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
Mentor: Rajesh Gupta rgupta@ucsd.edu

UC San Diego™
HALICIOĞLU DATA SCIENCE INSTITUTE

Motivation

Modern AI systems require increasingly large and unsustainable amounts of energy.

- ChatGPT required **564 MWh per day** (Feb 2023), with two days nearly amounting to the total 1,287 MWh used throughout the training phase. [3]
- Google reports **60% of ML energy use** goes to inference. [5]

Linear-Complexity Multiplication Algorithm ($\mathcal{L}$-Mul)

Floating-Point Numbers

BF16 is widely adopted in ML for its ability to retain the same dynamic range as FP32 while reducing memory usage and computational requirements.

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. bfloat16 sign, exponent, and mantissa breakdown

Multiplication

To multiply two floats together, the standard is to use the following equation:

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

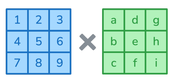
The output sign is determined by `xor` of the sign bits ($x_s \oplus y_s$)
In comparison, $\mathcal{L}$-Mul eliminates the multiplication of $m_x \times m_y$ by approximating it with a term $L(M)$ where M is the number of mantissa bits:

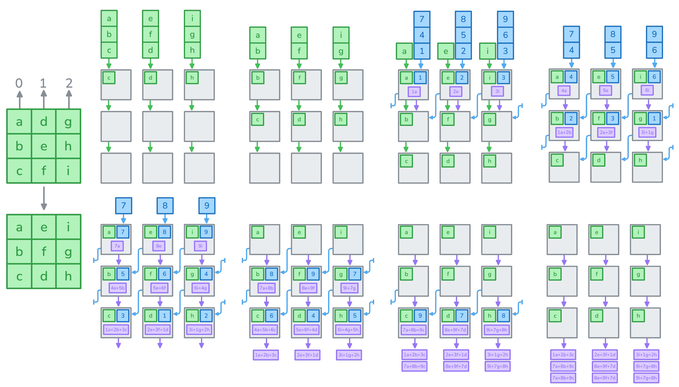
$$\mathcal{L}\text{-Mul} = (1+m_x+m_y+2^{-\mathbf{L(M)}})\cdot 2^{e_x+e_y}, \quad L(M)=\begin{cases} M, & M\le 3 \\ 3, & M=4 \\ 4, & M>4 \end{cases}$$

In hardware, we can execute $\mathcal{L}$-Mul with a single adder by adding the combined exponent and mantissa bits with the $L(M)$ term adjusted for bias. and performing a bitwise addition on the combined exponent and mantissa parts, adding the $L(M)$ term, and subtracting the bias bit shifted to the left to align with the exponent part. This enables us to skip the normalization step since the mantissa carry automatically adds to the exponent.

Systolic Arrays

Matrix multiplication is a fundamental operation in machine learning demanding high computation for large models. Systolic arrays are a spatial computing architecture that can be used to compute GEMMs and are composed of independent processing elements that coordinate data flow while minimizing global memory access and maximizing data reuse.





The diagram shows how GEMM is calculated using a systolic array with a diagonal input and permutated weight stationary dataflow.

- First weights are permutated by shifting columns up by their index value
- Weights are loaded one row per cycle, and the first row of data is input at the same time as the last weights to save a cycle
- Inputs flow down and diagonally, partial sums flow down
- Output rows are available synchronously after $2N+S-2$ cycles where N is the size of the array and S is the number of pipeline stages

System Architecture

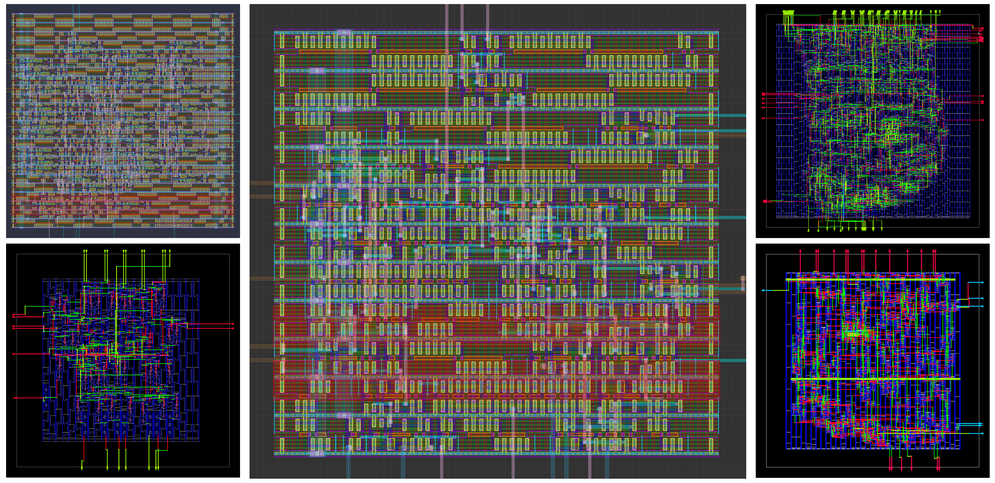


Figure 2. Physical designs of $\mathcal{L}$-Mul unit (top left, center) and routing (bottom left) for processing element (top right) and adder (bottom right)

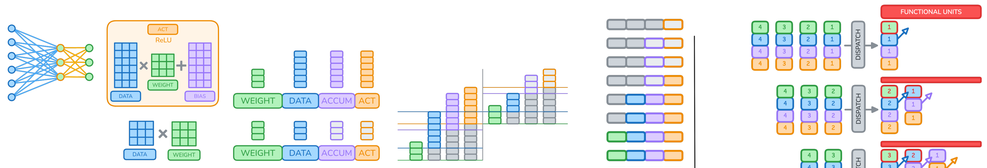


Figure 3. ML Compiler Flow

Machine learning models need to be compiled to instructions custom hardware can understand. We employ a VLIW approach.

