Investigating Integrated CPU/GPUs for Accelerating Reinforcement Learning

Alex Schiffman, Jacob Oakman, Michael Bentley

Abstract

Graphics Processing Unit (GPU) programming has increasingly provided inexpensive opportunities for utilizing large numbers of compute units. Current desktop architectures use separate physical memory for the Central Processing Unit (CPU) and GPU which imposes memory transfer latency on GPU operations. GPU programming also suffers from overhead imposed by GPU kernel startup cost. These inefficiencies are small compared to large traditional workloads carried out by GPUs but cause problems for high volume, small task workloads. These smaller workloads are characteristic of reinforcement learning algorithms. Systems like the NVIDIA Jetson single board computers (SBCs) integrate CPUs and GPUs on the same chipset with shared physical memory which can facilitate true “zero-copy” GPU memory. In this paper we investigate using these boards to speed up a cutting edge reinforcement learning algorithm known as Hindsight Experience Replay by utilizing systems with shared GPU/CPU memory and establishing persistence kernels. In our initial testing we found a 7x speedup when enabling CUDA in the PyTorch implementation on the Jetson Nano as opposed to a 2x speedup on a traditional PC. Later, an approach was implemented with the goal of evaluating a reinforcement model with heavy GPU utilization. This proved to be significantly more work than anticipated, so the results were unfortunately inconclusive.

1 Overview

Devices like the NVIDIA Jetson line of SBCs have great potential in edge computing and robotics applications due to their onboard GPU. These boards contain up to 512 CUDA cores alongside multi-core ARM CPUs. This available parallelism could prove invaluable in deep learning applications like reinforcement learning [7]. GPU acceleration is particularly important on systems which cannot support high-end CPUs due to size, power, and cost constraints. These boards have one more trick up their sleeve which is particularly interesting: their architecture shares physical memory between the CPU and GPU.

On a traditional computer, the GPU is typically on a completely separate card than the CPU and motherboard, connected over PCIe lanes. These GPU cards contain their own memory hierarchy consisting of high speed graphics memory and several layers of caches. There are several upsides to this architecture, particularly avoiding the need to maintain cache consistency between the CPU and GPU and allowing the GPU to have dedicated, purpose-built resources instead of competing with the CPU for use of traditional RAM. There are also downsides to this architecture. Important to our project is the memory transfer latency between the CPU and GPU which is inherent to these architectures. For any computation to be offloaded to the GPU, the relevant data will need to be transferred from main memory through a PCIe interface to the GPU. While this overhead is often small compared to compute time for traditional workloads, it becomes a higher percentage of total runtime for workloads based around a high volume of small tasks.

These workloads also suffer from kernel startup overhead. GPU programming has traditionally fit a model of preparing a large set of work, transferring it to the GPU, and creating a GPU kernel to complete that work and terminating. Then the result can be reclaimed by the CPU. This kernel startup overhead becomes a larger portion of overall execution time for small tasks making certain tasks ill suited for GPU programming in practice even if they are embarrassingly parallel. Reinforcement learning is an application which is often characterized by these kinds of workloads. Persistent kernels can solve this problem. These kernels are designed to never terminate and instead data is passed back and forth between the CPU and GPU and the kernel will perform operations on that data as it becomes available [4].

Our project features benchmarking an implementation of the Hindsight Experience Replay (HER) reinforcement learning algorithm. This is a particularly interesting algorithm because it allows for effective learning on a sparse reward space and performs better than other algorithms even when they are given shaped reward functions. The insight behind HER is that in human learning, we often learn as much from failures as we do successes. In particular, when an action is performed we

learn the result of that action under those circumstances. If our goal was to create the outcome that was achieved by our failed attempt, we would have succeeded. HER works by "replaying" actions as if the goal was to achieve whatever end state they caused and allowing the network to learn from these simulated successes [2].

