

GPU DVFS

Annie and Charlotte

Jetson Nano GPU

Nvidia Maxwell GPU:

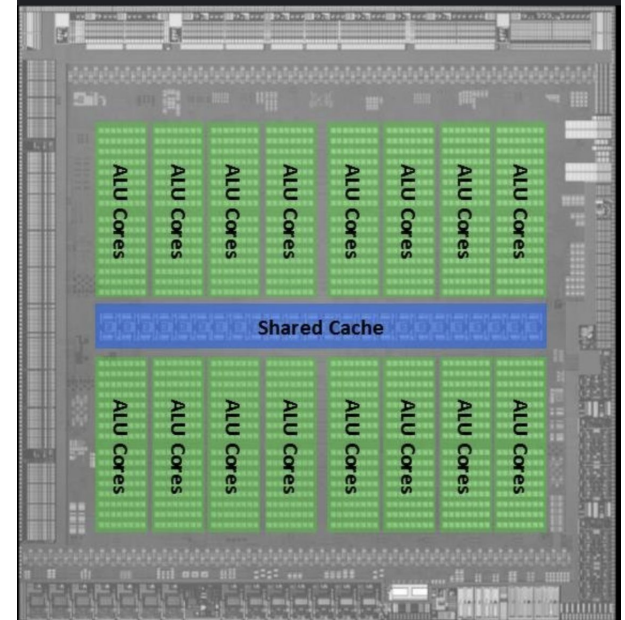
128-core GPU | Tile Caching

OpenGL 4.6 | OpenGL ES 3.2 | CUDA supported

OpenGL ES Shader (up to): 512 GFLOPS (FP16)

Operating Frequency:

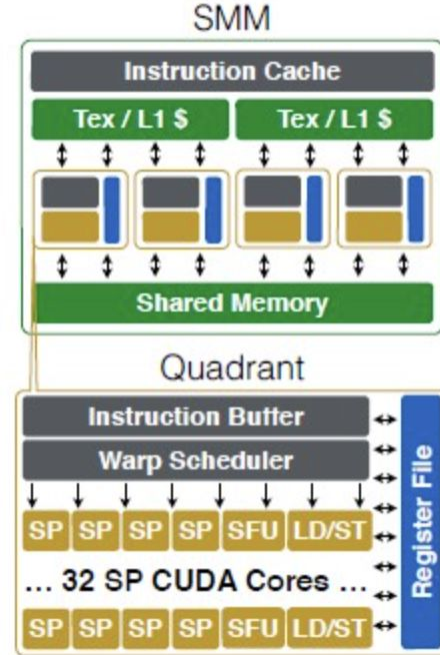
Base clock: 640 MHz | Boost Clock: 921 MHz



Features of the GPU:

- Graphics Processing Cluster (GPC)
 - Each GPC contains multiple Streaming Multiprocessors and a Raster Engine
- Each SMM has 4 independent processing blocks each with:
 - Its own instruction buffer
 - A dedicated scheduler
 - 32 CUDA cores
- DVFS (what we are trying to investigate)

Maxwell SMM

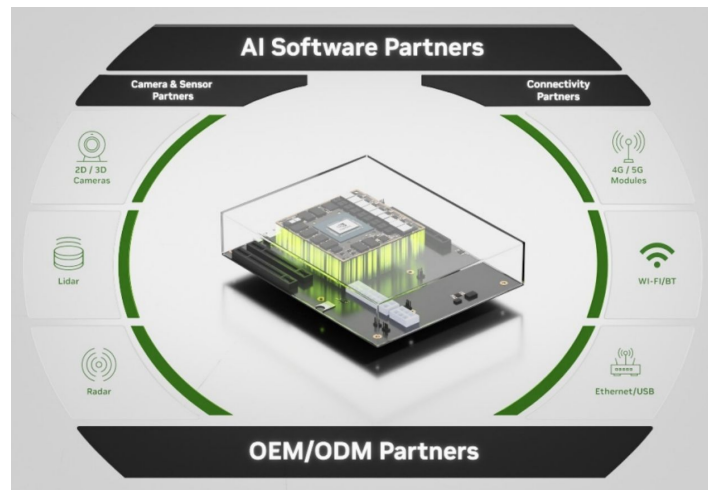


Why are we doing this?

