The University of Texas at Austin
Chandra Department of Electrical and Computer Engineering
Cockrell School of Engineering

This design explores a modified systolic array design, where rather than having inputs propagate through the computation cells the inputs are sent directly to each computation tile from memory. This was modeled entirely within behavioral Verilog, and then synthesized into a physical design using Design Vision. We tested this design by compiling a simple program that trains a neural network to 99% accuracy, and then running it on our architecture. After verifying the correctness of the design, it was then compared to existing architectures based on the maximum delay derived from the physical design.

Minimal Machine Learning Processor

VLSI 1 Course Project

Ian SymSmith, Michael Ebenstein

Table of Contents

Introduction

Decreasing transistor sizes have led modern processing units to have access to more area, and one use of this area is to implement architecture for specific operations. At the same time, one increasingly common task is machine learning, which does a large amount of parallel processing on matrices and vectors. An architecture designed to efficiently handle these parallel operations is systolic arrays. Systolic array architecture features a matrix of computation tiles connected to a config cell that interfaces with instructions and memory. Computation tiles receive inputs on the perimeter of the matrix, operate in parallel based on a single opcode, and pass information to adjacent tiles. The control tile chooses the opcode to send to the tiles based on what operation it is asked to perform, while also choosing what data from memory to transfer to the tiles.

You can find the source code for this project in this repository.

Motivation

The primary motivation for this project is that certain alternatives to systolic array architectures are underexplored, as well as not having comprehensive benchmarking. This is important because machine learning is an increasingly common tool in programming applications, and hardware can limit what machine learning programs can accomplish. Systolic arrays are one example of a way to build architecture for machine learning algorithms, due to their ability to work on large amounts of data in parallel and their ability to be scaled up or down for different use cases. Exploring different variations of systolic array designs can lead to new designs that perform better in specific applications.

Another motivation for this project is the change in hardware making specific hardware components for machine learning more practical. As modern chips have access to more transistors and things like power usage and performance become more important, a chip can have many accelerators for specific applications that are activated when the processor needs them. One example of an accelerator could be a systolic array architecture specifically for matrix operations and convolutions. Overall, the improvements in hardware have made the project more practical, while the growing frequency of machine learning applications makes it more important.

Project Scope

This project seeks to design an alternative input method to systolic arrays, where rather than inputting values on the perimeter and propagating the values through the array, the config cell will directly send information from memory to each individual computation tile. The design for this architecture will first be done in behavioral Verilog, tested, and then synthesized into a layout. This synthesized design will be used for benchmarking information to compare it to other systolic array designs, as well as CPUs and GPUs.

The design constraints for this project will include limited libraries and limited number of people working on the project. The design library the project uses will be the 45nm library provided in the labs, which means that is what the design will be synthesized with. Additionally, the limit on time and people means that certain features are not possible, such as a complete compiler and full pipeline. Overall, the scope of this project will be to design the modified systolic array architecture and compare it to other designs to determine whether it is a viable design.

Previous Work and Gap in Literature

Systolic array designs have been studied and iterated upon numerous times over the course of their history. Recently however, they have been receiving more attention after their use in modern designs, such as the Google TPU. This has led to many studies being done on what performance improvements the standard systolic array design gives, as well as how variations on the systolic array perform. Generally the tests the studies use to benchmark their designs are for machine learning processes, which line up with what the systolic array variation in this project will be tested with.

Systolic Array Design

Traditional systolic array processors feature a matrix of computation tiles that all perform the same function simultaneously, with inputs given at the perimeter, and data passed between the tiles. Normal systolic array architectures come in three forms, output stationary systolic array processors, input stationary systolic array processors, and weight stationary systolic array processors [1]. Output stationary systolic array processors propagate both inputs every cycle, and calculate the output inside a single computation tile until the work is complete. Input and weight stationary arrays load their respective stationary input before the operation, and then propagate the other input and the output. All three of these designs can be seen in Figure 1. All of these have their advantages and disadvantages with respect to matrices bigger or smaller than the size of the

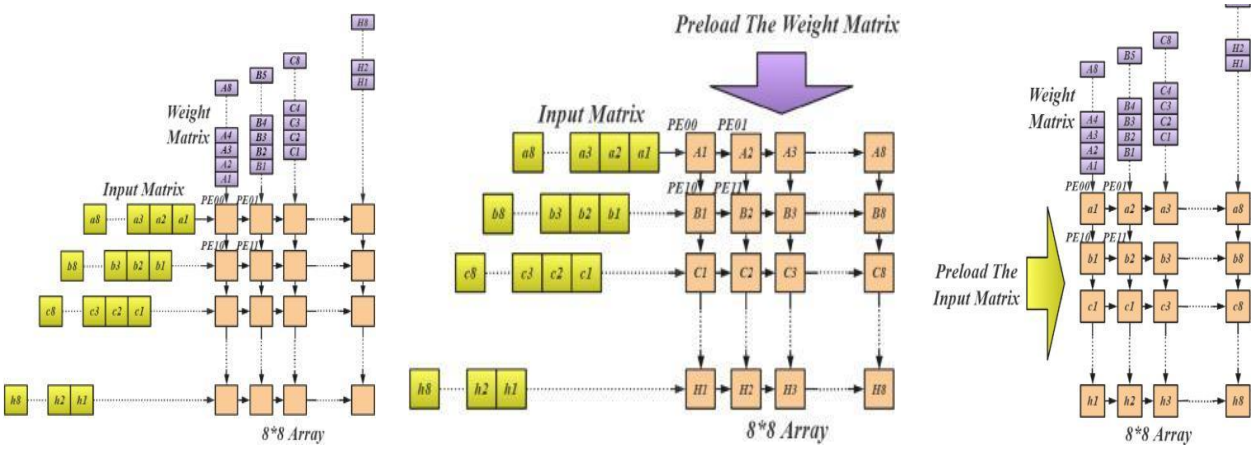


Figure 1: From Left to Right; Output Stationary, Weight Stationary, and Input Stationary Systolic Arrays

computation matrix.

