# Project F.A.S.T.R.

FPGA Accelerated System for Template Recognition

EECS 113 Final Project Report

Jordan Dickson 65582692

## Problem:

Modern image processing and machine learning algorithms often need to perform many matrix multiplications to complete their operation. Matrix multiplication is a computationally intensive operation that can put a lot of strain on a general purpose CPU, and this strain can have an exaggerated effect in the context of embedded systems processors.

The main problem is that as the matrices grow linearly in dimension, the complexity of the multiplication operation grows polynomially as $O(n^3)$. According to this growth factor, multiplying two 32x32 matrices is 512 time as difficult as multiplying two 4x4 matrices. The complexity (and execution time) quickly becomes unmanageable for larger matrices. For example, multiplying two 1024x1024 matrices is more than 32,000 times as complex as multiplying two 32x32 matrices.

## Proposal:

There is a push in recent technology towards using hardware acceleration by offloading certain tasks from the general purpose processor to a specialized circuit designed to handle that task more efficiently. Newer embedded systems-on-a-chip such as the Xilinx PYNQ board include an FPGA module for implementing that kind of hardware acceleration.

The F.A.S.T.R. project aims to design a hardware accelerator for performing matrix multiplication quickly on a platform such as the PYNQ board. Ideally we'll find a way to reduce the algorithmic complexity to something below $O(n^3)$ to make larger matrices more manageable.

## Potential Applications:

Since matrix multiplication is required by many different computer algorithms, the potential applications for the hardware matrix multiplier could benefit an array of computer operations. Two specific applications, machine learning and image processing, could benefit greatly from a faster matrix multiplication library that utilized a hardware accelerator.

## Materials:

There are few physical materials required for this project:

- Xilinx PYNQ Board
- Power source for the PYNQ Board
- Ethernet Cord
- SD Card with PYNQ-compatible embedded Linux installed

## Approach:

In defining the scope of the project, we decided concern ourselves only with multiplying $n$ x $n$ square matrices. In addition, we decided to design our circuits and baseline programs to handle only integer matrices. Floating point operations are outside the scope of this project.

In order to have a baseline comparison for the hardware accelerator's performance, we designed two software matrix multipliers - one in C and one in Python - and gathered data about their performance as the size of input matrices increased. Both programs exhibited an $O(n^3)$ complexity.

Since we're trying to simplify the complexity of the $O(n^3)$ operations using hardware, and each individual multiplication of matrix elements is independent of every other multiplication, we can design a circuit that can perform all of the multiplications in parallel. After all the multiplications, we can use a tree of adder circuits to sum up the appropriate products and get the corresponding elements of the output matrix.

Because all $n^3$ multiplications are performed in parallel, the multiplication complexity is now just $O(1)$. However, the adder tree that follows the multiplication step will have add a new complexity of $O(log_2 n)$. Regardless, this is still a massive reduction of complexity compared to the software solution.

Note: To be clear, I should explicitly state that for the software solutions I am equating algorithmic complexity with the execution time. For the hardware solutions I am equating the algorithmic complexity with the critical path length (signal delay time) of the circuit. In both cases the $n$ is referring to the size of the $n$ x $n$ matrices being multiplied.

It's important to remember that part of the hardware solution will require transferring the $n^2$ elements from the two input matrices to some buffer and later retrieving another $n^2$ elements from the output buffer once the operation is complete. This new $n^2$ complexity associated with the data transfer actually becomes the new bottleneck of our hardware solution. The hardware solution may not be as large a speedup as it first seemed, but it is still a whole polynomial order of complexity easier to perform.

## Design:

### Integer Multiplier:

The first step was to design an integer multiplier unit that could be used as a building block for the matrix multiplier later. Our design uses a masking and shifting stage followed by an adder tree to implement hardware integer multiplication.

