Designing an Energy-Efficient Hardware Accelerator for Deep Learning

Kai Breese
kbreese@ucsd.edu

Justin Chou
jtchou@ucsd.edu

Katelyn Abille
kabille@ucsd.edu

Lukas Fullner
lfullner@ucsd.edu

Rajesh Gupta
rgupta@ucsd.edu

Abstract

Efficient floating-point operations are a significant challenge in large neural networks and other computationally intensive machine learning algorithms, where energy consumption and latency are key constraints. In this report, we present an implementation of the linear-complexity multiplication ($\mathscr{L}$-Mul) algorithm designed to approximate floating-point multiplication using addition (Luo and Sun 2024). By leveraging this approximation, $\mathscr{L}$-Mul achieves high precision with significantly lower computational cost than traditional floating-point multiplication methods. Our goal with this project is to develop a working simulation of a processor that can run machine learning models such as a multilayer perception or a transformer. The core of this processor is a matrix multiplication module utilizing the $\mathscr{L}$-Mul algorithm to achieve faster and more efficient processing of machine learning models. This approach aims to reduce energy consumption and processing time, making AI systems more sustainable and cost-effective.

Website: https://nakschou.github.io/hardware-accelerators-site/
Code: https://github.com/ninjakaib/hardware-accelerators

1 Introduction

As modern artificial intelligence (AI) systems grow in scale and complexity, their computational processes require increasingly large amounts of energy. Notably, ChatGPT required an estimated 564 MWh per day as of February 2023. In comparison, the cost of two days nearly amounts to the total 1,287 MWh used throughout the training phase, highlighting that inference is a major long-term cost (de Vries 2023). Similar trends appear in other large-scale models, with Google reporting that 60% of its ML energy usage is dedicated to inference (Patterson et al. 2022). These concerns extend beyond cost, as the environmental impact of large-scale AI computation grows. Addressing these challenges requires new hardware solutions that optimize energy efficiency without sacrificing performance.

Addressing this challenge, we aim to investigate a new approach by building on the existing linear-complexity multiplication ($\mathscr{L}$-Mul) algorithm developed by Luo and Sun (2024), which achieves high precision while reducing computational overhead. Our advancements suggest that $\mathscr{L}$-Mul could play a critical role in optimizing neural network efficiency. By leveraging $\mathscr{L}$-Mul, we aim to design a hardware accelerator that optimizes floating-point operations, improving both energy efficiency and computational speed in neural network inference.

To achieve this, we will focus exclusively on PyRTL for development and expand our systolic array beyond the original 2×2 design for more efficient floating-point operations. Additionally, we will implement hardware activation units for machine learning functions, such as ReLU and Sigmoid, and benchmark our design against traditional floating-point multipliers. To assess real-world feasibility, we will run machine learning models on our processor and evaluate $\mathscr{L}$-Mul 's performance within an ONNX-based workflow. A comprehensive testing suite will be developed to measure performance metrics and optimize hyperparameters using the generated data. By systematically analyzing our model's efficiency with our data science background, we aim to refine our design for maximum energy savings and computational accuracy.

1.1 Prior Work

Already, we have seen the increasing development of specialized AI chips designed to optimize training and inference efficiency. Google's Tensor Processing Unit (TPU), for example, leverages a systolic array architecture to accelerate tensor operations, particularly matrix multiplication.[1] AWS Inferentia2 delivers 4x higher throughput, 10x lower latency, and 10x memory bandwidth than its original model with support for up to 190 TFLOPS of FP16 performance.[2] Alongside Intel Gaudi, Groq Language Processing Unit (LUP), and more, the market for new AI hardware continues to grow. These advancements highlight the growing need for more efficient computational methods that complement the increasing capabilities of AI hardware.

[1] https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[2] https://aws.amazon.com/ai/machine-learning/inferentia/

Literature Review

Addressing the need for sustainability, recent research has introduced novel algorithms and hardware implementations aimed at reducing computational overhead while maintaining high accuracy. The primary work informing our project by Luo and Sun (2024), for example, is where we draw the $\mathscr{L}$-Mul algorithm from and as such is the core motivation of our project. The key insight of $\mathscr{L}$-Mul is that traditional floating-point multiplication can be approximated using addition, significantly reducing computational cost while maintaining high precision. Floating-point multiplication typically involves mantissa multiplication, a computationally expensive step. $\mathscr{L}$-Mul simplifies this by replacing mantissa multiplication with a discovered offset term 2^{-4} to minimize error across a range of machine learning tasks. This approximation reduces the operation to a combination of addition, subtraction, and an XOR for the sign bit, achieving near-lossless accuracy while significantly improving energy efficiency. By leveraging $\mathscr{L}$-Mul in hardware, AI accelerators can eliminate costly multiplication operations, making large-scale model inference more efficient.

Building upon this foundation, Chen et al. (2024) propose a power-efficient hardware implementation of $\mathscr{L}$-Mul on FPGAs, targeting FP8 arithmetic. The authors design a custom FPGA-based approximate multiplier using lookup tables (LUTs) and carry chains to optimize energy efficiency and resource utilization. Their implementation, deployed on AMD UltraScale+ FPGAs, demonstrates that $\mathscr{L}$-Mul can be efficiently integrated in hardware while consuming 10% fewer resources than existing FPGA-based 8-bit multipliers. Furthermore, their results show that $\mathscr{L}$-Mul-based FP8 multipliers maintain accuracy in deep neural network inference tasks, making them a viable alternative to traditional floating-point multiplication in energy-constrained environments.

Expanding further on hardware and algorithmic efficiency, Zhou et al. (2021) proposes a framework for joint optimization of neural architectures and hardware accelerators. Rather than treating model architecture and hardware constraints as disjoint concerns, they develop and follow the Neural Architecture and Hardware Accelerator Search (NAHAS) method to optimally configure both simultaneously. Their findings suggest that co-designing model architectures alongside hardware configurations can improve both accuracy and energy efficiency, reducing power consumption by up to 2x under the same accuracy constraint. This approach targets industry-standard accelerators, demonstrating that a unified optimization framework can yield superior performance across diverse inference tasks. This research reinforces the importance of hardware-aware algorithm design, further motivating our investigation into $\mathscr{L}$-Mul as a multiplication-efficient approach tailored for deep learning accelerators.

1.2 Data Applications

Multiplication and its current algorithm are already fast on modern processors, so why do we want to accelerate such a basic mathematical process? The main reason: time and energy. The vast majority of data science and ML/AI applications rely heavily on matrix multiplication operations, to process structured numerical data (i.e. image pixels, text embeddings, and sensor inputs), so optimizing that core step has a domino effect, improving

the neural network's overall runtime. By optimizing multiplication, we can achieve faster processing, leading to quicker model inference and reduced response times for large language models (LLMs) and other transformer-based architectures. Additionally, minimizing the number of operations reduces energy consumption, which in turn lowers the cost of operating the model.

2 Background

2.1 Floating-Point Numbers

Floating-point numbers are a fundamental data type in computing to store decimal values accurately. Today, most machine learning models and algorithms use floating-point (FP) tensors, typically in FP32 or mixed-precision formats like FP16, to represent their inputs, outputs, and trainable parameters (Luo and Sun 2024).

2.1.1 IEEE-754 Standard

The IEEE-754 standard defines the representation and behaviors of floating-point numbers in most systems. A floating-point number at its core is represented in three sections of bits: the single sign bit, followed by the exponent bits, and mantissa bits (also known as the significand). The most significant bit (MSB) is always the sign bit, determining whether the number is positive or negative. The exponent scales the number by a power of two, and lastly, the least significant bits (LSB) representing the mantissa provide the fractional part of the number.

2.1.2 Floating-Point Formats

Full Precision (FP32) Full-precision floating-point numbers, often referred to as FP32, use 32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa. This format provides a wide dynamic range and high precision, making it suitable for a variety of scientific and engineering applications.

Brain Float Despite being reduced to 16 bits, Brain Floating-Point 16 (BF16) is widely adopted in machine learning for its ability to retain the same dynamic range as FP32 while reducing memory usage and computational requirements by using a lower precision mantissa. This format uses 8 bits for the exponent (like FP32) and 7 bits for the mantissa (compared to FP32's 23 bits), as shown in Figure 1, making it effective for deep learning tasks with minimal accuracy loss. By sharing the same number of exponent bits as FP32, BF16 enables stable training and inference for large models like LLaMA, Qwen, and Phi, despite its reduced mantissa size (Fujii, Nakamura and Yokota 2024).

Bit Type	S	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M
Bit #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 1: BF16 sign, exponent, and mantissa breakdown

The format follows this equation for normal numbers:

$$(-1)^{sign} 2^{\text{exp}-127} (1 + \frac{\text{mantissa}}{2^7})$$

And for subnormal numbers (when exponent = 0):

$$(-1)^{sign} 2^{-126} (0 + \frac{\text{mantissa}}{2^7})$$

FP8 FP8 is an 8-bit floating-point format designed for deep learning and hardware optimization, as discussed in Micikevicius et al. (2022); Noune et al. (2022). It comes in two variants:

- **E4M3**: 1 sign bit, 4 exponent bits, 3 mantissa bits.
- **E5M2**: 1 sign bit, 5 exponent bits, 2 mantissa bits.

Let's take a look at OCP 8-bit floating-point Specification (OFP8).[3]

Table 1: OFP8 exponent parameters

Parameter	E4M3	E5M2
Exponent bias	7	15
emax (unbiased)	8	15
emin (unbiased)	-6	-14

The value, v, of a normal OFP8 number is:

$$v = (-1)^S \times 2^{E-bias} \times (1 + 2^m \times M)$$

The value, v, of a subnormal OFP8 number (subnormals have E = 0 and M > 0) is:

$$v = (-1)^S \times 2^{1-bias} \times (0 + 2^{-m} \times M)$$

Exponent parameters and min/max values for both OFP8 formats are specified in Table 1. The E5M2 format represents infinities and NaNs, following IEEE 754 conventions, while the E4M3 format does not represent infinities and uses only two-bit patterns for NaN (a single mantissa-exponent bit pattern but allowing both values of the sign bit). This design choice increases the dynamic range of E4M3 by one binade, as shown in Noune et al. (2022), in

Table 2: OFP8 value encoding details

Parameter	E4M3	E5M2
Infinities	N/A	$S.11111.00_2$
NaN	$S.1111.111_2$	$S.11111.01, 10, 11_2$
Zeros	$S.0000.000_2$	$S.00000.00_2$
Max normal number	$S.1111.110_2 = \pm 448$	$S.11110.11_2 = \pm 57{,}344$
Min normal number	$S.0001.000_2 = \pm 2^{-6}$	$S.00001.00_2 = \pm 2^{-14}$
Max subnormal number	$S.0000.111_2 = \pm 0.875 \ \ 2^{-6}$	$S.00000.11_2 = \pm 0.75 \ \ 2^{-14}$
Min subnormal number	$S.0000.001_2 = \pm 2^{-9}$	$S.00000.01_2 = \pm 2^{-16}$
Dynamic range	18 binades	32 binades

order to increase emax to 8 and thus to increase the dynamic range by one binade. Various values for OFP8 formats are detailed in Table 2.

Our study originally focused on the implementation of the $\mathscr{L}$-Mul algorithm with 8-bit floating-point representation in mind, with later expansion to brain floating-point 16 (BF16).

