Roofline Model: Training a Convolutional Neural Network

Matteo De Sanctis, Riccardo De Sanctis
desanctis.1937858@studenti.uniroma1.it, desanctis.1937859@studenti.uniroma1.it

November 17, 2025

Abstract

This short report documents a roofline-analysis workflow for CUDA kernels that simulate a convolutional neural network (CNN) training. The work includes: a training simulator kernel (forward, backward-data, backward-weight, SGD update), data-collection pipelines, single- and multi-kernel roofline plotting scripts, and experiments run locally and on Sapienza HPC. A notable limitation encountered on the cluster is restricted access to NVIDIA hardware performance counters; this is discussed in a dedicated section.

1 Introduction

The roofline model relates a kernel's performance (GFLOP/s) to its arithmetic intensity (FLOP/Byte) and hardware peaks (peak bandwidth, peak compute). This project implements a synthetic training workload for a convolutional neural network and measures/estimates FLOPs, DRAM traffic, and runtime to place the kernel(s) on a roofline.

2 Background: the roofline model

The roofline model is defined by two hardware limits:

- **Peak compute** $P_{\max}$ (GFLOP/s): the device's maximum delivered floating-point throughput.
- **Peak bandwidth** $B_{\max}$ (GB/s): the sustainable peak main-memory bandwidth.

For a kernel we define:

$$\text{FLOPs} = \text{total floating-point operations executed by the kernel,}$$
$$\text{Bytes} = \text{total bytes transferred to/from DRAM by the kernel.}$$

The kernel's **arithmetic intensity** (AI) is

$$\text{AI} \;=\; \frac{\text{FLOPs}}{\text{Bytes}} \quad \text{[FLOP / Byte]}.$$

The roofline gives the theoretical attainable performance $P(\text{AI})$:

$$P(\text{AI}) \;=\; \min\big(P_{\max},\; \text{AI} \cdot B_{\max}\big).$$

Hence:

- If $\text{AI} \geq \dfrac{P_{\max}}{B_{\max}}$, the kernel is *compute-bound* and can reach up to $P_{\max}$.

- If $\text{AI} < \dfrac{P_{\max}}{B_{\max}}$, the kernel is *memory-bound* and limited by $\text{AI} \cdot B_{\max}$.

To place a measured kernel on the roofline we need:

1. **FLOPs** — the number of floating point operations actually executed (or the theoretical FLOPs implied by the algorithm). On NVIDIA hardware, accurate counts of executed instructions (FADD, FMUL, FFMA) are obtained from performance counters such as `sm__sass_thread_inst_executed_op_ffma_pred_on.sum`.
2. **DRAM bytes** — the number of bytes read/written to device memory; counters like `dram__bytes_read.sum` and `dram__bytes_write.sum` provide the authoritative values.
3. **Kernel runtime** — wall-clock execution time (measured with CUDA events or the profiler). From FLOPs and runtime we compute achieved GFLOP/s: $\text{GFLOP/s} = \dfrac{\text{FLOPs}}{\text{time}\,[\text{s}]}/10^9$.
4. **Hardware peaks** ($P_{\max}, B_{\max}$) — measured on the same device (e.g., via microbenchmarks) so the plotted roofline reflects the actual node characteristics.

In short: accurate AI requires both numerator (FLOPs) and denominator (Bytes) measured from the hardware; later we will see why it's fundamental to have proper access to the device being tested, and the limitations we faced on the HPC (Sapienza Supercomputer Center).

3 Methodology

1. Run the Convolutional Neural Network training simulation CUDA binary on a GPU node to produce wall-clock timings (and if needed print FLOP/DRAM estimates).
2. If available (Access to the GPU device requires superuser privileges (root)) run Nsight Compute (`ncu`) to gather hardware counters: instruction counts (FADD/F MUL/FFMA) and DRAM bytes read/write.
3. Use the Python extractor to compute arithmetic intensity (AI = FLOPs / DRAM bytes) and achieved GFLOP/s (FLOPs / runtime).
4. Plot roofline using measured peaks (device bandwidth, device peak GFLOPs) and mark each kernel's AI vs achieved performance.

4 HPC device counter permission issue

While running `ncu` on Sapienza's node `dgx005` (NVIDIA A100), Nsight Compute returned:

```
ERR_NVGPUCTRPERM - The user does not have permission to access NVIDIA GPU
Performance Counters
```

These counters (e.g., `sm__sass_thread_inst_executed_op_ffma_pred_on.sum` and `dram__bytes_read.sum`) are *privileged*. Without them, it is not possible to obtain hardware-level instruction counts and exact DRAM traffic. For authoritative roofline placement these counters are important because they provide the actual FLOP and byte counts seen by the hardware. We mitigated this limitation by running the code on our personal machines equipped with NVIDIA RTX 3050 (4 GB) and NVIDIA RTX 3070 (8 GB), where counter access was available. Note that results from these consumer GPUs may differ from the A100 on the HPC in absolute performance and bandwidth, so numbers are indicative rather than directly comparable.

5 Results

Figure 1 presents the roofline model for the four kernels evaluated on the NVIDIA RTX 3050. The *Matmul_timed* kernel, which implements a straightforward matrix multiplication, achieves an arithmetic intensity (AI) of 7.74 and a throughput of 32.67 GF/s, placing it well within the memory-bound region. The *compute_loop* kernel reaches an AI of 996 and a throughput of 4585.82 GF/s, effectively aligning with the empirical compute roof and nearly saturating the peak performance of the device. In contrast, the *smes_loop* variant, which executes the same iterative workload in shared memory, attains a higher AI of 2105 but only 405.46 GF/s, indicating that shared-memory pressure or instruction overhead (e.g. sync threads) dominates despite the improved operational intensity.