- Modern embedded systems—from drones and robots to smart cameras—rely heavily on GPUs for **compute-intensive tasks** such as deep learning inference and real-time image processing. These workloads are not only performance-critical but also energy-sensitive due to strict **power and thermal constraints** in mobile and edge environments.
- Just like CPUs, GPUs employ **DVFS** to manage power. However, unlike CPUs where DVFS behavior has been extensively studied, the DVFS behavior of embedded GPUs like the Maxwell GPU in Jetson Nano remains **poorly understood and largely undocumented**.
- In recent years, hardware-managed DVFS schemes—where the GPU autonomously adjusts its frequency and voltage—have become more common. These systems offer faster response times but **hide the control logic**, making it difficult to analyze or predict power-performance behavior without access to internal hardware IP.

Problem Addressed:

- There is **no clear model or documentation** explaining how embedded GPUs like the Jetson Nano's Maxwell core make DVFS decisions. This opacity creates challenges for:
 - Predicting thermal throttling behavior
 - Designing efficient and stable GPU workloads
 - Maximizing performance within a fixed power budget
- To address this, we develop parameterized GPU microbenchmarks that allow us to:
 - Systematically probe the GPU's power and frequency response across compute and memory workload types.
 - Create maximum-stress workloads that reveal upper bounds of GPU behavior.
 - Collect data from onboard monitors (via sysfs) to capture real-time power, frequency, and thermal states.



Our GOAL:

- To model and understand the GPU's DVFS mechanism by observing how its operating frequency, power consumption, and temperature respond to diverse workloads varying in:
 - Parallelism (via threads and blocks)
 - Arithmetic intensity
 - Memory access patterns
- To contribute a “TOOL” for programmable Jetson utilization, correlated with power measurements.

Power Management:

Power Domain	Power Island in Domain	Modules in Power Island
	XUSBA, XUSBB, XUSBC	USB 3.0
	VIC	VIC (Video Image Compositor)
	ADSP	APE (Audio Processing Engine)
	DFD	Debug logic
GPU (VDD_GPU)	GPU	3D, FE, PD, PE, RAST, SM, ROP
CPU (VDD_CPU)	CPU 0	CPU 0
	CPU 1	CPU 1
	CPU 2	CPU 2
	CPU 3	CPU 3
	Non-CPU	L2 Cache for Main CPU complex
	TOP	Top level logic

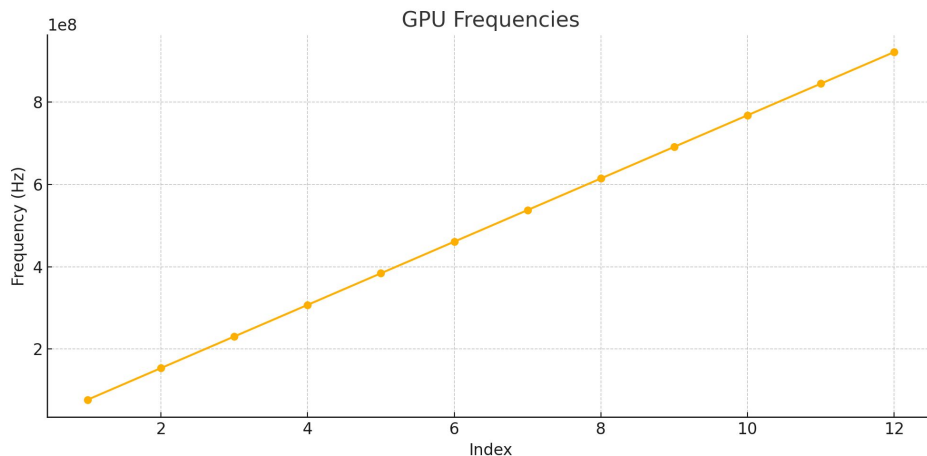
Finding Frequency

/sys/devices/gpu.0/devfreq/57000000.gpu/cur_freq

- /sys/devices/gpu.0/ — Refers to the GPU device (Maxwell GPU) registered as gpu.0.
- devfreq/ — Part of the Dynamic Voltage and Frequency Scaling (DVFS) framework in Linux.
- 57000000.gpu/ — This is the device name of the GPU node created by the device tree.
 - 57000000 is the base address of the GPU in physical memory, defined in the Jetson Nano's device tree (in the hardware definition).
 - It refers to the Tegra X1's GPU hardware block, typically the GPU controller.
- cur_freq — This file gives the current operating frequency of the GPU in Hz.