2.1.3 Multiplication

Before we dive into the $\mathscr{L}$-Mul algorithm, it's important to understand standard floating-point multiplication. As we discussed, a floating-point number can be represented with a sign bit s, exponent bits e, bias b, and mantissa bits m where the value v is calculated as:

$$v = (-1)^s \times 2^{(e-b)} \times (1+m)$$

To multiply two floats together, we use the following equation:

$$\begin{aligned} Mul(x,y) &= (1+x_m) \cdot 2^{x_e} \cdot (1+y_m) \cdot 2^{y_e} \\ &= (1+x_m+y_m+x_m \cdot y_m) \cdot 2^{x_e+y_e} \end{aligned}$$

Incorporating our sign bit, we have:

$$p = (s_1 \oplus s_2) \times 2^{(e_1+e_2-2b)} \times (1+m_1+m_2+m_1 \times m_2)$$

Because the exponents are biased, we must subtract the bias twice to compute the unbiased exponent, otherwise, the bias is counted twice:

$$\begin{aligned} e_{\text{unbiased}} &= e_1 + e_2 - 2b \\ e_{\text{bits}} &= e_{\text{unbiased}} + b \\ e_{\text{bits}} &= e_1 + e_2 - b \end{aligned}$$

[3] https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

2.1.4 The Linear-Complexity Multiplication Algorithm ($\mathscr{L}$-Mul)

The core innovation of the $\mathscr{L}$-Mul algorithm is replacing the computationally expensive $O(m^2)$ mantissa multiplication with a simple approximation that achieves linear complexity. In standard floating-point multiplication, we compute:

$$\begin{aligned} Mul(x, y) &= (1 + x_m) \cdot 2^{x_e} \cdot (1 + y_m) \cdot 2^{y_e} \\ &= (1 + x_m + y_m + x_m \cdot y_m) \cdot 2^{x_e + y_e} \end{aligned}$$

The $\mathscr{L}$-Mul algorithm replaces the $x_m \cdot y_m$ term with a constant offset $2^{-L(M)}$, where M is the number of mantissa bits. The function $L(M)$ is defined as:

$$L(M) = \begin{cases} M, & M \leq 3 \\ 3, & M = 4 \\ 4, & M > 4 \end{cases}$$

This gives us the $\mathscr{L}$-Mul approximation:

$$\mathscr{L}\text{-Mul}(x, y) = (s_1 \oplus s_2) \times 2^{(e_1 + e_2 - b)} \times (1 + m_1 + m_2 + 2^{-L(M)})$$

The values of $L(M)$ were empirically determined to minimize the average error across a range of machine learning tasks. For example, with 8-bit floating-point numbers using 3 mantissa bits (FP8E4M3), $L(M) = 3$, resulting in an offset of $2^{-3} = 0.125$. This approximation works particularly well for neural network weights, which typically have a specific distribution that makes the error minimal.

In hardware implementation, $\mathscr{L}$-Mul can be executed with remarkable efficiency. The entire operation requires just: 1. An XOR operation for the sign bit: $s_{out} = s_1 \oplus s_2$ 2. An integer addition of the exponent and mantissa bits: $em_{out} = (e_1 \ll M + m_1) + (e_2 \ll M + m_2) - \mathit{offset}$

Where *offset* combines both the bias correction ($b \ll M$) and the approximation term ($2^{M - L(M)}$). This can be precomputed as a constant for any given floating-point format.

The elegance of this approach is that the integer addition automatically handles the carry from mantissa to exponent, eliminating the need for separate normalization steps. When the mantissa sum exceeds 2, the carry naturally propagates to the exponent bits, correctly scaling the result. This allows $\mathscr{L}$-Mul to achieve $O(n)$ complexity compared to the $O(n^2)$ complexity of standard floating-point multiplication, while maintaining sufficient accuracy for neural network computations.

2.1.5 Addition

In contrast to integer operations, addition of floating-point numbers is more complex than the multiplication operation. Since floating-point representation is akin to binary scientific notation, operands must be normalized to have the same exponent (floating-point must be aligned) before addition. To do this, the steps are:

1. *Identify which of the two numbers has the smaller exponent.* This number will need its mantissa adjusted to align with the larger exponent.

2. *Shift the mantissa of the smaller number right by the difference in exponents.*
   Let e_1 and e_2 be the exponents of the two numbers, where $e_1 > e_2$. The difference in exponents is $\Delta e = e_1 - e_2$. Shift the mantissa of the number with the smaller exponent (m_2) right by Δe bits.

3. *Add or subtract the aligned mantissas based on the sign bits.*
   Perform the addition or subtraction of the aligned mantissas (m_1 and m_2') depending on the sign bits of the operands. If the signs are the same, add the mantissas. If the signs differ, subtract the smaller mantissa from the larger one:

   $$m_{\text{sum}} = m_1 \pm m_2'$$

   The sign of the result is determined by the sign of the larger operand.

4. *Normalize the mantissa such that the binary point implies a single leading 1.*
   After addition, the mantissa m_{sum} may not be normalized (i.e., it may not have a single leading 1). To normalize, shift the mantissa left or right until it has the form 1.xxx. Count the number of shifts s required. If the mantissa overflows (i.e., becomes ≥ 2), shift it right by 1 and increment the exponent.

5. *Adjust the final exponent based on the amount the mantissa was shifted to normalize.*
   Adjust the exponent e_1 by the number of shifts s performed during normalization:

   $$e_{\text{final}} = e_1 + s$$

   If the mantissa was shifted left, s is positive; if shifted right, s is negative.

6. *Calculate the final sign based on the sign of the mantissa sum.*
   The final sign of the result is determined by the sign of the larger operand or the result of the mantissa addition. If the result is negative, set the sign bit to 1; otherwise, set it to 0.

2.2 Systolic Arrays

Matrix multiplication requires $O(N^3)$ floating-point operations which induces significant overhead for sequential processors. A well-known technique in hardware design to make GEMM (general matrix multiplications) more efficient is the use of an architecture known as a systolic array (Kung and Leiserson 1979). The concept for the systolic array involves an array of processing elements that have data passed between them, allowing for efficient data reuse and fewer global memory accesses. Systolic arrays can be used for more than just matrix multiplications, it is a general architecture that describes a way of spatial computing with processing elements in which the flow of data between each processing element

defines the operation. The PE used for matrix operations consists of a multiply and an accumulate unit (MAC). A MAC unit computes the product of two inputs and adds it to an accumulator which stores partial sums.

The processing element receives a number from the top and left, multiplies them, and then adds them to a contained total. The numbers are then passed to processing elements down and right. This structure allows for a systolic array to perform matrix multiplication by having two matrices passed in row by row. The data is multiplied when it intersects in processing elements, similarly to how matrix multiplication is performed by hand. Once all the data is passed through, the accumulated sums in each processing element can be read out to view the multiplied matrix. This diagram showcases the systolic array process, where data is fed in row by row, and where it intersects in processing elements it is multiplied and accumulated.

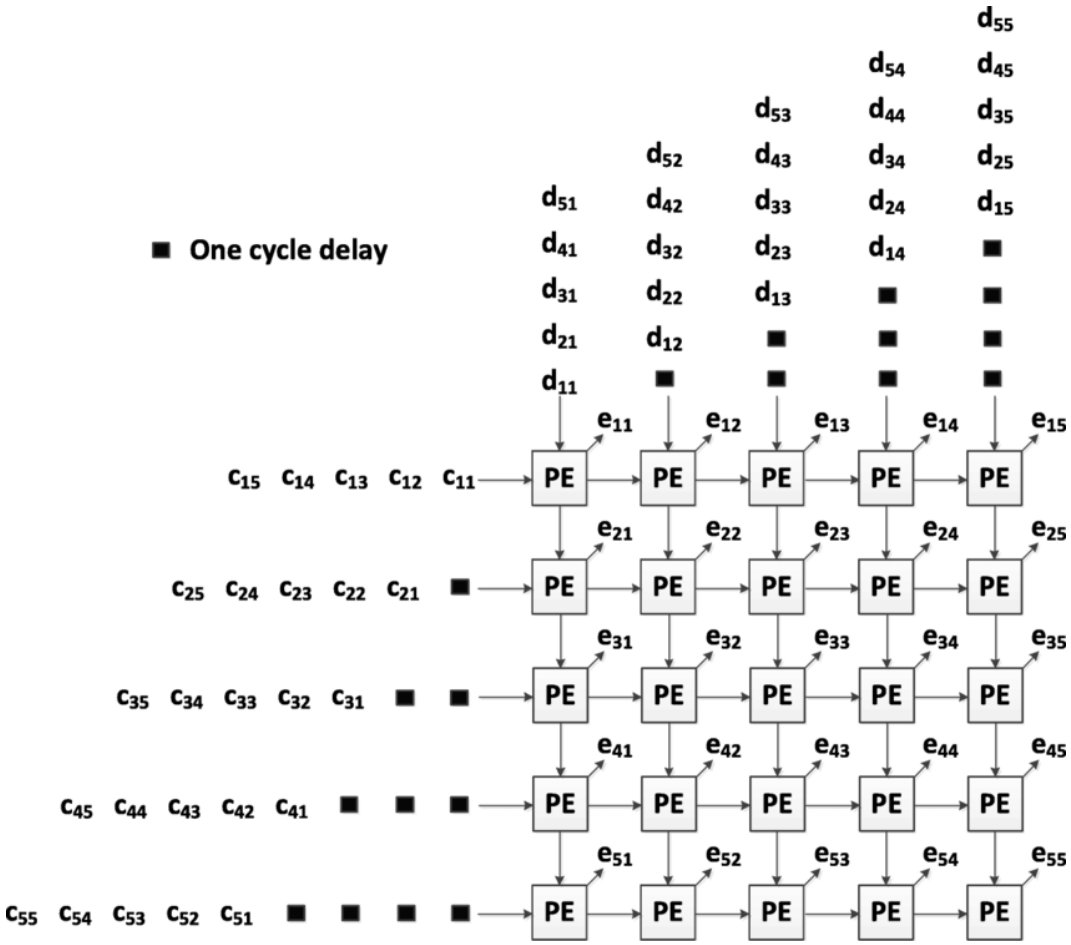

Figure 2: 5×5 Systolic Array Architecture from ResearchGate (2024)

2.2.1 Data Flow Patterns

There are various ways to employ systolic arrays for GEMMs (Raja 2024), each having their own performance tradeoffs and characteristics. Different data flows, such as Weight

Stationary (WS), Input Stationary (IS), and Output Stationary (OS), can be used in specific scenarios to optimize the speed and efficiency of the calculations.

Weight Stationary: In weight stationary (WS) data flow, weight matrix values are pre-filled into the systolic array, while the inputs and partial sums are propagated through the systolic array for each clock cycle. The spatial requirements, or the minimum size of the systolic array needed is the size of the weight matrix,

Input Stationary: Similar to WS data flow, the IS data flow pre-fills the systolic array with the input matrix values for an IS data flow is therefore the size of the input matrix, while the temporal dimension is mapped to M

Output Stationary: Finally, the output stationary data flow is where the output matrix is stationary while both the weight and input values are streamed into the array. In this flow, each PE accumulates the partial sum without propagating it until the entire operation is completed. After the multiplication, the output matrix is read to the output buffer. The spatial dimensions required to compute matrix multiplication through an OS data flow is the size of the output matrix, M×P, while the temporal dimension is mapped to N.