We have adapted an existing GitHub Python repository by user TianhongDai which implements HER using PyTorch. This implementation leverages HER to complete several tasks in the OpenAI Gym MuJoCo simulated robotic arm environment. The tasks are: FetchReach-v1, FetchPush-v1, FetchPickAndPlace-v1, and FetchSlide-v1 [3]. We have chosen to focus on FetchReach-v1 because it takes the least amount of training time so it fits best in our project timeframe. Our forked version of this repository can be found here: https://github.com/LehighCSE375-AJM/hindsight-experience-replay

We intend to measure the performance effects of eliminating memory transfer overhead by using the shared physical memory architecture of NVIDIA Jetson SBCs. This will be achieved by first porting an existing reinforcement learning algorithm to run on the Jetsons in a standardized Docker container. This ported version will then be benchmarked on several Jetson boards and a traditional PC architecture. Our initial goal was to rewrite parts of the algorithm in C++ and make these new implementations callable through the existing Python code. However, this proved too complicated for the project timeframe. Instead we constructed a simple reinforcement learning application to learn a basic linear function taking advantage of the code we originally wrote to replace functionality in the HER repository and benchmarked this application with different configurations.

2 Background

Key to our project are several main technologies which we will share more detail about in this section. These include persistent kernels and CUDA memory models. Before elaborating on these it is important to clarify several concepts. When speaking about GPU programming, it is common to refer to the CPU and supporting architecture as the **Host**. The GPU is referred to as the **Device**. Finally, the **Kernel** refers to a specific set of instructions sent to the GPU for execution [1]. Confusingly, this has very little to do with the use of Kernel in reference to operating system design. We will be centering our discussions around CUDA because, while it is a proprietary technology, it has become the defacto standard for GPU programming.

2.1 Persistent Kernels

As mentioned earlier, starting a kernel carries a significant overhead for small tasks. This overhead is also non-deterministic because it relies on the GPU scheduler which can take variable amounts of time to schedule the kernel which is often unacceptable for real-time applications. The persistent kernels model (also referred to as persistent threads) avoids these issues by launching a CUDA kernel at the start of the program and causing it to continue running until the program terminates. A queue of tasks is dynamically provided to the running kernel by the CPU over the execution time and the kernel will churn through this queue, returning the result to a section of memory which can be read back by the CPU. This results in reducing the kernel start overhead to a single instance, improving performance and determinism [4].

Persistent kernels come with several downsides. The kernel will be running at all times so it is not possible to launch multiple heterogeneous kernels throughout the application. These can be approximated by using a switch statement to select from predefined operations but this is not ideal for many cases. There is no official CUDA support for persistent kernels so the kernel must perform a busy wait while waiting for a workload from the CPU to recognize when there is more work given to it. This will consume more resources and power than if the GPU were allowed to go idle. Similarly, the CPU will also have to perform a busy wait when waiting for the GPU to complete a calculation. These waits can be minimized by designing programs such that the CPU prepares the next workload for the GPU while the GPU is working [4].

2.2 CUDA Memory Models

Memory models in CUDA fall into three main categories: Device Memory, Unified (or Managed) Memory, and Host-Pinned Memory. As an interesting sidenote, Bateni et al. found that when running multiple kernels simultaneously, converting from Device Memory to Unified or Host-Pinned Memory provides memory savings but some Device Memory kernels should still be maintained to pick up idle resources when kernels using these other methods must wait due to their associated overheads [1].

2.2.1 Device Memory

The Device Memory model works by maintaining a separate section of memory for the GPU kernel. It is allocated through the CudaMalloc() API and is the default

method for CUDA. In this model, memory should be copied from the host to the device section before the kernel is started and copied back after the kernel has completed. This model suffers from overhead because data must be copied between sections of memory before computation can take place which cannot be avoided even on integrated systems. There is also memory use overhead because maintaining two copies of data requires twice the memory which can be especially devastating on constrained systems. The benefit of this model is the GPU does not need to maintain cache coherency with the CPU during computation [1].

