Performance Analysis of Reinforcement Learning for Robot Tasks with GPU Acceleration in NVIDIA Isaac Lab

Submitted To:

Dr. Kaeli

EECE5640 – High-Performance Computing – Final Project

Submitted By:

Ruben Noroian

Due Date: 4/10/25

Rationale

Since April of 2024, I've been involved in Robotics research with the Silicon Synapse Lab under Northeastern University's Institute for Experiential Robotics. Recently, students at the lab have begun exploring the potential of reinforcement learning in developing functional motion planning for a quadrupedal robot project.

In addition, reinforcement learning has seen an increased presence in robotics. One example demonstrating this is the Robotics and AI (RAI) Institute's progress on Boston Dynamics' spot, more than tripling its top speeds with reinforcement learning as opposed to traditional model predictive control (MPC). While MPC develops a software model for approximating the robots' dynamics and executes in real-time onboard, reinforcement learning trains offline, then deploying a policy to efficiently runs on the robot.

This project analyzes the efficiency of reinforcement learning training for robotics applications on NVIDIA software platforms and GPUs.

Simulation Background

Rather than physically test the robot by training a reinforcement learning policy on it, the policy can be optimized based on simulated behavior of a robot. This concept, known as Sim2Real, has gained popularity as a more cost-effective and manageable approach to developing algorithms for autonomous navigation. To accurately portray a physical robot in simulation, there exists a variety of tools, including Robot Operating System's (ROS) Gazebo simulation platform and NVIDIA's Omniverse.

Since the mechanical design of a robot is completed through SolidWorks, simulation platforms are developed to accept a CAD assembly with different file formats, including URDF, SDF, and STL.

This project uses NVIDIA's Omniverse, arguably the industry's best tool for the development and test of "Digital Twins." This title is used by NVIDIA to emphasize the similarity of physics and visualization in simulated world and physical world.

NVIDIA Platforms

Omniverse

NVIDIA possesses a suite of platforms for Physical AI development, investing in tools for organized reinforcement learning workflows on different robots. NVIDIA Omniverse is a platform including multiple software development kits (SDKs), APIs, and services for physically accurate virtual worlds. Ray tracing is enabled in these visualizations for high quality, accurate imagery. They mention use cases in synthetic data collection, autonomous vehicle simulation and virtual warehouse management.

Image 1: This image demonstrates the accuracy and quality of simulation on the NVIDIA Omniverse platform.

Isaac Sim

Built on top of Omniverse is Isaac Sim, the application connecting Omniverse to robotics-specific workflows. It’s a relatively recent technology, introduced by NVIDIA in 2019. Isaac Sim utilizes GPUs with PhysX, an SDK for efficient modeling capabilities and scalable simulation. It requires RTX-supported GPUs, requiring raytracing and rendering. Though this project runs training in headless mode, without a GUI, RTX is still a requirement for synthetic sensor data generation relating to the training.

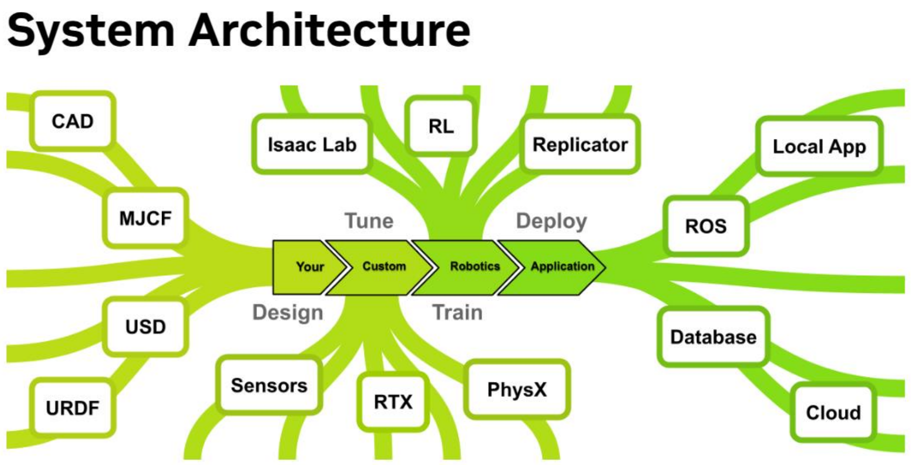


Image 2: An image illustrating the Isaac Sim workflow pipeline from importing a CAD model to sensor integration and accuracy, to training and physical deployment.

Isaac Sim possesses numerous open-source robot model humanoids, quadrupeds, manipulators, and autonomous mobile robots. As a newer platform, it is more experimental and introduces new features rapidly. This presented challenges in development environment setup, elaborated more below.

Isaac Lab

Introduced in 2023, Isaac Lab was developed initially as ORBIT with the aim to build a framework for modular and simple workflows in robotic environment setups and robot learning. Due to the growing popularity of reinforcement learning in teaching robot tasks, Isaac Lab has support for several different reinforcement learning frameworks and provides an abundance of open-source robot task training scripts to produce policies for different robots. It also enables simple incorporation into physical robot data processing through Sim2Real transfer and ROS support.

Image 3: An example of the parallelization of reinforcement learning training for a humanoid robot in simulation.

Due to the computationally intensive nature of reinforcement learning training, especially for complex tasks, I wanted to gain an understanding of how to use NVIDIA's increasingly popular training tools and how task training benefits from multi-GPU configurations.

Reinforcement Learning

RL Background

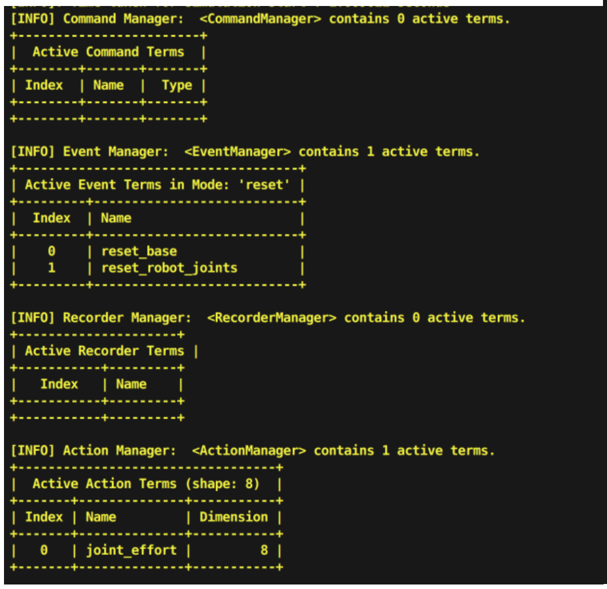
Before detailing the specific Isaac Lab reinforcement learning configurations, this section provides sufficient basic background to understand how training produces a policy.

Reinforcement learning is valuable in robotics for its ability to optimize certain tasks given to the robot. An example of a task would be to enable a humanoid to walk forward at 2 meters per second.

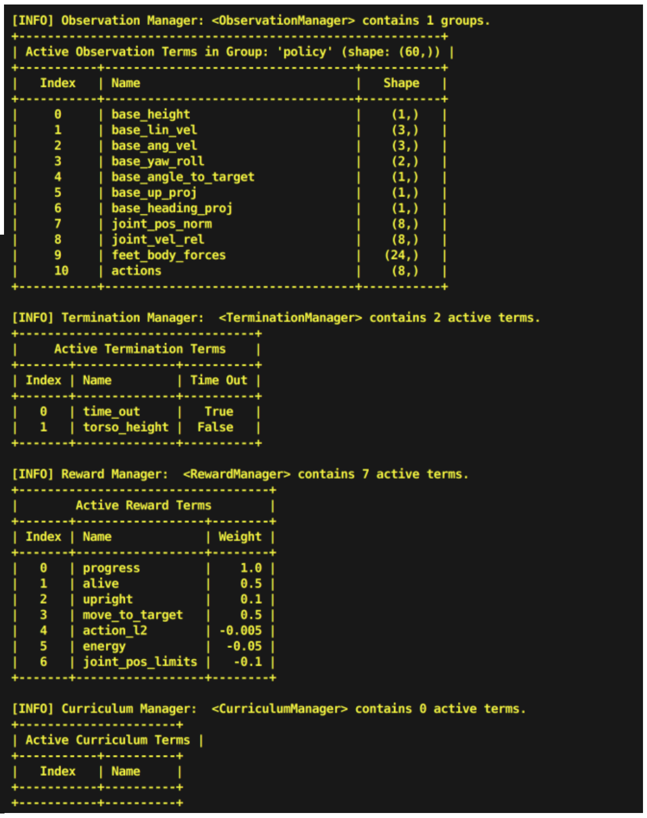
A reinforcement learning policy is the function that defines an agent's behavior based on a certain state.

The observation space defines the set of all observations and agent can receive from its environment. This is usually stored as a vector, containing data relevant to the movement or task being trained on the robot. Observations are passed to the policy, which outputs an action based on the optimization function. The action space is all actions the agent can take at any step, and the environment processes the action to update the current state and give a reward.

After the agent acts, a reward is a scalar value associating a value with the quality of the action. Since the agent's goal is to maximize the reward over time, the policy is reinforced to produce actions with higher rewards. Rewards can be associated with the deviation from an ideal state or progression of an ideal state based on a given task.



```
[INFO] Command Manager:  <CommandManager> contains 0 active terms.
+------------------------+
|  Active Command Terms  |
+--------+-------+-------+
| Index  | Name  |  Type |
+--------+-------+-------+
+--------+-------+-------+

[INFO] Event Manager:  <EventManager> contains 1 active terms.
+---------------------------------------+
| Active Event Terms in Mode: 'reset'   |
+---------+-----------------------------+
|  Index  | Name                        |
+---------+-----------------------------+
|    0    | reset_base                  |
|    1    | reset_robot_joints          |
+---------+-----------------------------+

[INFO] Recorder Manager:  <RecorderManager> contains 0 active terms.
+---------------------+
| Active Recorder Terms |
+-----------+---------+
|   Index   | Name    |
+-----------+---------+
+-----------+---------+

[INFO] Action Manager:  <ActionManager> contains 1 active terms.
+-----------------------------------+
|  Active Action Terms (shape: 8)   |
+-------+-------------+-----------+
| Index | Name        | Dimension |
+-------+-------------+-----------+
|   0   | joint_effort |         8 |
+-------+-------------+-----------+
```