2.2.2 Characterizing Systolic Arrays

To evaluate the power requirements of the systolic arrays we estimate the energy consumption as (Raja 2024):

$$E = N_{PE} \cdot P_{PE} \cdot N_C \cdot T_{clk}$$

Where E is the energy consumed, N_{PE} is the number of processing elements in the systolic array, P_{PE} is the power consumption of one processing element, N_C is the total number of clock cycles needed to compute the matrix multiplication and T_{clk} is 1 / the clock frequency of the processing elements. SCALE-Sim was proposed by Samajdar et al, and gives the equation for the number of clock cycles as:

$$N_C = 2 \cdot S_R + S_C + T - 2$$

Where S_R is the number of spatial rows, S_C is the number of spatial columns, and T is the temporal dimension. Each of these dimensions is mapped differently based on the type of data flow used.

Table 3: Data flow strategies and their spatial/temporal mappings.

Data flow	Spatial Rows	Spatial Columns	Temporal
Weight Stationary (WS)	M	N	P
Input Stationary (IS)	N	P	M
Output Stationary (OS)	M	P	N

3 Methods

3.1 PyRTL

All of our major hardware implementations for this project (standard floating-point multiplier, $\mathscr{L}$-Mul multiplier, floating-point adder, systolic array and its processing elements, and the accumulator buffers for tiled matrix operations) were done in PyRTL (Mirza, Dangwal and Sherwood 2019). This approach proved to be most viable for us because it allowed us to utilize Pythonic syntax for our hardware description. The framework is rich with built-in features that smoothened out our development process. We relied heavily on immediate representation throughout our pipeline-building process due to its ability to simulate and optimize our work. It also allowed us to work in our comfort language as we built out our utility libraries, testing suite, visualizations, and miscellaneous logic control in a language we're comfortable with (Clow et al. 2017).

One of the major concerns we had was that in the event we wanted to continue with hardware development on this project, we'd have to transfer most of our work over to another framework. Conveniently, PyRTL keeps this option open, as it can be used to generate Verilog code.

3.2 High-Level Architecture

We are designing and optimizing on three levels to create an efficient and performant accelerator. First is the register transfer level; this is the lowest level hardware design of basic components like adders, multipliers, multiplexers, and instruction decoders. The key implementation here is the $\mathscr{L}$-Mul unit for approximating floating-point multiplication.

We use these low-level designs and combine them and orchestrate their dataflow patterns into components that represent higher levels of abstraction. These components include the parametrize-able systolic array, accumulator memory buffer, and vectorized activation function module. These components are represented as classes in our main code and can be configured with various parameters to control things like float data type, size of systolic array, and memory size. A key insight into the code design is that these classes serve as "empty containers" which initialize what are essentially placeholders for the hardware that will be fully initialized during simulation, not necessarily at runtime which gives us the flexibility to control and build them dynamically. This design choice allows us to control

things like the way data flows through the systolic array, choosing different timing and control signal patterns, and determining what we want to handle in hardware vs software on a simulation level.

The final accelerator unit architecture is heavily inspired by the Google TPU architecture (Jouppi et al. 2017), which relies on minimal control logic, a large matrix engine with accumulators and memory attached to the output, with an activation function module connected to the accumulator memory. This design pattern in combination with the DiP data flow enables a FIFO-free design, reducing latency and power. Figure 3 shows a high-level architectural diagram of the components and the data flow.

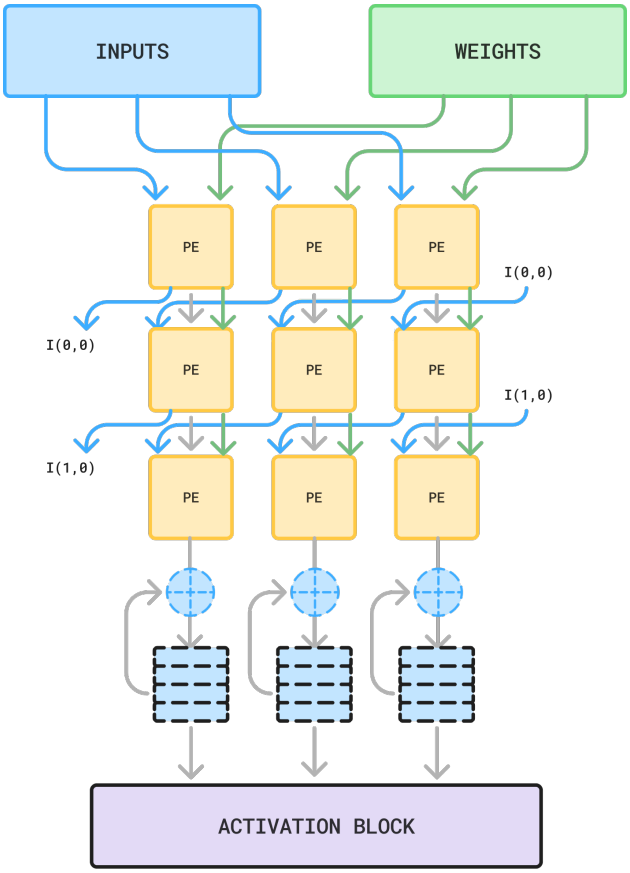

Figure 3: Architecture Diagram

3.3 Processing Elements

The core element of the systolic array is the processing element (PE). The PE utilizes a floating-point MAC unit and supports mixed precision weights, activations, and accumulation formats. FP8, Bfloat16, half-precision, and full-precision formats are supported for all components that make up a PE. Operands are upcast to the larger bitwidth of the two before the operation (multiply or add) is performed. Each PE contains 3 or 4 registers depending on the level of pipelining specified. Weight register, data register, and accumulation register

are the minimum requirements, and optionally a product register can be included to reduce the critical path distance from inputs to outputs, increasing the maximum frequency of the design. We leave the pipelining level configurable to explore the design space and tradeoffs between performance and efficiency. PEs also contain control signal inputs, but no control out wires meaning that control signals must be continuously sent from an external source and do not propagate between the PEs like the data. Figure 4 shows a block diagram of the processing element design.

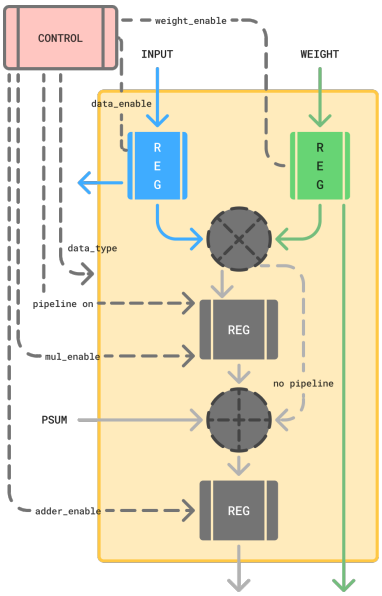

Figure 4: Processing Element Design

3.4 Adder Unit

A core component of the MAC unit is the adder, which features configurable pipelining to achieve high operating frequencies when required. Our design provides the flexibility to generate adders dynamically with either high-performance internal stages that consume more area and power, or simpler, more compact stages that maintain power efficiency with minimal delay penalties. The pipelined adder operates through five distinct stages:

Stage 1: Extracts the sign, exponent, and mantissa from both inputs and concatenates the hidden mantissa bit. This stage also computes status flags indicating whether any input is zero, NaN, or infinity.

Stage 2: Performs exponent subtraction to determine the alignment offset. The mantissa of the input with a smaller magnitude is identified based on the sign of the exponent difference. The offset between exponents determines the right-shift magnitude required for proper mantissa alignment.

Stage 3: Executes the alignment shift on the smaller mantissa and generates sticky, guard, and round (SGR) bits for precise rounding in subsequent stages. All other input components remain unchanged and propagate forward through this stage.

Stage 4: Performs addition or subtraction on the aligned mantissas, with the operation determined by the XOR of the two input signs. The resulting sum is processed by a

leading zero counter (LZC) module to determine the normalizing shift amount. The sign bit from this addition operation propagates to the final stage.

Stage 5: Subtracts the LZC value from the larger exponent and applies normalization to the final mantissa. This stage implements IEEE-compliant rounding using the SGR bits from stage 3, applies sign detection logic, and adjusts the exponent based on rounding overflow and leading zeros. The final components are concatenated to produce the IEEE 754-compliant result.

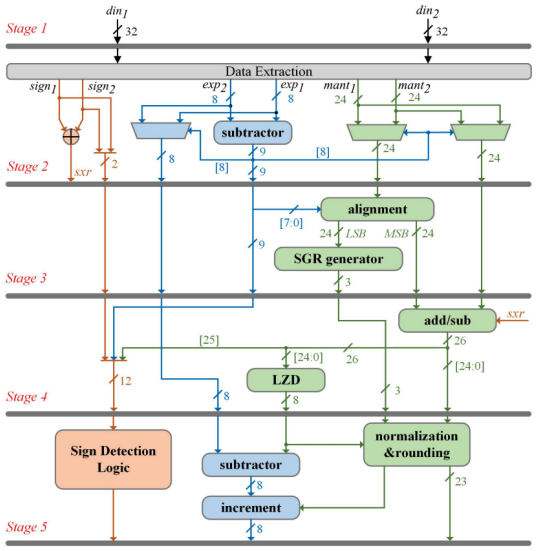

Figure 5: Block diagram of the pipelined adder unit showing data flow between stages and internal submodules.

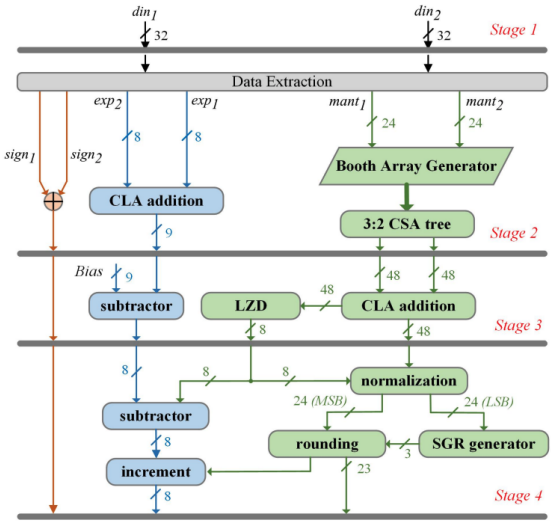

Figure 6: Block diagram showing the microarchitecture of the floating-point multiplier unit. Niknia et al. (2024)]

3.5 Multiplier Unit

We implement an IEEE 754-compliant floating-point multiplier to establish a baseline for comparison against our $\mathscr{L}$-Mul-based designs. The multiplication algorithm, while similar to addition, requires fewer pipeline stages due to its inherently simpler computational nature. Like the adder, our multiplier supports all target data types and offers configurations optimized for either performance or efficiency. The multiplier employs a four-stage pipeline architecture:

Stage 1: Extracts and registers the sign, exponent, and mantissa components from both inputs. This stage also calculates zero detection flags that propagate unchanged through the pipeline.

Stage 2: Adds the exponents using either a Kogge-Stone adder (in high-performance configuration) or a standard adder (in power-efficient configuration). Concurrently, the input mantissas undergo multiplication to calculate their product. The result sign is determined by computing the XOR of the input signs.

Stage 3: Detects leading zeros in the mantissa product and subtracts the exponent bias from the exponent sum calculated in the previous stage.

Stage 4: Utilizes the leading zero count to normalize the mantissa and adjusts the unbiased exponent sum accordingly. This stage generates sticky, guard, and round bits for IEEE-compliant rounding, which is then performed to produce the final result.