Finding Frequency

- `cur_freq` can be read as fast as my user space application allows. I.e. microsecond level.
- Range of freq in this GPU(Hz):
 - 76800000
 - 153600000
 - 230400000
 - 307200000
 - 384000000
 - 460800000
 - 537600000
 - 614400000
 - 691200000
 - 768000000
 - 844800000
 - 921600000



frequencymonitor.sh

```
#!/bin/bash
# This script continuously monitors the GPU frequency with a high-resolution timestamp.
# It reads the current frequency from the sysfs file and prints it alongside a nanosecond-precision timestamp.
# Run as root (e.g., using sudo).
# Ensure the script is run as root.
if [ "$(id -u)" -ne 0 ]; then
    echo "Error: Please run this script as root."
    exit 1
fi
# GPU frequency file (adjust this path if necessary)
GPU_FREQ_FILE="/sys/devices/gpu.0/devfreq/57000000.gpu/cur_freq"
# Check that the frequency file exists
if [ ! -f "$GPU_FREQ_FILE" ]; then
    echo "Error: GPU frequency file not found at $GPU_FREQ_FILE"
    exit 1
fi
# Loop forever, reading the GPU frequency as fast as possible with a nanosecond-precision timestamp
while true; do
    GPU_CUR_FREQ=$(cat "$GPU_FREQ_FILE")
    TIMESTAMP=$(date '+%Y-%m-%d %H:%M:%S.%N')
    echo "${TIMESTAMP} ${GPU_CUR_FREQ}"
done
```

Finding Power

/sys/bus/i2c/devices/6-0040/iio:device0/in_power1_input

- /sys/bus/i2c/ - This is the sysfs directory for I²C devices.
- /devices/6-0040/ - This indicates a device at: Bus 6, and Address 0x40 (hexadecimal 40). So this refers to an I²C device connected at I²C bus 6, address 0x40.
 - In Jetson devices, this is often a power monitoring IC such as the TI INA3221, which is a triple-channel current and power monitor.
- /iio:device0/ - This means the device is registered under the Industrial I/O (IIO) subsystem, which Linux uses for sensors and ADC-type devices.
 - iio:device0 is the name given to the first registered IIO device (can be renamed or changed if other devices exist).
- In_power1_input - This file reports the instantaneous power measurement from channel 1 (usually GPU).
 - in_power1_input → Typically measured in microwatts (μW)
 - The numbering convention:
 - in_power1_input → channel 1 (often GPU rail)
 - in_power2_input → channel 2 (often CPU rail)
 - in_power3_input → channel 3 (e.g., system or SoC power)
 - So, in_power1_input gives the real-time GPU power consumption.

Finding Power

Access Frequency:

Just like `cur_freq`, this file is a virtual interface exposed by the kernel.

Can read it as fast as your application allows, even in microseconds.

However:

The INA3221 sensor itself samples data at a fixed rate (typically 1–2 Hz by default).

So reading this file faster than ~once every 500 ms (~2 Hz) usually won't give new values.