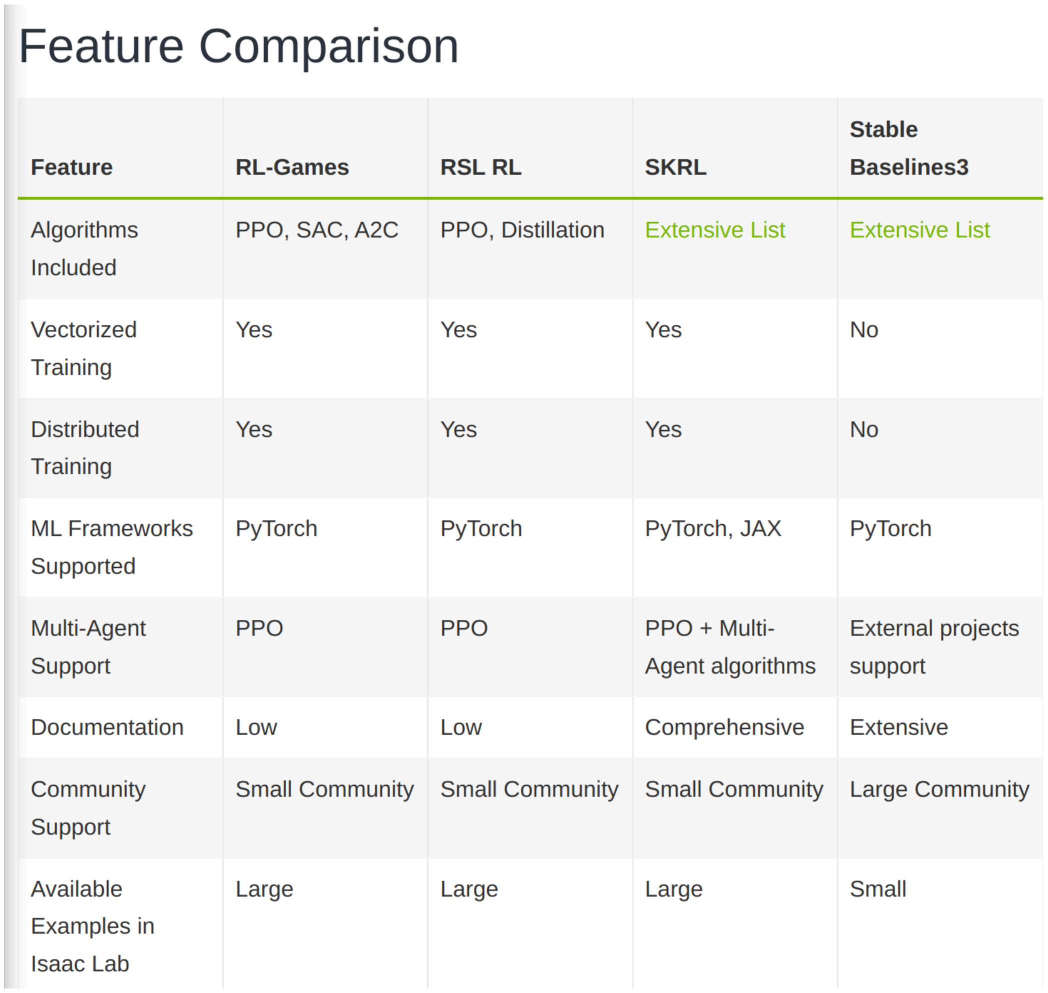


```
[INFO] Observation Manager: <ObservationManager> contains 1 groups.
+---------------------------------------------------------+
| Active Observation Terms in Group: 'policy' (shape: (60,)) |
+-----------+------------------------------------+-----------+
|   Index   | Name                               |   Shape   |
+-----------+------------------------------------+-----------+
|     0     | base_height                        |    (1,)   |
|     1     | base_lin_vel                       |    (3,)   |
|     2     | base_ang_vel                       |    (3,)   |
|     3     | base_yaw_roll                      |    (2,)   |
|     4     | base_angle_to_target               |    (1,)   |
|     5     | base_up_proj                       |    (1,)   |
|     6     | base_heading_proj                  |    (1,)   |
|     7     | joint_pos_norm                     |    (8,)   |
|     8     | joint_vel_rel                      |    (8,)   |
|     9     | feet_body_forces                   |   (24,)   |
|     10    | actions                            |    (8,)   |
+-----------+------------------------------------+-----------+

[INFO] Termination Manager:  <TerminationManager> contains 2 active terms.
+--------------------------------+
|     Active Termination Terms   |
+-------+------------+---------+
| Index | Name       | Time Out |
+-------+------------+---------+
|   0   | time_out     |   True   |
|   1   | torso_height |  False   |
+-------+------------+---------+

[INFO] Reward Manager:  <RewardManager> contains 7 active terms.
+--------------------------------+
|        Active Reward Terms       |
+-------+-----------------+--------+
| Index | Name            | Weight |
+-------+-----------------+--------+
|   0   | progress        |    1.0 |
|   1   | alive           |    0.5 |
|   2   | upright         |    0.1 |
|   3   | move_to_target  |    0.5 |
|   4   | action_l2       | -0.005 |
|   5   | energy          |  -0.05 |
|   6   | joint_pos_limits |   -0.1 |
+-------+-----------------+--------+

[INFO] Curriculum Manager:  <CurriculumManager> contains 0 active terms.
+----------------------+
| Active Curriculum Terms |
+-----------+----------+
|   Index   | Name     |
+-----------+----------+
+-----------+----------+
```

Image 4 & 5: These images depict the Isaac Lab initialization of different observations, rewards, actions, and commands. This is the setup for Isaac-Ant-v0, an environment training an ant to walk around.

Isaac Lab Frameworks

Isaac Lab supports four reinforcement learning libraries, SKRL, RSL-RL, RL-Games, and Stable-Baselines3.



Feature Comparison

Feature	RL-Games	RSL RL	SKRL	Stable Baselines3
Algorithms Included	PPO, SAC, A2C	PPO, Distillation	Extensive List	Extensive List
Vectorized Training	Yes	Yes	Yes	No
Distributed Training	Yes	Yes	Yes	No
ML Frameworks Supported	PyTorch	PyTorch	PyTorch, JAX	PyTorch
Multi-Agent Support	PPO	PPO	PPO + Multi-Agent algorithms	External projects support
Documentation	Low	Low	Comprehensive	Extensive
Community Support	Small Community	Small Community	Small Community	Large Community
Available Examples in Isaac Lab	Large	Large	Large	Small

Image 6: The specifications of each of the RL frameworks supported by Isaac Lab.

In this project's experimentation, RSL-RL, Stable Baselines3, and SKRL are not included, as their efficiency for multi-GPU parallelism was determined to be worse, where RL-Games and was more optimized for distributed PyTorch training. SKRL was initially chosen due to its detailed documentation and versatility in running different algorithms, but it showed mediocre performance when testing speed-up with multiple vs one GPU.

PyTorch is the software of choice for each framework due to its simplicity in developing deep learning models and its support for GPU acceleration.

RL Computation

Beyond the benefit of reinforcement learning in robotics, it’s important to understand what makes RL computationally intensive and why GPUs are involved in producing a policy.

Most RL algorithms, including Proximal Policy Optimization (PPO) used during experimentation, utilize deep neural networks, and GPUs are beneficial for speeding up the massively parallel matrix operations executed in neural networks.

In traditional RL training, one agent runs in one environment, optimizing its policy step by step within the environment. To maximize GPU usage, thousands of environments can be parallelized, and physics simulation can take place with tremendous speed-up.

However, not all training tasks require the same amount of memory and time to optimize the policy. In Isaac Lab, environments include humanoid locomotion training, ant locomotion training, cart balancing training, and Franka arm robot training. Each of these contains a different number of parameters for observations and action execution, and therefore they may not all perform the same provided different amounts of computer. The experiments detail later provide context into the questions being answered for Isaac Lab.

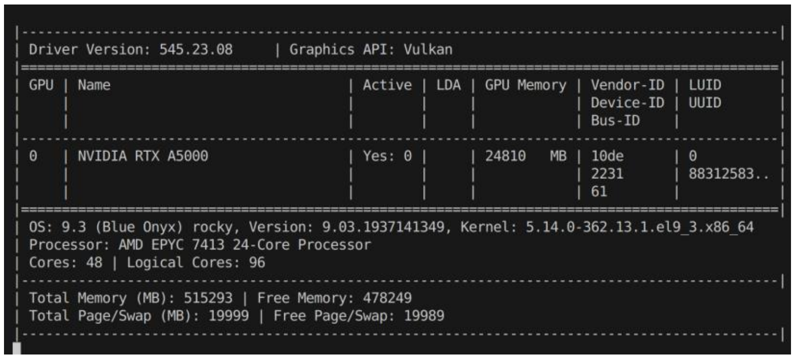
Environment Setup

Before installing Isaac Lab, Isaac Sim must be installed, and it has strict requirements. Its minimum specifications include 32 GB of RAM, an RTX3070 GPU, and 8 GB of VRAM. I encountered several issues when trying to install the most recent version of Isaac Sim, Isaac Sim 4.5.0, due to memory issues on my personal computer.

To use Isaac Lab, I instead followed its specific installation instructions, which created an Anaconda environment on the Northeastern University Explorer Cluster. For installation, an RTX-supported GPU node was acquired, and the CUDA 12.3 module was loaded. The conda environment was installed with Isaac Sim 4.5, and Isaac Lab's GitHub repository was cloned and installed with a bash script.

This initial trouble exposed to me the lack of available resources and guides for navigating the software. I will mention other issues later, many of which slow the speed of the project as solutions to potential bugs aren't always available.

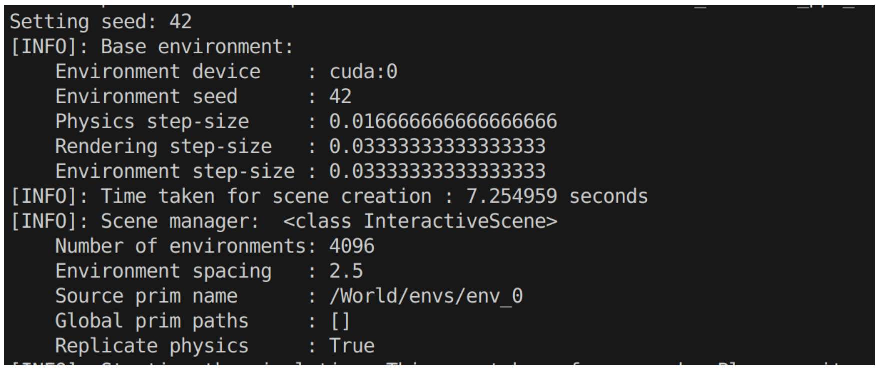
Raytracing is required for visual Isaac Lab training with Isaac Sim and Omniverse, but with the virtual Explorer Cluster headless training was executed. Despite this, errors with the PhysX SDK were encountered while attempting to train on a regular GPU, so RTX-supported GPUs were selected for each experiment. The Explorer Cluster provides L40, L40S, A5000, and A6000 GPUs with configurations up to 8 GPUs per node.



```
|---------------------------------------------------------------------------------------|
| Driver Version: 545.23.08     | Graphics API: Vulkan
|=======================================================================================|
| GPU | Name                         | Active | LDA | GPU Memory | Vendor-ID | LUID       |
|     |                              |        |     |            | Device-ID | UUID       |
|     |                              |        |     |            | Bus-ID    |            |
|---------------------------------------------------------------------------------------|
| 0   | NVIDIA RTX A5000             | Yes: 0 |     | 24810   MB | 10de      | 0          |
|     |                              |        |     |            | 2231      | 88312583.. |
|     |                              |        |     |            | 61        |            |
|=======================================================================================|
| OS: 9.3 (Blue Onyx) rocky, Version: 9.03.1937141349, Kernel: 5.14.0-362.13.1.el9_3.x86_64
| Processor: AMD EPYC 7413 24-Core Processor
| Cores: 48 | Logical Cores: 96
|---------------------------------------------------------------------------------------|
| Total Memory (MB): 515293 | Free Memory: 478249
| Total Page/Swap (MB): 19999 | Free Page/Swap: 19989
|---------------------------------------------------------------------------------------|
```

Image 7: The GPU configuration printed to the screen when initializing a task training.

