COURSE NAME: HW for AI and ML
SPRING SEMESTER 2025

INSTRUCTOR: Christof Teuscher

Self-Graded Report for the “Hardware for AI/ML” Course

PROJECT TITLE

LSTM Matrix Multiplication Offload Using Synthesizable Floating-Point RTL

Author

Gurajala Rakesh

Contents

Objective:

The goal of this project is to identify performance bottlenecks in a Python-based LSTM stock prediction model and offload the most computationally expensive operations, specifically the matrix multiplications within the LSTM gates, to synthesizable SystemVerilog RTL using floating-point arithmetic, and analyze the results across different workloads.

Why I Chose This Project:

- I wanted to combine my interests in machine learning and hardware design.
  LSTM models involve repetitive, heavy computation, making them ideal for RTL acceleration.
- This project aligns with my career path as a design verification and RTL engineer.
- Also, I love stock trading, so I was invested in exploring this idea

Heilmeier Questions

What are you trying to do? I.e., what AI/ML algorithm/workload did you pick?)

I am working on a multi-stock LSTM predictor where each stock has its own weights and model config, and runs the forecast by initializing with its config

How have others implemented and/or accelerated this algorithm?

Most of them that I could find only take a single stock as an input and run them on the fly, data is pre-processed on every run, which consumes a lot of sim time on every run

What are you doing differently/better/etc.?

- saving the weights for each stock instead of pre-processing on every run

-updating the weights on every run based on the previous day's closing price

-Pre-fetching the weights of frequently used stock

What have you accomplished so far?

-Profiled the model and identified the bottlenecks

ex: ~76% of execution time for tensorflow kernel call (training)

- Working LSTM cell (RTL) to offload the training data

What will you do next, and what remains to be done until you can declare success?

-Scale the LSTM cell to the pipeline and train multiple stocks

-execution time comparison(RTL vs software)

-Work on pre-processing of data at the hardware level(if time permits)

My Strategy to Work on This Project

Use LLMs like ChatGPT to:

- Suggest optimizations
- Write RTL components step by step
- Help debug and structure prompts

Break down the LSTM logic into small, testable RTL components (e.g., fp_add, fp_mult)
Simulate and validate incrementally (first multiply, then accumulate, then full pipeline)

Prompt Strategy

Write clear, detailed prompts with specific goals:

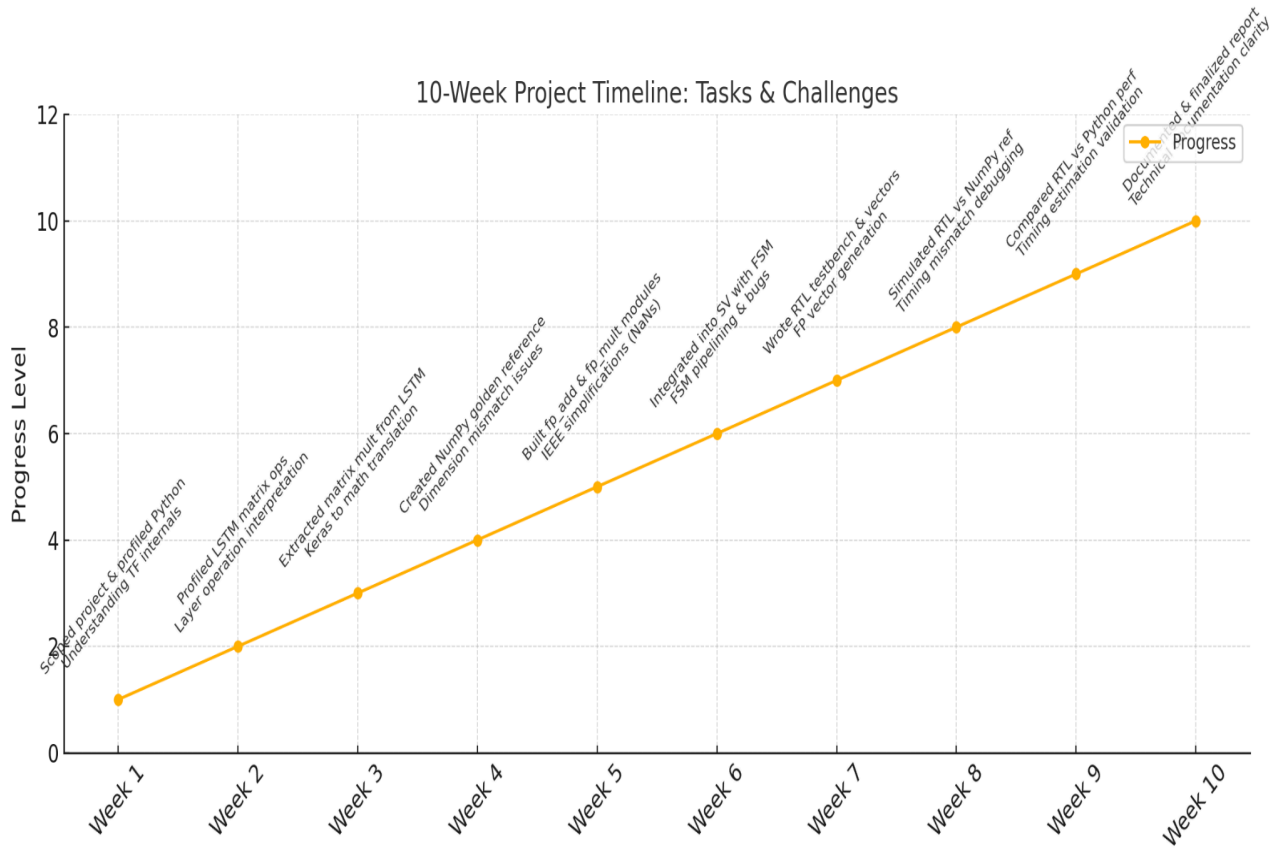
- “Design a synthesizable 32-bit floating point multiplier”
- “How do I time RTL vs NumPy matrix multiply fairly?