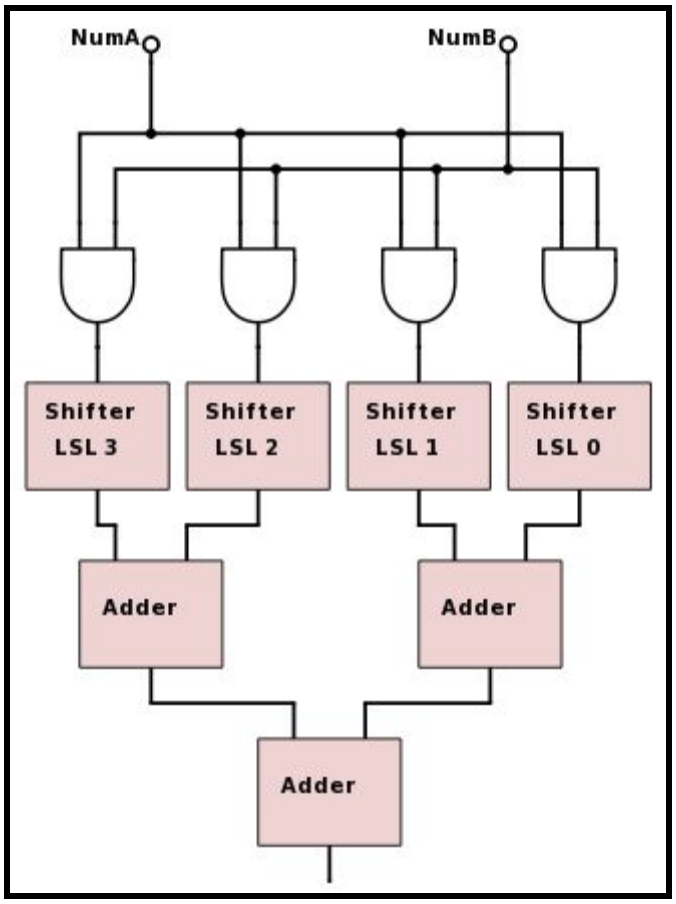


Figure 1: Four Bit Hardware Integer Multiplier

In the figure above, the left input to each AND gate is the entirety of NumA. The right input is a mask based on the corresponding bit in NumB. For example if NumB is 0b0101, then the masks will be (from left to right) 0b0000, 0b1111, 0b0000, and 0b1111. The shifter stage that follows simply reindexes the signals to put them in the correct binary place value. The adder tree at the end sums up each of the shifted values and outputs the product of NumA and NumB. This adder tree also determines the critical path complexity to be $O(log_2 W)$ where $W$ is the bit width of the input numbers being multiplied. For this project we will set $W$ to be a constant 32 bits.

## Matrix Multiplier:

With the multiplier circuit for individual integers complete, we designed a matrix multiplier circuit that could perform all of the necessary multiplications in parallel. In order to do this in an easily scalable way for arbitrarily large $n \text{ x } n$ matrices, we created a system where all of the $n^2$ elements of both matrices are arranged along the axes of a two dimensional space. Matrix A's elements are listed along the horizontal axis incrementing by column first, and elements from Matrix B are on the vertical axis incrementing by row first. Integer multipliers are then placed in the 2D space in the appropriate locations for each multiplication that will take place. After placing each of the multipliers, we can see that a pattern emerges. The $n^2 \text{ x } n^2$ 2D space is divided into $n \text{ x } n$ cells, each cell with $n$ multipliers in it. The sum of the products of those multipliers gives the value of an element in the output matrix.

Stated mathematically, if $A * B = C$ where $A, B, C$ are all $n \text{ x } n$ matrices, then each element of $C$ can be found using the following equation.

$$\sum_{k=0}^{n-1} A_{k,\,i} * B_{j,\,k} = C_{i,\,j}$$

The variables $i$ and $j$ refer to the column and row (respectively) of the cell in the 2D multiplication space, and variable $k$ refers to the individual multiplier inside that cell.