The CPU for each node was also initially thought to be negligent, but due to communication between GPUs in multi-GPU configurations, was included in the experiment setups.



```
Setting seed: 42
[INFO]: Base environment:
    Environment device    : cuda:0
    Environment seed      : 42
    Physics step-size     : 0.016666666666666666
    Rendering step-size   : 0.03333333333333333
    Environment step-size : 0.03333333333333333
[INFO]: Time taken for scene creation : 7.254959 seconds
[INFO]: Scene manager:  <class InteractiveScene>
    Number of environments: 4096
    Environment spacing   : 2.5
    Source prim name      : /World/envs/env_0
    Global prim paths     : []
    Replicate physics     : True
```

Image 8: The parameters in the configuration file printed to the screen during task training initialization.

Experimentation

Experiment 1 – Strong Scaling

I chose to first investigate the strong scaling capabilities of an Isaac Lab environment, determining the performance versus the number of GPUs. To test strong scaling, the training workload remained constant across different GPU counts.

The task used was Isaac-Cartpole-Direct-v0, where the objective is to keep a pole vertical on a mobile cart. This was selected due to its relative simplicity compared to tasks with larger, more complex observation spaces such as Humanoid training. Cartpole was executed using the RL_Games framework with PyTorch distributed for multi-GPU configurations.

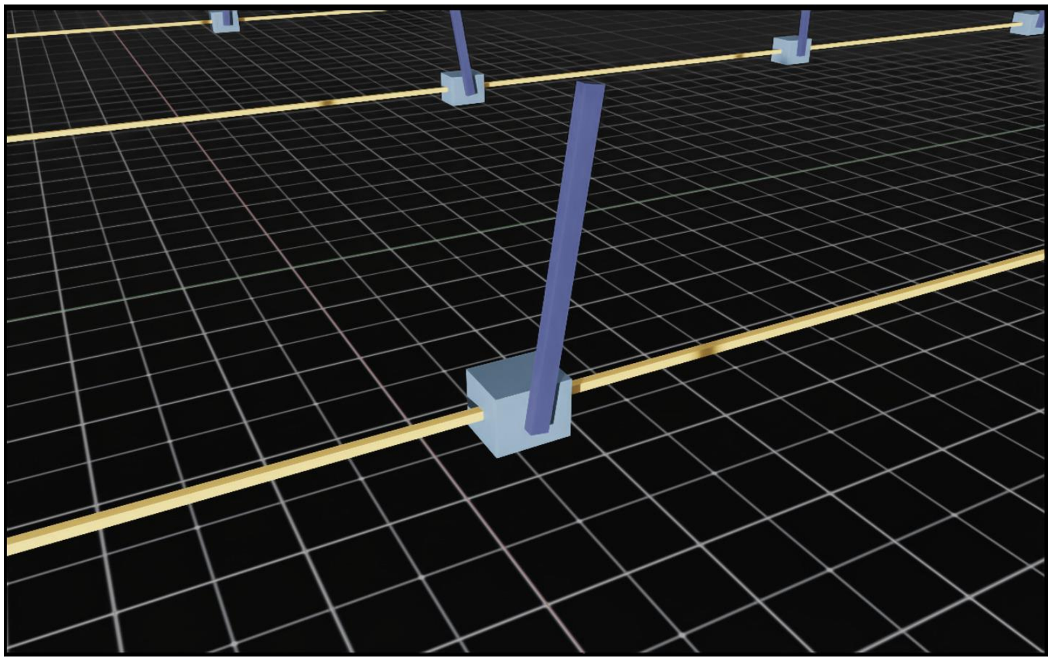


Image 9: A visual of the cartpole during parallel task training.

This experiment ran the same task training with RL_Games at 1, 2, 4, and 8 GPUs. It used 4096 parallel environments per GPU, with 150 epochs of training. The GPU selected for this training was the L40, with host CPU Intel Xeon Gold with 36 total cores across 2 sockets.

The L40, released in 2022, uses NVIDIA's Ada Lovelace Architecture, with 142 third-generation raytracing cores, 568 fourth-generated tensor cores and larger, 48GB GPU memory. Each graphics processing cluster (GPC) possesses 6 texture processing clusters (TPCs), with 12 streaming multiprocessors (SMs). Within each SM are four tensor cores and an RT core. Though these experiments did not use the simulation platform, raytracing is essential for collecting synthetic sensor data for training environments. In addition to better performing processing units, this architecture's memory hierarchy contains a 16x larger L2 cache, improving ray-tracing operations over previous GPUs. More detail will be provided on GPU architectures in experiment 3, which evaluates their differences more closely.

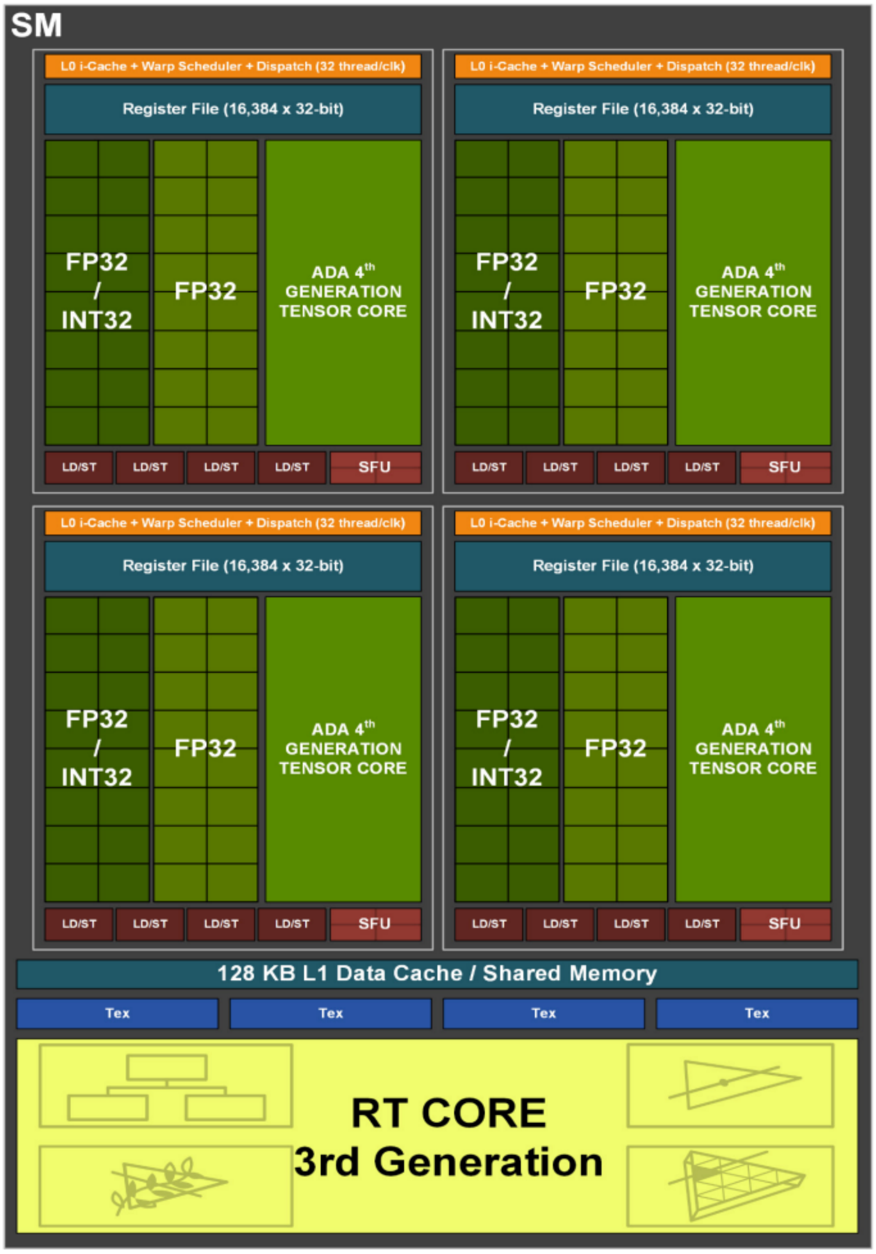
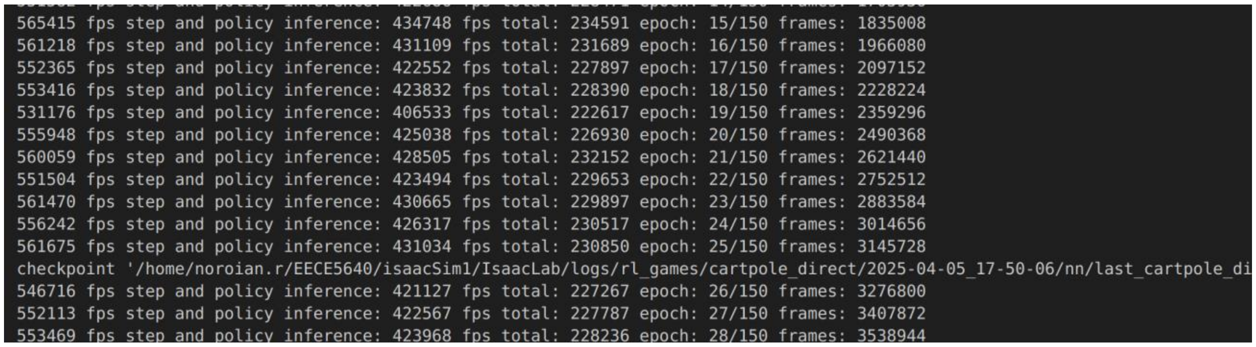

SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
ADA 4th GENERATION TENSOR CORE
LD/ST
SFU
128 KB L1 Data Cache / Shared Memory
Tex
RT CORE 3rd Generation

Image 10: The streaming multi-processor (SM) diagram for Ada Lovelace architecture provided by its whitepaper.