2.2.2 Unified Memory

Also referred to as Managed Memory, Unified Memory allows the CPU and GPU to share sections of memory. This model is allocated through the cudaMallocManaged() API. Unified Memory was implemented in NVIDIA's Pascal architecture with CUDA 6.0 and maps a specific section of memory to be accessed by the host and device without explicit memory transfer [1]. Pre-Pascal GPUs can still use cudaMallocManaged() but there is significantly more overhead because they cannot page fault [6]. Without page faults the CPU is also unable to access Unified Memory while a kernel is running. This means Unified Memory is a very ineffective memory model for persistent kernels on Pre-Pascal GPUs. There are still memory transfer costs in this architecture on devices with separate physical host and device memory pools but the Jetson SBCs will not experience this because they have shared physical memory. This model also suffers from overhead from caching. Cache flushing is often required to maintain coherency between caches. Concurrent access to the same data is not allowed between the CPU and GPU for the same reason [1].

2.2.3 Host-Pinned Memory

Similar to Unified Memory, Host-Pinned Memory maintains a section of memory which is the same across the host and device. Host-Pinned Memory is allocated by using the CudaHostAlloc() API. The difference between this and Unified Memory is that it ensures coherency by forcing both the CPU and GPU to disable their caches. This allows both to access the same memory space concurrently. The clear downside here is that disabling caches makes accessing memory significantly more expensive. This memory model should only be used when concurrent data access is required. Algorithms which rely heavily on repeatedly accessing the same memory (i.e. those that benefit from cache optimization) will suffer greatly under Host-Pinned Memory [1].

2.3 OpenAI Gym and MuJoCo

The Hindsight Experience Replay repository utilizes simulated environments from OpenAI Gym. OpenAI Gym provides a wide range of these environments for the express purpose of training and comparing reinforcement learning algorithms [9]. The robotic arm simulation in Gym requires a physics engine called MuJoCo which stands for Multi-Joint dynamics with Contact. It was developed in part to facilitate robotics research and development and can be accessed with a free educational license [10].

3 Implementation

Our implementation took place over several iterations. This section will outline steps taken to complete our goal, details of our implementation, and challenges faced over the course of the project. We will start by explaining the details of porting the HER repository to the Jetson. Then explain the PyTorch functionality we have replaced by custom C++ code. We will talk through the process of integrating our code with the Python repository, challenges faced in this process, our eventual decision to exclude HER from final testing, and the reinforcement learning algorithm we tested.

3.1 HER Jetson Port

A Docker container was implemented to run the existing HER Python repository on the Jetson boards. This container relies on the boards running a recent version of JetPack. We have tested with JetPack v4.4.1. Additionally, the repository was modified to use an external host server for the MuJoCo robot simulation in order to isolate the machine learning performance. If this were not implemented, the simulation would be competing for scarce computational resources with the algorithm we are trying to benchmark leading to inconsistent and artificially poor performance. The server component was tested successfully on Ubuntu 18.04 and macOS (Big Sur) 11.0.1 with academic MuJoCo licenses, but did not work on Windows 10.

3.2 Replaced Functionality

We have attempted to rewrite the most processor intensive sections of the Python codebase. This code handles the core functionality of the neural network implementation. Hindsight Experience Replay can work with an arbitrary off-policy reinforcement learning algorithm [2]. The repository we are working with uses the Deep Deterministic Policy Gradients (DDPG) algorithm which is an actor-critic, model-free algorithm based on the deterministic policy gradient (DPG) algorithm [8].

Internal to the implementation of DDPG is an Adam optimizer. Our implementation, which lives in the cpp directory, replaces the PyTorch Tensors, Linear Layers and Adam Optimizer. We have also implemented a simple gradient descent optimizer to compare Adam with. We could have instead forked the PyTorch C++ implementation but this was deemed far more complicated than purpose-building our own version.