Our decision: Poll every 250–1000 ms to get fresh power data without wasting CPU cycles.

powergpu.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#define SAMPLE_INTERVAL_MS 50 // Adjust as needed
int main() {
    const char *gpu_power_path =
"/sys/bus/i2c/devices/6-0040/iio:device0/in_power1_input";
    while (1) {
        // 1) Read GPU power
        FILE *fp = fopen(gpu_power_path, "r");
        if (!fp) {
            perror("Failed to open GPU power file");
            return 1;
        }
        int power_microwatts;
```

```
        if (fscanf(fp, "%d", &power_microwatts) == 1) {
            float power_mw = power_microwatts / 1000.0f;
            // 2) Get high-resolution timestamp
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            struct tm tm = *localtime(&ts.tv_sec);
            // 3) Print timestamp + power
            printf("%04d-%02d-%02d %02d:%02d:%02d.%09ld
GPU Power: %.2f mW\n",
                tm.tm_year + 1900,
                tm.tm_mon + 1,
                tm.tm_mday,
                tm.tm_hour,
                tm.tm_min,
                tm.tm_sec,
                ts.tv_nsec,
                power_mw);
        } else {
            fprintf(stderr, "Failed to parse power value\n");
        }
        fclose(fp);
        // 4) Sleep for the specified interval
        usleep(SAMPLE_INTERVAL_MS * 1000);
    }
    return 0;
}
```

Finding Temperature

/sys/devices/virtual/thermal/thermal_zone2/temp

- `/sys/devices/virtual/`
 - The virtual directory is for devices not tied directly to hardware in the traditional bus sense.
 - These are kernel-managed abstractions, like thermal zones, power regulators, and others.
- `/thermal/`
 - This directory contains thermal management interfaces.
 - It's part of the Linux thermal subsystem, which monitors and controls device temperatures.
- `/thermal_zone2/`
 - This represents one specific thermal sensor or temperature zone.
 - On Jetson Nano (and other Tegra devices), the thermal zones are numbered and mapped to specific parts of the SoC.
- Typically:
 - `thermal_zone0` = CPU
 - `thermal_zone1` = GPU (or power supply)
 - `thermal_zone2` = GPU (in some Jetson builds, confirmed by checking `/sys/class/thermal/thermal_zone2/type`)

Finding Temperature

Like other sysfs files, we can read temp as fast as we want, but there are sampling limits behind the scenes:

- The temperature data is updated based on an internal polling interval set by the thermal governor.
- On Jetson platforms, the update frequency is typically ~ 1 Hz (every 1000 ms).
- Reading faster won't give us newer data and just consumes CPU cycles.


We decided on: Sampling every 500–1000 ms.

tempmonitor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

int main() {
    // Adjust this path if your GPU thermal zone index is different
    const char *gpu_temp_path =
"/sys/devices/virtual/thermal/thermal_zone2/temp";
    while (1) {
        // 1) Read the GPU temperature in millidegrees
        FILE *fp = fopen(gpu_temp_path, "r");
        if (!fp) {
            perror("Failed to open GPU temperature file");
            return 1;
        }
        int temp_milli;
        if (fscanf(fp, "%d", &temp_milli) == 1) {
            float temp_c = temp_milli / 1000.0f;
            // 2) Get a high-resolution timestamp
            struct timespec ts;
            clock_gettime(CLOCK_REALTIME, &ts);
            struct tm tm = *localtime(&ts.tv_sec);

            // 3) Print timestamp + temperature
            printf("%04d-%02d-%02d %02d:%02d:%02d.%09ld GPU
Temperature: %.2f °C\n",
                tm.tm_year + 1900,
                tm.tm_mon + 1,
                tm.tm_mday,
                tm.tm_hour,
                tm.tm_min,
                tm.tm_sec,
                ts.tv_nsec,
                temp_c);
        } else {
            fprintf(stderr, "Failed to parse temperature value\n");
        }
        fclose(fp);
        // Sleep ~50 ms (20 samples per second)
        usleep(50 * 1000);
    }
    return 0;
}
```


Milestone: Produce first set of CUDA microbenchmarks displaying (micro)architectural parameters. 