The RL_Games produces several different metrics for efficiency. It provides FPS Step, the number of physics simulation steps processed per second, Environment Step & Inference FPS, the speed at which the policy network evaluates actions, and FPS total, the slowest process, which encapsulates the speed of end-to-end training speed.



```
565415 fps step and policy inference: 434748 fps total: 234591 epoch: 15/150 frames: 1835008
561218 fps step and policy inference: 431109 fps total: 231689 epoch: 16/150 frames: 1966080
552365 fps step and policy inference: 422552 fps total: 227897 epoch: 17/150 frames: 2097152
553416 fps step and policy inference: 423832 fps total: 228390 epoch: 18/150 frames: 2228224
531176 fps step and policy inference: 406533 fps total: 222617 epoch: 19/150 frames: 2359296
555948 fps step and policy inference: 425038 fps total: 226930 epoch: 20/150 frames: 2490368
560059 fps step and policy inference: 428505 fps total: 232152 epoch: 21/150 frames: 2621440
551504 fps step and policy inference: 423494 fps total: 229653 epoch: 22/150 frames: 2752512
561470 fps step and policy inference: 430665 fps total: 229897 epoch: 23/150 frames: 2883584
556242 fps step and policy inference: 426317 fps total: 230517 epoch: 24/150 frames: 3014656
561675 fps step and policy inference: 431034 fps total: 230850 epoch: 25/150 frames: 3145728
checkpoint '/home/noroian.r/EECE5640/isaacSim1/IsaacLab/logs/rl_games/cartpole_direct/2025-04-05_17-50-06/nn/last_cartpole_di
546716 fps step and policy inference: 421127 fps total: 227267 epoch: 26/150 frames: 3276800
552113 fps step and policy inference: 422567 fps total: 227787 epoch: 27/150 frames: 3407872
553469 fps step and policy inference: 423968 fps total: 228236 epoch: 28/150 frames: 3538944
```

Image 10: The output of the RL_Games framework training over time during Cartpole training.

These trainings were produced the following performance results:

# GPUs	FPS Step	Environment Step & Inference FPS	FPS total	Per Epoch Time
1	590,322	463,455	243,988	2.14 seconds
2	1,146,000	864,000	422,000	1.24 seconds
4	1,672,000	1,208,000	608,000	0.862 seconds
8	3,640,564	2,647,220	896,156	0.585 seconds

Image 11: A table of results from the Issac-Cartpole-Direct-v0 trainings with different #'s of GPUs.

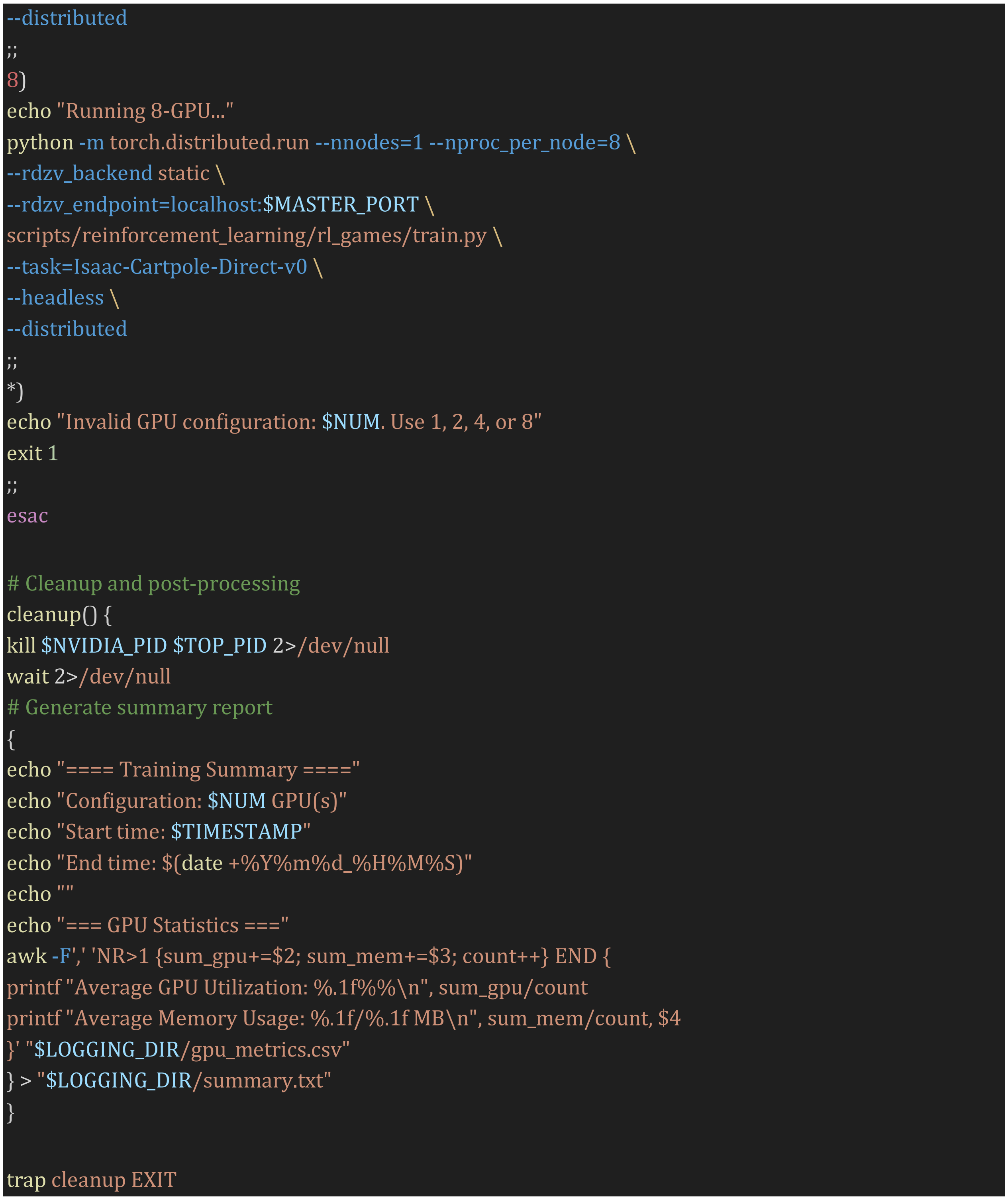
Based on the appearance of these results, parallel training did introduce efficiency, but a bash script was written to better profile the CPU and GPUs while running.

```
#!/bin/bash
#set -x

find_available_port() {
local max_attempts=100 attempt=0 port
while (( attempt++ < max_attempts )); do
port=$((29500 + RANDOM % 100)) # 29500-29599 range
if (echo >/dev/tcp/127.0.0.1/$port) 2>/dev/null; then
sleep 0.1
continue
fi
echo $port
return 0
done
echo "ERROR: Failed to find available port after $max_attempts attempts" >&2
```

```
return 1
}

if ! MASTER_PORT=$(find_available_port); then
exit 1
fi
export MASTER_PORT

ISAACLAB_PATH="$HOME/EECE5640/isaacSim1/IsaacLab"
LOGGING_DIR="$HOME/EECE5640/FinalProj/Logging"

NUM="$1"

# Create logging directory with timestamp
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
LOGGING_DIR="$LOGGING_DIR/run_${NUM}_${TIMESTAMP}_GPU"
mkdir -p "$LOGGING_DIR"

cd "$ISAACLAB_PATH" || { echo "Failed to cd to $ISAACLAB_PATH"; exit 1; }

# Clear port conflicts
fuser -k 29500/tcp 2>/dev/null

# Enhanced GPU monitoring
{
echo "timestamp,utilization.gpu,memory.used,memory.total,temperature.gpu,power.draw" >
"$LOGGING_DIR/gpu_metrics.csv"
nvidia-smi \
--query-gpu=timestamp,utilization.gpu,memory.used,memory.total,temperature.gpu,power.draw \
--format=csv -l 1 | \
grep -v "timestamp" >> "$LOGGING_DIR/gpu_metrics.csv" &
NVIDIA_PID=$!
}

{
# Extended timeout with progress reporting
TIMEOUT=120 # Increased from 30 to 120 seconds
ATTEMPT=0
echo "Waiting for training processes to start..." > "$LOGGING_DIR/cpu_metrics.log"
while [ $TIMEOUT -gt 0 ]; do
```

```
TRAINING_PIDS=$(pgrep -f
"python.*$ISAACLAB_PATH/scripts/reinforcement_learning/rl_games/train.py")
# Alternative process detection if pgrep fails
if [ -z "$TRAINING_PIDS" ]; then
TRAINING_PIDS=$(ps -ef | awk '/[p]ython.*train\.py/{print $2}' | tr '\n' ' ')
fi
if [ -n "$TRAINING_PIDS" ]; then
echo "Found PIDs: $TRAINING_PIDS" >> "$LOGGING_DIR/cpu_metrics.log"
break
fi
# Progress reporting every 10 seconds
if (( ATTEMPT % 10 == 0 )); then
echo "[$(date)] Waiting... ($TIMEOUT seconds remaining)" >> "$LOGGING_DIR/cpu_metrics.log"
# Debugging info:
ps aux | grep python >> "$LOGGING_DIR/cpu_metrics.log"
fi
sleep 1
((TIMEOUT--))
((ATTEMPT++))
done

if [ -z "$TRAINING_PIDS" ]; then
{
echo "CRITICAL: No training processes found after 120 seconds"
echo "Debug info:"
echo "--- Active Python Processes ---"
ps aux | grep -i python
echo "--- GPU Status ---"
nvidia-smi
echo "--- Recent Errors ---"
dmesg | tail -n 20
} >> "$LOGGING_DIR/cpu_metrics.log"
exit 1
fi

# Header for CPU metrics
echo "timestamp,user,system,idle,thread_pid,thread_cpu,thread_mem,command" >
"$LOGGING_DIR/cpu_metrics.tmp"
# Monitor with detailed output
top -b -d 1 -H -p $(echo $TRAINING_PIDS | tr ' ' ',') | \
awk '
BEGIN {print_header=1}
```

```
/^top -/ {
timestamp=$3
if(print_header) {
print "timestamp,user,system,idle,thread_pid,thread_cpu,thread_mem,command"
print_header=0
}
}
/^%Cpu/ {cpu_user=$2; cpu_system=$4; cpu_idle=$8}
/^[0-9]/ {printf "%s,%s,%s,%s,%s,%s,%s,%s\n",
timestamp, cpu_user, cpu_system, cpu_idle,
$1, $9, $10, $12}
' >> "$LOGGING_DIR/cpu_metrics.tmp" 2>&1
# Rotate log file to ensure header is preserved
mv "$LOGGING_DIR/cpu_metrics.tmp" "$LOGGING_DIR/cpu_metrics.log"
} &
TOP_PID=$!

# Run training
case "$NUM" in
1)
echo "Running single-GPU..."
./isaaclab.sh -p scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Cartpole-Direct-v0 \
--headless
;;
2)
echo "Running 2-GPU..."
python -m torch.distributed.run --nnodes=1 --nproc_per_node=2 \
--rdzv_backend static \
--rdzv_endpoint=localhost:$MASTER_PORT \
scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Cartpole-Direct-v0 \
--headless \
--distributed
;;
4)
echo "Running 4-GPU..."
python -m torch.distributed.run --nnodes=1 --nproc_per_node=4 \
--rdzv_backend static \
--rdzv_endpoint=localhost:$MASTER_PORT \
scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Cartpole-Direct-v0 \
--headless \
```



```bash
--distributed
;;
8)
echo "Running 8-GPU..."
python -m torch.distributed.run --nnodes=1 --nproc_per_node=8 \
--rdzv_backend static \
--rdzv_endpoint=localhost:$MASTER_PORT \
scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Cartpole-Direct-v0 \
--headless \
--distributed
;;
*)
echo "Invalid GPU configuration: $NUM. Use 1, 2, 4, or 8"
exit 1
;;
esac

# Cleanup and post-processing
cleanup() {
kill $NVIDIA_PID $TOP_PID 2>/dev/null
wait 2>/dev/null
# Generate summary report
{
echo "==== Training Summary ===="
echo "Configuration: $NUM GPU(s)"
echo "Start time: $TIMESTAMP"
echo "End time: $(date +%Y%m%d_%H%M%S)"
echo ""
echo "=== GPU Statistics ==="
awk -F',' 'NR>1 {sum_gpu+=$2; sum_mem+=$3; count++} END {
printf "Average GPU Utilization: %.1f%%\n", sum_gpu/count
printf "Average Memory Usage: %.1f/%.1f MB\n", sum_mem/count, $4
}' "$LOGGING_DIR/gpu_metrics.csv"
} > "$LOGGING_DIR/summary.txt"
}

trap cleanup EXIT
```