In order to feed information into the computation tiles, standard systolic arrays have specific modules to receive information from memory, called data feeders, and send the output back to memory, called data collectors [2]. While there are variations on how these are designed such as using FIFOs or caches, the goal remains consistent, to input data from memory into the systolic array's perimeter when beginning an operation. Reading data from memory and storing data back into memory is controlled by a separate config cell, which determines what addresses to read from memory. Additionally, this cell will control what operation the computation tiles perform, and handle dependencies [2].

Systolic Array Variations

In addition to the normal systolic array design, much work has been done with adding new components to the design or changing its functionality. One variation is Neuflow, which is a design specialized in machine

learning networks that work on image processing. A small subsection of this design is pictured in Figure 2. One major change that Neuflow implemented is that the processor configures each tile with specific operations at runtime, allowing it to achieve higher performance. This also allows for tiles to have more complex functions that are specialized for the task the processor is doing. Another change it makes is that while in traditional systolic arrays each computation tile performs the same operation, not all tiles in Neuflow perform the same function [3]. All of these additional features allow the chip to be more power efficient, while also being extremely optimized for dense image processing [4].

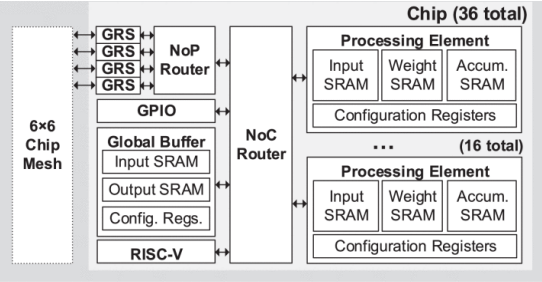


Figure 3: MCM Architecture Diagram

Another variation is a multi-chip-module, where multiple systolic arrays are connected together to either work on single large matrices, or many small matrices independently. The systolic arrays in multi-chip-modules are smaller, and only activated when needed for larger matrices. This makes them more energy efficient when dealing with smaller matrices. Additionally, it also makes them more practical for manufacturing and cheaper, since smaller chips are easier to mass produce. These benefits do come with penalties to performance, area, and power when working with larger matrices however [5].

Gap in Literature

While these variations on systolic array design do present significant improvements, there are still variations that can be tested. One thing that all these variations have in common is the propagation of data between the cells of the matrix. It would be possible to connect each cell directly to memory, and directly read memory into the cells without the need of propagation. The lack of interconnects between the tiles also allows for more flexible floor planning, since the tiles are unrelated, which can could lead to more compact layouts and reduced wire delays. Additionally, many of these designs were made to be implemented on FPGAs. This unexplored design may have potential in specific applications, and currently there is not a clear source of benchmarking information on it.

Proposed Work

This paper proposes a design similar to an output stationary systolic array, but with data sent directly from memory to each computation tile, rather than inputted on the perimeter and propagated through. This design will include a config cell to control computation tiles and a memory module to simulate off-chip memory.

Top Level Design Description

The overall design of the system consists of three core parts, a matrix of computation tiles, a config cell, and memory. As shown in Figure 5 below, computation tiles are laid out in a matrix, with two 16 bit input busses connecting each cell to memory and one 16 bit output bus. These cells perform basic operations on data passed to them by memory, and the load that information back into memory when signaled to by the config cell. Computation tiles do not receive addresses to memory, rather they send and receive information on a bus, while the address they are sending it to is sent from the config cell to memory simultaneously. Which operation they perform is given to them by a 4 bit opcode from the config cell. Both the addresses the config

cell sends to memory and what 4 bit opcode the config cell sends to the computation tiles is determined from compiled instructions given as inputs to the config cell.

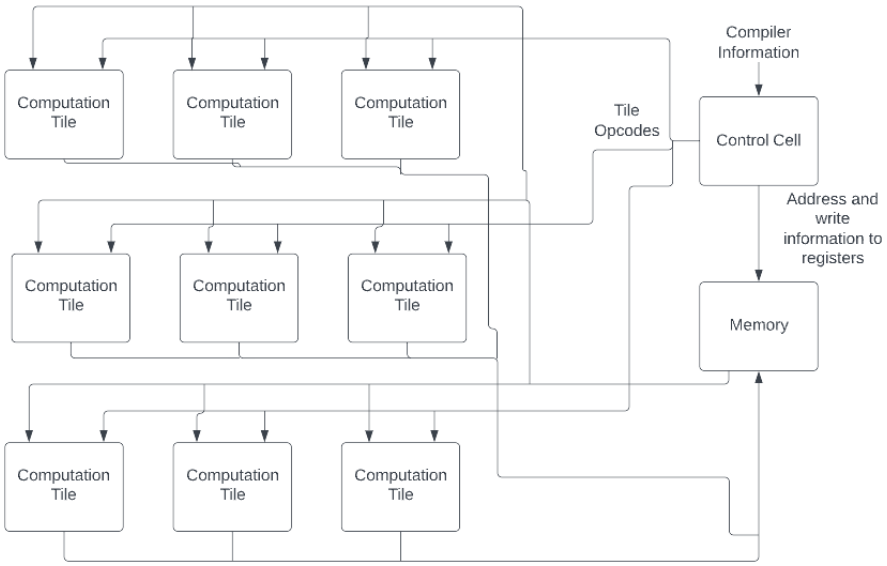
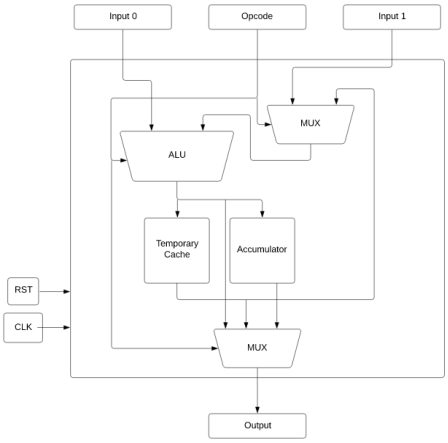