Figure 6 outlines the microarchitecture of the multiplier components and their data paths. Figure **??** provides a waveform visualization of value transitions through the various pipeline stages.

In our subsequent analysis, we refer to stages 1 and 2 of both the adder and multiplier collectively as "stage 2," since stage 1 performs no logical operations beyond input registration.

3.6 $\mathscr{L}$-Mul Unit

The $\mathscr{L}$-Mul hardware block is the easiest and smallest to implement thanks to the simplicity of the algorithm. We create a pipelined design for $\mathscr{L}$-Mul, though it is not strictly necessary due to the short critical path relative to other components. A fully combinational circuit is viable for energy-efficient settings, while the pipelined version demonstrates maximum theoretical gains over standard multiplication.

The original paper, "Addition is All You Need," demonstrates a software implementation of $\mathscr{L}$-Mul with just two assembly instructions. Our hardware design leverages the insight that the exponent bias subtraction and $\mathscr{L}$-Mul offset term can be combined ahead of time into a single value based on the input data type, reducing the need for an additional adder unit.

The method for calculating the combined bias and offset is as follows:

$$L(M) = \begin{cases} M, & \text{if } M \leq 3 \\ 3, & \text{if } M = 4 \\ 4, & \text{if } M > 4s \end{cases}$$

For a floating-point format with E exponent bits and M mantissa bits:

$$\begin{aligned} \text{bias} &= 2^{E-1} - 1 \\ \text{aligned bias} &= \text{bias} \ll M \\ \text{aligned offset} &= (1 \ll M) \gg L(M) = 2^{M - L(M)} \\ \text{unsigned offset} &= \text{aligned bias} - \text{aligned offset} \\ \text{signed offset} &= \sim \text{unsigned offset} + 1 \text{ (two's complement)} \end{aligned}$$

To achieve the most efficient design, we use the signed version represented as the two's complement of the unsigned version, allowing us to complete $\mathscr{L}$-Mul using only unsigned integer operations.

Below we calculate the offset for various floating-point formats:

FP8E4M3 ($E = 4$, $M = 3$, bias $= 7$)

$$\begin{aligned}\text{unsigned offset} &= (7 \ll 3) - ((1 \ll 3) \gg 3) \\ &= 56 - 1 = 55 = \texttt{0b0110.111} \\ \text{signed offset} &= \sim \texttt{0b0110.111} + 1 = \texttt{0b1001.001} = 73\end{aligned}$$

Float16 ($E = 5$, $M = 10$, bias $= 15$)

$$\begin{aligned}\text{unsigned offset} &= (15 \ll 10) - ((1 \ll 10) \gg 3) \\ &= 15360 - 128 = 15232 \\ \text{signed offset} &= \sim 15232 + 1 = \texttt{0b10001.0001000000} = 17472\end{aligned}$$

BFloat16 ($E = 8$, $M = 7$, bias $= 127$)

$$\begin{aligned}\text{unsigned offset} &= (127 \ll 7) - ((1 \ll 7) \gg 4) \\ &= 16256 - 8 = 16248 \\ \text{signed offset} &= \sim 16248 + 1 = \texttt{0b10000001.0001000} = 16520\end{aligned}$$

Float32 ($E = 8$, $M = 23$, bias $= 127$)

$$\begin{aligned}\text{unsigned offset} &= (127 \ll 23) - ((1 \ll 23) \gg 4) \\ &= 1065353216 - 524288 = 1064828928 \\ \text{signed offset} &= \texttt{0b10000001.00010000000000000000000} = 1082654720\end{aligned}$$

A clear pattern emerges in the two's complement representations. The signed offset can be expressed as:

$$\text{signed offset} = 2^{E+M-1} + 2^M + 2^{M-L(M)}$$

This bit pattern has 1's at exactly three positions: the most significant bit (MSB), the M-th bit, and the $(M - L(M))$-th bit, with all other bits set to 0.

3.6.1 Hardware Implementation

The design for the $\mathscr{L}$-Mul unit utilizes the precomputed offset terms as hardcoded constants in the circuit. To efficiently add 3 terms, a carry-save adder is used. As the algorithm leverages approximation, we assume any zero-exponent to represent a zero and automatically pass through a zero. Later analysis of neural network inference shows this method of handling denormals by clipping to 0 has little to no impact on accuracy.

When working with floating-point numbers, typically normalization is need after an operation by either adjusting the exponent or mantissa. By combining the entire operation into a single add, the mantissa carry is automatically propagated to the exponent. If the mantissa sum is greater than 2, the carry is added allowing $\mathscr{L}$-Mul to skip the rounding process. Overflow and underflow cases may occur, which can be detected by the 2 extra

Table 4: $\mathscr{L}$-Mul Carry Out Final Result Selection

$[em+1, em]$	**Final Exp Mantissa**	Info
`2'b00`	`0b0`	Underflow or denormal, clip to 0
`2'b01`	`sum[em-1:0]`	Normal result
`2'b1x`	MAX	Overflow, clip to max

carry out bits of the overall sum. Table 4 shows how these cases are handled. The sign out is calculated as $S_a \cdot S_b$, leaving $e+m$ bits from the inputs and the offset term.

Figure 7 shows a block diagram of the $\mathscr{L}$-Mul unit.

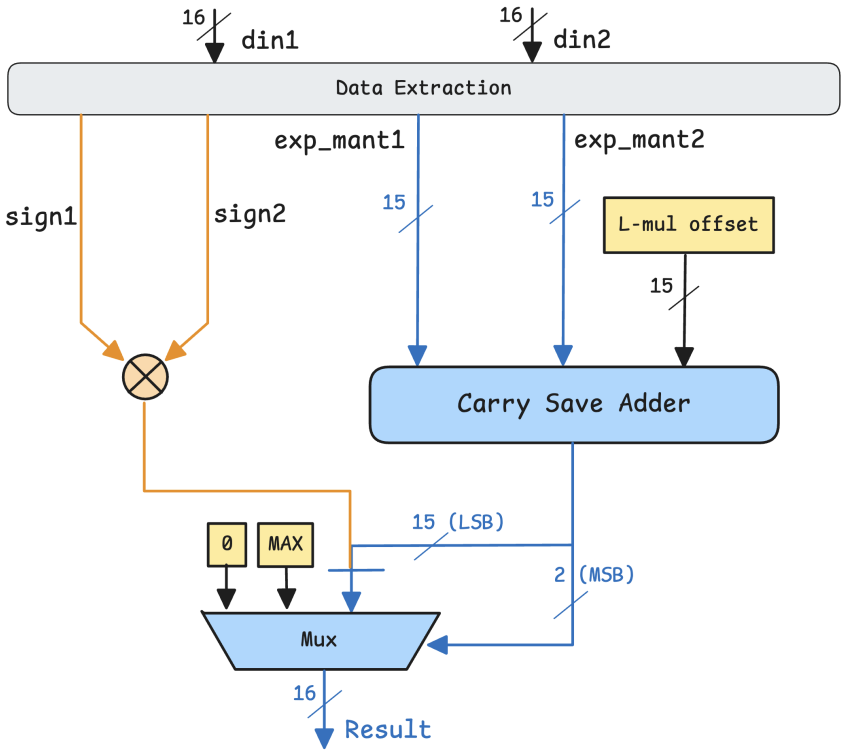

Figure 7: $\mathscr{L}$-Mul Block Diagram

3.7 Systolic Array

Diagonal-Input and Permutated weight-stationary (DiP) The DiP systolic array (Abdelmaksoud, Agwa and Prodromakis 2024) is a novel architecture designed to optimize matrix multiplication by eliminating the need for input and output synchronization FIFOs, which are typically required in traditional weight-stationary systolic arrays. This architecture improves energy efficiency by leveraging a unique dataflow pattern.

The key innovation in DiP is the diagonal movement of inputs and the permutation of weights, which eliminates the need for FIFO buffers and reduces latency. The inputs move diagonally across the PE rows, transitioning from one row to the next, while the weights are permuted and loaded vertically into the PEs. This design allows for efficient data reuse and minimizes idle cycles, leading to higher throughput and energy efficiency.

Figure 3 illustrates the architecture of our DiP systolic array, showing how inputs move diagonally through the PEs and how weights are permuted and loaded, and 8 shows a practical example of how a matrix multiply is computed.

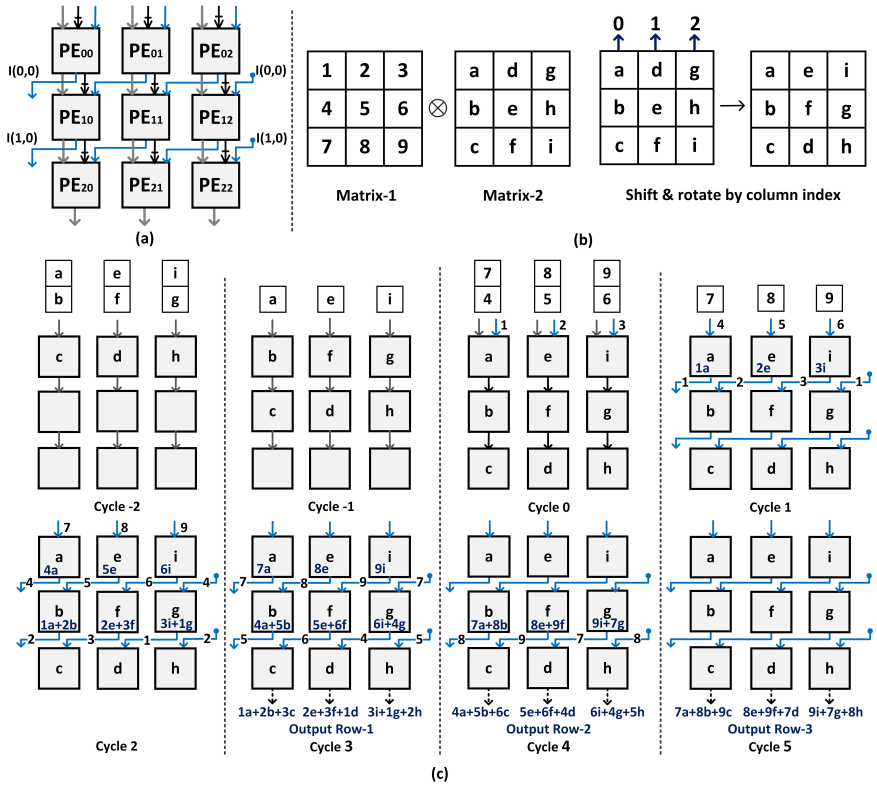

Figure 8: DiP example dataflow for matrix multiplication (Abdelmaksoud, Agwa and Prodromakis 2024)

3.8 Accumulators

The accumulator system is a critical component of the MAC unit, responsible for storing and summing partial results from the systolic array's processing elements. It is designed for high-throughput tiled matrix multiplication. The system employs parallel memory banks, one for each column of the systolic array, enabling simultaneous accumulation. (Abdelmaksoud, Agwa and Prodromakis 2024)

The accumulator operates in two modes: *accumulate mode*, where new results are added to the existing stored value, and *overwrite mode*, where new results replace the existing value. A dedicated address generator manages memory access, calculating addresses for tiled data access. This generator uses a base address ROM for efficient tile address calculation and finite state machines (FSMs) for sequential access within tiles. Independent read-and-write FSMs allow for overlapped read-and-write operations.