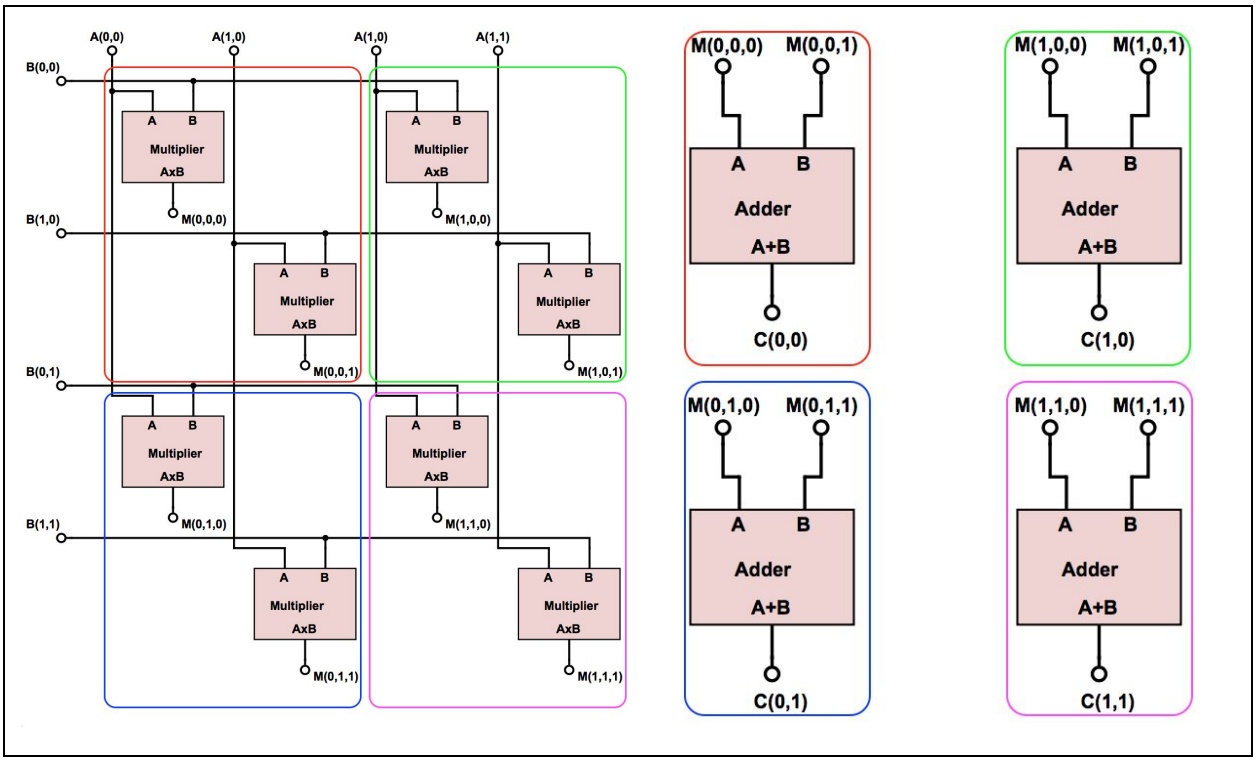


Figure 2: 2x2 Matrix Multiplier Example

Figure 2 shows the multiplication and addition stages side by side to present a cleaner view of the circuit. In the case of a 2x2 matrix, the circuit requires a 2x2 matrix of multiplier cells, with 2 multipliers in each cell. Figures 3 and 4 show how this design can scale to suit 4x4 matrix multiplication.

