Accelerated Matrix Multiplier

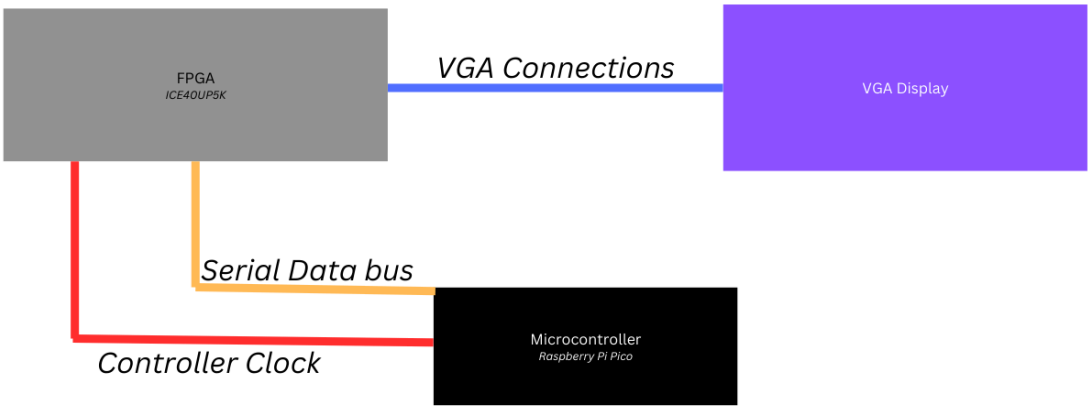
ES4: Introduction to Digital Logic

Jack Burton, Michael Zhou, Ha Nguyen, Camran Aduli

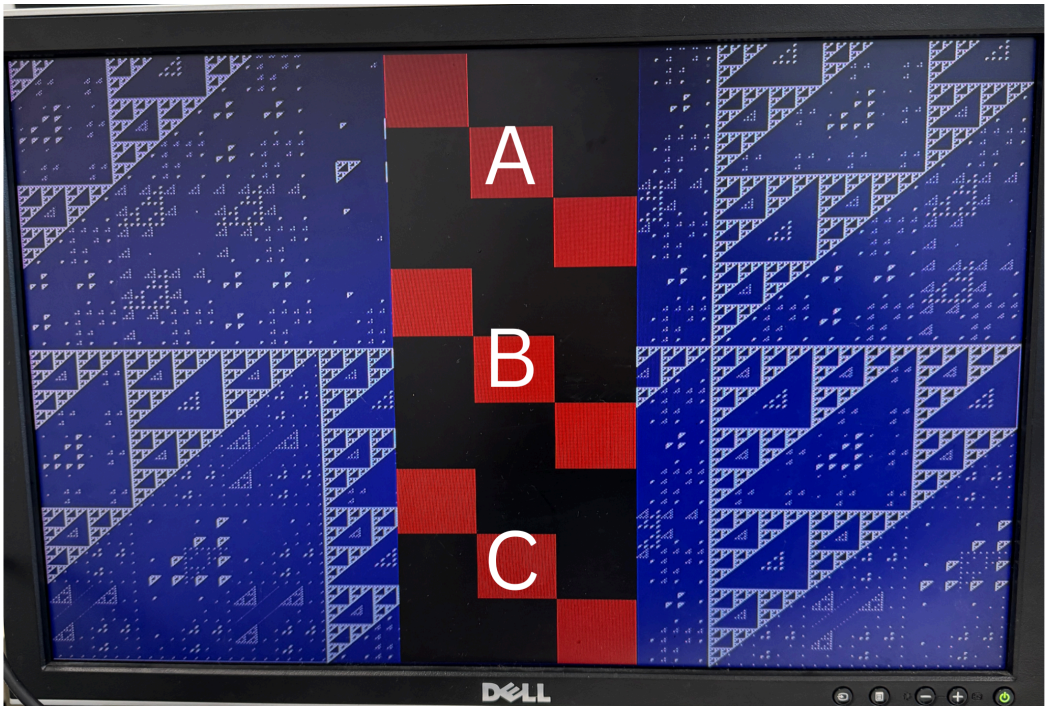
Overview **2**
Technical Description **3**
Pico **3**
Overview 3
Faster Execution 4
Dependencies 4
Initialization Logic 4
Driver Loop 5
Compilation 5
FPGA **5**
Overview 5
top 6
matrix_mult 6
vga 6
pattern_gen 7
Results **7**
State 7
Timing 8
Known Bugs 8
Photos/Videos 9
Reflection **11**
Work Division **12**