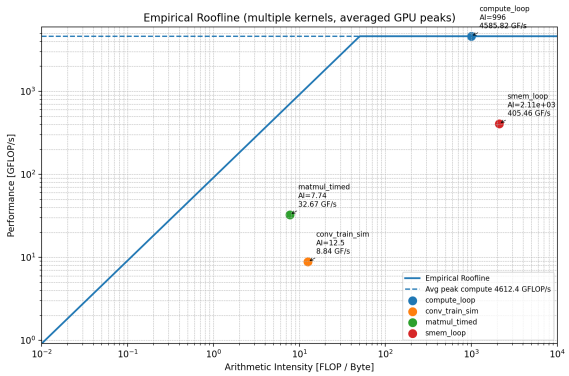

Figure 1: NVIDIA RTX 3050 - Roofline model for CNN training and 3 benchmarks. (GPU peaks: 91.57 GB/s, 4612.36 GFLOP/s)

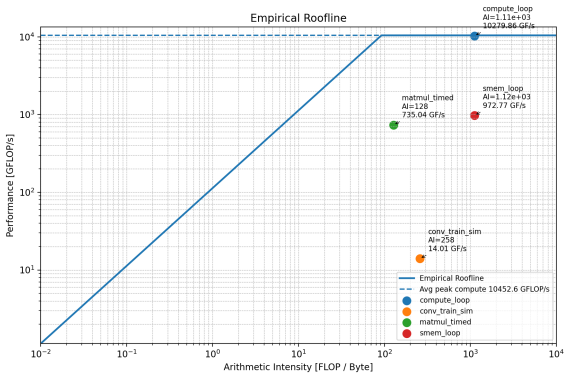

Figure 2: NVIDIA RTX 3070 - Roofline model for CNN training and 3 benchmarks. (GPU peaks: 113.03 GB/s, 10452.56 GFLOP/s)

The convolutional neural network training kernel exhibits the lowest performance among the four, reaching 8.84 GF/s at an AI of 12.55. Although its intensity is slightly above that of the matrix multiplication kernel, it remains far from both rooflines. This is expected as other kernels are just benchmarks, while this kernel simulates the full training pipeline of a CNN, very much heavy in computation. To improve on this, one could experiment with larger batch size, so each weight and activation would be used more, use mixed precision (e.g. FP16/TF32), improve code to reduce memory traffic (shared-memory, cache), or even improve on the CNN architecture itself, like reducing large-kernel convolutions and use lower-precision training algorithms.
All code of the kernels can be found in the repo shared during the submission.

The same relationship between kernels may be observed when operating the NVIDIA RTX 3070 in Fig. 2, although the higher compute capability raises both the achievable arithmetic intensity and throughput. The *matmul_timed* kernel reaches an AI of 127.55 and a throughput of 735.04 GF/s, moving markedly closer to the compute roof and to the shared memory loop kernel in terms of performance; the *compute_loop* attains an AI of 1114.71 and 10279.86 GF/s, effectively aligning with the empirical compute peak; the *smem_loop* records a similar AI (1117.04) but only 972.77 GF/s, indicating shared-memory pressure or synchronization overhead still limits it, note also the presence of a decrease in AI wrt the RTX 3050, that shows a memory-compute balance change, likely due to differences in cache/shared-memory use or compiler decisions; finally, the *conv_train_sim* kernel achieves 14.01 GF/s at an AI of 258.14, remaining well below the rooflines and the 3 benchmarks, yet improving on the RTX 3050. Note also shift in the kernels relative positions in the roofline models between the two GPUs: as AI depends on memory traffic and FLOPs, architectural and compiler differences (cache/shared-memory behavior, register spills, occupancy) and measurement artifacts can change the measured memory traffic and thus the AI; obviously also performance changes with the 3070's higher GFLOP/s and compute roof.

6 Implemented components

Key artifacts in the repository:

- `kernels/conv_train_sim.cu` – CUDA training-simulator kernel (forward, backward-data, backward-weight, update).
- `plot_roofline3.py` – single-kernel extractor + one-shot roofline plot.
- `plot_roofline_multi.py` – plot multiple saved kernel info files.
- `pipeline_cnn.sh`, `pipeline_multi.sh` – helper bash pipelines.
- `kernels_data/*.json` and `kernels_data/*.csv` – extracted kernel info and profiler outputs (if available).

7 Conclusions

We implemented a small but realistic training-simulation kernel and an end-to-end pipeline to measure and plot roofline data. On local machines the workflow completes fully; on Sapienza HPC, access to NVIDIA hardware counters is blocked by policy — acquiring admin permission is required for exact hardware-counter-based roofline measurements. Overall, this work allowed us to gain a deeper understanding of GPU performance by implementing and profiling representative CUDA kernels, collecting hardware-level metrics, and positioning them within the roofline model to reason quantitatively about the balance between computation and memory throughput.

A Reproducibility / How to run

On the cluster interactive GPU node:

```
# request a GPU node (example)
srun --gres=gpu:1 --mem=32G -c 8 --time=01:00:00 --pty /bin/bash

# load modules, compile, and run pipeline
module load cuda
module load python/3.11.7
source ~/roofline_env/bin/activate   # if using a venv with numpy/pandas/matplotlib

cd ~/Roofline_model
chmod +x pipeline_cnn.sh
./pipeline_cnn.sh
# then run the multi-plot:
python plot_roofline_multi.py ./kernels_data/*.json
```