The accumulator's operation comprises several stages. First, the address generator computes the memory address. In accumulate mode, the existing value is read from memory. Then, either the new data is added to the existing data (accumulate mode) or the new data overwrites the existing data (overwrite mode) using a dedicated floating-point adder. Finally, the result is written back to the memory bank.

Key features include parallelism (via multiple memory banks), support for various floating-point formats (e.g., FP8, BF16, FP32), data reuse to minimize memory accesses, pipelined operation for high frequency, and scalability to accommodate larger matrices. Future work may explore hybrid accumulators using mixed-precision arithmetic and alternative dataflow patterns.

3.9 Activation Module

Our activation model was based on the ReLU activation function, and gives the same output as ReLU:

$$\text{val} <= 0 : 0$$

$$\text{val} > 0 : \text{val}$$

The module takes in the input value, and if the sign bit is 1 the module returns 0 otherwise it returns the value passed to it. Additionally, the module has an enable signal that is latched when the start signal is received and then stored in the enable_reg register. If the module is enabled it works as intended, otherwise it passes the input value out directly.

3.10 Control Logic and VLIW

We utilize a VLIW hardware approach. VLIW (very long instruction word) is a design methodology in contrast to typical superscalar design found in many modern processors

like CPUs and GPUs, where hardware consists of many independent functional units that are controlled independently of each other. In a typical CPU, instructions can be executed out of order and it is up to the hardware to efficiently manage resources and optimize for things like reducing cache misses and stalls.

In VLIW, multiple instructions for the different functional units are packed together into a single instruction "word" which moves control from the hardware to the compiler. This means the hardware is fully deterministic and it is up to the compiler to efficiently manage operations. We chose this approach because it greatly simplifies the hardware design, and reduces the amount of space the chip needs to dedicate to control logic. This approach has historically been found in specialized processors like DSPs and AI focused chips like the Google TPU and the Groq LPU. Since for this project we are focusing on working with a small subset of neural network architectures, this approach works very well since it allows us to fine-tune at the assembly level to squeeze the highest level of performance and utilization out of the design during simulation. The compiler will be responsible for simultaneously dispatching instructions to all the functional units including the systolic array, accumulators, memory controllers, and activation units. A single instruction carries information about the source and destination registers for each of these units as well as the operations to be carried out by each. Since some units complete operations in more or fewer cycles than other units, in many cases passing a NOP (no operation) instruction to some units is required while they wait for the results of another unit. This functionality is fully dependent on the compiler to correctly schedule the order of operations.

3.11 Composable Hardware Blocks

Designing in Python provides the flexibility of being able to combine both object-oriented and functional programming paradigms, enabling rapid and relatively easy creation of complex hardware. Each major functional unit to be mapped to hardware is encapsulated in its own class, with attributes defining the IO ports and standardized methods to allow blocks to easily connect with one another. Components or units with a very specific purpose that might not be large enough to justify using a class can be encapsulated with reusable functions. Hardware design differs greatly from software as code that's written does not only describe functional behavior but a physical structure. Calling a function twice that creates hardware will not pass the data through the hardware twice, it creates two separate instances of the hardware itself.

PyRTL also provides integration with open-source EDA tools enabling exporting designs built entirely in Python to Verilog. Language constructs like introspection allow for techniques such as automatic insertion of pipeline registers, and inferring the bitwidth of wires or depth of tree-based reducers based on type annotations. We employ a bottom-up methodology in our design by first creating the smallest components in the design (i.e., the logic to split a floating-point input into its component parts: sign, exponent, and mantissa) and combine them together in increasing levels of complexity to form circuits such as adders, MAC units, the entire systolic array, and a top-level accelerator design. The use of the aforementioned object-oriented classes also allows the creation of "empty" repeated ele-

ments where we can later connect their inputs and outputs. Since hardware just "exists" and does not do anything at runtime unless we simulate it, this allows defining a template or placeholder of modules, then defining their dataflows after instantiation, and finally connecting control logic.

3.12 Neural Network Architecture and Implementation

To validate our hardware design and substantiate the claims presented in the original $\mathscr{L}$-Mul paper, we implement and evaluate a simple multi-layer perceptron (MLP) on the MNIST dataset **?**. MNIST serves as a canonical benchmark in machine learning, comprising 60,000 labeled training examples and 10,000 test cases of handwritten digits. While achieving high accuracy on this dataset has become relatively straightforward with modern machine learning techniques, our implementation required careful consideration of the constraints imposed by hardware simulation.

Given that hardware simulations operate at speeds several orders of magnitude slower than native execution, we deliberately designed a compact network architecture with a single hidden layer of 128 neurons. This architectural decision balances computational feasibility with sufficient model capacity to demonstrate meaningful results. The network follows a standard feedforward structure:

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{1}$$

$$\mathbf{y} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 \tag{2}$$

where $\mathbf{x} \in \mathbb{R}^{784}$ represents the flattened input image, $\mathbf{h} \in \mathbb{R}^{128}$ is the hidden layer activation, and $\mathbf{y} \in \mathbb{R}^{10}$ produces the logits for classification. The weight matrices $\mathbf{W}_1 \in \mathbb{R}^{128\times784}$ and $\mathbf{W}_2 \in \mathbb{R}^{10\times128}$ along with bias vectors $\mathbf{b}_1 \in \mathbb{R}^{128}$ and $\mathbf{b}_2 \in \mathbb{R}^{10}$ constitute the learnable parameters of the network.

To comprehensively evaluate our hardware's performance across different precision regimes, we trained two variants of this network: one using standard 32-bit floating-point precision (FP32) and another using 16-bit brain floating-point format (BF16). Training directly in these native precisions circumvents the complexities associated with post-training quantization, which can introduce additional performance degradation and implementation challenges. It is worth noting that quantization research has advanced significantly in recent years, with sophisticated techniques now capable of preserving model performance even at reduced precision **??**.

Our approach embodies principles of hardware-software co-design. The hidden layer dimension of 128 was specifically selected to align with common systolic array implementations, which are typically designed with power-of-2 dimensions. For instance, NVIDIA's Tensor Cores in recent GPU architectures perform matrix multiply-accumulate operations on 16×16 matrices **?**. This dimensional alignment enables efficient tiling of weight matrices without computational overhead from zero-padded tiles.

We employ the Rectified Linear Unit (ReLU) activation function, defined as $\text{ReLU}(x) = \max(0, x)$, for the hidden layer. This choice is motivated by both its computational simplicity—requiring only a comparison and conditional assignment—and its empirical effectiveness in neural network training **?**. The ReLU function's non-saturating gradient characteristic facilitates more efficient backpropagation during training compared to traditional sigmoid or hyperbolic tangent activations.

The output layer employs no activation function during inference, as the raw logits are sufficient for determining the predicted class through an argmax operation. During training, however, these logits are passed through a softmax function and evaluated using cross-entropy loss to enable gradient-based optimization.

This deliberately simplified architecture serves as an ideal testbed for our hardware accelerator, allowing us to focus on the computational efficiency and numerical precision aspects of our design while maintaining sufficient model capacity to demonstrate meaningful acceleration on a standard machine learning task.

3.13 Virtual Instruction Set

Our accelerator architecture is specifically designed for inference rather than training. We utilize gate-level simulations to validate both the behavior of hardware components and the accuracy of the $\mathscr{L}$-Mul algorithm and module. The aforementioned VLIW approach gives us complete control over the hardware and shifts complexity from the execution model to the programming model.

Our simulations run on a per clock-cycle basis, but for tasks requiring numerous individual operations, we must abstract to a slightly higher level to balance fine-grained control with programming ease. To achieve this, we define a rudimentary VLIW instruction set that operates at the simulation level but could be implemented directly in hardware via an instruction decoder and control unit. The Python simulation API essentially functions as this control unit.

A single instruction containing multiple sub-instructions for each independent functional unit is dispatched simultaneously, coordinating both the operations performed on data and the flow of data between components. An instruction contains:

`data`: A matrix or vector whose column dimension matches the systolic array size.
`weights`: A matrix with the same dimensions as the systolic array.
`accum_addr`: Address in the accumulator memory to write the data vector to. If the data is a matrix, the address auto-increments via an internal address generator to avoid overwriting previous vectors.
`accum_mode`: Controls whether to accumulate with or overwrite the existing value at a given memory address.
`activation_enable`: Controls whether the activation unit should read the value at a given memory address in the accumulator or do nothing.
`activation_func`: Controls which activation function is applied to the data if the activation unit is enabled. Currently only supports `None` (passthrough) and `ReLU`.

`flush_pipeline`: Controls how many cycles should be executed after the last data vector is given as input to the top of the systolic array. This is useful when preparing the array to load new weights on the next instruction dispatch without interfering with partially computed previous results.

`NOP`: All units do nothing for one cycle.

For a systolic array of size $N \times N$ with pipeline depth p, an input matrix $\boldsymbol{A} \in \mathbb{R}^{M \times N}$, and weight matrix $\boldsymbol{W} \in \mathbb{R}^{N \times N}$, a single instruction can take between 1 and $2N + M + p - 1$ cycles. Note that the `flush_pipeline` instruction does not flush activations out after the instruction has been dispatched; it only ensures data reaches the output of the systolic array. Therefore, additional `NOP` or other instructions must be executed to collect the results.

3.14 Tiling Strategies

To achieve high computational utilization of the resources available within the systolic array, we must carefully consider how to partition matrices for efficient processing. For an $N \times N$ systolic array, optimal utilization requires input matrices (non-stationary) to be at least $N \times N$ in dimension. When operating with an accumulator of address width B, the input should maximize the dimension that can be shared with a stationary weight matrix (up to 2^B), thereby utilizing all available accumulator memory addresses while minimizing weight loading overhead. This optimization is effectively achieved through batched matrix operations.

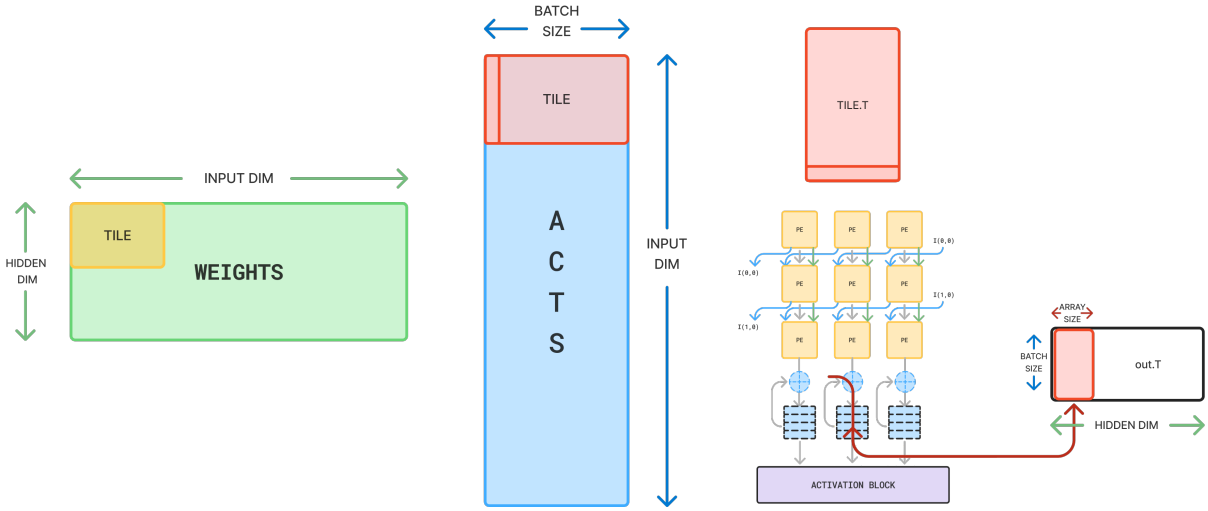

Figure 9: Batch GEMM tiling strategy showing how matrices are partitioned into tiles for efficient systolic array processing. The weight matrix is divided into tiles of size $N \times N$, while the input batch is processed in corresponding chunks to maximize computational throughput.