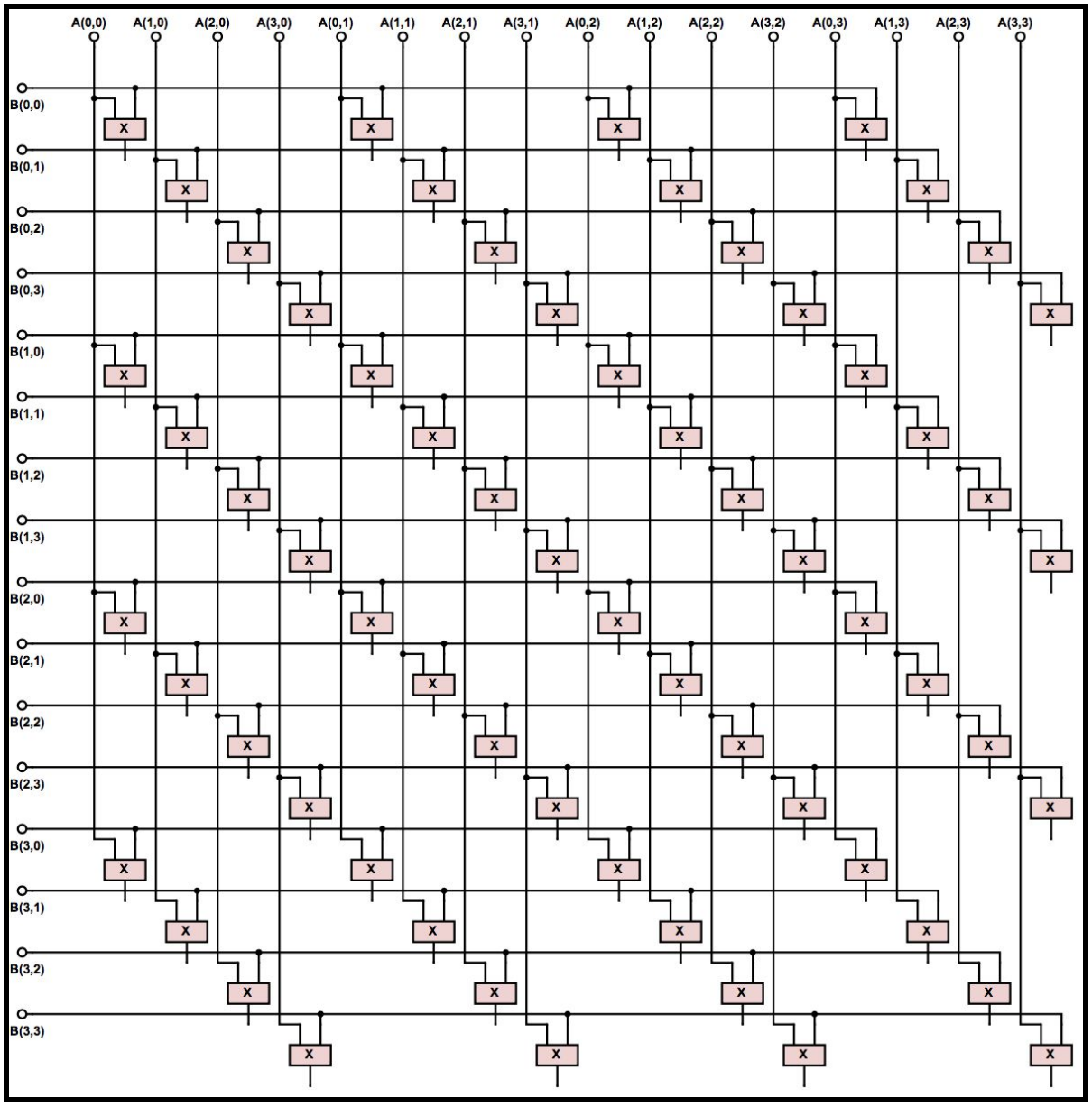


Figure 3: Multiplier Matrix for Multiplying 4x4 Matrices

While the number of multipliers is increasing rapidly as $O(n^3)$, all $n^3$ of those multipliers are executing in parallel, so the critical path remains constant. In Figure 3 there are 4x4 multiplication cells, and each cell contains 4 multipliers.