Figure 5: Top Level Diagram

Computation Tile Design

Computation tiles' primary task is to perform simple operations on data sent to them by memory, and send that information back to memory once the operation is complete. Computation tiles have 5 inputs, two 16 bit inputs, a 4 bit opcode, a reset signal, and the system clock. The only output is a 16 bit output that connects to a bus that sends information to memory. Inside the cell is a simple ALU, a 16 bit register to store temporary results, and an accumulator. The connections between these components and the list of opcodes are shown in Figure 6. The register is used if the cell needs to use its current result as an input to the next instruction or if the output from the current instruction needs to be held for additional cycles. The accumulator has another register so it can accumulate multiple cycles in a row.



Opcode	Operation	Opcode	Operation
0000	NOT	1000	READ
0001	AND	1001	XXX
0010	OR	1010	XXX
0011	ADD	1011	XXX
0100	LESS THAN	1100	MIN
0101	GREATER THAN	1101	MAX
0110	EQUAL TO	1110	XXX
0111	MULTIPLY	1111	MULTIPLY-ACCUMULATE

Figure 6: Computation Tile Diagram and Opcode List

The computation cell's ALU is kept simple for two reasons, to minimize the area of the cell so many can be laid out in a matrix, and because convolutions and matrix multiplication do not require many functions. The ALU can add, multiply, multiply-accumulate, compare two values, and perform bitwise logic operations on values. There is also an instruction to signify that the operation is complete. The first cycle after reset or it gets the read operand, the tile uses the two inputs to calculate the output. It then uses the output received as one of the inputs until the cell receives the read instruction. If the instruction is a multiply-accumulate, then it will use the two memory inputs regardless.

Memory Design

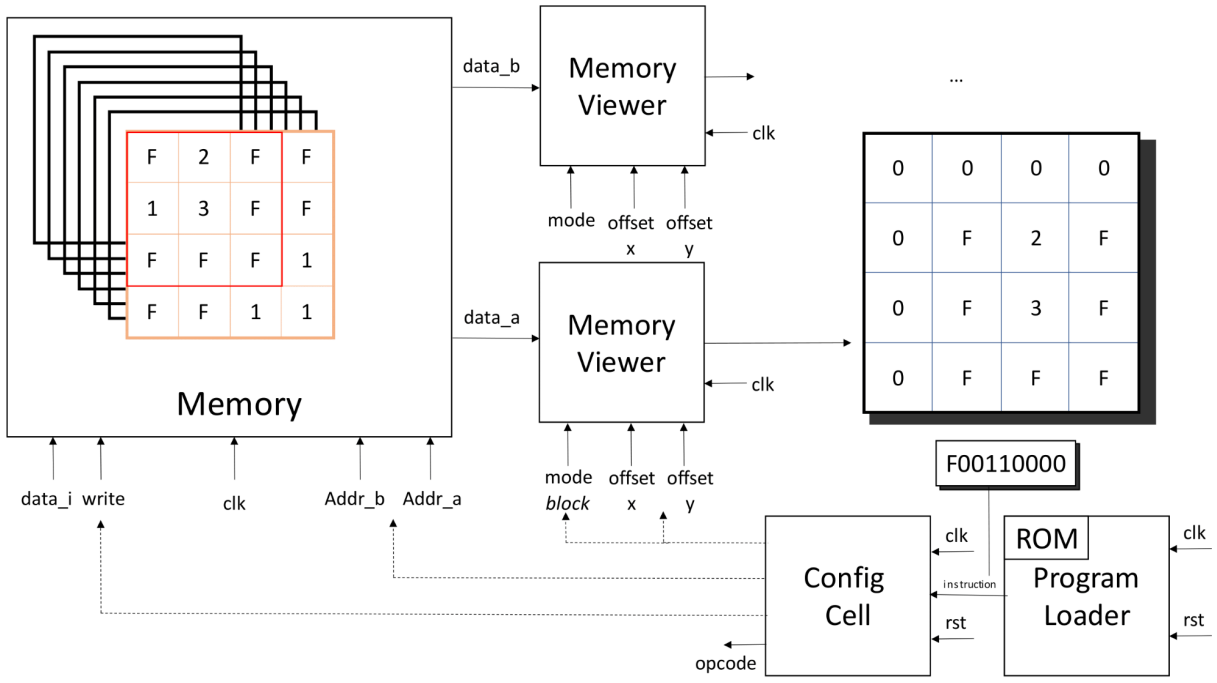


Figure 7: Config Cell and Memory Diagram

The compute tile grid makes it quite easy to concurrently execute matrix operations, the main challenge is in reducing the various high-level operations to this architecture. In each cycle we need to send 2 unique operands to the tile and thus organize the memory as memory blocks, where each block is a consecutive area of memory with $N \times N$ words, and allow the config cell to simultaneously retrieve 2 independent blocks. In the simplest scenario we can compute for example a matrix addition of two $N \times N$ matrices by storing the matrix values in two separate pages, selecting them for the computation and sending the additional op-code. For more complex operations, or for matrices that do not align well with the tile layout, the memory needs to be padded, or values need to be ignored. For an addition with a $N \times 1.5N$ matrix we would need to split the operation into two hardware operations, using a total of 4 pages.