An approximation of a tensor was created using a flattened 2D array of doubles. Linear layers were built to use a tensor for weights and a tensor for biases. Both the critic and actor networks use four linear layers. The two hidden layers have 256 nodes. This translates to a 256 x 256 tensor object for the weights and another 256 x 1 tensor object for the weights per internal layer. Thus, operations on these objects are both processor intensive and heavily parallel. Tensor operations were initially implemented sequentially using OpenBLAS which is an optimized C library for Basic Linear Algebra Subproblems (BLAS). This library does not have bindings for element-wise multiplication or square root which is used in the Adam optimizer so these were implemented manually as basic loops. Later we will outline the changes made to implement these operations with CUDA acceleration.

The optimizers are designed to mimic the functionality in PyTorch. When initialized they are given a vector of tensor pointers which they store as a field. These tensors are retrieved from the layers of the neural network through calling the parameters function which has been designed to return the weights and bias tensors from each layer in order as a vector. Tensor objects contain a pointer to another tensor which represents the gradient. This is generated through back propagation using the mean squared error and will otherwise point to NULL. Calling the step function on the optimizer will loop through all tensors stored in the optimizer and adjust the internal values for those which have associated gradients using the corresponding algorithm. We implemented both Gradient Descent (GD) and Adam as children of a generic optimizer class so they could both be tested against each other.

3.3 Integrating Python with C++

In order to run CUDA optimized functions with the Hindsight Experience Replay code, we needed to implement a wrapper layer to interface the provided Python code with our custom C++ code. This was accomplished by using the ctypes library, which is built into the Python standard library. Since we replaced objects from or inherited from the PyTorch library such as the Actor network, Critic network, and Adam optimizer with custom implementations of these objects in C++, wrapper classes were created for these objects which could be interchanged with the use of the objects in the Hindsight Experience Replay code.

Wrapper classes were implemented in both Python and C++ in order to create a communication layer between the two codebases. The wrapper classes in C++ interact with other C++ classes we have created which are modeled around the PyTorch library. The Python wrapper classes provide the functions that are needed by the Hindsight Experience Replay code, and serve to convert data between Python data types and data types that can be passed to the C++ wrapper functions. The ctypes library limits the types of data that can be passed to C++ functions to primitive data types and arrays of primitives. Because of this, and the overhead of transferring data, we sought to minimize the amount of data transferred through the wrapper layers and do most of the processing on the C++ side. In order to do this effectively, the Python wrapper classes store pointers to the associated C++ wrapper classes, and pass mostly numeric data types and flat array representations of PyTorch tensors. We were able to get the Hindsight Experience Replay code to run with the wrapper classes, but it did not learn properly as explained in the following section.

3.4 Challenges

PyTorch hinges around a system for tracking and computing the gradients of tensor operations called AutoGrad. This system dynamically generates a graph of operations as these operations are performed on tensors so they can be tracked backwards for gradient calculations. Originally we used a basic mean squared loss function to calculate the gradient instead of manually reimplementing AutoGrad. Towards the end of our implementation when taking the final steps to integrate the repository with our C++ code, it was found that our implementation was not sufficient to replace AutoGrad.

3.5 Solution

Since we didn't implement AutoGrad, we were unable to replace PyTorch in the Hindsight Experience Replay Github repository. Instead we focused on a basic task we initially used for testing our C++ machine learning code: training it to add three numbers and multiply them by two. We trained an identical network to the hindsight experience replay critic model. Although our machine learning model is trained on the seemingly simple task of basic addition and constant multiplication, it still needs to

perform the same matrix operations as training the same model on a more complex problem. The primary difference between our model and Hindsight Experience Replay's is not the operations it performs, but rather the amount of networks it has. Hindsight Experience Replay uses both a Critic and Actor network and a copy of each for a total of four networks, whereas, we only use one critic network.