Overview

Our project is an accelerated matrix multiplier with a visual display. Matrix multiplication involves many repeated multiplication and addition operations, and on a standard computer these operations take at least one, if not more, clock cycles, even excluding the cycles pertaining to moving numbers around in memory. With an UPduino 3.1 FPGA board (FPGA), these multiply and add operations can be performed concurrently since many FPGAs feature multiple built-in digital signal processing units (DSPs), equipped to perform math operations very quickly. This project takes advantage of these DSPs to perform quick matrix multiplication, and show the two input matrices and the product matrix on a VGA display, representing the numbers with 6-bit RGB values. The input matrix numbers are written into two arrays in some embedded C code, which is flashed onto a microcontroller, in our case the Raspberry Pi Pico ("Pico", going forward). This C code produces a controller clock used for serially streaming the input matrix numbers to the FPGA, as unsigned 8-bit binary values. When the serial bus and the controller clock are connected from the Pico to the FPGA, and the FPGA's VGA output is connected to a display, the two input matrices will populate and the resulting matrix will appear just below these. Each input array, and the output array, are 3×3 with each value being an 8-bit unsigned integer.



Overview of the major components and their connections



Loading in two input matrices, A and B (identity matrices), and showing the result, C = AB (also the identity matrix by definition)

Technical Description

Pico

Overview

The Pico drives the matrix multiplication, acting as a controller for the FPGA. The solution outlined below is written in embedded C, but we also have a similarly functioning solution in microPython. The Pico stores the input matrices in arrays, converts them from 8-bit unsigned integers to their 8-bit binary representations, and serially streams the values to the FPGA over a controller clock which it generates:

```
// Input matrices in unsigned base-10 form
uint8_t A_nums[] = {5, 0, 4, 1, 2 ,3, 5, 4, 0};
uint8_t B_nums[] = {1, 0, 0, 0, -1, 0, 0, 1, 0};

// num_to_bit_array
// Purpose: Converts array of base-10 numbers into array of 8-bit binary representations
// Returns: None; sets bit_array pointer to bit array
void num_to_bit_array(uint8_t *nums, uint8_t num_count, uint8_t *bit_array) {
```

The two input matrices A and B, and the function which converts them into binary bit arrays

Each value in the number arrays is stored as *uint8_t* since they are unsigned 8-bit numbers, and then they are also stored in the bit array as *uint8_t* just consisting of 8 bits of "0" or "1", to represent the binary value.

Num_to_bit_array has to perform a bit of math to convert from base-10 to base-2 representations, so this occurs prior to any input matrices being sent to the FPGA, otherwise the serial data stream would be significantly slower.

We have built an identically functioning script in microPython, but using the C solution is highly advised because it is much faster.

Faster Execution

The system clock on the Pico runs at a standard 125MHz, but can be safely overclocked to ~250MHz without issue, so it is overclocked to 250MHz to allow the C code to execute faster.

Dependencies

This embedded C relies on two libraries from the Pico Hardware APIs; the standard library to set GPIO pins on the Pico, and hardware clocks to overclock the Pico's system clock.

Initialization Logic

At runtime, the C code first performs some setup tasks:

- Each GPIO pin on the Pico is mapped:

Pin Global Variable Name	Purpose
PIN_OUT	Serial data bus
PIN_CLK	Controller clock generated by the Pico
PIN_RESET	Resets matrices stored on FPGA
PIN_START	Begins serial stream of values

- The base-10 number arrays are converted to base-2 bit arrays using the aforementioned num_to_bit_array function
- The reset pin is driven high, start is driven low, then start is driven high and reset is driven low. We sleep for 100ms here for the memory to be fully cleared on the FPGA.

- The driver loop begins

Driver Loop

The primary goal of the C code is to serially stream the matrix values to the FPGA, which occurs through a counter and a loop:

```
while (1) {

    if (counter < total_bits) {
        if (counter < 72) { // First matrix being sent
            gpio_put(pin_out, A[counter]);
        } else { // Second matrix being sent
            gpio_put(pin_out, B[counter - 72]);
        }
        counter++; // Iterating counter
    } else { // Reset counter back to 0 after both inputs sent
        counter = 0;
    }

    // Clock cycle
    gpio_put(pin_clk, 1);
    sleep_us(1);
    gpio_put(pin_clk, 0);
    sleep_us(1);
}
```