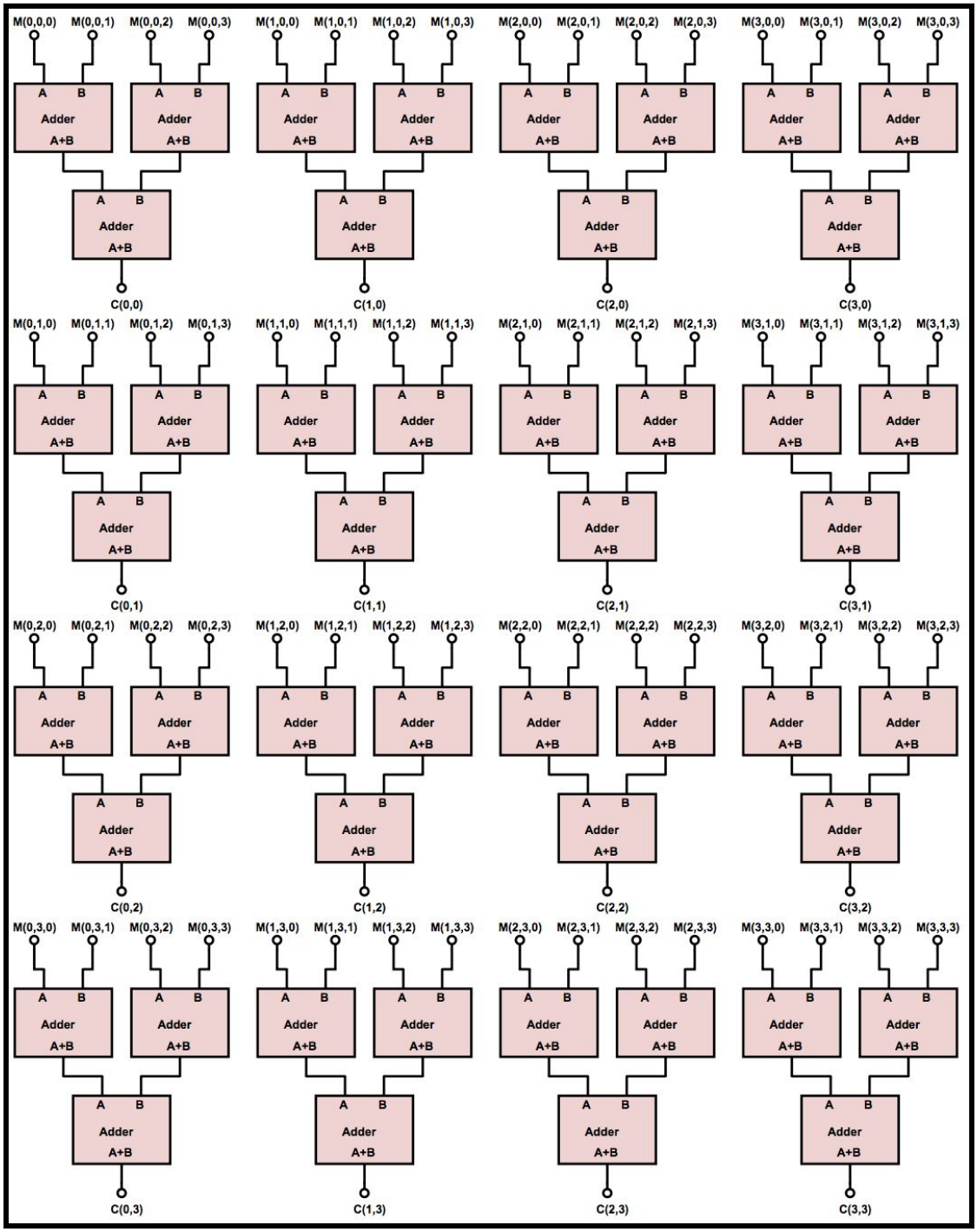


Figure 4: Adder Tree Stage for 4x4 Matrix Multiplication

The number of adders is also increasing as $O(n^3)$, but because $n^2$ of them are performed in parallel and the remaining $n$ are arranged in a tree structure, the critical path only grows as $O(log_2 n)$

The design basically works by trading algorithmic time complexity for hardware resource complexity. Rather than performing $n^3$ serial multiplications on one processor, the hardware design uses $n^3$ "processors" to perform all of the multiplications in one step. The new performance bottleneck becomes simply shifting in all of the data for the $n^2$ integer entries per matrix.