Matrix multiplications make this even more complicated, since matrix operations that consist of many accumulation steps require that the values are mapped to the same tiles in each step. For example, if we take a matrix multiplication for a linear layer with operands $N + 2 \times M$ for the weights ($M << N$) and M for the input. A simple way to map this to our architecture is by dedicating the first $N + 2$ tiles to the accumulation and in each cycle supplying one row of the weight matrix as one operand and one single value for the other,

for a total of M steps. If each operand requires its own memory block, we would need M blocks containing $N + 2$ words, and another M blocks with $N + 2$ identical values. It is obvious that the number of memory blocks quickly rises, even for simple operations, and that many memory cells would remain unused. Since in machine learning the memory requirements are often a limiting factor, we need to provide a more flexible way to access memory that allows us to use less padding and avoid redundant data.

To achieve this we devised a Memory Viewer (MV) module. The MV receives data blocks from the memory like before and returns a modified view of the memory block. It allows for modification in two ways:

Offset

The core mechanism is providing a view of the memory block that is shifted by a certain number of words in the x and y direction, where values outside the block perimeter are substituted by 0. This makes it possible to easily compress memory into a single block with minimal padding. In the above example, we could put multiple $N + 2$ weight rows into the same block and for each cycle we just need to move down 2 rows in the block, thus only wasting $N - 2$ padding words instead of $(N + 1) * (N - 2)$ like before. This mechanism also makes it easy to implement convolutions with padding and thus avoids wasting memory for padding values in matrices.

Mode

We allow each memory access to specify an access mode. Currently we only implemented two modes, but left space in the config cell opcode for extension, since this mechanism can greatly expand the flexibility of our processor (e.g. we had row- and column-wise access patterns in mind that allow us to replicate the systolic array behavior more closely). The two implemented modes are *block* view and *word* view. Block view provides the default memory view where each value is unique as above. The *word* view instead broadcasts the first value to all other elements, which is used for many applications where the same value is required in multiple computations at once. Combining this with the offset modification allows us to effectively address each word individually and thus zero padding is needed for many operations.

Each operand in our architecture has its own independent MV. A visualization of the MV can be seen here.

Config Cell Design

The config cell (CC) is responsible for decoding instructions from the program loader (PL) and outputting memory addresses and signals, MV parameters and the opcode for the tiles. The PL is a simple module with a ROM filled with instructions, one of which is read in each clock cycle. The opcode is as follows for a $N \times N$ grid, where $N = 2^w$, with $B = 2^b$ memory blocks:

	Operand A				Operand B			
tile opcode	mode	address	offset x	offset y	mode	address	offset x	offset y
4-bit	2-bit	B bits	1+W bits (signed)	1+W bits (signed)	2-bit	B bits	1+W bits (signed)	1+W-bit (signed)

All the information gets forwarded without modification to the appropriate outputs of the CC, but with different delays. When the CC sets the tile opcode, the operands need to be available as well, which require 2 cycles to arrive after setting the memory signals. Thus all outputs are put in a queue of varying length, the addresses are forwarded instantly, the MV modes and offset are delayed by one cycle (queue size 2) and the opcode by two cycles (queue size 3).

We also implemented a mechanism for storing data from the tiles in the memory. For the store operation we need to set the *write* output to high and provide a storage address, which prompts the memory to store the values of the tile data_out lines in the specified memory block. For the store operation we ignore all MV attributes and only use the address of operand A to specify the destination address. The storage address and the write bit are thus also put in queues with a 2 cycle delay.

Compiler

To utilize this architecture we need to have a compiler that is able to reduce a computation graph of matrix operations to a set of instructions. There is a growing interest in machine learning compilers with custom hardware support. We considered using existing technologies such as MLIR[6] or Glow[7], but after a brief exploration phase we decided that the technologies are too immature to use for our project. But in the near future we expect frameworks like that to be an easier way to target custom processors, especially with the advancements in the upcoming PyTorch 2.0 release[8].

Our compiler was implemented in Python and returns a list of opcodes for the CC. We will briefly describe the methods we used to reduce high-level compute graph operations to instructions. The description will only include descriptions of operations used in the example neural network or for the evaluation.

Addition

All simple binary operations are very easy to reduce to our architecture with usually 100% resource utilization. Since none of the values in either matrix are interrelated or being reused we can just store the values of both $K \times M$ matrices in $n = \lceil N^2/(KM) \rceil$ of blocks each. At most the last few words in the last block are wasted, but in some computations it might be acceptable to put values of different matrices into the same block, if there is no risk of commingling them. The resulting instructions will thus n addition instructions, each specifying the two operands without MV modifications, and n store instructions..

ReLU

ReLU is handled almost identically to addition. The only main difference is that the second operand is a broadcasted 0 (using *word* mode) and the tile opcode is set to max. For our implementation we simply looked at the existing memory block items and selected the first 0 we could find as the second operand. For a more mature system we could alternatively either have a hardware-level switch for ReLU, since it is a very common operation, or dedicate one memory block to global constants.

For any computation, if a result block would only ever be read once for a subsequent operation, he subsequent operation would be inserted before the store operation, using the tile cache as the operand, which effectively removes the need for n store instructions. This is almost always the case for when applying ReLU, wherefore it is often inserted before the store instruction of other instruction blocks.