- Parse and sort operations from model's computational graph
- Use algorithms like matrix tiling to construct low level operations
- Scheduling is managed ahead of time by the compiler

Execution Model: Instructions are sent to independent functional units in numerical order. Red boxes indicate currently executing instructions.

The Processing Element is the core computational unit in the systolic array. It features weight and data registers, an accumulation register, and optionally a product register to support additional pipelining. The accumulation is calculated as the sum of the input accum. value from the north PE and the product. It is designed to support mixed-precision operations (fp8, bf16, fp32). Control signals manage the flow of data by enabling reads and writes to the internal registers.

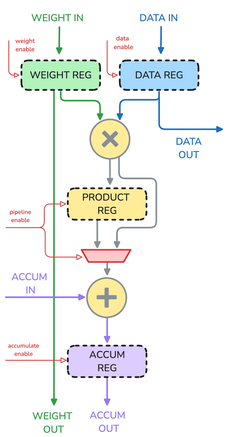


Figure 4. PE

Figure 5 shows a system level architecture of a single matrix acceleration unit based on a DiP systolic array. Weights and data are streamed directly without the need for synchronization FIFOs. Inputs move diagonally through the array, while weights are loaded vertically into PEs. Each input arrives with instructions that are propagated alongside the array, signaling the write mode (overwrite or accumulate) and destination address in the accumulator unit, and a signal for the activation function unit.

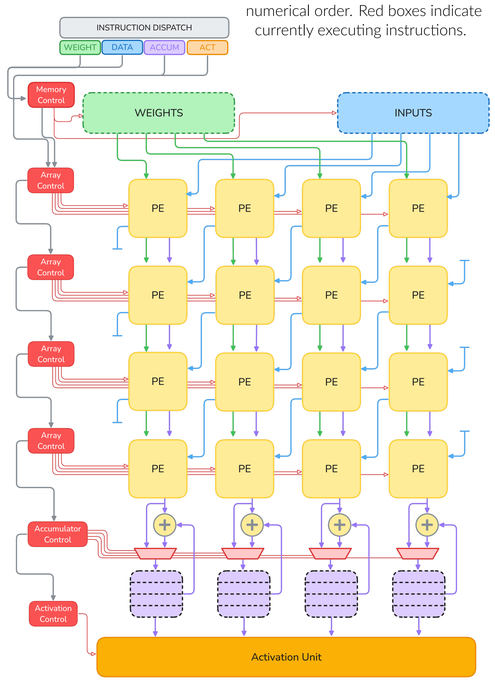
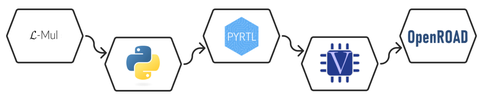


Figure 5. Systolic Array DiP Architecture [1]

Open-Source Roadmap: From Software to Silicon



Our project took a novel mathematical algorithm, converted it to code, constructed it in register-transfer level design, then synthesized it into a real hardware chip design:

- $\mathcal{L}$-Mul for approximating floating-point multiplication using addition operations.
- Python for rapid prototyping and validation of our core algorithms.
- PyRTL for hardware description in Python.
- Verilog for industry-standard hardware description converted from PyRTL.
- OpenROAD for ASIC implementation showing power, area, and timing benefits.

Results

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

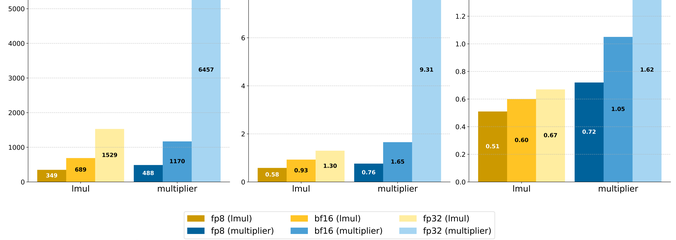


Figure 6. Area (μm^2), Power (**mW**), and Delay (**ns**) for different designs & data types (fp32, bf16, fp8)

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

MNIST accuracy by MLP (single hidden layer $d_h = 128$) running on simulated hardware

Conclusion and Website

When comparing the $\mathcal{L}$-Mul algorithm to the standard IEEE FP multiplication algorithm, we saw significant jumps in all key metrics: area, power, and delay. Specifically, we saw increasing percentage improvements as we scaled up in data types (Figure 6). This makes sense when considering the time complexity of each algorithm – $\mathcal{L}$-Mul is O(n) while IEEE is O(n^2), meaning the difference in operations should scale up as bits increase.

The loss in accuracy (Table 1) from IEEE to $\mathcal{L}$-Mul (0.092% for BF16) was extremely small compared to other key metrics (52% in area, 56% in power, 55% in delay for BF16). This is a strong indication that the hypothesis we were testing – that the speed/efficiency for accuracy tradeoff we were making would be worth it – indeed rang true.

Future Projections & Feasibility

- We see practical performance of $\mathcal{L}$-Mul on a small scale, but we would like to evaluate its performance on larger and more complex models such as LLMs.
- As the models expand, we would be able to evaluate trends in memory usage and compute requirements.
- We would also to scale up our power savings estimates to the size of a data center to see practical cost and power savings.
- Beyond the theoretical, we could tapeout our designs from a theoretical simulation to a physical chip.

github.com/ninjakaib/hardware-accelerators nakschou.github.io/hardware-accelerators-site/