Tool (usage message)

```
fprintf(stderr,  
    "Usage: %s <run_seconds> <threads_per_block> <num_blocks> <arithmetic_intensity>  
<memory_pattern> <stride>\n"  
    "Arguments:\n"  
    "  run_seconds          - Duration (in seconds) to run the kernel\n"  
    "  threads_per_block   - Number of threads per CUDA block (e.g., 128, 256)\n"  
    "  num_blocks          - Number of CUDA blocks (e.g., 1 - 65535)\n"  
    "  arithmetic_intensity - Number of floating-point operations per thread (1–10 recommended)\n"  
    "  memory_pattern      - Memory access pattern:\n"  
    "                        0 = Sequential\n"  
    "                        1 = Pseudo-random\n"  
    "                        2 = Strided (with stride specified)\n"  
    "                        3 = Reverse\n"  
    "  stride              - Stride value for pattern 2 (ignored otherwise)\n",  
    argv[0]);
```

Tool (num_blocks & threads_per_block)

In CUDA, the kernel is launched via:

- `workloadKernel<<<num_blocks, threads_per_block>>>(...);`

This tells the GPU to run:

- `num_blocks` blocks of threads,
- With `threads_per_block` threads in each block.

The total number of threads launched is:

- `total_threads = num_blocks × threads_per_block`

Its global thread index is given by:

- `int idx = blockIdx.x * blockDim.x + threadIdx.x;`

Threads are activated automatically by the GPU hardware, and they operate in parallel (as far as resources allow), mapped onto the GPU's CUDA cores via thread schedulers in each Streaming Multiprocessor (SM).

Tool (Arithmetic Intensity)

Arithmetic_intensity – Number of floating-point operations performed per thread.

- Increasing this value simulates compute-bound workloads.
- Low values simulate memory-bound workloads.
- This allows the analysis of GPU power scaling under varying FLOPs-to-bytes ratios.
- User input 1-10

Tool (Arithmetic Intensity)

```
for (int i = 0; i < arithmetic_ops; ++i) {  
    val = sinf(val) * cosf(val);  
}
```

Arithmetic Intensity (in our tool) refers to the number of floating-point operations (FLOPs) per memory access.

This loop performs:

- 2 floating-point transcendental operations (sinf, cosf)
- 1 floating-point multiplication (*)

So, for each i , we're doing approximately 3 FLOPs.

By adjusting this parameter, the tool changes whether the kernel is:

- Memory-bound (low AI: more memory access, fewer ops)
- Compute-bound (high AI: more math per memory)

Milestone: Create a maximum power stressmark for the Jetson's GPU 

Tool (Memory Access Patterns)

Pattern	Code	Description
0	<code>accessIdx = idx;</code>	Sequential access: Each thread reads/writes its own index. Fastest, coalesced.
1	<code>accessIdx = lcg_random(idx) % dataSize;</code>	Random access: Simulates scattered loads, cache misses, and unpredictable access.
2	<code>accessIdx = (idx * stride) % dataSize;</code>	Strided access: Each thread accesses a memory location <code>stride</code> steps apart. Simulates bank conflicts.
3	<code>accessIdx = dataSize - idx - 1;</code>	Reverse access: Threads go backward through memory. Poor spatial locality.

Sweep Configuration (sweep.sh)

The script explores a dense range of kernel launch parameters:

● <u>Parameter</u>	<u>Values Covered</u>
● Threads per block	32, 64, 256, 512, 1024
● Blocks	1 to 3000 (in steps of 50)
● Arithmetic intensity	1 to 10
● Memory pattern	0 = Coalesced, 1 = Random, 3 = Reverse
● Stride	Fixed at 1 (no strided memory pattern tested)

This setup enables control over occupancy, memory locality, and compute-to-memory ratios.

Sweep Configuration (sweep.sh)


For each configuration, the script:

- Runs gputool3.cu for 2 seconds
- Samples system metrics mid-run, every 100 ms for 1 second:
 - GPU frequency (cur_freq)
 - GPU power (in_power1_input)
 - GPU temperature (thermal_zone2/temp)

It records the peak values seen during that window — making it suitable for thermal/power ceiling analysis.

Outputs csv:

run_id,threads_pb,num_blocks,arith_intensity,mem_pattern,gpu_freq_hz,power_mw,temp_c

Milestone: Method for programmable Jetson utilization, correlated with power measurements. 

Machine Learning Models Results

Three models and its r^2 value

Optimizing RandomForest for GPU frequency prediction...

Best parameters: {'max_depth': None, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 200}

MSE: 387982171464782.62, R^2 : 0.9873

Optimizing GradientBoosting for power consumption prediction...