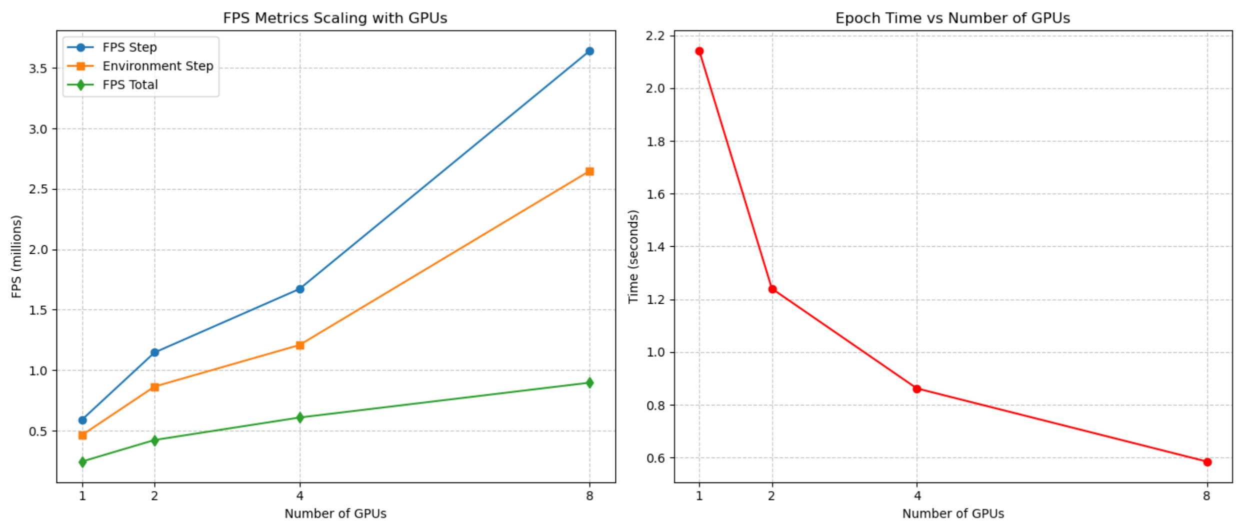
Image 12: The bash script for testing in experiment 1. Running ./exp1_bench.sh <# GPUs> enabled users to select the number of GPUs to test with.

This bash script enabled profiling by finding an available port for training to run on while logging CPU and GPU usage took place in the background. In GPU profiling, each GPU's utilization percentage, memory usage, total memory, temperature, and power drawn were collected into a CSV file. For the CPU, each process ID's CPU usage was logged in real-time across the entire training of the RL policy. A summary text file was created to list the configuration of the specific experiment and metrics including average memory and GPU usage across the training.

This data collection made understanding the results of the first experiment much easier.

Analysis of Experiment 1

To exhibit effective strong scaling, an application should see proportional speedup with the number of resources applied to a constant problem size. Diagnosing the causes of imperfect results was a multi-step process. Visuals were developed to better interpret the test outputs.



Images 13 & 14: The FPS scaling per GPU workload, and the Epoch time per GPU.

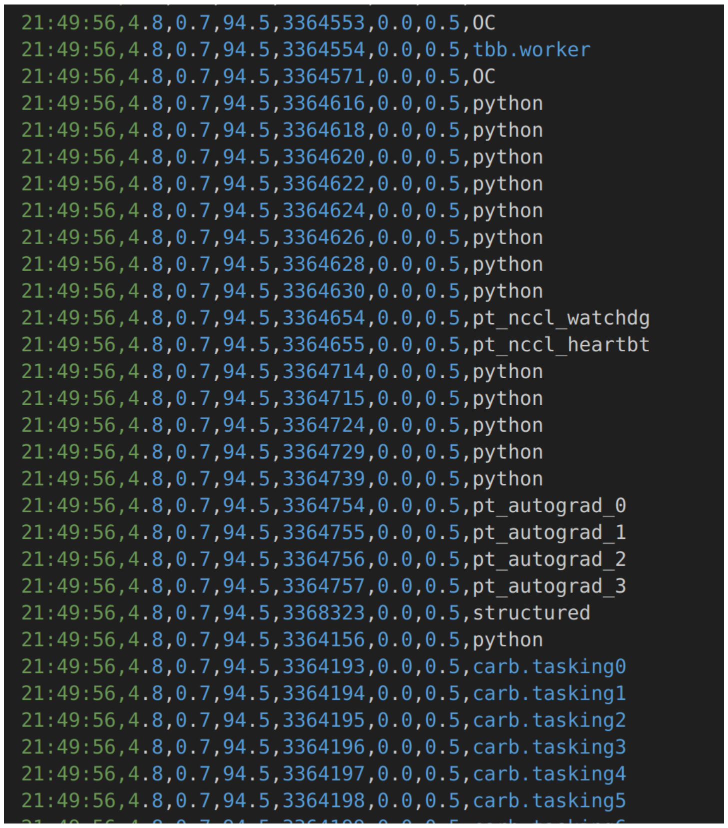
Though NVIDIA does provide its own benchmarks, which include the Cartpole task training, the results received in this project were not exactly replicated.

These results do show good strong scaling up to 4 GPUs, with diminishing returns at 8 GPUs. From 1 to 2 GPUs, the FPS Step, FPS Total, and Epoch Time scale by 1.94 and 1.73 times respectively, slightly below the expected 2x speedup. From 1 to 4 GPUs, the FPS values are 2.83 and 2.48 times, further below the expected 4x speedup. At 8 GPUs, speedup is 6.17x for FPS Step but decreases to below 4x for other metrics.

Similarly, per-epoch time improved from 2.14 seconds on 1 GPU to 0.862 on 4 GPUs but did not scale well from 4 to 8 GPUs with 0.585 seconds per epoch.

The inconsistency in scaling up to 8 GPUs was investigated using the bash scripts data outputs for CPU and GPU.

Initially, it was unclear what role the CPU played in coordinating parallelism for policy training, as NVIDIA's benchmarking results show clean scaling with scripts that do not modify CPU thread count or utilization.



```
21:49:56,4.8,0.7,94.5,3364553,0.0,0.5,OC
21:49:56,4.8,0.7,94.5,3364554,0.0,0.5,tbb.worker
21:49:56,4.8,0.7,94.5,3364571,0.0,0.5,OC
21:49:56,4.8,0.7,94.5,3364616,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364618,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364620,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364622,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364624,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364626,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364628,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364630,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364654,0.0,0.5,pt_nccl_watchdg
21:49:56,4.8,0.7,94.5,3364655,0.0,0.5,pt_nccl_heartbt
21:49:56,4.8,0.7,94.5,3364714,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364715,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364724,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364729,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364739,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364754,0.0,0.5,pt_autograd_0
21:49:56,4.8,0.7,94.5,3364755,0.0,0.5,pt_autograd_1
21:49:56,4.8,0.7,94.5,3364756,0.0,0.5,pt_autograd_2
21:49:56,4.8,0.7,94.5,3364757,0.0,0.5,pt_autograd_3
21:49:56,4.8,0.7,94.5,3368323,0.0,0.5,structured
21:49:56,4.8,0.7,94.5,3364156,0.0,0.5,python
21:49:56,4.8,0.7,94.5,3364193,0.0,0.5,carb.tasking0
21:49:56,4.8,0.7,94.5,3364194,0.0,0.5,carb.tasking1
21:49:56,4.8,0.7,94.5,3364195,0.0,0.5,carb.tasking2
21:49:56,4.8,0.7,94.5,3364196,0.0,0.5,carb.tasking3
21:49:56,4.8,0.7,94.5,3364197,0.0,0.5,carb.tasking4
21:49:56,4.8,0.7,94.5,3364198,0.0,0.5,carb.tasking5
```

Image 15: A snapshot of the CPU logging output during training.

Image 15 is the appearance of each of the CPU logging files recorded for each GPU count during testing. The CPU, in this case, is 94.5% idle, and was almost always during training, indicating that the workload was GPU bound. With this information, I pivoted to GPU analysis.

The most probable cause for diminishing returns at 8 GPUs was determined to be PyTorch multi-GPU communication through NVIDIA Collective Communication's Library (NCCL) and global interpreter lock (GIL) contention for too simple of a task.

The Isaac-Cartpole-Direct-v0 task state space includes four features, the pole angle, pole velocity, and cart position and velocity. This is a small state space, and the action space is only one-dimensional, the force to direct the cart. Additionally, the reward function for the policy includes only simple scalar evaluations, with small neural networks. This was initially selected due to its inclusion in Isaac Lab's own benchmarks, but their results did not include 8 GPUs.

The simplicity of this task was underscored by the GPU utilization for each GPU count. GPU utilization was collected using the previously mentioned bash script.

