); TensorFlow r0.11; *Profilers:* nvprof, nvvp

Table 1: System configuration for profiling.

we expect diminishing returns from unnecessary context switching overheads after exceeding some threshold dependent on the number of CPU cores. Additional agents will tend to fill the training queue, introducing an additional time delay between agent experiences and corresponding model updates, threatening model convergence. In short, N_T, N_P, and N_A encapsulate many of the complex dynamics in recent deep RL training.

When studying various settings for N_T, N_P and N_A, we measure their effects on resource utilization and learning speed. *Training per second* (TPS) is a measurement of the rate at which we remove batches from the training queue, which corresponds to the rate of model updates. Given a fixed learning rate and agent batch size, updates directly drive learning and convergence. Another metric is *predictions per second* (PPS), the rate of issuing prediction queries from the 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 TMAX$=5$ actions [9]. Hence, in a balanced configuration, PPS$\approx$TPS$\times$TMAX. Since each action is repeated four times as in [9], the number of frames per second is $4\times$PPS.

Setting N_P, N_T and N_A to maximize TPS also depends on the amount of computation to run the simulation environment, the size of the DNN, the available hardware, and any shared load on the system. As a rule of thumb, we match agents N_A to the number of available CPU cores, with predictors and trainers $N_P = N_T = 2$. Furthermore, we propose an annealing process to configure the system dynamically. Every minute we randomly change N_P and N_T by ± 1, monitoring change in TPS to accept or reject the new setting. This way, the optimal configuration is automatically identified in a reasonable time for different environments or systems, even in the case of unpredictable external factors. Figure 2 shows two cases of the automatic adjustment procedure on a real system: starting from a sub-optimal configuration ($N_A = 32$, $N_T = N_P = 1$), the TPS—and, consequently, also PPS—increases while N_T and N_P converge towards their optimal values.

3 Results and Discussion

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 server in Table 1 and monitored with CUDA profilers and custom profiling code based on performance counter timing within Python. We disable the automatic adjustment of N_T and N_P for profiling.

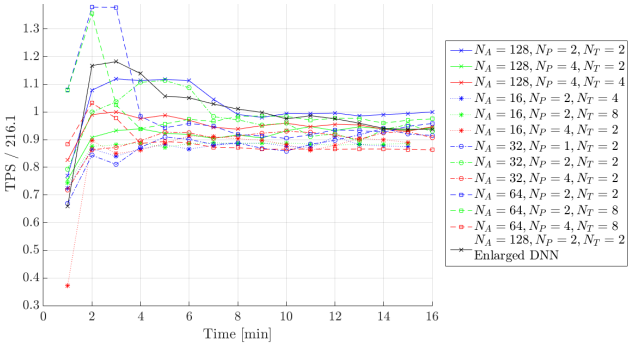

Figure 3: TPS of the top three configurations of predictors N_P and trainers N_T for several numbers of agents N_A, for the system in Table 1 while learning PONG. TPS is normalized by the best sustained performance (TPS $=$ 216.1). Also shown is a larger DNN model, operating under settings described in the text.

Maximizing TPS. Figure 3 shows TPS for the first 16 minutes of training on PONG, for agents $N_A \in \{16, 32, 64, 128\}$ and the top 3 performing combinations of predictors and trainers, $N_P, N_T \in \{1, 2, 4, 8, 16\}$. On this system, increasing N_A yields a higher TPS 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 $N_P = N_T = 2$, yielding a speed-up of $\sim 3.5\times$ relative to the CPU-only implementation. Figure 4 clearly shows the benefit of achieving a higher TPS: the optimal

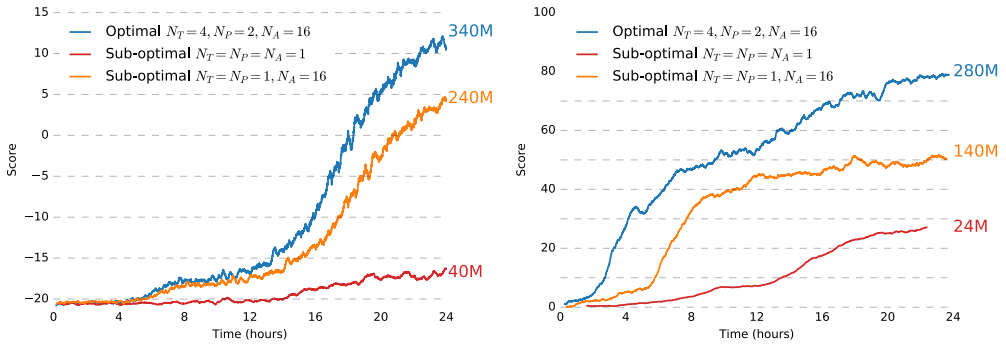

Figure 4: Effect of TPS on convergence speed, for different settings of N_A, N_T, and N_P on the system described in Table 1, all starting from the same DNN initialization. The cumulative number of frames played among all agents for each setting is reported on the right: configurations playing more frames converge faster.

configuration of N_A, N_T, and N_P allows the system to play more frames and make more model updates in the same amount of time, leading to faster learning.

GPU utilization and DNN size. The fastest configuration ($N_A = 128$, $N_P = N_T = 2$) has an average GPU utilization time of only 63%, with average and peak occupancy[3] of 76% and 98%, respectively. This suggests there is computational capacity for a larger network model, and we profile GA3C on a deeper and wider DNN[4] to evaluate this hypothesis.