## Results:

We worked out the following equations to accurately predict the execution time of the software and hardware solutions:

Software: $T_{mult} * n^3 + T_{add} * (n-1) * n^2 \approx n^3$

Hardware: $T_{xfer} * n^2 + T_{mult} + T_{add} * log_2 n \approx n^2$

Testing with the Python and C software benchmarks yielded expected results. Both programs execution times grew roughly as $O(n^3)$ and the C benchmark performed consistently better than the Python benchmark.

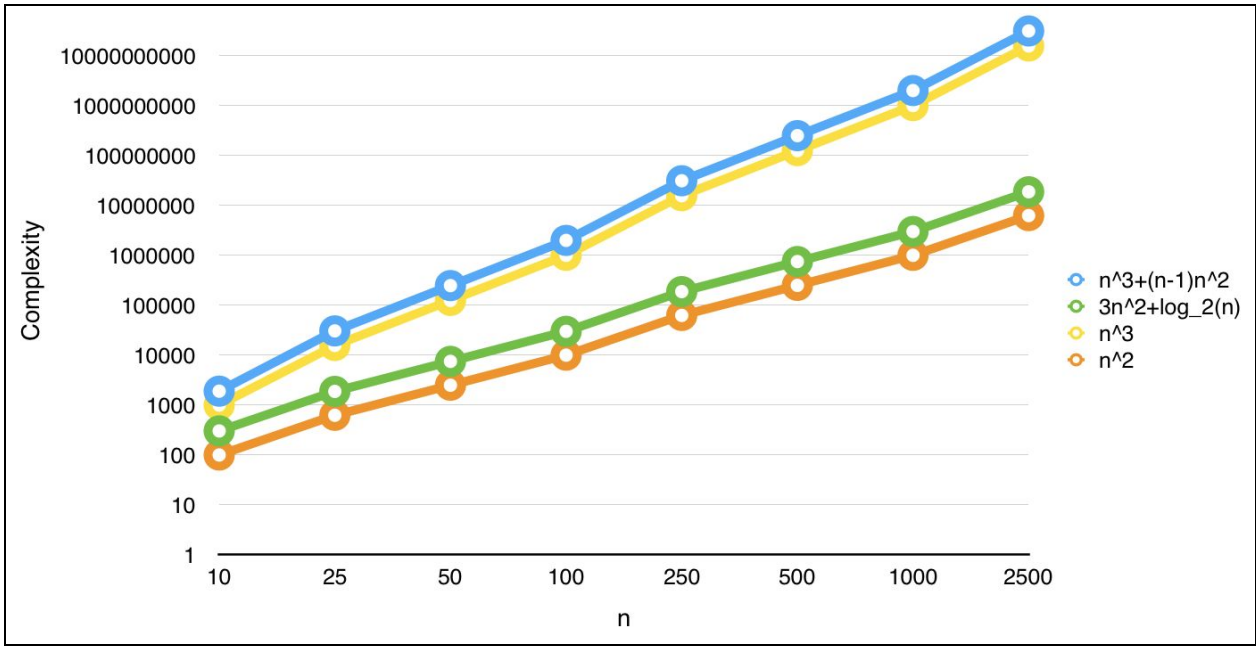


Figure 5: Expected Complexity Growth For Different Complexites

The blue and yellow lines predict the performance of the software solution, and the green and orange lines predict the performance of the hardware solution. We can see that the speedup of the hardware solution is expected to increase with n because the complexity of the hardware solution grows slower than the complexity of the software solution.