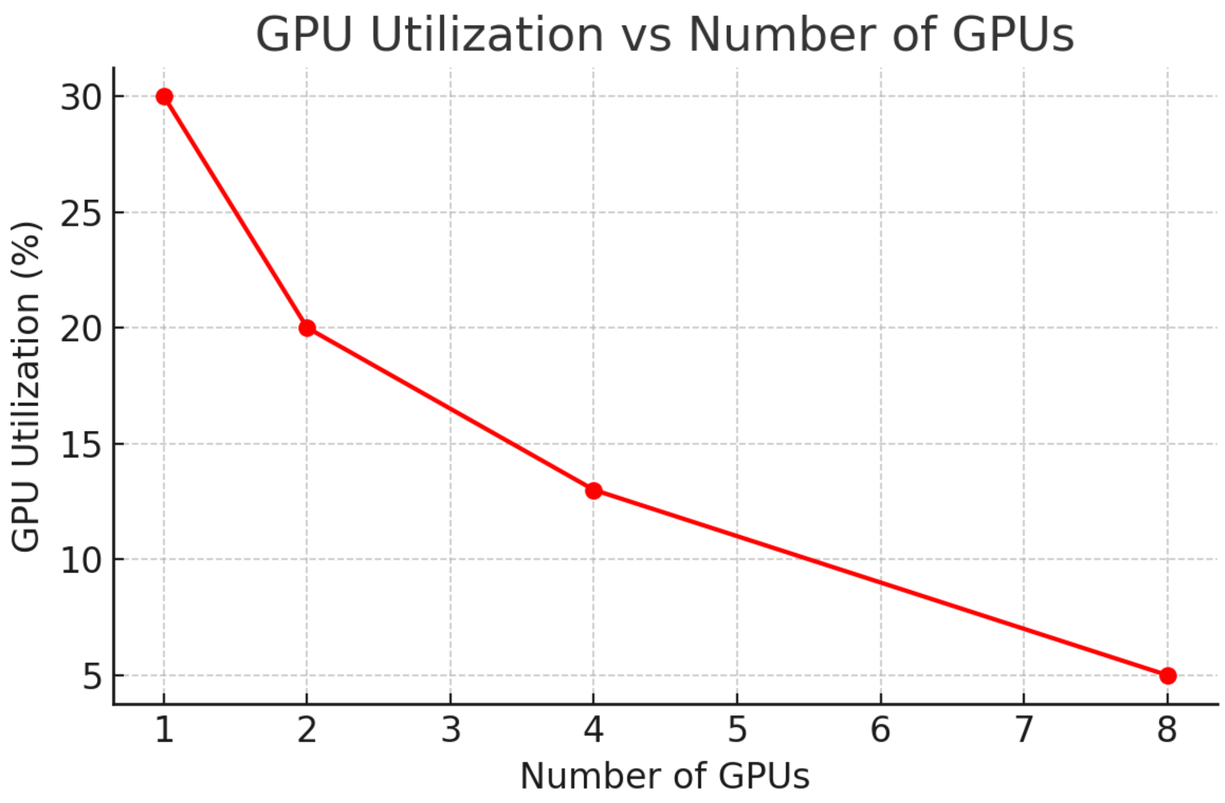


Image 16: The GPU utilization for each GPU count given the Cartpole training task.

Though the total GPU utilization was greater for 8 GPUs, distributed training introduces communication and atomic operation overhead.

NCCL enables communication between GPUs and nodes, optimized to achieve low latency over NVLink and PCIe interconnects. This communication is utilized during reinforcement learning for sharing parameters and synchronizing gradients for each step. With 8 GPUs, the amount of data shared grows, and the collective operations on this data introduce higher latency that outweighs benefits in GPU parallelization.

In addition to NCCL, Python's GIL introduces bottlenecks in multi-threaded environments as well. GIL is a mutex allowing one thread to control the Python interpreter, and though not often a bottleneck for larger tasks, PyTorch spawns separate GPU processes with inter-process contention, which for tiny workloads, becomes significant at 8 GPUs.

Experiment 2 – Weak Scaling

In this experiment, weak scaling was evaluated by scaling GPU count with the number of environments for a robot task.

Isaac Lab enables argument modification of num_envs when deploying the command for RL training, allowing the user to set the number of environments for training. The number of environments is the number of simultaneous simulations for training, and GPUs split up the number of environments to manage training with greater parallelism.

The task for this experiment was Isaac-Ant-v0. This is a quadrupedal robot simulation for training legged locomotion. The observation space is a 36-dimensional vector including base pose, velocities, joint measurements, sensor readings, and goal locations. The action space is an eight-dimensional vector representing the degrees of freedom and is evaluated upon a reward function encouraging upright balance and movement in the desired direction. Its greater complexity compared to cartpole made it the choice for weak scaling, as I believed the computation would outweigh any major overhead.

During the setup of this experiment, I determined the minimum number of environments is dependent on two metrics, mini-batch size, and horizon length. These values, initially set to 16384 and 16 in the PPO configuration file, required the environment count to be, at minimum, 1024. The mini-batch size is the number of samples per gradient update step, and horizon length is the number of steps an environment is run for before collecting samples for evaluation. Understanding these parameters was time consuming due to their unclear implications on training efficiency. After some modifications, they were left unchanged, and environments were scaled to 1024, 2048, 4096, and 8192 for GPU counts of 1, 2, 4, and 8.

In this case, an Explorer Cluster node was acquired with 8 L40 GPUs as well. The RL_Games framework was reused for consistency across experiments and produced the same FPS related results that are shown below. The table below, for weak scaling, is implemented into a graph, where the performance per GPU is determined.

# GPUs	# Envs	FPS Step	Environment Step & Inference FPS	FPS Total
1	1024	119561	100864	89792
2	2048	243482	219432	183778
4	4096	537280	435808	344864
8	8192	994,840	817,192	485,944

Image 17: The resulting performance given the number of GPUs and number of parallel environments.

Analysis of Experiment 2

These results, shown in the graph below, demonstrated particularly good weak scaling, though there was a repeated reduction in performance with 8 GPUs.

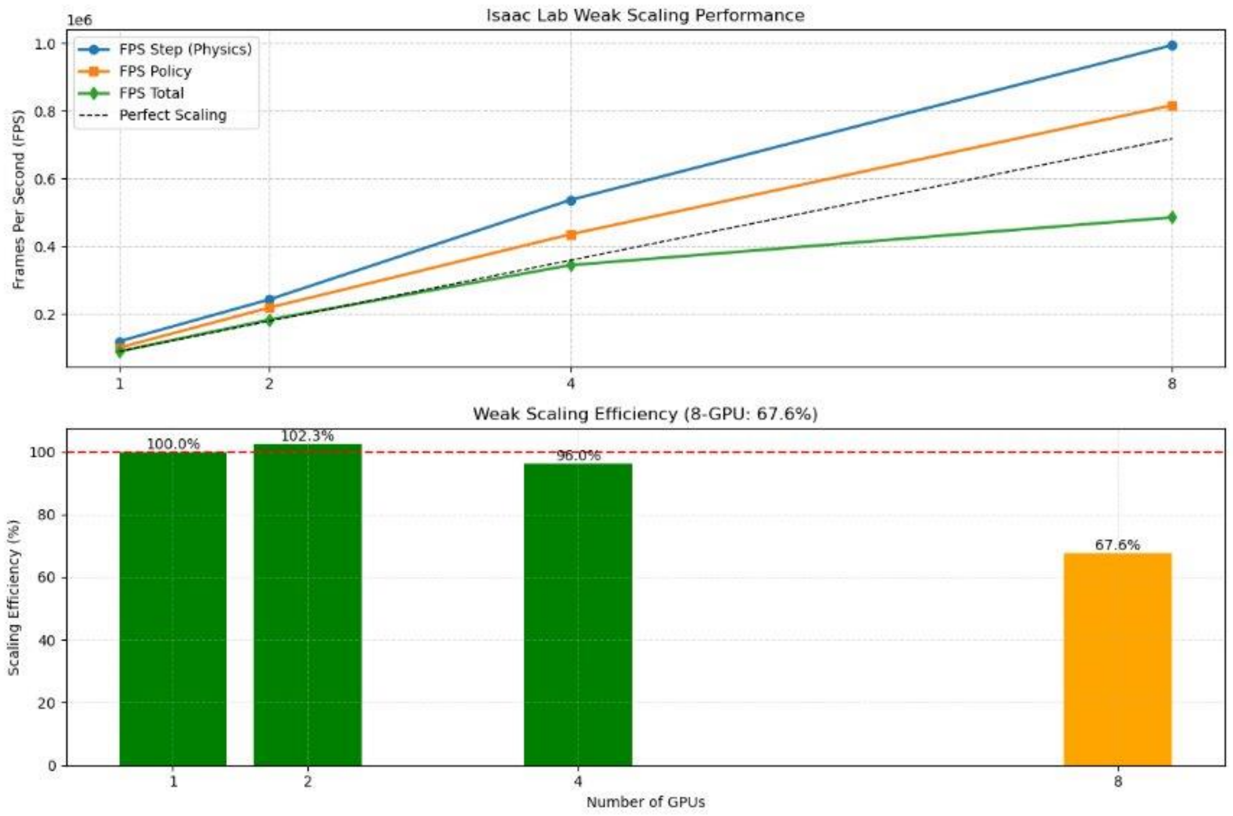
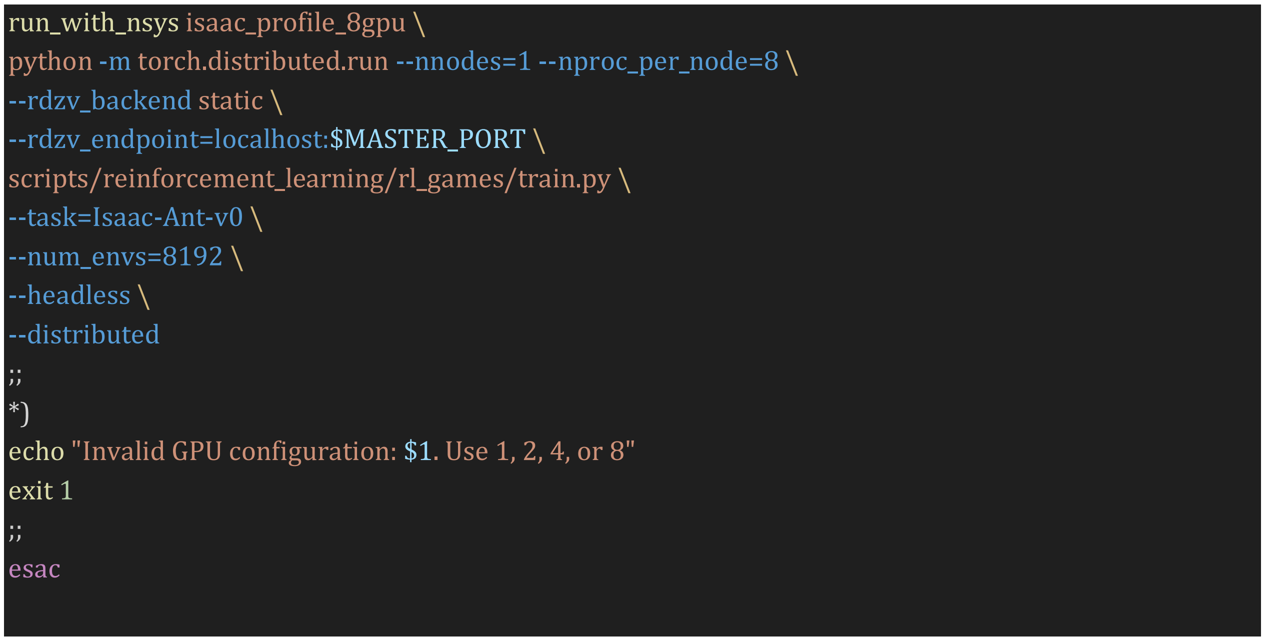


Image 18: The results of performance per GPU for scaled workloads, and a bar chart for comparisons to expected performance.