Our model consists of two primary methods, forward and backprop. The forward function feeds a Tensor input into the first linear layer of the network, then it gets that layer's output and feeds it to the next linear layer. It does this until it reaches the last layer. The output of the last layer is the model's prediction. Each linear layer consists of a weight Tensor, bias Tensor, and an activation function. When an input Tensor is fed into a layer it will multiply the input by the transpose of the weights Tensor, then it will add the bias matrix to that result. Then before returning the resulting matrix it will apply the activation function, which can be either ReLu, tanh, or no activation function. The backprop method is used to compute the error gradients with respect to the weight and bias Tensors of each linear layer. This is done by calculating the initial mean squared error gradient then feeding it to the compute_gradient method of the last linear layer. That will calculate the gradient with respect to the error for that linear layer's weight and bias matrix. It will also return the error gradient with respect to the output of the previous layer. This can then be fed to the previous linear layer's compute_gradient method. This is done until all the layers have computed their gradients.

The math done by each layer to compute the gradient consists of multiple highly parallel matrix operations including two matrix multiplications, and is a clear example of where GPU programming would provide speedup. After all the gradients are computed the backprop method runs the optimizer, which updates all the layer weights and biases based on their gradients. We train our critic model by feeding three random numbers into the forward method for it to add and multiply by two. Then we pass the prediction returned by the forward method to the backprop method along with the actual value that represents the sum of the three numbers multiplied by two. Repeating this process many times accurately trains the model to predict the correct value with minimal error.

3.6 GPU Programming

Nearly all of the operations required to train a machine learning model are matrix operations, which are highly parallelizable. We implemented a GPU accelerated version of our machine learning model by writing a device code version of each function we implemented for our initial C++ code. Many of our operations consisted of basic element-wise Tensor operations, such as multiplying each value by a constant. These are rather trivial to implement in a way that exhibits excellent memory coalescence. We also implemented the more complicated matrix operation of matrix multiplication. We programmed this operation in a rather naive way. There are options in our matrix multiplication method which say whether we want to multiply using the transpose of the first or second matrix, in a similar way to BLAS. However if either the first matrix is transposed, or the second one is not, then the matrix multiplication exhibits very poor memory coalescence. It also limits the amount of multiplications in parallel to that of the height or width of the second matrix, depending on whether it is transposed. This is monumentally bad for some operations where the height or width of the second tensor is one as only one thread does the work. This causes poor scaling when increasing the number of threads for matrix multiplication. There are also other additional more minor improvements to matrix multiplication that can be made, such as storing values that will be reused in shared memory, lowering their future read latency. When our code had a lower ratio of matrix multiplication operations we exhibited greater speedup which lends credence to the idea that our matrix multiplication operation is a significant bottleneck.

Currently our kernel is extremely persistent, as the entire program is executed on the GPU and it never has to wait for communication from the CPU. Due to time constraints we never implemented a communication channel between the CPU and GPU, but instead we run all the machine learning code directly on the GPU. Implementing the communication layer between the CPU and GPU may speed up our program, as currently a single GPU thread calculates three random values for each iteration which is a very slow GPU operation. It is unclear how this implementation tests the actual capabilities of the Jetsons' shared physical memory but it's kinda cool that it can be done.

4 Testing and Results

We conducted two rounds of testing during this project. Our initial testing looked at the HER repository and how the code performed when ported to different platforms with and without GPU acceleration activated in PyTorch. To ensure accuracy, simulations were hosted on separate machines as described in section 3.1. We found

impressive speedup characteristics across the board with the most gains seen in the low-end Jetson boards.

4.1 Test Platforms

Several platforms were used to test the efficiency of our implementations. These platforms included the Jetson Nano, Jetson TX1, Jetson AGX Xavier, and a high-end traditional desktop PC. The relevant specifications for these platforms are outlined below. The Jetson Nano and Jetson TX1 use the Maxwell architecture which does not support GPU page faulting which is needed for Unified Memory. Unified Memory was implemented in its successor, Pascal, which is implemented on the TX2. The AGX uses Volta, a descendant of Pascal, which features tensor cores.

4.1.1 Jetson Nano

The Jetson Nano boasts both a low cost and an incredibly low power consumption which makes it ideal for hobbyist and mobile projects. NVIDIA recently released a new version of the development kit board called the b01 which uses the same compute module as the previous version, a02, but has several changes to IO ports.