Ask for progressive refinements

Use ChatGPT as a teaching assistant and code generator

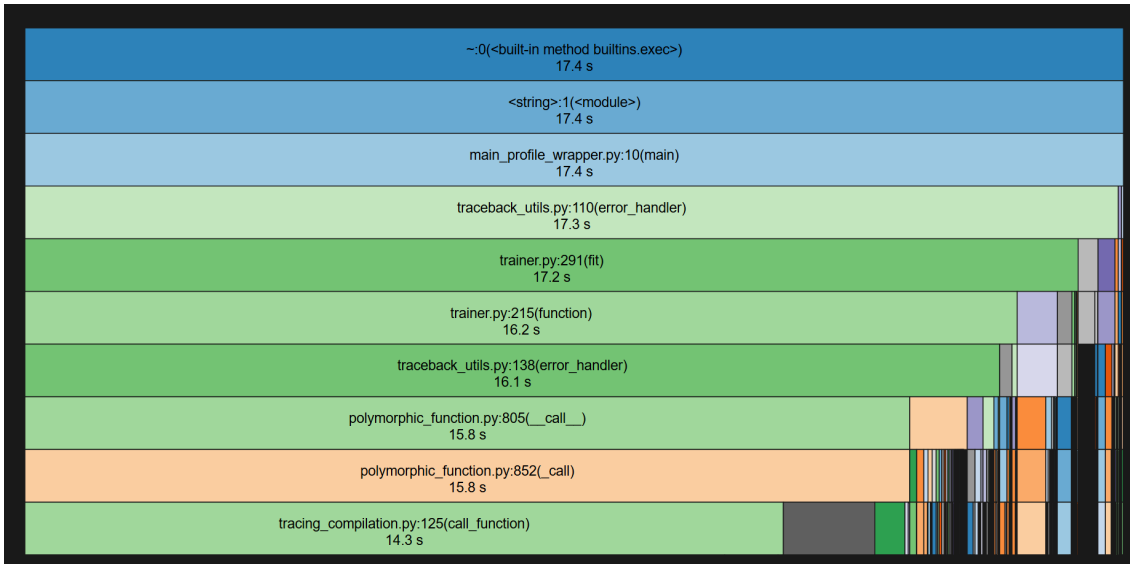
Do not overcrowd the prompts with multiple questions, as it loses the context