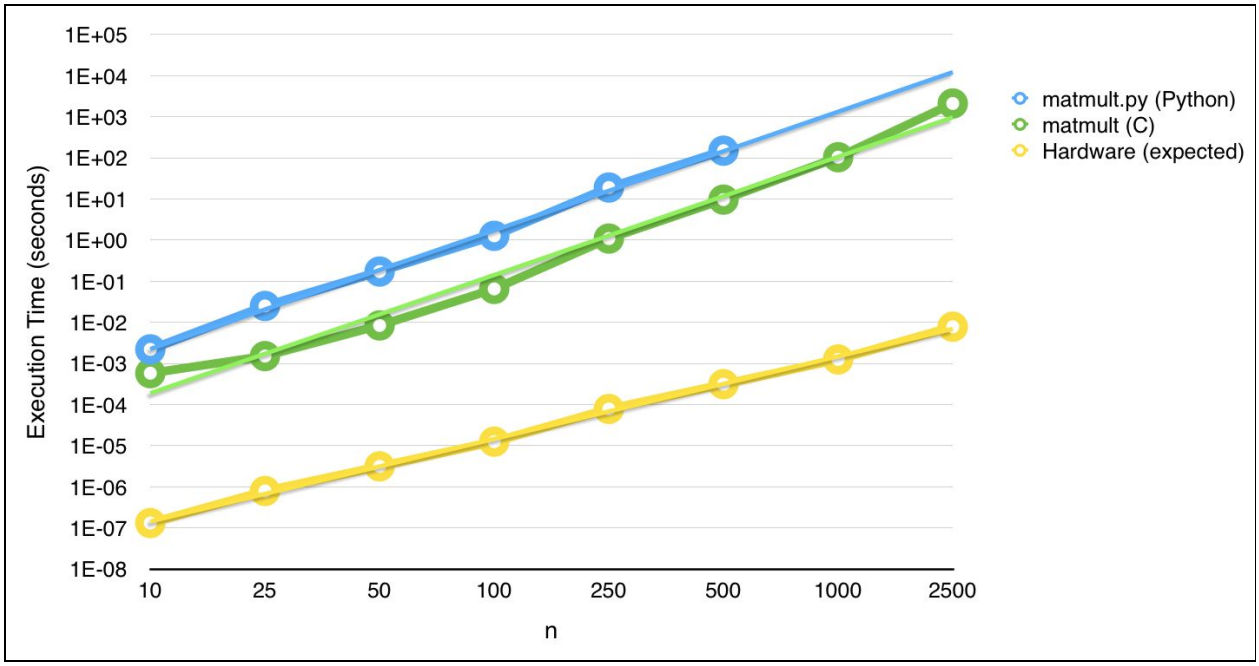


Figure 6: Experimental and Predicted Results for Benchmarks and Hardware Design

We can see that there is a similar improvement in performance from using the hardware multiplier. Since we never got the multiplier running the the FPGA itself, the data for the multiplier was collected using predictions based on simulation results and the PYNQ datasheet.

- According to the dc_shell-t synthesis and simulation tool, the integer multiplier has a critical path length of 1.48 ns.
- We will make an overestimate for the adder delay by simply saying that adders also have a critical path of 1.48 ns.
    - We know it will be less than this because the multiplier's critical path itself actually contains 5 adders, but we'll use the approximation anyway.
- According to the PYNQ board datasheet, the maximum data transfer rate between the PC and PL using the High Performance AXI ports is 1200MB/s
    - 1200M Bytes/second ⇒ 300M Integers/second ⇒ 3.33 ns/Integer

Using this information, we can find values for the coefficients in our complexity equation.

$$3 * T_{xfer} * n^2 + T_{mult} + T_{add} * log_2 n$$

$$T_{xfer} = 3.33ns$$

$$T_{mult} = 1.48ns$$

$$T_{add} \leq 1.48ns$$

Putting all these predictions together can give a good estimate of how the hardware solution could perform. If it had been implemented on the PYNQ's FPGA.

## Obstacles:

While the initial predictions all looked good, it turns out there are other limitations on the PYNQ board to consider aside from just the data transfer rate of the AXI port. The hardware resources on the PYNQ are limited, and this design's resource demand grows extremely fast with $n$.

According to the datasheet for the XC7Z020, the PYNQ board has 106,400 flip flops and 85,000 programmable logic cells available in its FPGA.