Our implementation employs a systematic tiling algorithm for batch matrix multiplication that optimizes the utilization of the systolic array. The algorithm partitions both the

weight matrix and batched inputs into appropriately sized tiles, ensuring efficient data flow through the hardware.

Algorithm 1: Batch GEMM tiling algorithm for systolic array processing

`BatchGemmTiling(`W`,` X`,` N`);`

Input : Weight matrix $W \in \mathbb{R}^{M \times K}$, batched input $X \in \mathbb{R}^{K \times B}$, systolic array size N
Output: Sequence of weight-input tile pairs for systolic array processing

$M_{pad} \leftarrow M + (N - M \bmod N) \bmod N$; // Padded output dimension
$K_{pad} \leftarrow K + (N - K \bmod N) \bmod N$; // Padded input dimension
$W_{pad} \leftarrow \text{Pad}(W, (0, M_{pad} - M), (0, K_{pad} - K))$; // Zero-pad weight matrix
$X_{pad} \leftarrow \text{Pad}(X, (0, K_{pad} - K), (0, 0))$; // Zero-pad input matrix
$n_M \leftarrow M_{pad}/N$; // Number of output dimension chunks
$n_K \leftarrow K_{pad}/N$; // Number of input dimension chunks

for $i \leftarrow 0$ **to** $n_M - 1$ **do**
  for $j \leftarrow 0$ **to** $n_K - 1$ **do**
    $W_{tile} \leftarrow W_{pad}[i \cdot N : (i+1) \cdot N, j \cdot N : (j+1) \cdot N]$; // Extract weight tile
    $X_{tile} \leftarrow X_{pad}[j \cdot N : (j+1) \cdot N, :]$; // Extract input tile
    isFirst $\leftarrow (j = 0)$; // First tile in row accumulation
    isLast $\leftarrow (j = n_K - 1)$; // Last tile in row accumulation
    yield $(W_{tile}, X_{tile}, \text{isFirst}, \text{isLast})$;
  end
end

The tiling algorithm (Algorithm 1) operates as follows:

1. First, we determine the necessary padding to ensure all dimensions are multiples of the systolic array size N.
2. The weight matrix W and input batch X are padded accordingly to dimensions $M_{pad} \times K_{pad}$ and $K_{pad} \times B$, respectively.
3. We then partition the padded matrices into tiles: the weight matrix into tiles of size $N \times N$ and the input batch into corresponding tiles of size $N \times B$.
4. For each weight tile, we process the corresponding input tile, tracking whether it's the first or last tile in a particular output row computation.
5. This information is crucial for the accelerator's control logic, as it determines whether to initialize (first=true) or accumulate (first=false) results, and whether to activate the output (last=true) after accumulation is complete.

During neural network inference, this tiling strategy is applied to both layers of the MLP. For the first layer, we process tiles of W_1 against batched inputs, accumulating results and applying ReLU activation when the last tile in a row is processed. For the second layer, we similarly process tiles of W_2 against the hidden layer outputs.

Without batching in the weight-stationary dataflow pattern, computation efficiency drops significantly—only approximately $\frac{1}{2}$ of cycles can be dedicated to computation for a pair of tiles, with merely $\frac{1}{N}$ rows of the array being active at once. For single-batch inference

that utilizes a GEMV (General Matrix-Vector multiplication) operation rather than GEMM, a large square systolic array with the DiP (Diagonal input Partitioning) dataflow is suboptimal, and alternative configurations should be considered for such workloads.

3.15 Simulation

Custom Data Types We implement our own versions of previously mentioned data types to easily convert data from human-readable decimal numbers stored as floats to the binary bits representing those values. As hardware can only understand integers, the custom dtypes are useful in simulation testing, type hinting, and runtime validation.

We also implement custom dunder methods that help us verify the correctness of operations implemented at the RTL level by emulating their behavior in software.

Unit Testing Each major hardware block worthy of its own class has a wrapper simulation class. The base classes do not explicitly define inputs and outputs, but leave that open-ended as previously described. The simulation class uses the methods defined to create strongly defined Input and Output wires which are needed by the simulation to interact with the hardware. We implement a somewhat standardized template for all simulation classes that reduces the originally large amount of boilerplate code required to set up a simulation and test a circuit. By abstracting away this boilerplate, it simplifies testing any changes to a component and being able to immediately see the results, for example we can run SystolicArraySimulator.matmul(A, B) which behaves just like A @ B, except the result is calculated entirely using a gate-level simulation. We validate these results against true results in an automated testing pipeline that prevents any code that does not give the same results as ground truth from being merged into the main codebase, ensuring all designs are verified and correct. The use of random values generated for tests also helps ensure adequate coverage. Tests are built with the pytest framework.

JIT Compilation to C and Caching As the size and complexity of the hardware grows, so does the time it takes to construct the hardware. A limitation of PyRTL is that while it does support modular design and reusability with functions and classes, this is not reflected under the hood. A ”block” of hardware is stored as a netlist of operations between individual wires. To put this in perspective, the internal representation of an 8x8 systolic array does not have any indication that it is made up of a grid of 8x8 processing elements, it is simply a huge logical operation 4x larger than a 4x4 array. This inefficiency becomes especially apparent when larger, complex structures are involved. The amount of operations needed to be computed by the simulation is sufficiently large that Python’s intrinsic limitations as an interpreted language begin to show.

PyRTL provides utilities to compile a block of hardware into significantly faster C code, which is accessed as a shared library. Unfortunately, every time a new simulation is created, the library is compiled from scratch again which can take several minutes. A compiled simulation also has additional limitations compared to the pure-Python version, such as lack of

ability to inspect internal wires, only explicit Inputs and Outputs. To overcome this challenge, we designed an interface that allows saving a compiled library for a given hardware block and reusing later. Compiled code is automatically identified by a configuration hash which allows testing different inputs and programming strategies quickly without having to reconstruct the internal netlist or recompile C. This was an important step not only for running big hardware for deeper evaluation, but to provide users of the demo a fast and enjoyable experience.

The standard simulation for an 8x8 systolic array top-level accelerator module requires little setup time, but can take over a minute to run a single inference of the simple MLP described above. The built-in compiled simulation requires between 10 seconds to several minutes of compilation time depending on the data types (therefore the internal bitwidths and complexity of components), but executes inference in around 5 seconds. With the custom caching, compiling a new design for the first time is the same, but a cold start only adds about 1 second of additional latency to load the library and logic net state if an existing configuration has already been saved. We include automated scripts to generate compiled libraries for all configurations of the accelerator in a single line of code.

3.16 Compiler Architecture

To validate the $\mathscr{L}$-Mul algorithm's performance, we'll be checking its performance on ONNX models. ONNX is a format for representing machine learning models. It uses an acyclic graph to represent neural networks where nodes take an input and have one or more outputs, and those outputs are edges connecting to other nodes.

When we load in the model, we traverse this graph to keep track of the required operations, layer dependencies (or, in other words, we build a topological sort), and data flow between layers. This information is later used to map the neural network to our hardware accelerator components. Notably, it has certain built-in functions such as type conversions that we used to help support our project.

Because we keep track of the required operations as we load in the neural net, once we have a complete set of the required operations, we map those to the hardware components. This is shown in our utility library.

Each hardware component we develop has a corresponding simulation handler class, which provides state tracking (of the weights, data, accumulators, and outputs), cycle-accurate simulation, and debug and inspection capabilities. To simulate, we create a simulator object and iterate through the steps. To verify and test, we can simply use this state dataclass to verify component behavior by validating we have the proper outputs.

3.16.1 EDA Tools and ASIC Hardening Flow

Moving beyond theoretical hardware designs presented several challenges. The ASIC design industry remains largely closed-source, with Professional Design Kits (PDKs), chip design libraries, and EDA tools locked behind licensing fees far beyond our budget for this

project. We were thus forced to rely on open-source tools. Recent open-source initiatives have recently emerged to democratize this process, enabling our hardware development workflows.

Yosys Yosys is an open-source synthesis tool. Specifically, it can convert Verilog (generated by PyRTL) into gate-level netlists that can be implemented on hardware. It also has built-in optimization and verification which we use in our synthesis process.

OpenROAD We also use the open-source software OpenROAD, a fully automated RTL to GDS flow, to take our design from RTL to a physical layout. OpenROAD uses Yosys as an intermediary step to synthesize the RTL (in combination with FreePDK45, an open-source process design kit). It then performs floor planning, placement, and routing to harden the designs. The flow generates reports and data along with these hardened designs, which we used to analyze the power, area, and delay performance between our implementations.

SiliconCompiler Some visualizations were generated with SiliconCompiler, which provides an easy to use Python API integrating both open and closed source EDA software and automated flows (like OpenROAD) to take a design from concept to production entirely in the comfort of a high level programming language. It includes built in collections of tools and libraries for hardware, including 7nm, 45nm, and 130nm PDKs.

3.17 Containers

The success of this project is thanks in-part to container based workflows. Because hardware design software is so often closed-source cross-platform development was not an option for FPGA workflows. This in part motivated our decision to pursue an ASIC-based design over FPGA, and thus it was critical that all members of the team had access to the same software and development environment. We utilize devcontainers as a way to standardize the environment for the team, ensuring consistency in package versioning, extensions, editor configuration, code formatting, testing, dependencies, and tool availability by defining a Dockerfile that automatically rebuilds the container and keeps everyone in sync. This was a big step forward for us as previously some members of the team were completely isolated in their work due to OS or other conflicts.

We also use containers to run the previously mentioned EDA tooling, run our interactive demo, and as a way for others to easily reproduce our results.