Best parameters: {'learning_rate': 0.2, 'max_depth': 7, 'min_samples_split': 2, 'n_estimators': 200}

MSE: 0.005855, R^2 : 0.9932

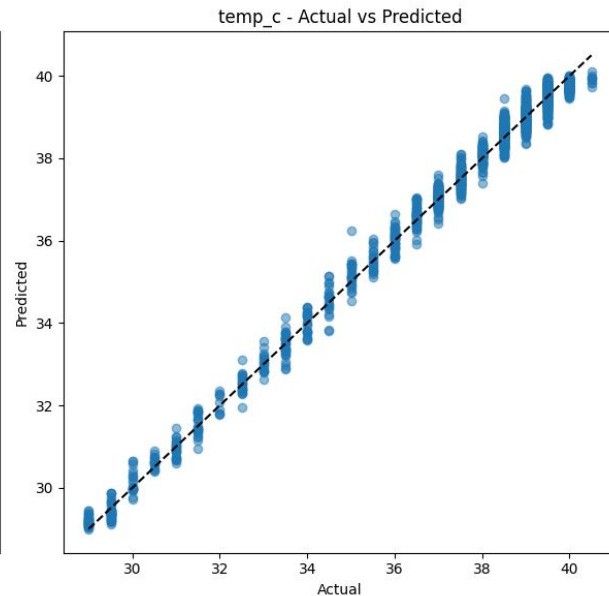
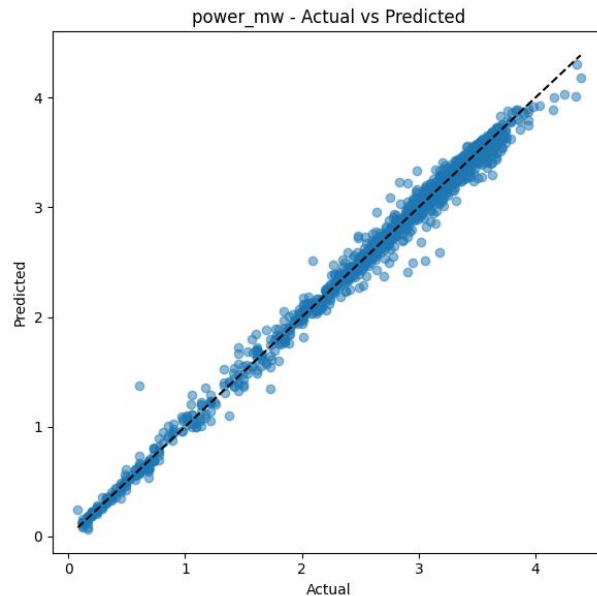
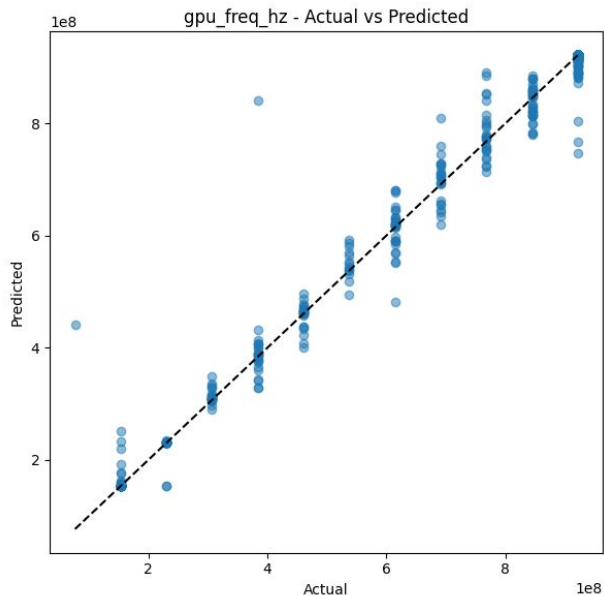
Optimizing GradientBoosting for temperature prediction...