CPU: 4-core ARM A57 @ 1.43 GHz
GPU: 128-core Maxwell
RAM: 4GB LPDDR4 25.6GB/s

4.1.2 Jetson TX1

The Jetson TX1 was the first board developed in the Jetson line. While it has impressive computing power, NVIDIA is encouraging developers to use the newer Jetson TX2 4GB instead by releasing it at the same price point with significantly better specs.

CPU: 4-core ARM Cortex A57 @ 1.73 GHz
GPU: 256-core Maxwell
RAM: 4GB LPDDR4 25.6GB/s

4.1.3 Jetson AGX Xavier

The Jetson AGX Xavier is the highest end Jetson board in both price and performance. It is targeted towards computation intensive but power-constrained corporate and research applications with an “industrial” version explicitly offered.

CPU: 8-core ARM Carmel @ 2.26 GHz
GPU: 512-core Volta GPU with Tensor Cores
RAM: 32GB 256-bit LPDDR4 137GB/s

4.1.4 Traditional PC

Alex has a nice PC so we used that too.

CPU: 4-core Intel i7-6700k @ 4.6 GHz
GPU: GeForce GTX 1080 (2560-core)
RAM: 32GB DDR4 25.6GB/s

4.2 Initial HER Speed Evaluation

Benchmarking under several conditions was carried out to determine the performance characteristics of the HER Jetson Port. The repository was tested with different numbers of threads, with and without the GPU. These systems were executed within the same local network so there would be minimal network latency. The GPU acceleration used in this port was not updated from the default CUDA bindings in PyTorch. Interestingly, across all testing platforms, using more than a single thread for running the HER code completely destroyed performance. Consequently, the “Without GPU” sections in Fig. 1. refer to a completely sequential implementation. This CPU parallelism slowdown is worth investigating further but was not deeply analyzed because it is beyond the scope of this project.

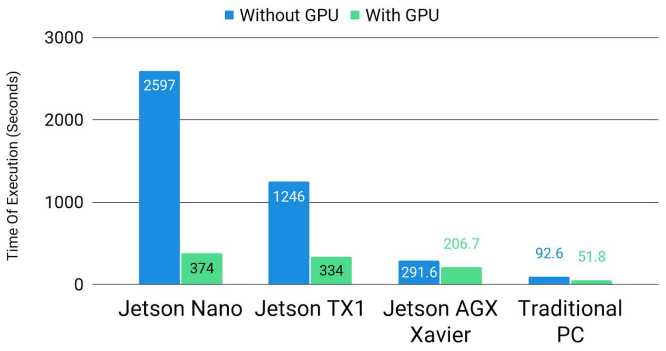

Fig. 1. Initial Results of FetchReach-v1

Fig. 1. shows the initial results of our testing with the simple HER Jetson Port. According to the README of the HER repository, GPU acceleration is not recommended unless on a powerful machine. However, our test data shows the GPU acceleration provides significant speedup for all Jetson boards with the most significant benefits for even the low-end Jetsons with a 4x speedup on the TX1 and a 7x speedup for the Nano. This shows that either the author of the repository was mistaken or the unique characteristics of the Jetsons allow them to see benefits from GPU acceleration despite being far less powerful than traditional medium and low end CUDA capable PCs. We could identify this for sure if Alex’s computer wasn’t so damn nice. This increased speedup for lower-end

systems seems to follow the idea of diminishing returns where adding more cores does not linearly increase performance because there is still a section of the overall runtime that is not parallelizable. This test is not sufficient to measure the effects of shared physical memory because this element was not sufficiently isolated, but these results are promising.