4 Results

4.1 Understanding $\mathscr{L}$-Mul Accuracy: Initial Interpretations with FP8

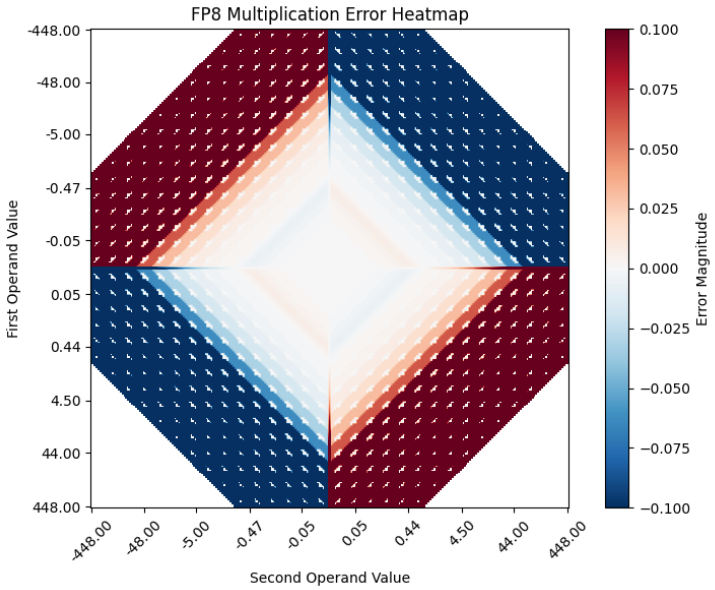

Figure 10: Multiplication Error

From Figure 10, we can see that the largest magnitude errors generally occur when multiplying a small and large number together. Generally, errors tend to be lower when multiplying smaller values, which is good since the application of this algorithm is for machine learning models whose weights tend to be in well-defined ranges like $[0, 1]$ or $[-1, 1]$. Compared to the basic $\mathscr{L}$-Mul algorithm:

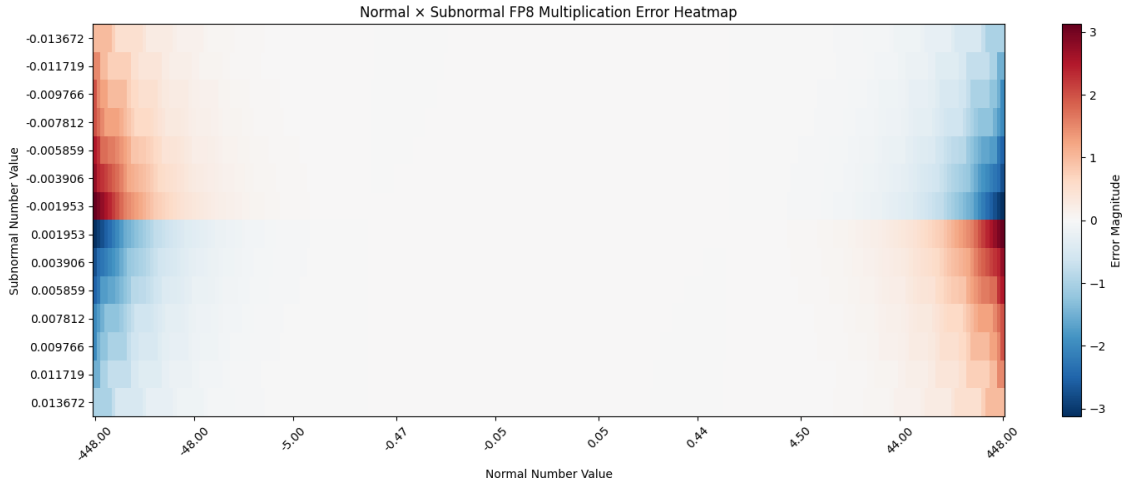

Figure 11: $\mathscr{L}$-Mul Error

And our modified algorithm to handle subnormals:

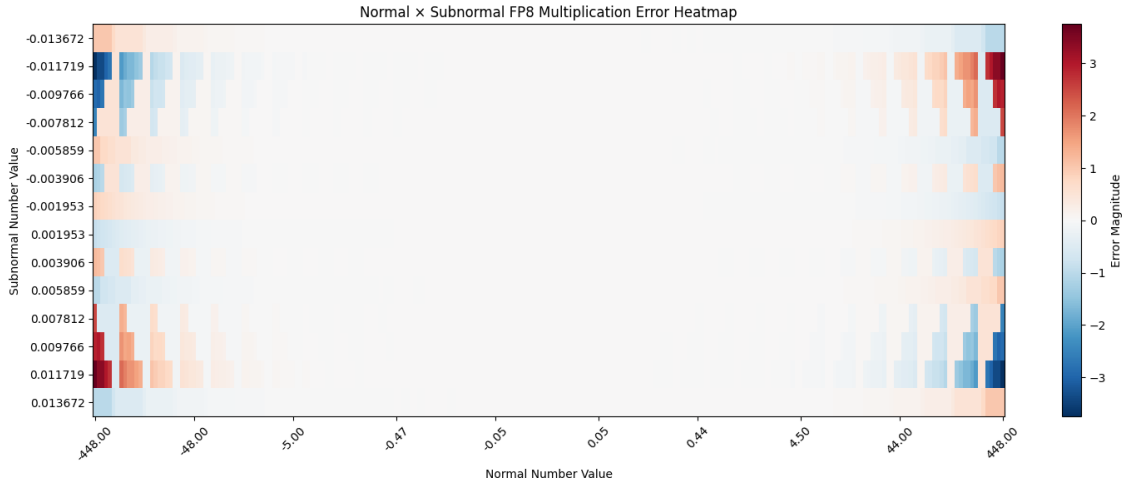

Figure 12: $\mathscr{L}$-Mul Error Optimized

4.2 Comparison Metrics

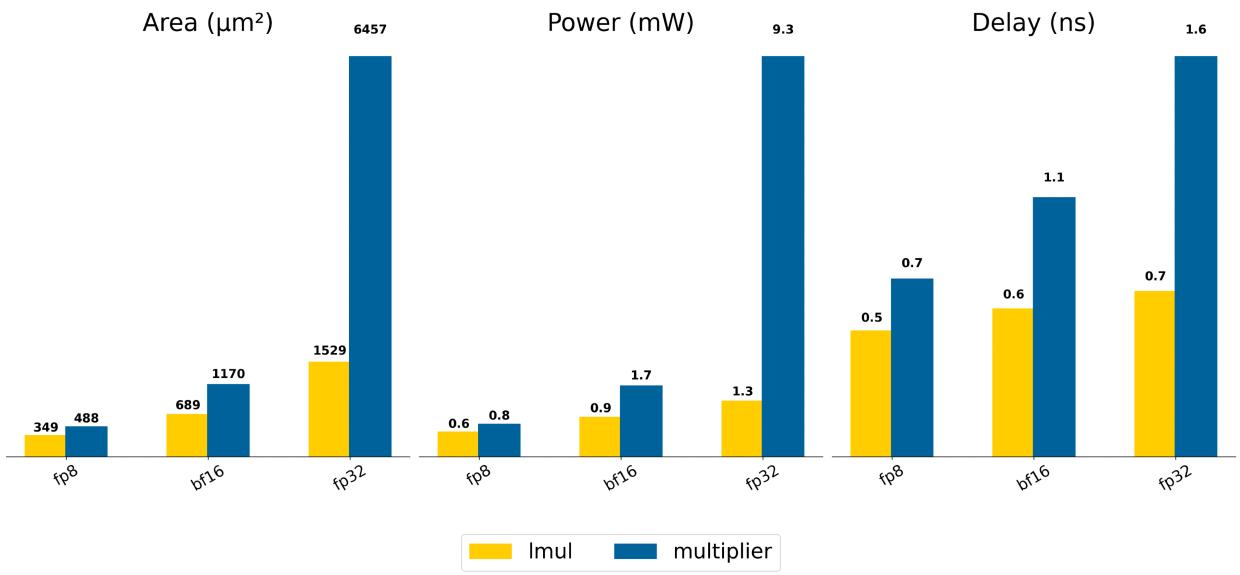

Figure 13: Area (μm^2), Power (mW), and Delay (ns) for Different Designs & Data Types (fp32, bf16, fp8)

Figure 13 shows the comparison between the IEEE standard floating-point multiplier and $\mathscr{L}$-Mul implementations across different data types.

Table 6 shows the model accuracy results for different configurations.

Table 5: Area, Power, and Delay Metrics for Different Designs

Design	fp8			bf16			fp32		
	Area	Power	Delay	Area	Power	Delay	Area	Power	Delay
L-mul Comb.	112.784	0.111	0.360	255.626	0.253	0.480	702.506	0.532	0.550
L-mul Pipelined	348.726	0.583	0.510	688.674	0.928	0.600	1529.230	1.300	0.670
Multiplier Comb.	347.396	1.055	1.290	1067.720	7.460	1.940	6311.910	133.398	2.850
Multiplier Pipelined	487.578	0.762	0.720	1169.600	1.654	1.050	6457.420	9.311	1.620
Multiplier Stage 2	162.260	0.161	0.550	552.482	1.184	0.910	4149.600	29.274	1.460
Multiplier Stage 3	71.820	0.027	0.230	134.064	0.048	0.290	319.466	0.080	0.420
Multiplier Stage 4	160.132	0.118	0.650	352.982	0.216	0.690	1253.660	0.553	1.070

Table 6: Accuracy Results for Different Model Configurations

Multiplier	Weight Type	Activation Type	Accuracy (%)
Baseline	Float32	Float32	97.81
Baseline	Float8	BF16	97.43
Baseline	BF16	BF16	97.46
L-mul	Float8	BF16	97.41
L-mul	BF16	BF16	97.42

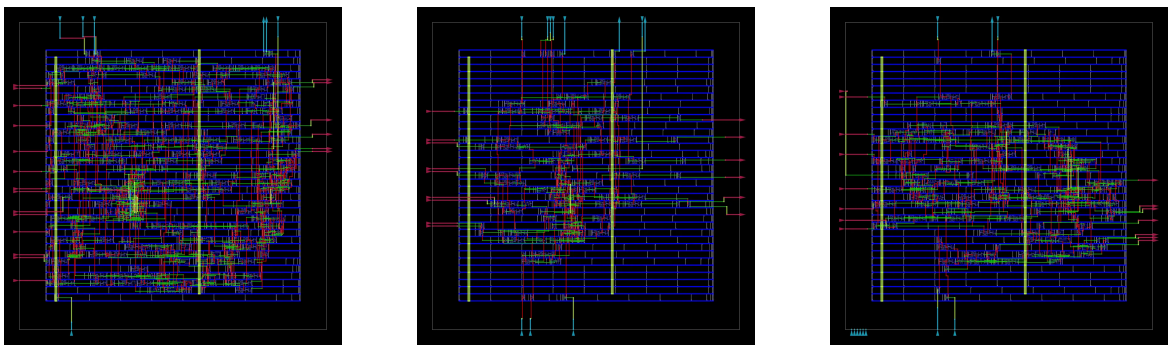

Figure 14: Left to right: adder, $\mathscr{L}$-Mul, multiplier pipelines

5 Discussion

Our evaluation of the $\mathscr{L}$-Mul algorithm relative to the standard IEEE floating-point multiplier is a classic speed/efficiency vs. precision trade-off. Specifically, the original $\mathscr{L}$-Mul paper claimed that the losses in accuracy were so small that the gains in speed and efficiency couldn't be ignored; our results aimed to test that hypothesis. To analyze speed and efficiency, we gathered three key metrics using OpenROAD: area, power, and delay. To analyze the loss in precision, we simply compared the accuracy of each multiplier on the test set of the MNIST data set using a neural network we trained.

5.1 Significance of Results

As shown in Table 5, the combinational $\mathscr{L}$-Mul implementation for FP8 achieves a 67.5% reduction in area (112.784 μm^2 vs. 347.396 μm^2), an 89.5% reduction in power consumption (0.111 mW vs. 1.055 mW), and a 72.% reduction in delay (0.36 ns vs. 1.29 ns) relative to the combinational IEEE implementation. These improvements show across all formats, with the FP32 $\mathscr{L}$-Mul implementation reaching an 88.9% reduction in area, 99.6% reduction in power, and 80.7% reduction in delay.

The pipelined implementations show similar trends, with the $\mathscr{L}$-Mul pipelined design consistently achieving better numbers in all key metrics across the tested dtypes. For example, for BF16, the $\mathscr{L}$-Mul pipelined implementation uses 41.1% less area, 43.9% less power, and achieves 42.9% lower delay than the standard pipelined multiplier.