Memory demand grows as $O(3 * n^2)$, and each integer requires 32 flip flops to store its value, so the largest number $n$ they PYNQ can support is:

$\sqrt{\frac{106400}{32 * 3}} \approx 33 \Rightarrow$ The PYNQ's FPGA can store 3 33x33 matrices in its flip flops

Logical cell complexity grows as $O(n^3)$, mainly just trying to keep up with the rapidly expanding number of multipliers. According the the dc_shell-t tool, each 32-bit integer multiplier requires about 1200 logical cells to implement. This also limits us in how large our matrices can be:

$\sqrt[3]{\frac{85000}{1200}} \approx 4.137 \approx 4 \Rightarrow$ The PYNQ's FPGA can only support enough multipliers to multiply 4x4 matrices.

## Next Steps:

While we are limited to multiplying 4x4 matrices, we can still employ algorithms to build our way back up to larger matrices. A new complexity equation can be devised based on the original hardware solution.

$$(3 * T_{xfer} * 4^2 + T_{mult} + T_{add} * log_2 4) * (\tfrac{n}{4})^3 + T_{add} * \tfrac{1}{2} * (\tfrac{1}{4})^3 * 1.25 * n^3 \approx O(n^3)$$

Notice this solution is still no simpler than $O(n^3)$, but using the same predictive strategy as before we can still estimate its performance to be somewhere between that of the original (impossible) design and that of the software solution.

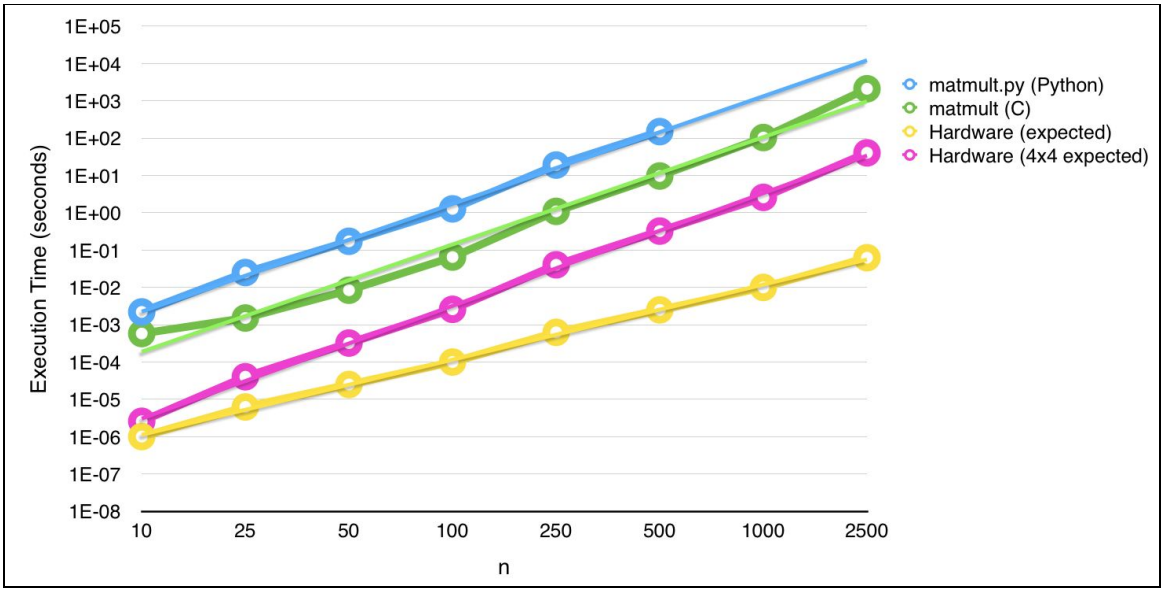


Figure 7: Expected Performance of the 4x4 Multiplier/Accumulator

We can see that the expected performance of the 4x4 multiplier starts off being very similar to the original design, but as n increases it begins to approach the same performance as the C solution.

Perhaps a more advanced solution could share resources the reduce the number of logical cells required. In addition the PYNQ comes with specialized DSP modules that could be used in a more elegant accelerator.