Best parameters: {'learning_rate': 0.1, 'max_depth': 5, 'min_samples_split': 5, 'n_estimators': 200}

MSE: 0.060699, R^2 : 0.9927

Machine Learning Models Results

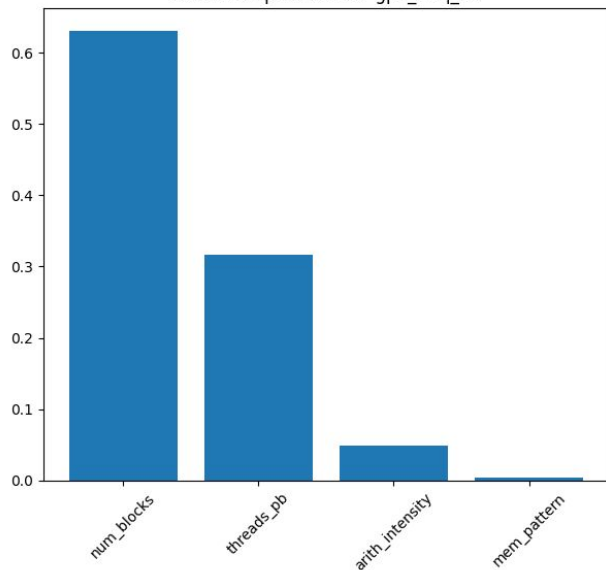
Actual vs Predicted results



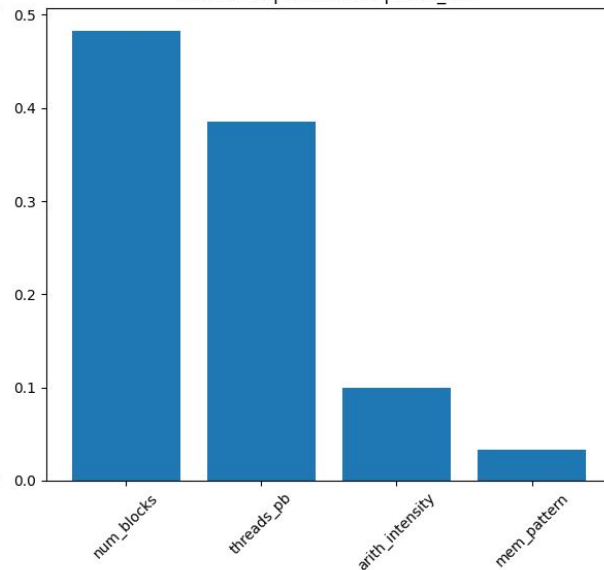
Machine Learning Models Results

Feature Importance

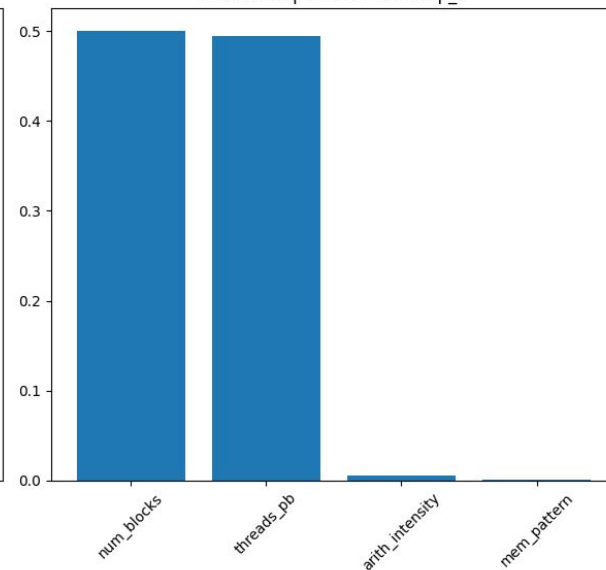
Feature Importance for gpu_freq_hz



Feature Importance for power_mw



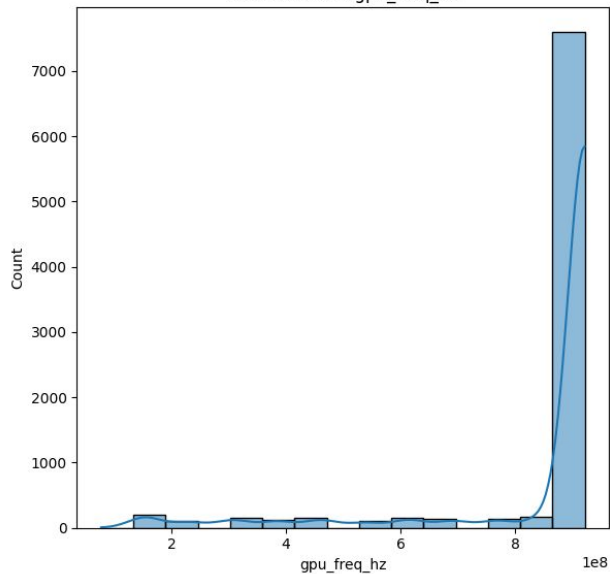
Feature Importance for temp_c



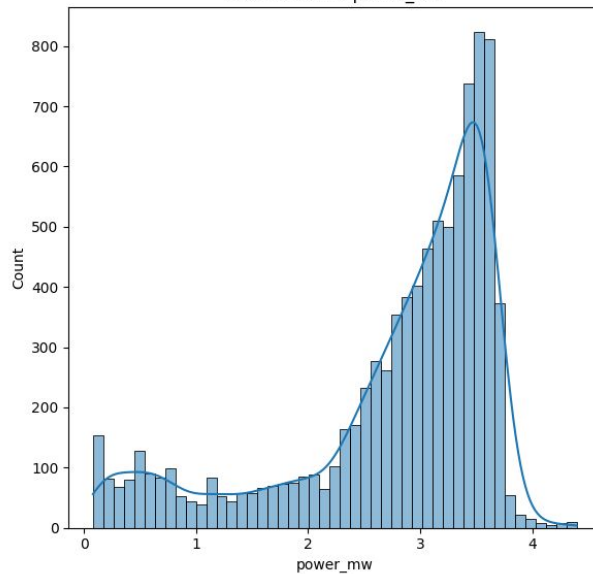
Machine Learning Models Results

Data Variation

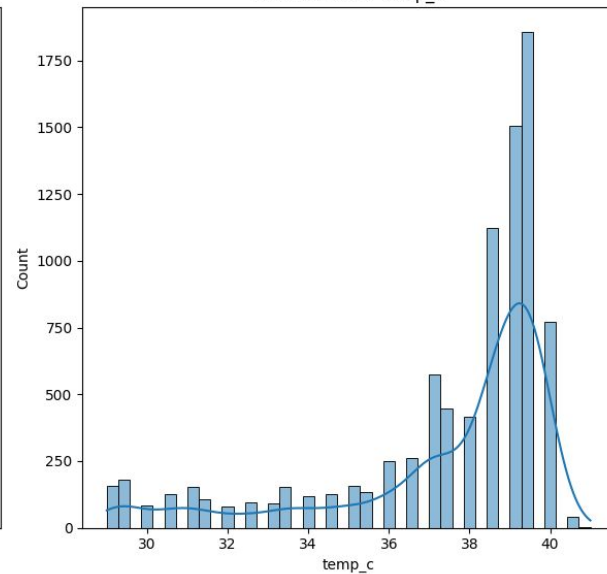
Distribution of gpu_freq_hz




Distribution of power_mw



Distribution of temp_c



Milestone: Validated power measurements corresponding to simple CUDA microbenchmarks. 

Next Steps

- Document additional performance counters (e.g., L2 hit rate, stall reasons).
- Use power and frequency measurements to estimate internal GPU voltage.
- Build more predictive models to predict frequency/power from workload metrics.
- Design dynamic benchmarks that vary load over time to analyze DVFS responsiveness.
- Identify and characterize hidden throttling behaviors under thermal or memory stress.
- Create a structured dataset of workload parameters and corresponding GPU telemetry.
- Investigate GPU behavior during idle/sleep states and low-load operation.
- Extend benchmarks to include real-world GPU workloads (e.g., AI inference models).
- Cross-compare DVFS behavior across Jetson platforms (Nano, Xavier, Orin)