This good weak scaling demonstrates the consistency of a parallel workload in Isaac Lab and its ability to handle larger problems without compromising efficiency. Though the reasoning behind these results follows the previous experiment's, I wrote a bash script for launching tests with Nvidia Nsight analysis.

```
#!/bin/bash

nsys --version
#set -x

find_available_port() {
local max_attempts=100 attempt=0 port
while (( attempt++ < max_attempts )); do
port=$((29500 + RANDOM % 100)) # 29500-29599 range
if (echo >/dev/tcp/127.0.0.1/$port) 2>/dev/null; then
sleep 0.1
continue
fi
echo $port
return 0
done
echo "ERROR: Failed to find available port after $max_attempts attempts" >&2
return 1
}

if ! MASTER_PORT=$(find_available_port); then
exit 1
fi
export MASTER_PORT

ISAACLAB_PATH="$HOME/EECE5640/isaacSim1/IsaacLab"
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
OUTPUT_DIR="${SCRIPT_DIR}/Exp2Nsight"
mkdir -p "$OUTPUT_DIR"
# Clear port conflicts
fuser -k 29500/tcp 2>/dev/null

cd "$ISAACLAB_PATH" || { echo "Failed to cd to $ISAACLAB_PATH"; exit 1; }

run_with_nsys() {
local outfile=$1
shift
echo ">>> Running Nsight Systems: ${OUTPUT_DIR}/${outfile}.qdrep"
nsys profile -o "${OUTPUT_DIR}/${outfile}" \
--trace=cuda,nvtx,osrt \
--sample=cpu \
--stop-on-exit true \
"$@" 2>&1 | tee "${OUTPUT_DIR}/nsys_${outfile}.log"
echo ">>> Nsight profile attempt finished"
echo ">>> Expected output: ${OUTPUT_DIR}/${outfile}.qdrep"
}

case "$1" in
1)
echo "Running single-GPU with Nsight Systems..."
run_with_nsys isaac_profile_1gpu \
./isaaclab.sh -p scripts/reinforcement_learning/rl_games/train.py \
--task Isaac-Ant-v0 \
--num_envs=1024 \
--headless
;;
2)
echo "Running 2-GPU with Nsight Systems..."
run_with_nsys isaac_profile_2gpu \
python -m torch.distributed.run --nnodes=1 --nproc_per_node=2 \
--rdzv_backend static \
--rdzv_endpoint=localhost:$MASTER_PORT \
scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Ant-v0 \
--num_envs=2048 \
--headless \
--distributed
;;
4)
echo "Running 4-GPU with Nsight Systems..."
run_with_nsys isaac_profile_4gpu \
python -m torch.distributed.run --nnodes=1 --nproc_per_node=4 \
--rdzv_backend static \
--rdzv_endpoint=localhost:$MASTER_PORT \
scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Ant-v0 \
--num_envs=4096 \
--headless \
--distributed
;;
8)
echo "Running 8-GPU with Nsight Systems..."
```



```
run_with_nsys isaac_profile_8gpu \
python -m torch.distributed.run --nnodes=1 --nproc_per_node=8 \
--rdzv_backend static \
--rdzv_endpoint=localhost:$MASTER_PORT \
scripts/reinforcement_learning/rl_games/train.py \
--task=Isaac-Ant-v0 \
--num_envs=8192 \
--headless \
--distributed
;;
*)
echo "Invalid GPU configuration: $1. Use 1, 2, 4, or 8"
exit 1
;;
esac
```

Image 19: The bash script for NVIDIA Nsight deployment of RL training with Isaac Lab.

To utilize the script, its command was run with an input of 1, 2, 4, or 8 GPUs. I downloaded the nsys-rep files generated from this script and copied them to my home computer, running nsys-ui to visualize the output.

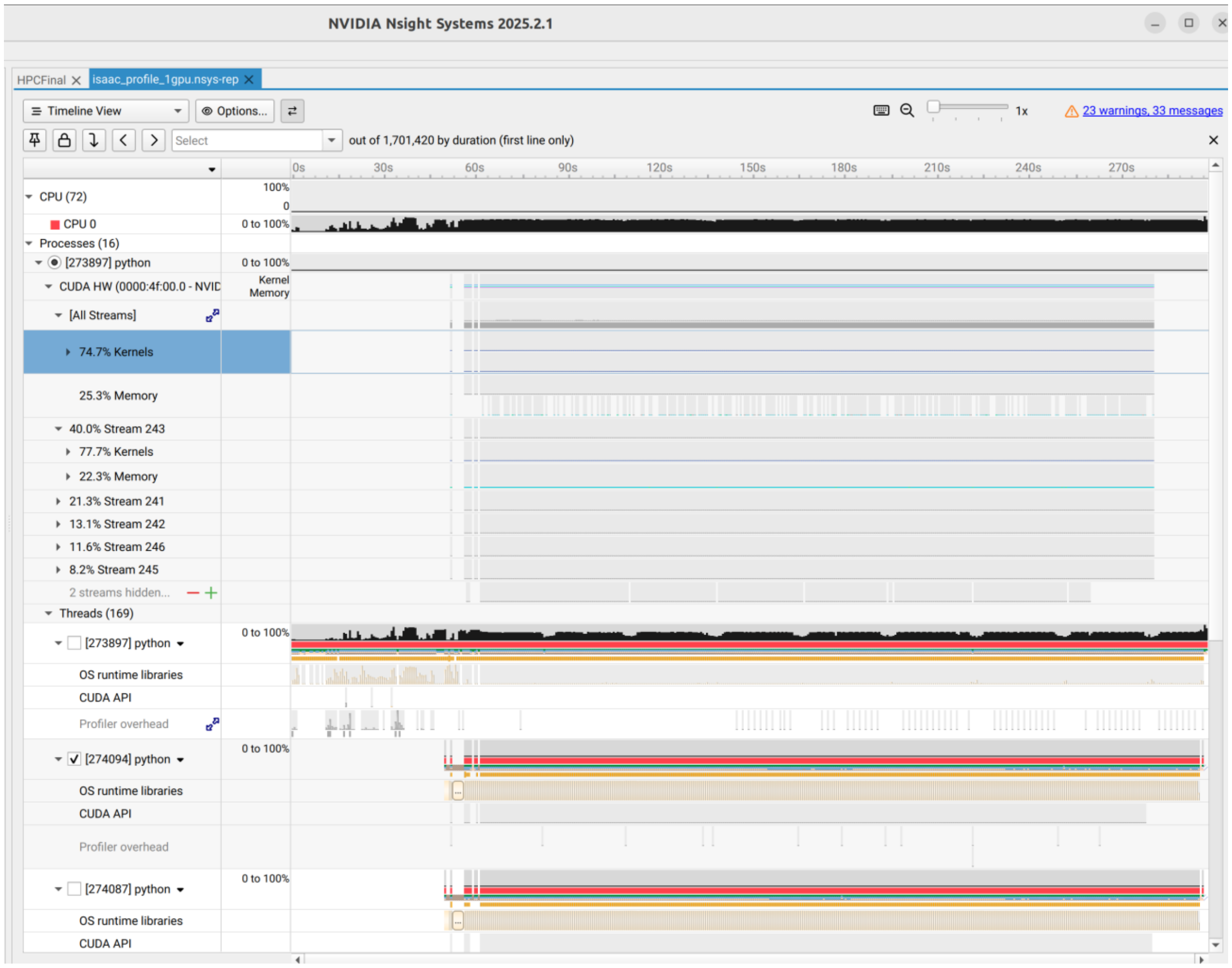


Image 20: The Nsight Systems output with details for a test on one GPU, showing the kernel usage and CPU processes during training.

This profiling allowed me to analyze training execution at a finer granularity. Each individual kernel can be seen, allowing users to get a better understanding of how the training is done sequentially and how the CPU is incorporated.

The streams are ordered from 241 to 246, displaying the kernels usage associated with specific processes within these streams and their memory usage. This is done to separate physics simulation, the running of multiple agents, and potential memory copies. Stream 241, to begin, contains radixSortMultiCalculateRankLaunchWithCount kernel and radixSortMultiBlockLaunchWithCount kernel that launch the training, and use up most of the kernel, at greater than 40%. Most of the memory in this stream is copying from the host to the GPU device. This is similar for the other first streams, and during training artiSolveInternalConstraintsTGS1T, computeUnconstraintedAccelerationsLaunch1T, and computeMassMatrix1T. The final stream is almost entirely a transfer of memory to the host, as DtoH memcpy takes up 90% of the memory.

Though this experiment's results were as expected, using NVIDIA's Nsight was a valuable experience to gain a better understanding in how RL training executes under the hood.

Experiment 3 – GPU Comparisons

The final experiment investigated the differences between GPU architecture on RL training, with the aim to understand what features in the GPU are most valuable to the application.

The Explorer Cluster provides L40, L40S, and A5000 GPUs, each of which were used individually and compared for the Isaac-Ant-v0 training with 4096 environments.

The L40 and L40S GPUs are based on the Ada Lovelace architecture, while the A5000 is built on the Ampere architecture.

The A5000, released in April 2021, is built on the GA102 silicon die, optimized for ray tracing. This die improved power efficiency versus the previous Turing architecture by almost 2 times the graphics power per watt. It introduced second generation ray tracing cores, of which the A5000 possesses 64. This architecture's SM enabled async compute, or simultaneous compute and graphics, with dedicated ray tracing cores, leaving the SM processing power for other workloads. It also introduced ray-traced motion blur, with multiply instruction multiple data (MIMD) hardware-based bounding volume hierarchy (BVH) traversal to significantly accelerate moving geometry motion blur. The A5000 also saw improved memory bandwidth at 768 GB/s, and 8,192 CUDA cores with 256 tensor cores. It contains 24 GB of GPU memory as well.

The Ada Lovelace architecture advanced upon Ampere, incorporating ray-triangle intersection testing, an intensive part of raytracing, at double the speed. The newer architecture's third-generation RT core contains an Opacity Micromap engine, evaluating the mesh of micro-triangles to characterize images more accurately. It also supports FP8 operations and performs at almost double the TFLOPS for FP16. The memory size of the Ada Lovelace architecture is also much greater, though it varies between L40 and L40S. The AD102 architecture has 16 times the size of the L2 cache as Ampere and almost double the L1 cache size.

The L40 has a memory bandwidth of 864 GB/s, with 142 RT cores, 568 tensor cores, 18175 CUDA cores, and 48 GB of GPU memory. These numbers all significantly exceed that of the A5000.