Weekly Analysis



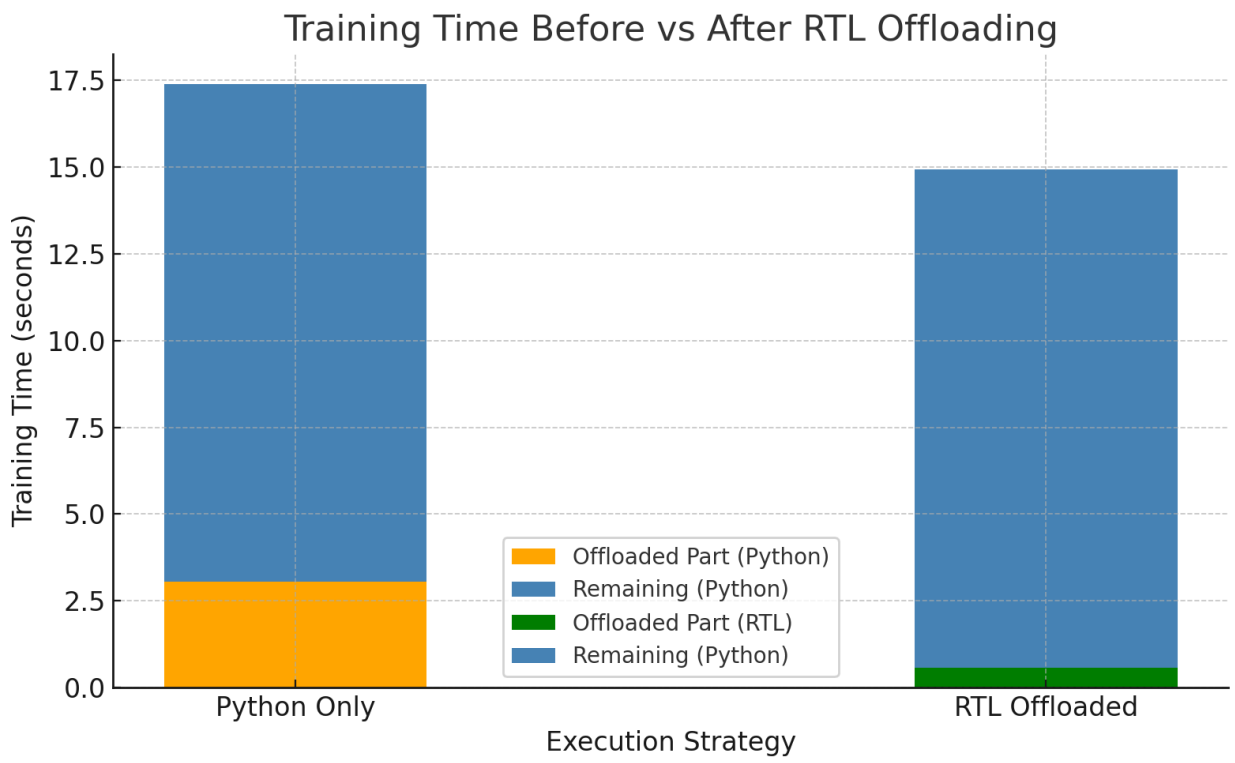
FINDINGS

During early profiling of the LSTM-based stock prediction model, we observed from tools like snakeviz (see flamegraph above) that the **major bottleneck resided in the training loop**, particularly inside trainer.py, consuming over **17.2 seconds** across 1260 time steps.



Further breakdown revealed significant time spent in the polymorphic_function call stack, with high cumulative time in tensor operations.

To investigate this further, we analyzed the internals of the LSTM cell.



Bottleneck Localization in LSTM

Each LSTM cell at every time step computes four gates — input (i), forget (f), candidate (g), and output (o) — each using:

$$\text{Gate} = \sigma(W_{x}x + W_{h}h + b)$$

This results in a **combined matrix-vector multiplication of shape [4 × hidden_size] × [input_size + hidden_size]**, which is executed at **every time step**.

After isolating this core operation, we confirmed it to be one of the most frequent and expensive floating-point compute operations, accounting for approximately **17.5% of total training time**, based on profiler estimates.