Figure 3 shows TPS drops by 7% with the larger DNN controller while the average occupancy (*i.e.*, the portion of computational resources actually used by the GPU) increases by 0.5% and GPU utilization time increases 5%. The small 5% increase in the GPU utilization (associated with the decrease of TPS) is caused by the increased depth of the DNN, which forces the GPU perform an additional set of serialized computational tasks. On the other hand, the negligible 0.5% increase in occupancy shows the efficient resource management of CUDNN when kernels are running, *i.e.* there is capacity to run additional parallel tasks without needing additional time. In short, we can use a wider DNN at almost no additional cost, while expecting a small decrease of TPS with a deeper DNN.

System	DNN	A3C	GA3C	Speed-up
[See Table 1]	small	352	1080	3×
	large	11	1004	91×
Intel Core i-3820, CPU 360GHz x8, NVIDIA GeForce 980	small	116	728	6×
	large	12	336	28×

Table 2: PPS on two systems, for small and large DNN controllers.

On the contrary, our CPU implementation of A3C does not scale as well with the size of the DNN controller: the larger DNN network achieves TPS ≈ 11, which is approximately $91\times$ slower than GA3C. This behavior is consistent across different systems, as shown in Table 2. These experiments point to the importance—and capacity—of GPUs for experimentation with larger DNNs. This may be particularly critical when exploring real world problems, such as autonomous driving [13].

Significant latency. Profiling reveals that the average time an agent waits for the result of a prediction call is 108ms, only 10% of which is spent in 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.

Balancing the components. Agents, predictors, and trainers all share the GPU as a resource, thus balance is important. This is demonstrated in the top performing configurations in Figure 3. The best

[3]Occupancy reflects the parallelism of the computation. It captures the fraction of GPU threads which are active while the GPU is being utilized. So, for example, occupancy of 76% with utilization of 63% implies roughly 48% overall thread utilization.

[4]We reduce the stride of the first DNN layer in A3C from 4 to 1, increase its filters from 16 to 32, increase the filters in the second layer from 16 to 32, and add a third convolutional layer with $64 \times 4 \times 4$ filters and stride 2. These changes also increase the number of parameters in the fully-connected layer by $4\times$.

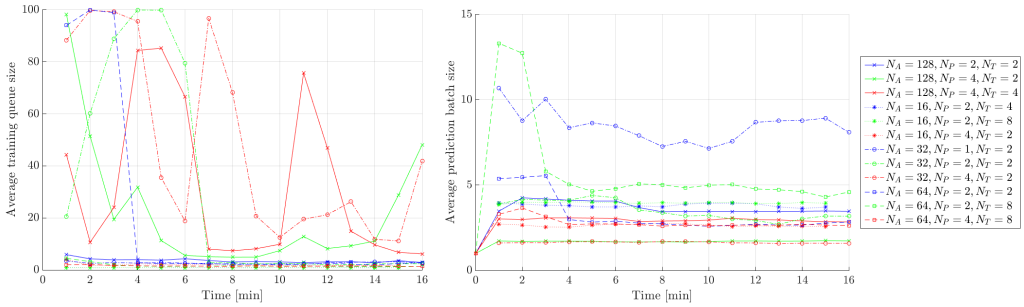

Figure 5: The average training queue size (left) and prediction batch size (right) of the top 3 performing configurations of N_P and N_T, for each N_A, for PONG and the system in Table 1.

	Atari Game Scores									Attributes		
	AMIDAR	BOXING	CENTIPEDE	NAME THIS GAME	PACMAN	PONG	QBERT	SEAQUEST	UP-DOWN	Time	PPS*	System
Human [2]	1676	10	10322	6796	15375	16	12085	40426	9896	–	–	–
A3C-1 [9]	284	34	3773	5614	3307	11	13752	2300	54525	1 day	352	CPU
A3C-4 [9]	264	60	3756	10476	654	6	15149	2355	74706	4 days	352	CPU
GA3C	218	92	7386	5643	1978	18	14966	1706	8623	1 day	1080	GPU

Table 3: Average scores on a subset of Atari games achieved by: a human player [2]; A3C after one and four days of training on a CPU [9]; and GA3C after one day of training. (*Note: predictions per second (PPS) is measured on our TensorFlow CPU and GPU implementations of A3C, while scores are from the cited publications.)

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 the speed of prediction. At the same time if the training queue is too full then training calls will dominate GPU computation thereby throttling the prediction speed. This is further confirmed by the finding that TPS and PPS plots track closely (plot omitted for space). Figure 5 shows the training queue size and prediction batch size for the top configurations. In all cases, the training queue stabilizes well below maximum capacity. Additionally, the fastest configuration has one of the largest average prediction batch sizes, yielding higher utilization of the GPU.

Game performance. Table 3 compares scores achieved by A3C on the CPU (as reported in [9]) with our TensorFlow implementation GA3C. A direct speed comparison is not possible due to the lack of reported timing results for the original A3C implementation. Results in this table show that, after one day of training, our open-source implementation achieves similar scores to A3C.

Discussion. Analysis of the computational aspects of GA3C reveals potential for using GPUs for training deeper RL models. The significant speed-up achieved with respect to a CPU implementation of the same algorithm (especially for larger DNN models) comes as a result of a flexible system capable of finding a reasonable allocation of computational resources. We believe this may be a consistent theme in deep RL implementations of the future, motivating further studies like this one and subsequent support in DNN frameworks. By open-sourcing GA3C, we allow other researchers to explore this space and investigate in detail the computational aspects of deep RL algorithms.

References

[1] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 02 2015.

[3] 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.

[4] G. Lample and D. Singh Chaplot, "Playing FPS Games with Deep Reinforcement Learning," *ArXiv e-prints*, Sept. 2016.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[6] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[7] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015.

[8] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, "Massively parallel methods for deep reinforcement learning," *CoRR*, vol. abs/1507.04296, 2015.

[9] 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.

[10] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015.

[11] ., "Open AI Gym." `https://gym.openai.com/`.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.