The L40S is fine-tuned for AI-enabled applications. It has the same specifications as the L40 with enhanced fourth-generation tensor cores and higher power draw of 350W compared to the L40 of 300W. It also possesses five times better inference performance for high-quality image generation and visuals.

Based on these primary differences between the three architectures, the performance in RL_Games with a single GPU for Isaac-Navigation-Flat-Anymal-C-v0. Since I'm running a headless training, I chose a task employing sensors using ray tracing cores. This task trains a quadruped to navigate and includes a ray caster sensor, performing like RTX based rendering. The results for this are below.

GPU	CPU	FPS Step	Environment Step & Inference FPS	FPS Total
L40	Intel Xeon Gold (36 cores)	175,712	166,537	147,413
L40S	Intel Xeon Gold (96 Cores)	215,543	206,693	185,967
A5000	Intel Xeon Gold (64 cores)	119,640	113,944	102,349

Image 21: The results for each metric for each GPU.

Analysis of Experiment 3

Based on the previously mentioned GPU architectures, these results align with the expected performance. The Ampere architecture's A5000 GPU performs the worst of the three GPUs due to its older generation ray tracing cores, and smaller overall size. It has significantly less memory and compute for intensive workloads such as reinforcement learning, which is why it performs the worst of the three. The A5000 has half as many CUDA cores, RT cores, and tensor cores. For faster performance, an L40S has almost double the FPS across each metric used by RL_games.

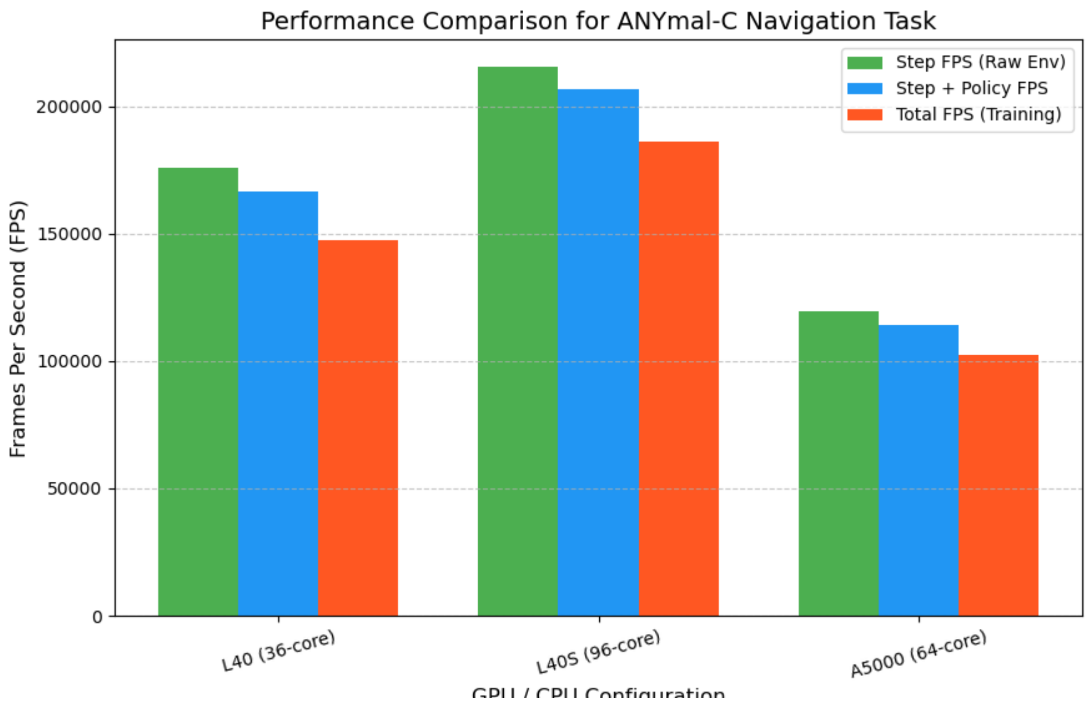


Image 22: A bar chart illustrating the difference in performance across each metric for the three GPUs.

To explain the differences between the L40 and L40S, more research was done into the GPU usage and CPU importance. For a single GPU, the following image is provided by Isaac Lab to show workload deployment.

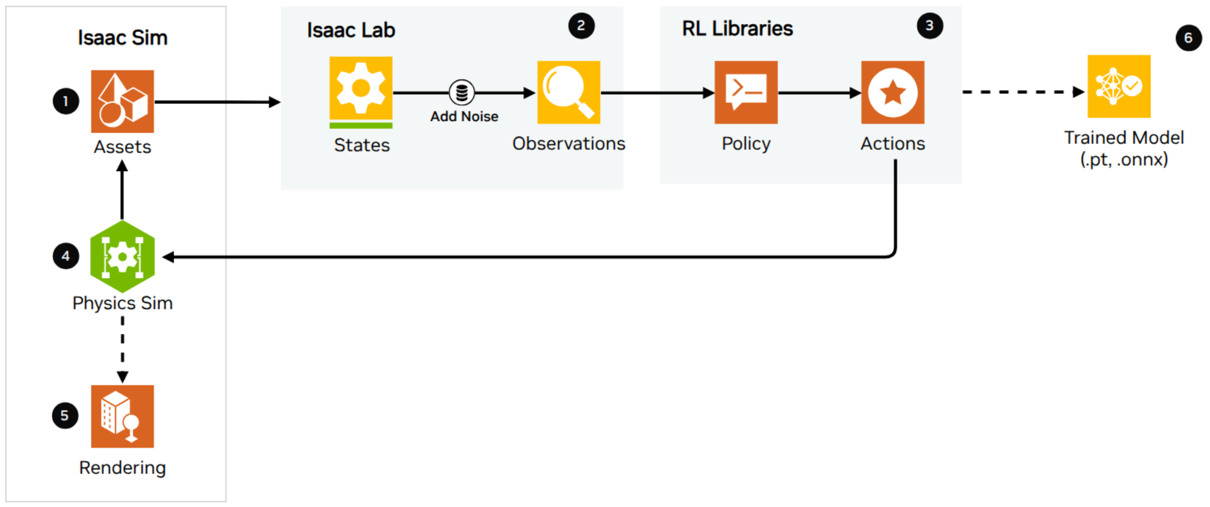


Image 23: The steps of execution during task training with a single GPU.

Based on this, the primary reason for L40S performance enhancement compared to the L40 is the benefits in physics simulation and rendering. The given task trains a quadruped, an Anymal, to navigate based on a ray casting sensor, frame transformer, and camera sensor. Though the task training was run headless, these sensors benefit from improved ray-tracing capabilities in GPUs.

In addition to differences in simulation, the node with the L40S GPU has a 96 core CPU, with 2 sockets of 48 cores each. This enables better environment deployment, preparing data more efficiently and handling environment resets as well.

Conclusions & Final Thoughts

I found this project to be valuable as a student interested in the intersection of high-performance computing and physical artificial intelligence.

NVIDIA's platforms, still in their infancy, have emerged as promising tools for reinforcement learning in robot tasks, and continue to advance rapidly. This project introduced me to reinforcement learning, NVIDIA's Isaac Sim and Isaac Lab applications, and allowed me to explore the implications of parallel computing on efficiency in RL. Through experiments and analysis in strong scaling and weak scaling, it can be concluded that the scalability of these GPU applications is effective, though it is dependent on the complexity of task training. Through the development of scripts for data collection and the utilization of profiling tools such as Nvidia Nsight, I gained greater familiarity with how to organize benchmarks and set up experiments for analyzing GPU performance.

Without widespread adoption of Isaac Lab or prior RL experience, one of the greatest challenges of this project was differentiating relevant and irrelevant factors in training performance. The RL_Games PPO configuration file contains numerous different parameters that, for me to understand were irrelevant, required tweaking and re-testing not included in this paper. This process was time-consuming and made complete analysis more difficult.

Future work in benchmarking these training tasks can include multi-node testing and varying the RL frameworks and algorithms used.

Grading

Based on the grading expectations from the proposal below, I believe I satisfied my expectations for an A. The first two experiments meet the A criteria, and the third experiment adds extra depth to the paper's analysis of Isaac Lab.

A: 2 different robot tasks, locomotion and aerial control, each tested over the different, given RL examples across >= 3 multi-GPU configurations

A-: 2 different robot tasks, locomotion and aerial control, each tested over the different, given RL examples across 1 or 2 multi-GPU configurations

B+: 1 robot task, locomotion or aerial control, tested over >= 3 multi-GPU configurations with the different examples

B: 1 robot task, locomotion or aerial control, tested over 1 or 2 multi-GPU configurations with the different examples

Extra Credit Opportunities:

-Significant changes made to codebases of frameworks to increase speedup without performance degradation

-RL policies generated on CPUs as well, which is much more time consuming

References

1. NVIDIA Corporation. (2023, January). *NVIDIA L40 GPU Datasheet*. Retrieved from https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/support-guide/NVIDIA-L40-Datasheet-January-2023.pdf
2. NVIDIA Corporation. (n.d.). *NVIDIA Ada Lovelace GPU Architecture*. Retrieved from https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf
3. Liu, Z., Lee, J., & Ahuja, N. (2024). *What Do Vision-Language Models Know About Perception?* arXiv preprint arXiv:2408.03539. https://arxiv.org/pdf/2408.03539
4. Xie, H., Lin, H., Zhang, Y., et al. (2024). *PiCode: Unsupervised Dense Visual Representation Learning with Pixel Denoising Codebooks*. arXiv preprint arXiv:2408.15633. https://arxiv.org/html/2408.15633v1
5. Kober, J., Bagnell, J. A., & Peters, J. (2013). *Reinforcement Learning in Robotics: A Survey*. *The International Journal of Robotics Research, 32*(11), 1238–1274. https://www.ri.cmu.edu/pub_files/2013/7/Kober_IJRR_2013.pdf
6. Levine, S., Finn, C., Darrell, T., & Abbeel, P. (2017). *End-to-End Training of Deep Visuomotor Policies*. *Journal of Machine Learning Research, 17*(39), 1–40. https://arxiv.org/abs/1707.06347

7. DeepMind. (2024). *From tokens to actions: scaling up the learning of robotic manipulation skills* [Video]. YouTube. https://www.youtube.com/live/_UHxP0FbOws
8. NVIDIA Corporation. (n.d.). *Nsight Systems Documentation*. Retrieved from https://docs.nvidia.com/nsight-systems/index.html
9. NVIDIA Corporation. (n.d.). *NVIDIA Collective Communications Library (NCCL) Documentation*. Retrieved from https://docs.nvidia.com/deeplearning/nccl/index.html
10. NVIDIA Corporation. (n.d.). *NVIDIA RTX A5000 Datasheet*. Retrieved from https://resources.nvidia.com/en-us-briefcase-for-datasheets/nvidia-rtx-a5000-dat-1
11. NVIDIA Corporation. (2021). *NVIDIA Ampere GA102 GPU Architecture Whitepaper (v2.1)*. Retrieved from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf