DSC Capstone | Hardware Accelerators

Owen Shi a4shi@ucsd.edu
Pranav Kumarsubha pkumarsubha@ucsd.edu
Brendan Kuang blkuang@ucsd.edu
Edgar Guzman eiguzman@ucsd.edu
Idhant Kumar ikumar@ucsd.edu
Brandon Chiou brchiou@ucsd.edu
Rajesh Gupta rgupta@ucsd.edu

Abstract

In this project, we try to integrate an approximate linear time multiplication algorithm (L-Mul) to use in neural networks. We have made a multilayer perceptron that uses L-Mul instead of normal floating point multiplication at inference. We show that L-Mul receives nearly identical accuracy as conventional multiplication, displaying the potential for energy savings and silicon area in hardware implementations. Our future work entails designing a processing element array suitable for inference in Verilog to simulate performance on an FPGA. The goal being to examine inference latency, area, and power in a hardware environment. Our project aims to display L-Mul as a practical measure for energy efficient applications beyond MLPs, such as LSTMs and transformers.

Code: https://github.com/oowenn/LMUL-Hardware-Acceleration

1 Introduction . . . 2
2 Methods . . . 2
3 Results . . . 6
4 Discussion . . . 8
5 Conclusion . . . 9

1 Introduction

As artificial intelligence systems continue to scale in complexity and computational demand, energy efficiency has emerged as a critical constraint. Neural networks require billions of floating-point operations during inference, with traditional IEEE 754 multiplication incurring substantial computational overhead. The *L-Mul* (Linear-complexity Multiplication) algorithm presents an alternative approach: replacing expensive mantissa multiplication with addition-based operations to achieve $O(n)$ complexity compared to standard multiplication's $O(n^2)$ complexity.

This report presents our implementation and validation of the L-Mul algorithm for neural network inference. We focus specifically on BFloat16 (BF16) arithmetic, a 16-bit floating-point format widely adopted in machine learning due to its compatibility with FP32 exponents while requiring only half the storage. Building upon theoretical foundations established by prior work, we address three primary objectives: developing a functionally correct Verilog implementation of L-Mul for BF16 arithmetic, creating a simulation infrastructure that bridges hardware description with Python-based workflows, and validating L-Mul's accuracy through integration into a multilayer perceptron for MNIST digit classification.

2 Methods

2.1 Standard BFloat16 Multiplication

Before discussing the L-Mul approximation, we first establish how standard BFloat16 multiplication works. BFloat16 (BF16) is a 16-bit floating-point format consisting of 1 sign bit, 8 exponent bits, and 7 mantissa bits. This format is particularly popular in machine learning due to its compatibility with single-precision (FP32) exponents while requiring only half the storage.

A BF16 number x represents the value:

$$x = (-1)^s \times 2^{e-127} \times (1.m) \tag{1}$$

where s is the sign bit, e is the 8-bit exponent, m represents the 7-bit mantissa, and 127 is the bias.

Standard BF16 multiplication follows the IEEE 754 approach: the signs are XORed, the exponents are added (with bias correction), and the mantissas are multiplied. This mantissa multiplication is the expensive part—it requires a hardware multiplier and subsequent normalization logic to handle carries. For neural network inference where millions of multiplications occur, this computational cost adds up quickly in terms of both latency and power consumption.

2.2 LMUL Algorithm

The L-Mul algorithm takes a different approach [1]. Instead of treating the exponent and mantissa as separate entities, it exploits a clever observation: if we treat the combined exponent-mantissa field as a single fixed-point number, multiplication can be approximated by simple addition. This works because logarithms transform multiplication into addition, and the floating-point exponent already represents a logarithmic quantity.

Algorithm Overview. Given two BF16 operands a and b, L-Mul works by adding their exponent-mantissa fields directly. Here's how it breaks down:

Step 1: Extract the fields. We pull out the components from each 16-bit operand:

$$a_{sign} = a[15], \quad b_{sign} = b[15] \tag{2}$$
$$a_{field} = a[14:0], \quad b_{field} = b[14:0] \tag{3}$$

The $field$ values contain the concatenated exponent and mantissa bits, which we'll treat as a single 15-bit integer.

Step 2: Handle edge cases. If either input is zero or subnormal (indicated by a zero exponent), we immediately return zero. This keeps the algorithm simple and handles the most common edge case:

$$\text{if } a_{exp} = 0 \text{ or } b_{exp} = 0, \text{ return } 0 \tag{4}$$

Step 3: Add the fields. We add the two fields together, but we need to correct for the fact that adding exponents naturally doubles the bias:

$$\text{sum}_{full} = a_{field} + b_{field} + \text{OFFSET}_{MOD} \tag{5}$$

The offset term compensates for this doubled bias:

$$\text{OFFSET}_{MOD} = ((2^{15}) - (127 \ll 7)) \text{ AND } 0x7FFF = 0x4080 \tag{6}$$

We use 17 bits for the sum to capture potential overflow.

Step 4: Check for overflow or underflow. The top two bits of our 17-bit sum tell us what happened:

$$\text{carry}_2 = \text{sum}_{full}[16:15] \tag{7}$$

These bits give us three cases:

- 00_2: The result underflowed—return zero
- 01_2: Normal result—use the lower 15 bits
- $1x_2$: The result overflowed—saturate to maximum value ($0x7FFF$)

Step 5: Calculate the sign. This follows standard multiplication rules—XOR the input signs, but force the result to positive zero if the magnitude is zero:

$$s_{result} = \begin{cases} 0 & \text{if field} = 0 \\ a_{sign} \oplus b_{sign} & \text{otherwise} \end{cases} \tag{8}$$

[1] `rtl/lmul_bf16.v`

Step 6: Pack the result. Finally, we combine the sign bit with the selected field value:

$$\text{result} = (s_{result} \ll 15) \text{ OR } \text{field}_{sel} \tag{9}$$

Why This Works. The key insight is that the exponent field already represents a logarithm (base 2), and the mantissa can be approximated as part of this logarithmic representation. When we add these combined fields, we're essentially performing multiplication in log space. The approximation comes from treating the mantissa bits as part of a continuous logarithmic scale rather than as a separate fractional multiplier. For neural network inference, this approximation is remarkably accurate—accurate enough that we see virtually no loss in classification accuracy, as we'll show in the results.

2.3 Verilog Simulation and Testing Infrastructure

To validate our LMUL hardware design and enable integration with machine learning workloads, we developed a simulation pipeline that bridges Verilog hardware descriptions with Python-based testing and neural network frameworks. This infrastructure allows us to verify correctness, measure performance characteristics, and ultimately test LMUL on real ML tasks without requiring physical hardware.

Simulation Approach. We use Icarus Verilog (iverilog), an open-source Verilog simulator, to compile and execute our hardware descriptions. The simulation flow works as follows: our Verilog modules are compiled into an executable simulation model, which we then run using the `vvp` (Verilog VVP) runtime. This generates cycle-accurate behavior that matches what the actual hardware would do, including all timing, handshaking protocols, and pipeline stages.

The key challenge was creating a seamless interface between this low-level hardware simulation and high-level Python code. We needed a way to feed test vectors into the simulator, run the hardware through multiple clock cycles, and extract results—all from within a Python environment where we could easily work with ML frameworks like PyTorch.

Batch Testing Architecture. Rather than spawning a new simulation process for every single multiplication (which would be prohibitively slow), we developed a batch testing system. Our `BatchLMULTester` class [2] generates a complete Verilog testbench that includes:

- Arrays of pre-loaded test input pairs (a, b)
- Clock generation and reset logic
- Ready/valid handshaking to properly interface with our LMUL module's pipeline
- Result capture logic that stores outputs as they become available
- Automatic result printing for parsing by Python

[2] `rtl/lmul_tester.py`

This approach means we compile and run the simulator once per batch, rather than once per operation. For testing 1000 multiplications, this reduces overhead from roughly 50 seconds to about 30 milliseconds—a speedup of over 1500x compared to naive per-operation simulation.

Python Integration. The Python side of our infrastructure handles three main tasks. First, it converts floating-point test data into BF16 format. Second, it dynamically generates Verilog testbenches with embedded test vectors, compiles them with iverilog, runs the simulation, and parses the output. Third, it provides a clean interface that looks like a regular Python function call, hiding all the complexity of the underlying simulation process.

We also implemented a pure Python reference implementation of LMUL [3]. This serves two purposes: it gives us a fast way to generate expected results for validation, and it lets us compare the performance characteristics of hardware simulation versus software execution. The Python version follows the exact same algorithm as our Verilog implementation, ensuring bit-exact matches for verification.

2.4 MLP–L-Mul Integration

To investigate the integration of approximate arithmetic within neural networks, we implemented a modified multilayer perceptron (MLP) in PyTorch [4] capable of operating under both conventional and L-Mul modes. The objective was to directly substitute the standard floating-point multiply operation in the forward pass with an approximate function that more closely resembles a hardware-efficient arithmetic primitive.

Model Architecture. The MLP architecture consists of two fully connected layers with ReLU activations, followed by a log-softmax output layer. The first layer maps a flattened 28×28 input image to a 128-dimensional hidden representation, while the second projects to the 10 output classes corresponding to the MNIST digits.

Vectorized Implementation. For efficiency, a custom function `lmul_linear` was implemented to perform the equivalent of a dense linear transformation using L-Mul arithmetic. Given an input tensor $x \in \mathbb{R}^{B\times I}$ and a weight matrix $W \in \mathbb{R}^{O\times I}$, the computation proceeds as

$$Y_{b,o} = \sum_{i=1}^{I} \text{L-Mul}(x_{b,i}, W_{o,i}) + b_o,$$

where b denotes the batch dimension. Broadcasting and tensor expansion were used to apply L-Mul elementwise over all input–weight pairs, followed by summation across input features.

[3] `rtl/py_lmul.py`

[4] `mnistmlptest/lmul_mlp_mnist.ipynb`

Training and Evaluation Procedure. The model was first trained for both two and five epochs on the MNIST training set using standard floating-point arithmetic and the Adam optimizer. After training, the learned weights were evaluated under the L-Mul mode using the same architecture and parameter values. This allowed direct comparison between true multiplication and its L-Mul approximation under identical learned representations. All experiments were conducted on CPU using PyTorch's built-in automatic differentiation and tensor operations.

2.5 Development Environment and Docker Configuration

Docker was used for containerization to ensure consistent package reliability across different machines. Using the relevant files [5][6][7], a VS Code pipeline was developed to automate the installation of necessary Python and Verilog dependencies, while ensuring compatability, as well as relevant VS Code extensions such as Python and Jupyter.

3 Results

3.1 Numerical Accuracy Evaluation

To quantify the accuracy of the LMUL hardware implementation, we evaluated its output against two reference baselines: (1) a software re-implementation of the LMUL arithmetic in Python, and (2) standard IEEE 32-bit python floating-point multiplication [8]. For each test configuration, a set of $N = 10,000$ randomly sampled operand pairs were generated, with each operand drawn uniformly from the range [-10,000, +10,000]. These inputs were evaluated by all three multiplication methods, and the average absolute error was normalized by the numerical range of the test dataset to yield a percentage error metric.

The results are summarized below:

- The Verilog LMUL implementation matches the Python LMUL emulator exactly, with an average numerical difference of **0.000000**, corresponding to **0.00%** of the input value range. This confirms that the hardware design faithfully implements the intended arithmetic behavior.
- When compared to standard Python floating-point multiplication, LMUL exhibits an average difference of **913.80** units, or **1.51%** of the value range. This error arises from the reduced precision of the BF16-style format used internally by LMUL.