Max-Pooling

Any matrix reduction/pooling operation is reduced in the same way, except for the opcode. For channel-wise pooling with d channels we have $d - 1$ pooling instructions per block and one store instruction. The first instruction takes the first block of two channels and each subsequent pooling is using the cached results as the second operand. This is repeated for all n blocks.

For in-channel pooling with kernel size k for each block we similarly have $k^2 - 1$ instructions, where the two operands are views of the same memory block with different offers.

Matrix-Multiplication

There are many different algorithms for matrix-multiplication. For our purposes we implemented a naive algorithm that accumulates over the output nodes. N tiles, corresponding to the outputs, are used for the computation and the operation is reduced to M, the size of the input vector, MAC instructions. The first operand is a broadcasted word from the input vector and the second operand is a row of the weight matrix. After all M instructions a single store instruction is added.

2D Convolution

Convolution is basically just a special case of weighted pooling with in-channel and channel-wise operations at the same time. The pooling instruction is MAC and the second operand is a broadcasted weight. Only after the in-channel and channel-wise instructions is a store instruction added, leading to $(d-1)(k^2-1)$ MACs per block.

Simulation and Measurement Results and Analysis

After modeling our design in Verilog, we synthesized and simulated the design. Using the clock speed we got from the maximum delay in the synthesized design, we then created multiple testbenches for our processor. These testbenches tested multiple specific applications of our design. We then used these benchmarks to compare to other processors running the same program.

Simulation Environment

For simulating our design, the first step was using Verilog HDL to model our design. We then compiled this design using VCS, and simulated it in DVE, using sample programs that were compiled from the PyTorch library. After verifying its correctness, we used the same design in Design Vision using the 45 nm libraries to synthesize a hardware design. After this, we used the Design Vision tools for calculating area and maximum delay to measure the results of our design. We repeated the synthesization process for different sizes of the matrix in order to get different areas and delay results.

Measurement Results

The two primary results we measured in our physical design were area and maximum delay, which we will use to calculate the clock speed of our design. Because the design can be resized with different amounts of

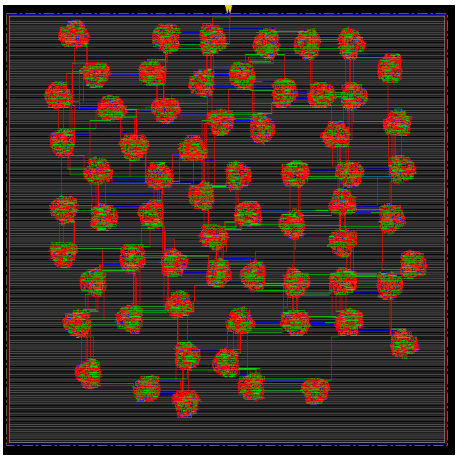


Figure 8: Synthesized Layout

Area Measurements	
Matrix Size	Size (μm^2)
1x1	4,604
4x4	63,705.6
8x8	253,663.22
16x16	1,012,896.82

Extrapolated Area Measurements	
Matrix Size	Size (μm^2)
2x2	16,395.96
32x32	4,050,550.83
64x64	16,201,956.56

Tables 1&2: Area and Extrapolated Area Measurements

computation tiles, it was important to take area measurements of multiples sizes and then extrapolate those results so that the sizes of designs that were too big to synthesize could be estimated. The exact areas and the estimated areas can be seen in Tables 1 and 2 respectively. These results show a quadratic relationship between the size of the matrix and the area of the design, which is expected. A graph showing the area function and the equation can be found in Figure 8. Overall, the area of the design is relatively small at 16x16 and lower, however larger computation matrix sizes can make it prohibitively large for some applications. In comparison to other prior work, the area of our design is smaller, however this comparison is not entirely accurate due to the limited functionality of our design. The Neuflow architecture measured 12,500,000 μm^2 in area for a 10x10 matrix, and the scalable MCM architecture 6,000,000 μm^2 in area for a single chip.

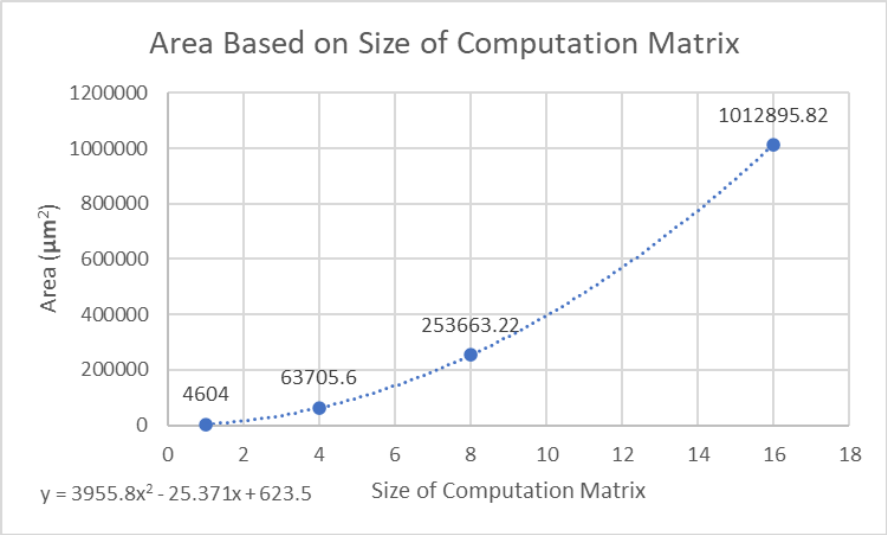


Figure 9: Area Graph

For delay, the primary concerns were how long the maximum delay was to calculate the clock speed of our design, and where the maximum delay was so that it could be decreased in the future. We found that for all sizes that were synthesizable, the maximum delay was 2.56 ns, and the path for the maximum delay was through the multiplier and the accumulator in the computation tile. This path is visually shown in red on the diagram of the computation tile in Figure 9. Because the maximum delay is only through the computation tile, the size did not have a significant effect on the maximum delay. It is possible, however, that very large matrix sizes will affect the delay to larger muxes being required in the config cell.