4.3 Our Implementation

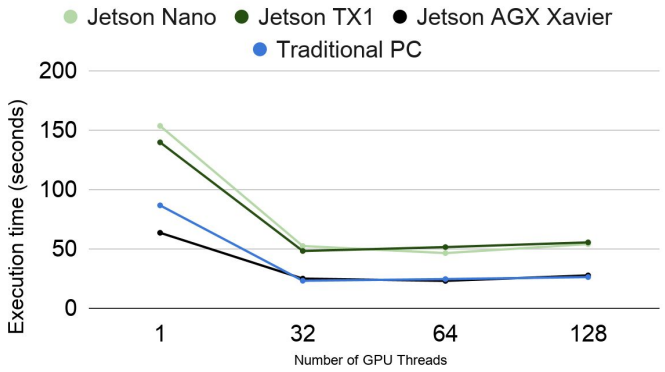

Fig. 2. Comparison of Speed with Different Numbers of GPU Threads

Our implementation was tested with 100 iterations using the bash time function. Fig. 2. shows the runtime comparisons between each platform with different numbers of allocated GPU threads. We can see that there is an initial speedup when we move from a sequential gpu program to using an entire warp but that speedup does not continue when adding more threads after that. We believe this is because adding additional warps will add an additional amount of overhead which offsets any speed gain. While this graph looks fine, it is important to note that the sequential version we wrote which runs entirely on the CPU is 40x faster than any of these. This is almost certainly due to the fact that we implemented too much code on the GPU and neglected what was more suited to the CPU.

5 Conclusion

While it is regrettable that we were not able to get HER fully integrated with our code due to issues mimicking PyTorch's AutoGrad, our initial results still speak to the efficacy of NVIDIA's Jetsons. We have shown that for the FetchReach-v1, not only does using GPU increase the performance for every system we have tested which disproves the advice in the repository README file. The speedup ratio, in fact, becomes more beneficial for lower-end hardware in our test scenario. This lends credibility to our intuition that reinforcement learning applications can benefit from GPU parallelism on this architecture despite their small-task workflows. It also demonstrates that complicated memory models and persistent kernels are not required to see significant performance improvements through GPU acceleration on these boards.

Although our implementation did not result in the kind of speedup we were looking for, we believe this is still able to be found with a more nuanced approach. Writing effective GPU acceleration requires a level of care which we sadly did not achieve under the time constraints. Going forward, it is worth reevaluating the difficulty of implementing, or approximating, the AutoGrad system so we can move forward with applying our implementations with the HER repository. Also, making further improvements to our CUDA code, specifically the matrix multiplication, will yield significant improvement in performance. Finally, it is worth testing a more traditional GPU programming model rather than trying to handle the data flow entirely within a CUDA kernel. Looking back, knowing what we know now, we likely could have done a better job in completing our initial goals. You know what they say, Hindsight Experience Replay is 20/20.

References

[1] Bateni, S., Wang, Z., Zhu, Y., Hu, Y., & Liu, C. (2020). Co-Optimizing Performance and Memory Footprint Via Integrated CPU/GPU Memory Management, an Implementation on Autonomous Driving Platform. *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* doi:10.1109/rtas48715.2020.00007

[2] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., . . . Zaremba, W. (2017). Hindsight Experience Replay. *31st Conference on Neural Information Processing Systems (NIPS 2017)*. doi:arXiv:1707.01495

[3] Hindsight Experience Replay PyTorch Implementation. Retrieved from https://github.com/TianhongDai/hindsight-experience-replay

[4] Allen, T. (2018). Improving Real-Time Performance with CUDA Persistent Threads (CuPer) on the Jetson TX2. Concurrent Real-Time White Paper. Retrieved from https://www.concurrent-rt.com/wp-content/uploads/2016/09/Improving-Real-Time-Performance-With-CUDA-Persistent-Threads.pdf

[5] PyTorch Adam Optimizer Implementation. Retrieved from https://github.com/pytorch/pytorch/blob/master/torch/csrc/
[6] Harris, M. (2017). Unified Memory for CUDA beginners. *NVIDIA Developer Blog*. Retrieved from https://developer.nvidia.com/blog/unified-memory-cuda-beginners/
[7] https://developer.nvidia.com/embedded-computing
[8] Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning.
[9] More information about OpenAI Gym can be found here https://gym.openai.com/
[10] More information about MuJoCo can be found here http://www.mujoco.org/