In both the combinatorial and pipelined versions, we also observed that the percentage differences in all key metrics scaled up as the precision increased. For instance, moving from FP8 to BF16 to FP32, the area reduction for the combinational implementation increases from 67.5% to 76.1% to 88.9%, respectively. Similarly, power consumption reduction scales from 89.5% to 96.6% to 99.6%, and delay reduction from 72.1% to 75.3% to 80.7%. This makes sense from a logical perspective; the standard floating-point multiplication algorithm is of $O(n^2)$ complexity while the $\mathscr{L}$-Mul algorithm is O(n), where n is the number of bits in the floating-point representation. Just as n^2 increases faster than n, so too should the area, power, and delay for the IEEE algorithm relative to $\mathscr{L}$-Mul.

From an accuracy standpoint, Table 6 demonstrates that the $\mathscr{L}$-Mul algorithm maintains competitive performance on our image classification task. Using BF16 for both weights and activations, the $\mathscr{L}$-Mul implementation achieves 97.37% accuracy on MNIST, only 0.09 percentage points lower than the baseline with the same data types (97.46%). When using FP8 weights with BF16 activations, the $\mathscr{L}$-Mul shows a larger accuracy gap of 0.54 percentage points (96.89% vs. 97.43%), which is a sizable but still reasonable gap for most applications. Though we were not surprised that reducing the number of bits in the floating-point representation resulted in lower accuracy, we did find it noteworthy that differences in accuracy slightly increased as the precision decreased.

The results of our evaluation validate the $\mathscr{L}$-Mul paper's hypothesis: the multiplication approximation had significant advantages in our three key efficiency/speed metrics at a small

loss in accuracy. In other words, the hardware benefits are substantial enough that the minor accuracy trade-offs become acceptable for certain real-world applications, in environments from edge devices to large-scale data centers where power efficiency is paramount. The consistent performance improvements across different data types additionally demonstrate its versatility $\mathscr{L}$-Mul approach, making it applicable to a wider range of machine learning workloads.

5.2 Future Work

Our analysis of individual pipeline stages (shown in Table 5) provides insights into where computational bottlenecks occur in floating-point multiplication. Though this was more secondary to our paper, these numbers can serve as a diagnosis for the targets of further optimization in the future. In particular, the size, power, and delay of stage 2 as the precision increases accounts for an increasing portion of its area, power, and delay, indicating it could be a stage that is looked at in the future.

We also considered heterogeneous processing elements, where both $\mathscr{L}$-Mul and IEEE multipliers could exist in the same architecture. For example, more critical parts of the network requiring higher precision could use the standard, more accurate multiplier, while the less sensitive parts could benefit from the increased speed and efficiency of our $\mathscr{L}$-Mul unit.

The most obvious next step we'd take, however, is an implementation of other functions that could increase the range of applications for our hardware. We trained an MLP on the MNIST dataset as a proof-of-concept; if we were to take this project further, we'd begin by building popular architectures such as the transformer. For example, building out a hardware-optimized Swish (activation) function would be one of the first ideas we look at. We would then test on a model like BERT or a smaller LLM to validate the $\mathscr{L}$-Mul algorithm, particularly in generative use cases.

On a similar note, while our ASIC simulation is useful as a proof-of-concept, implementing the design on FPGA would offer stronger validation of our performance characteristics, and would get us one step closer to direct measurement of inference latency on a realistic workload.

6 Conclusion

Our project presented a means of accelerating machine learning, using an MLP simulation of our hardware implementation of the linear-complexity multiplication ($\mathscr{L}$-Mul) algorithm as a proof-of-concept for further work. In other words, we validated the speed/efficiency and precision tradeoff presented in Hongyin Luo and Wei Sun's *Addition is All You Need* paper.

The $\mathscr{L}$-Mul hardware units we developed outperformed standard IEEE-754 multipliers across all data types in our key speed/efficiency metrics. Our smallest implementation on FP8 numbers reduced silicon footprint by more than two-thirds, consumed nearly 90% less

power, and processed computations in under a third of the time compared to the conventional approach. These benefits became even more pronounced at higher precisions, with our FP32 implementation showing nearly order-of-magnitude improvements in resource utilization and energy efficiency.

The systolic array architecture we designed uses these optimized multipliers within a comprehensive accelerator featuring configurable processing elements, accumulation buffers, and activation units. Our VLIW-based control strategy eliminated the need for complex scheduling, instead exploiting instruction-level parallelism to optimize our work. Our testing of the MLP on the MNIST classification dataset demonstrated a minimal loss in accuracy in fractions of percentage points when comparing the $\mathscr{L}$-Mul algorithm to the IEEE version.

While most of research in hardware acceleration today focuses on optimizing memory bandwidth, data transfer, and attention mechanisms, our findings confirm that targeting core mathematical operations can also yield substantial efficiency benefits at a low cost of precision. Indeed, the $\mathscr{L}$-Mul approach effectively linearized what is traditionally a quadratic scaling problem, with the advantage gap widening as computational precision increases.

Moving forward, we've outlined several possibilities: building on our architecture to support more complex (and more widely used) networks such as transformers, implementing hybrid approaches that selectively apply $\mathscr{L}$-Mul where precision requirements allow, and physical implementation on FPGA platforms to validate real-world performance.

In short, our $\mathscr{L}$-Mul-based accelerator demonstrates the potential of algorithmic innovation as another avenue to be explored for energy efficiency. We've demonstrated its energy savings and speedups are strong enough to be worth the accuracy trade-off in certain scenarios — on a larger, we hope a fundamental reconsideration of computing primitives can offer another path towards hardware acceleration.

7 Contributions

Kai Breese wrote the bulk of the hardware code and made the demo.

Justin Chou developed the project site, built utility functions for matrix operation in the main package, conducted the analysis (power, area, delay) through OpenROAD, and created the development environment and reproducible docker containers.

Katelyn Abille designed the poster and implemented hardware compatibility for two data types FP16 and FP32.

Lukas Fullner provided support for various parts of the project but was absent for part of the quarter.

References

Abdelmaksoud, Ahmed J, Shady Agwa, and Themis Prodromakis. 2024. "DiP: A Scalable, Energy-Efficient Systolic Array for Matrix Multiplication Acceleration." *arXiv preprint arXiv:2412.09709*

Chen, Ruiqi, Yangxintong Lyu, Han Bao, and Bruno da Silva. 2024. "A Power-Efficient Hardware Implementation of L-Mul." *arXiv preprint arXiv:2412.18948*

Clow, John, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. "A pythonic approach for rapid hardware prototyping and instrumentation." In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE

Fujii, Kazuki, Taishi Nakamura, and Rio Yokota. 2024. "Balancing Speed and Stability: The Trade-offs of FP8 vs. BF16 Training in LLMs." *arXiv preprint arXiv:2411.08719*

Jouppi, Norman P, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers et al. 2017. "In-datacenter performance analysis of a tensor processing unit." In *Proceedings of the 44th annual international symposium on computer architecture*.

Kung, Hsiang Tsung, and Charles E Leiserson. 1979. "Systolic arrays (for VLSI)." In *Sparse Matrix Proceedings 1978*. Society for industrial and applied mathematics Philadelphia, PA, USA

Luo, Hongyin, and Wei Sun. 2024. "Addition is All You Need for Energy-efficient Language Models." *arXiv preprint arXiv:2410.00907*

Micikevicius, Paulius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu et al. 2022. "Fp8 formats for deep learning." *arXiv preprint arXiv:2209.05433*

Mirza, Diba, Deeksha Dangwal, and Timothy Sherwood. 2019. "Pyrtl in early undergraduate research." In *Proceedings of the Workshop on Computer Architecture Education*.

Niknia, Farzad, Ziheng Wang, Shanshan Liu, Pedro Reviriego, Ahmed Louri, and Fabrizio Lombardi. 2024. "ASIC Design of Nanoscale Artificial Neural Networks for Inference/Training by Floating-Point Arithmetic." *IEEE Transactions on Nanotechnology*

Noune, Badreddine, Philip Jones, Daniel Justus, Dominic Masters, and Carlo Luschi. 2022. "8-bit numerical formats for deep neural networks." *arXiv preprint arXiv:2206.02915*

Patterson, David, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. 2022. "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink." *Computer* 55 (7): 18–28. [Link]

Raja, Tejas. 2024. "Systolic Array Data Flows for Efficient Matrix Multiplication in Deep Neural Networks." *arXiv preprint arXiv:2410.22595*

ResearchGate. 2024. "5×5 Systolic array architecture." [Link]

de Vries, Alex. 2023. "The growing energy footprint of artificial intelligence." *Joule* 7 (10):

2191–2194

Zhou, Yanqi, Xuanyi Dong, Berkin Akin, Mingxing Tan, Daiyi Peng, Tianjian Meng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. 2021. "Rethinking co-design of neural architectures and hardware accelerators." *arXiv preprint arXiv:2102.08619*

Appendices

A.1 Project Proposal

Efficient floating-point operations are a significant challenge in large neural networks and other computationally intensive machine learning algorithms, where energy consumption and latency are key constraints. In this report, we present an implementation of the linear-complexity multiplication ($\mathscr{L}$-Mul) algorithm designed to approximate floating-point multiplication using addition (Luo and Sun 2024). By leveraging this approximation, $\mathscr{L}$-Mul achieves high precision with significantly lower computational cost than traditional floating-point multiplication methods. Our goal with this project is to develop a working simulation of a processor which can run machine learning models such as a multilayer perception or a transformer. The core of this processor will be a matrix multiplication module using the $\mathscr{L}$-Mul algorithm in order to achieve faster and more efficient processing of machine learning models.

In quarter 1 we utilized various implementation methods in order to get a working version of the $\mathscr{L}$-Mul algorithm. Utilizing PyRTL and Vivado, we first implemented the $\mathscr{L}$-Mul algorithm, and then developed a systolic array utilizing the $\mathscr{L}$-Mul module. Through this process we studied various data formats as the key to the $\mathscr{L}$-Mul algorithm is exploiting a property of floating-point numbers. We started with keeping our data in an eight-bit float as this was the data format that the algorithm was shown to work on, but we also had to consider that most machine learning models are not run on fp8, and so the process of converting the models to fp8 would add additional overhead that we would like to avoid. This lead us to the bf16 format, which was easier to convert to and from while still being usable in our algorithm. All of this research into the $\mathscr{L}$-Mul algorithm lead us to the central problem that we are trying to solve, that the greatest slowdown of processing a machine learning model is the multiplication step, so by accelerating the multiplication we would gain a noticeable speedup in model performance. Our quarter 1 project allowed us to perform a deep dive into various ways of writing Verilog code, using PyRTL and writing Verilog directly. We learned much about both the syntax of design as well as the logic that goes into designing a circuit or module. This led to us developing our workflow, and determining how we can most efficiently create code.

This leads us to our goal for quarter 2, to develop and simulate an accelerator for machine learning models. We will be using PyRTL to construct our hardware, which will take the form of a processor with its own instruction set and series of compute modules. By basing the multiply off of $\mathscr{L}$-Mul, we will be able to run actual machine learning models (A

pytorch or ONNX model) on our processor, and will be able to benchmark performance to see if the $\mathscr{L}$-Mul based multiply is practical. We will also develop a comprehensive testing suite for our processor, both to display the performance of the processor but also to allow us to optimize various hyperparameters of the design by using the performance data we generate. In essence we will perform data science on our model, generating data about its performance so we can further optimize the performance.