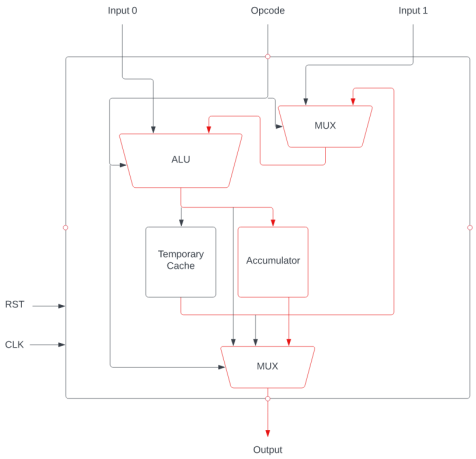


Figure 10: Critical Path

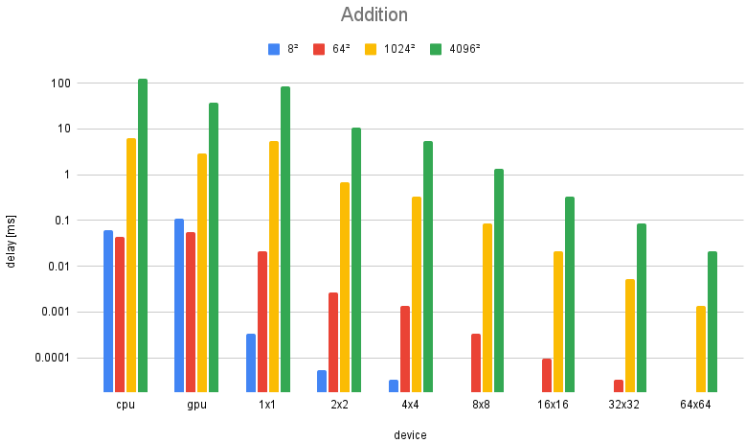
Result Analysis

To compare our architecture to existing processors we ran various computations on matrices with exponentially increasing size to observe how well the computations scale. We compare the performance to multiple versions of our processors with varying grid sizes to highlight the effect the parallelism has on the compute time.

The times for CPU and GPU were measured with PyTorch and averaged across 500 samples each, to allow caching mechanisms to show their effect, which especially for GPUs can have a pronounced effect. The CPU we compare to is an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, and the GPU is a Nvidia GeForce GTX 1050 Mobile.

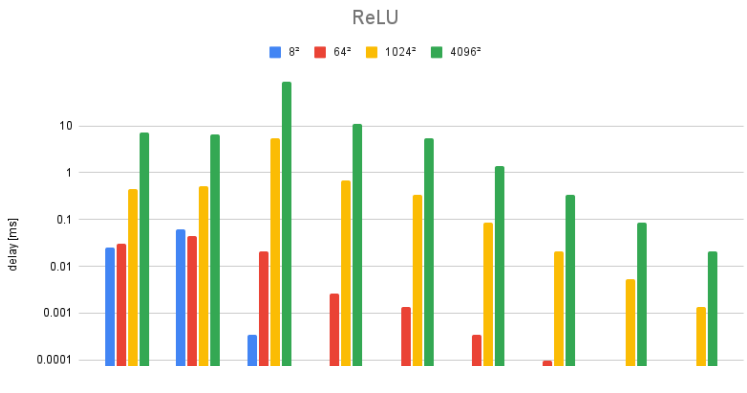
Addition

Primitive binary operations scale really well on our architecture. We can see a linear increase in performance with an increase in computation tiles. As we can see even the smallest configuration of our architecture outperforms other processors, which shows us that the magnitude of the differences has to be taken into consideration with scrutiny. Since we do not have physical processors and have to rely on simulation results, the timings for our processors will probably be favorable. On the other hand, the times taken from the other processors will be unfavorable since the CPU and GPU are part of an active OS that has to handle other tasks as well, and they are part of a much larger system which leads to all kinds of delays and caveats. But we still think that the scaling effects can give an idea of how the different processor architectures scale and where their weaknesses are.



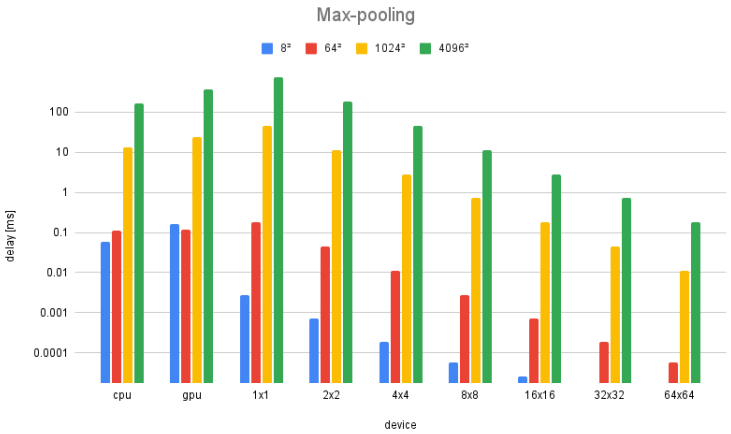
ReLU

For ReLU the picture is similar to above, but we can see that the other processors drastically outperform our small configurations for large matrix sizes, the 4x4 device is the first device with a comparable performance for the largest matrix.