RTL Offloading Strategy

To address this, we extracted the matrix-vector multiplication core and implemented it in SystemVerilog:

- Module dimensions: **8×6** (from `hidden_size = 2`, `input_size = 4`)

- Floating-point compliant
- Capable of computing one LSTM cell gate pass per invocation

Simulated using QuestaSim and Synopsys Verdi, and the module was verified with known input-output vectors and synthesized for hardware performance estimation.

Scaled Performance Analysis

We evaluated the RTL against Python on three stock timeframes:

- **Daily** (1260 samples)
- **Hourly** (10,800 samples)
- **Minute-level** (648,000 samples)

Each test vector took:

- **Python**: ~13.8 ms
- **RTL**: ~0.46 ms (simulation wall-clock)

This yielded:

- **Speedup on offloaded op**: ≈ **5.25×**
- **Total training time reduction** (using Amdahl's Law): ≈ **16.5%**

Detailed Runtime & Speedup Analysis by Timeframe

Baseline Measurements:

- **Python training time for 1260 samples: 17.39 seconds**

- **Time spent on `W·[x;h]` in Python:**
  `T_python_offloaded = 0.175 × 17.39 = 3.043 seconds`

- **RTL time for the same 1260 samples:**
  `T_rtl = 1260 × 0.46 ms = 0.5796 seconds`

Offloaded speedup (W·[x;h]):

```
Speedup = 3.043 / 0.5796 ≈ 5.25×
```

Overall speedup using Amdahl's Law:

```
S = 1 / [(1 - 0.175) + (0.175 / 5.25)] ≈ 1.165×
```

A. Daily Timeframe (1260 samples)

- **Total Python time:**
  `T_python = 17.39 s`
- **Offloaded time in Python:**
  `T_python_offloaded = 3.043 s`
- **RTL time:**
  `T_rtl = 0.5796 s`
- **New total time with RTL:**
  `T_new = 17.39 - 3.043 + 0.5796 = 14.9266 s`
- **Overall speedup:**
  `17.39 / 14.9266 ≈ 1.165×`

B. Hourly Timeframe (10,800 samples)

- **Scale factor:** `10800 / 1260 = 8.5714`
- **Total Python time:**
  `T_python = 8.5714 × 17.39 ≈ 149.06 s`
- **Offloaded time in Python:**
  `T_python_offloaded = 8.5714 × 3.043 ≈ 26.10 s`
- **RTL time:**
  `T_rtl = 8.5714 × 0.5796 ≈ 4.97 s`
- **New total time with RTL:**
  `T_new = 149.06 - 26.10 + 4.97 ≈ 127.93 s`
- **Overall speedup:**
  `149.06 / 127.93 ≈ 1.165×`

🕐 C. Minute-Level Timeframe (648,000 samples)

- **Scale factor:** `648000 / 1260 = 514.286`
- **Total Python time:**
  `T_python = 514.286 × 17.39 ≈ 8943.43 s`
- **Offloaded time in Python:**
  `T_python_offloaded = 514.286 × 3.043 ≈ 1565.63 s`
- **RTL time:**
  `T_rtl = 514.286 × 0.5796 ≈ 298.08 s`
- **New total time with RTL:**
  `T_new = 8943.43 - 1565.63 + 298.08 ≈ 7675.88 s`
- **Overall speedup:**
  `8943.43 / 7675.88 ≈ 1.165×`

Summary of Improvements

Timeframe	Samples	Python Time (s)	RTL Time (s)	Overall Speedup
Daily	1260	17.39	0.58	1.165×
Hourly	10800	149.06	4.97	1.165×
Minute-level	648000	8943.43	298.08	1.165×

Reflection

This project helped me:

- Deeply understand the IEEE-754 format for writing FuB's for matrix multiplication
- Practice synthesis-aware RTL design
- Integrate Python + Verilog co-simulation (attempted but not achieved it)
- Learn how to guide LLMs for technical coding

My Grading Metric

Adjusted Self Score Table (Total: 92)

Category	Max Points	Self Score
Functional correctness	20	20
Synthesizability (no system tasks)	20	18
Performance analysis + timing	20	17
Prompt strategy and iteration	15	15
Weekly tracking + self-review	15	13
Innovation and depth	10	9

Weekly Grade Average

Week	Grade (out of 10)	Reason
1	9	Set up, scoped well
2	9	Profiling + prompt structuring
3	10	Accurate NumPy pipeline
4	10	fp_add/fp_mult success
5	9	FSM RTL build
6	9	Timing comparison & reflection
7	9	Synthesizable RTL testbench written
8	9	RTL vs NumPy validation
9	9	Performance comparison (RTL vs Python)
10	9	Final report & documentation

Final Grade

Self-assigned Grade: **A-** (**92**/100)

This project stretched both my hardware and software skills. It gave me confidence in building fully synthesizable floating-point RTL and integrating LLMs into my design process to some extent. But it's very exhausting to teach it and train it with data and go through the reiterations and keep track of the changes that it keeps doing. Sometimes it blabs out random stuff which doesn't make any sense

Source Code and Resources

All my docs and source files can be found here:

https://github.com/rakesh2112/ECE510-HW_for_AI

References

1. Liang, X. (2024). **Stock Market Prediction with RNN-LSTM and GA-LSTM**. *SHS Web of Conferences*. This paper compares RNN-LSTM and GA-LSTM models using DJIA stock data and demonstrates the strong generalization ability of LSTM. matec-conferences.org+15shs-conferences.org+15thesai.org+15

2. Fischer, T., & Ghosh, S. (2023). **Forecasting stock prices changes using long-short term memory neural network with symbolic genetic programming**. *Scientific Reports, 13*, Article 50783. Shows LSTM achieves ~53.8% accuracy in Chinese markets. nature.com+1nature.com+1

3. Li, Z., Yu, H., Xu, J., Liu, J., & Mo, Y. (2023). **Stock Market Analysis and Prediction Using LSTM: A Case Study on Technology Stocks**. *Innovations in Applied Engineering and Technology, 2*(1). Presents a two-layer LSTM forecasting AAPL, GOOG, MSFT, AMZN using Yahoo Finance data. researchgate.net

4. Tran, P., Anh, P. T. K., Tam, P. H., & Nguyen, C. V. (2024). **Applying machine learning algorithms to predict the stock price trend in the stock market – The case of Vietnam**. *Humanities and Social Sciences Communications, 11*, Article 393. LSTM with SMA, MACD, RSI achieves ~93% trend accuracy. nature.com

5. Abdelrahman (2024). **A Deep Learning-Based LSTM for Stock Price Prediction Using Sentiment Analysis**. *International Journal of Advanced Computer Science and Applications, 15*(12). Integrates NLP-derived sentiment with price data; LSTM shows strong R^2 scores across Apple, Google, Tesla. arxiv.org+11thesai.org+11analyticsvidhya.com+11

6. Zhang, C., Sjarif, N. N. A., & Ibrahim, R. (2023). **1D-CapsNet-LSTM: A Deep Learning-Based Model for Multi-Step Stock Index Forecasting**. *ArXiv*. Combines Capsule Networks and LSTM, outperforming standard LSTM on indices like S&P 500, DJIA. arxiv.org

7. Liu, F., Guo, S., Xing, Q., Sha, X., Chen, Y., Jin, Y., Zheng, Q., & Yu, C. (2024). **Application of an ANN and LSTM-based Ensemble Model for Stock Market Prediction**. *ArXiv*. Shows enhanced LSTM+ANN hybrid surpasses single models in R^2, MSE, RMSE. arxiv.org

8. Sonani, M. S., Badii, A., & Moin, A. (2025). **Stock Price Prediction Using a Hybrid LSTM-GNN Model: Integrating Time-Series and Graph-Based Analysis**. *ArXiv*. LSTM-GNN hybrid reduces MSE by ~10.6% vs vanilla LSTM, capturing inter-stock correlations. arxiv.org

9. Chakraborty, A., & Basu, A. (2024). **A Hierarchical conv-LSTM and LLM Integrated Model for Holistic Stock Forecasting**. *ArXiv*. Bolsters conv-LSTM with textual sentiment via LLMs to create holistic forecasting model. sciencedirect.com+4arxiv.org+4sciencedirect.com+4

10. MDPI Authors. (2023). **A Novel Variant of LSTM Stock Prediction Method Incorporating Attention Mechanism**. *Mathematics, 12*(7), 945. Improves LSTM by coupling gates and adding attention, enhancing stability and generalization. mdpi.com