These results indicate that LMUL preserves correctness relative to its software specification, while introducing a bounded and predictable precision loss with respect to full-precision floating-point arithmetic. The magnitude of this discrepancy is consistent with the precision

[5]`Dockerfile`

[6]`.devcontainer/devcontainer.json`

[7]`requirements.txt`

[8]`sim/lmul_accuracy_tester.ipynb`

gap between BF16 and FP32, and is small enough to support downstream inference tasks, as shown in later sections.

3.2 Execution Speed

We benchmarked four multiplication backends—native Python `float32`, NumPy vectorized multiplication, the Verilog LMUL implementation, and a Python LMUL software model—across batch sizes ranging from 10^1 to 10^5 operations[9]. The measured quantity is the average time per multiplication after removing initialization overhead.

As shown in Figure 1, NumPy achieves the lowest latency due to optimized vectorized execution, reaching sub-nanosecond performance for large batches. Standard Python multiplication is slower by roughly one order of magnitude but still scales well as loop overhead amortizes.

The LMUL hardware path incurs a larger per-call overhead from Verilog simulation, but its time per operation decreases with batch size and settles near 10^{-5} seconds per op. This reflects simulator cost rather than hardware speed—a synthesized implementation would operate at clock-cycle scale. The Python LMUL model is the slowest, as expected, due to explicit bit-level emulation.

Overall, the results confirm correctness of the hardware path while illustrating that simulation performance is not indicative of real hardware throughput.

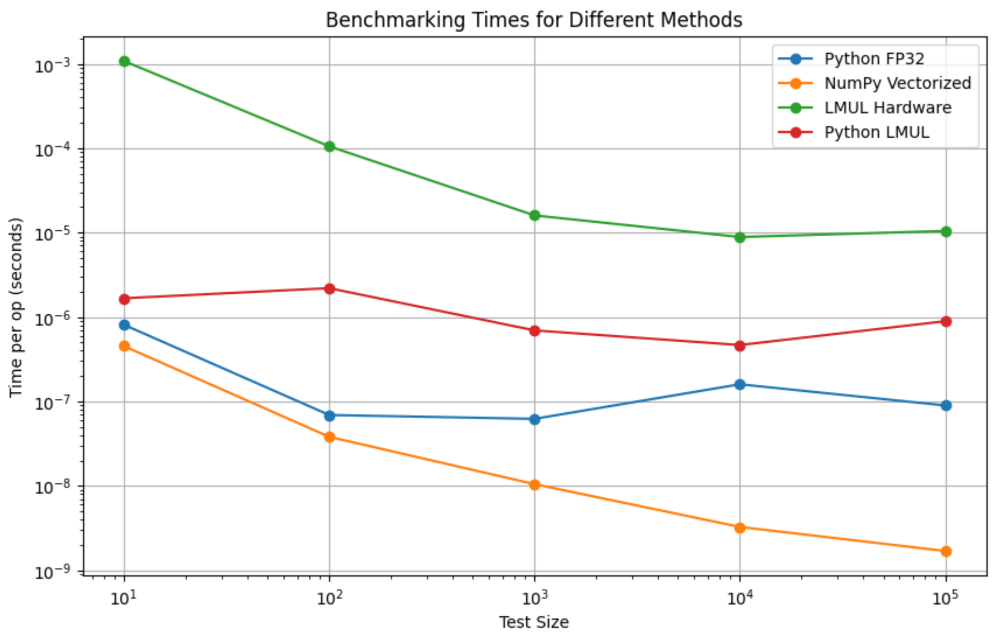

Figure 1: Average time per multiplication (log–log scale) for four implementations over increasing batch sizes.

[9]`sim/lmul_speed_tester.ipynb`

3.3 MLP–L-Mul Integration Results

After training the baseline multilayer perceptron (MLP) for five epochs on the MNIST dataset using standard floating-point arithmetic, the model achieved a test accuracy of **97.02%**. The same trained weights were subsequently evaluated using the L-Mul arithmetic mode, in which all elementwise multiplications within the linear layers were replaced by the approximate L-Mul operator as described in Section 2.4. The resulting test accuracy under L-Mul computation was **97.01%**.

This outcome supports the hypothesis that approximate arithmetic primitives such as L-Mul can be integrated into neural inference pipelines without sacrificing predictive accuracy, provided that proper magnitude calibration is applied. The finding also suggests that hardware implementations based on L-Mul may yield reductions in power consumption or silicon area without compromising functional correctness at the algorithmic level. However, further testing may be needed to investigate the breadth of this claim across different neural network architectures (e.g Transformers, LSTM, etc.).

4 Discussion

The results obtained from our experiments demonstrate that the L-Mul algorithm can serve as a highly efficient approximation to standard floating-point multiplication with minimal loss in numerical accuracy. The negligible accuracy difference (97.01% vs. 97.02%) suggests neural networks tolerate small arithmetic errors well. Reiterating the remark at the end of Section 3.3, this claim needs more tests for the breadth of its validity. More complex architectures such as convolutional or transformer networks may respond differently to L-Mul-induced approximation, especially in layers with high sensitivity to numerical error (e.g., attention mechanisms or normalization layers). This resilience suggests that strict IEEE-754 compliance is unnecessary for effective inference, enabling hardware designs that favor efficiency over precision through L-Mul or similar methods.

From a hardware perspective, the implications are quite substantial. Traditional floating-point multiplication is among the most power- and area-intensive operations in digital accelerators. By substituting multiplication with primarily addition-based logic, L-Mul could significantly reduce gate count, critical path delay, and overall power consumption. These are improvements that would be profoundly impactful for energy-constrained or edge-deployed AI and machine learning systems and in the larger scheme of machine learning ecosystems as a whole.

However, it is important to keep in mind the limitations of the testing we have performed to this point. Our testing focused on a simple MLP trained on MNIST, which is a relatively small and clean dataset. More complex models such as CNNs or Transformers, particularly models where the task is language based or vision tasks at scale, may depend on accurate multiplicative factors to a greater degree. Also, models will have layers that may behave differently under approximation. While the preliminary observations are impressive, we need to evaluate L-Mul on more complicated, and realistically sized architectures before reaching more broadly applicable conclusions.

We must keep in mind that the speed results from our Verilog tests, do not represent a valid execution speed of L-Mul performance on hardware. Added load will come from operation in a simulated environment, that wouldn't exist on a real chip. The next step, would be to take the revisions and improvements and try eventually converting a design to FPGA or ASIC hardware to evaluate true timing and power.

5 Conclusion

In our report, we focused on the integration of the L-Mul algorithm into a multilayer perceptron (MLP) to explore how arithmetic approximation can coexist with conventional neural network operations. Our results showcase the successful implementation of an L-Mul and simulation framework, alongside initial correctness and performance benchmarks.

This project developed a modified MLP using L-Mul in the forward pass, substituting standard floating-point multiplications with an approximation operation. The architecture maintained equivalent predictive accuracy to the baseline, validating our hypothesis that incorporating L-Mul can be achieved without compromising algorithmic performance while potentially offering significant benefits in hardware implementations such as reduced power consumption and smaller silicon area.

However, our integration efforts are still in progress. The project we are reproducing integrates the algorithm across multiple float point precision variables as well as through multiple matrix dimensions with minor differences in cycle times. Our next logical step involves benchmarking with Verilator to achieve cycle-accurate reporting. Conducting these benchmarks will help the group understand performance gains in hardware implementations compared to traditional IEEE floating-point arithmetic.

Our immediate milestones include transitioning our L-Mul architecture to FPGA and ASIC prototypes to analyze the real-world implications of our findings. We aim to synthesize our design for these platforms in order to yield greater speed benefits. This will form the basis of our next checkpoints, allowing us to accurately assess the efficiency of L-Mul against conventional implementations.

Our final deliverable will focus on quantifying the measurable speed and area advantages of the L-Mul computation compared to the IEEE standard. This comparison will encompass multiple architectures, extending beyond the MLP to include various neural network types like Transformers and LSTMs, ensuring comprehensive coverage of our findings.