Max-Pooling

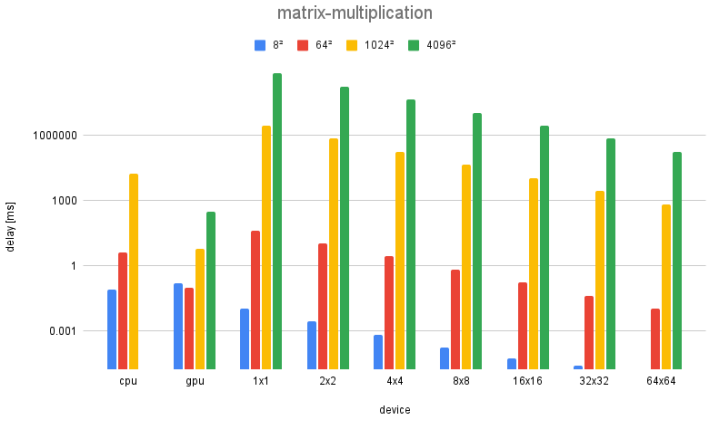
Simple pooling scale is very similar to ReLU, only with a big difference in magnitude.



Matrix-multiplication

Matrix-multiplication is an operation with many inter-related operations and exponentially increasing complexity. We can see that the CPU is struggling a lot with the large matrix sizes and took too long to finish the last operation (more than 30 minutes). Similarly our architecture did not scale well with increasing matrix size and only configurations starting from

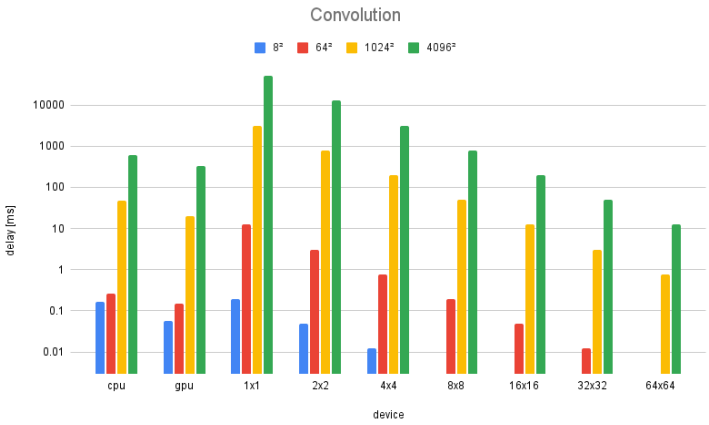
16x16 are comparable to the CPU performance. All of our tested configurations fail to come even close to the performance of the GPU. We estimate we would need a grid of about 1500^2 tiles to achieve a comparable performance.



We expect that our processors would be twice as fast as ones with a traditional systolic array, since we can immediately utilize all compute units and do not need to queue the operands. Yet if we would compare our processors to a real world TPU or other processors with a systolic array, our performance would likely lack behind them as well. The main bottleneck for our architecture is the algorithm the compiler implemented, which has complexity $O(N^3)$ (which translates to $68.7 * 10^9$ MAC operation for the most demanding case), while PyTorch implements an algorithm with a lower exponential and thus has to process fewer operations. We also expect other modern processor mechanisms, like caching, to play a role in the large performance difference. It's also worth mentioning that our computation is based on signed int16 values, but since PyTorch does not implement integer matrix-multiplications we had to compare it to FP16 operations. Yet we do not think this represents a significant factor in the performance difference.

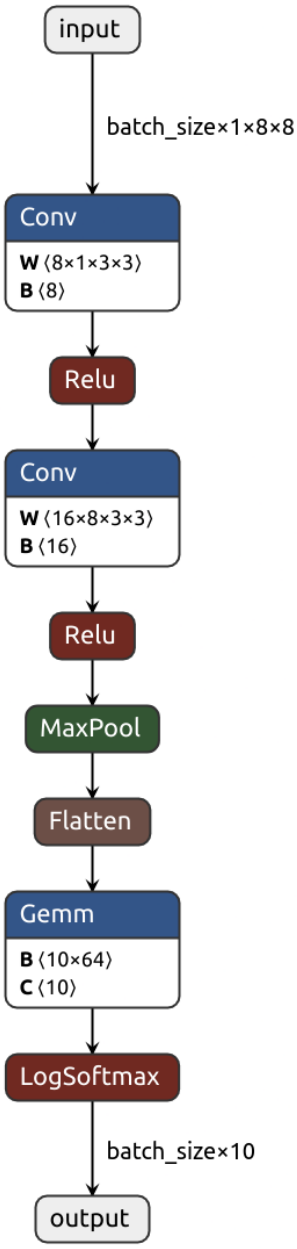
Convolution

Convolutions compare similarly to matrix-multiplications and suffer from the same bottlenecks. But since convolutions do not scale as rapidly with growing matrix size, we can see that our $16x16$ processor is comparable to the other CPU and GPU and bigger processors significantly outperform them.



Testing and Verification of Proposed Work

To test and verify our implementation we created a simple neural network (shown on the right) that is representative of common ML operations and use-cases. The network is a small Resnet trained to recognize handwritten digits from a 8x8 single-channel image, as a simplified version of the MNIST dataset. We chose a very limited architecture with small layer sizes to be able to manually verify each step and ensure we can actually run it on within the simulation environment. We implemented the network in PyTorch and trained it to 99% accuracy.



The trained network was converted to a set of instructions and initial memory blocks using the compiler described above, which resulted in 1356 instructions and 37 memory blocks. The instructions and memory data were stored in text files as HEX numerals and loaded via Verilog for simulation.

Before verifying the network inference works, we verified the behavior of the ALU, MV and CC independently with a few manual test-cases. The compute graph computations were verified by identifying the timestamps when the output data would be written to the memory, observing the values and comparing them to the ground-truth values computed with PyTorch on a conventional CPU. We modified our Python environment to display NumPy arrays as HEX numerals so we can directly compare the values we observe in the simulation to the correct ones.

Problems and Solutions

While the initial design of this was straightforward, there were some problems that we encountered due to the limited amount of time and smaller team size. One of these issues was not being able to use other machine learning compilers for our design. Another issue was a design issue based on the amount of data our computation tiles needed to receive each cycle.

Roadblocks Encountered

The primary roadblock we encountered was the lack of a usable machine learning compiler. We were planning to use Glow or MLIR in some capacity to reduce more complex graphs for us, but after initial attempts we found that those frameworks are too immature and lack substantial documentation.

Another issue we encountered was minimizing the amount of information transferred between the computation tile and the rest of the architecture. As our design involves connecting each computation tile directly to memory, we did not want to further bloat the size of the opcode with addresses and additional bits for deciding when to read from cache. Including these extra bits would also require us to process them, which would increase the length of our critical path.

Workarounds

To solve the lack of a compatible compiler we decided to implement our own primitive compiler. This has the consequence that our compiler is not very flexible and does not implement optimized algorithms, which limits the overall system performance as described below.

In order to reduce the amount of data needed by each computation tile, we implemented multiple strategies to reduce the size of the opcode. First, the config cell handles all of the addressing and memory activation on its own, and will not send the memory information to the computation tiles. The other strategy we implemented was having the temporary memory for using a previous output as an input in the next cycle be automatic based on the sequence of opcodes. This meant that we did not need to have a specific set of bits for using information from the cache, and we could do this because a common pattern in machine learning is to perform a convolution or matrix multiplication, perform another operation on it, and then store that information.

Summary and Next Steps

Overall, the design successfully models a systolic array architecture with computation tiles connecting directly to memory. The benchmarks that this design achieved shows that there could be benefits to using this type of design for specific applications, however there is still more room to explore the design before it is ready for commercial use.

Project Summary

This project implemented a systolic array alternative where each computation tile is directly connected to memory. Each computation tile performed simple arithmetic operations, while a config cell organized memory reading and writing as well as what operation each cell would perform. We then verified the correctness of our design by compiling a simple machine learning program and then running it on our design. After this, we synthesized the design to gain an estimation of the area and maximum delay, and used that information to benchmark it compared to similar designs. This comparison showed that our design performed better on simpler instructions, comparably on convolutions, and worse on matrix multiplication. In conclusion, our design is a decent alternative to traditional systolic array designs for lightweight machine learning.

Project Impact

The biggest impact of this project is contributing to the set of benchmarking information that exists for machine learning architecture. As this is a field that is becoming increasingly important in both hardware and software, benchmarking unique designs and comparing them to existing ones can contribute to important developments. From our results, our design shows that it has practical applications in machine learning machines that focus on simple instructions and convolutions. Additionally, it is smaller than some of the other systolic array architectures. All of these indicate that our design is best for low intensity machine learning applications, potentially in situations where a full module is not frequently needed.

Next Steps

The next major steps primarily revolve around expanding the features of our design and developing more tools to better compare to existing architectures. First, a computation tile with more features would be

important for accurate comparisons. Currently most systolic array architectures support nonlinear functions, which our current machine does not support. This makes some of the measurements and benchmarking for our design inaccurate, since there may be more delay or a greater area cost for supporting these functions. Another important goal will be to develop a compiler specifically for our design. Currently the design only uses simple scripts and a small amount of hand compilation to convert machine learning instructions to code for our compiler. By developing a complete compiler, it would allow us to rapidly benchmark our designs with different programs. Additionally, it would improve the accuracy of our verification due to the fact that it would allow us to test significantly longer programs. Overall, the subject of a systolic array architecture with direct connections to memory has a significant amount of topics left to explore.

References

[1] Wang, Bo, Sheng Ma, Guoyi Zhu, Xiao Yi, and Rui Xu. “A Novel Systolic Array Processor with Dynamic Dataflows.” *Integration (Amsterdam)* 85 (2022): 42–47.

[2] Jia, Liancheng, Liqiang Lu, Xuechao Wei, and Yun Liang. “Generating Systolic Array Accelerators With Reusable Blocks.” *IEEE MICRO* 40, no. 4 (2020): 85–92.

[3] Farabet, C, B Martini, B Corda, P Akselrod, E Culurciello, and Y LeCun. “NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision.” In *CVPR 2011 WORKSHOPS*, 109–116. IEEE, 2011.

[4] P. -H. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun and E. Culurciello, "NeuFlow: Dataflow vision processing system-on-a-chip," *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2012, pp. 1044-1047, doi: 10.1109/MWSCAS.2012.6292202.

[5] R. Venkatesan *et al*., "A 0.11 PJ/OP, 0.32-128 Tops, Scalable Multi-Chip-Module-Based Deep Neural Network Accelerator Designed with A High-Productivity vlsi Methodology," *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1-24, doi: 10.1109/HOTCHIPS.2019.8875657.

[6] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., & Zinenko, O. (2020). MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv*. https://doi.org/10.48550/arXiv.2002.11054

[7] Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., Montgomery, J., Maher, B., Nadathur, S., Olesen, J., Park, J., Rakhov, A., Smelyanskiy, M., & Wang, M. (2018). Glow: Graph Lowering Compiler Techniques for Neural Networks. *arXiv*. https://doi.org/10.48550/arXiv.1805.00907

[8] *Pytorch 2.0*. PyTorch. (n.d.). Retrieved December 6, 2022, from https://pytorch.org/get-started/pytorch-2.0/