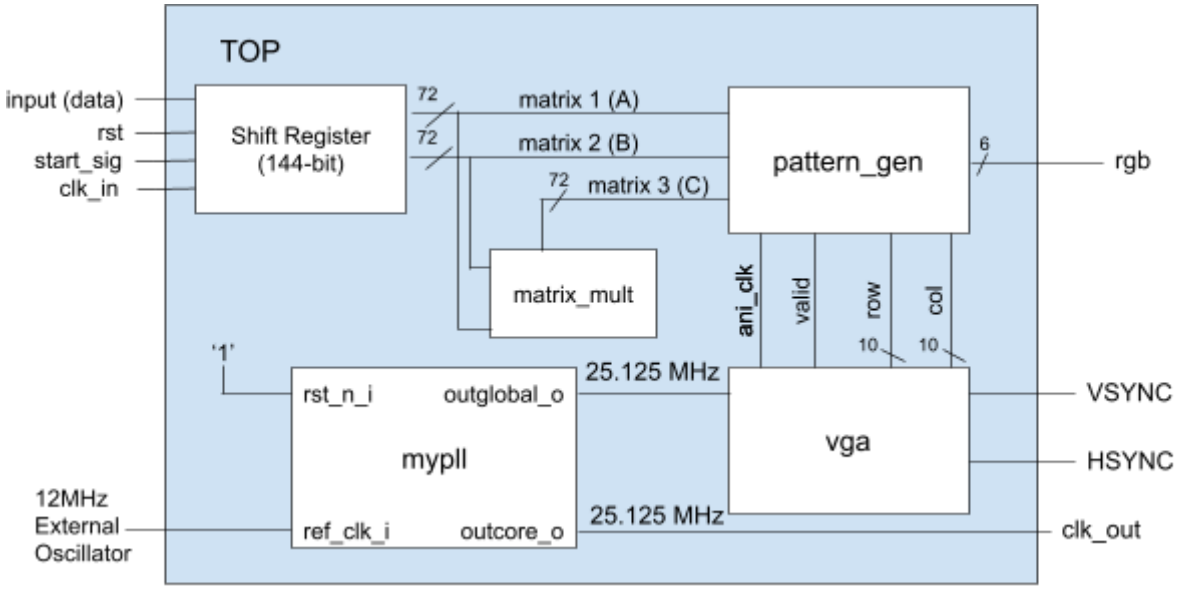
In the driver loop shown above:

- counter sends the first 72 bits (input matrix A), then sends the second 72 bits (input matrix B)
- pin_clk is driven high, then a small sleep occurs, then it is driven low–this simulates a square wave clock signal, used as the controller clock to stream values to the FPGA.

Compilation

The C code must be compiled with the Pico SDK to build a USB Flashing Format (uf2) file to flash onto the Pico.
The microPython solution is not advised for use because it runs significantly slower and microPython is no longer maintained for PyCharm, but it can be opened as a project in PyCharm versions before 2024.3 and flashed onto the Pico through there.

FPGA



Overview

The FPGA is dependent upon the Pico for its functionality, and relies on its clock signal to read in matrix data. The structure of the FPGA may be broken down into four modules, which are illustrated in the above block diagram:

- **top** : This module is the highest-level module, and aside from instantiating the other modules as components, contains the shift registers which write the serial data to memory .
- **matrix_mult** : This module takes the bit vectors storing the matrix data as input and outputs their product.
- **vga** : This module contains several clocks which simultaneously output to FPGA pins the necessary synchronization signals, and drive an internal FPGA signal indicating when it is valid to send graphical data.
- **pattern_gen** : This module dictates the pixel configuration of the output VGA signal based on the coordinates of the output pixel coordinate and the current state of the matrix values in memory.
- **mypll** : This phase-locked loop (PLL) module simply converts the 12 MHz clock of the FPGA into a 25.125 MHz clock used to drive the VGA module.

top

The FPGA uses shift registers to receive the input serial data and write it to memory, shifting in the two matrices multiplied in the operation $A \times B = C$, ("A" and "B") into two 72-bit unsigned bit

vectors. Since the input data bus is 1 bit in width, the total read-in operation takes 144 clock cycles, which is executed in specific intervals using a *count* signal. One additional cycle drives a *pause* signal high to freeze the counting operation after 145 cycles, inhibiting the further writing of data to memory. Without this pause signal, the FPGA would continue to read the current input state as new data to write to memory, and subsequently overwrite both "A" and "B" with the same value.

matrix_mult

The operation of the matrix multiplication module is independent of a clock on the FPGA and relies on combinational logic. This enables the output matrix "C" to be computed as soon as the final bit of matrix "B" is shifted in. Assuming that each matrix has dimensions 3×3 , the each element of "C" is equal to the sum of three products between elements of matrices A and B. Since a multiply operation between two unsigned bit vectors of n-length will result in a product with width of 2n-bits, the value for a single element of "C" must be resized using the *resize* function to 8 bits from16 bits before being written to the 72-bit unsigned bit vector serving as memory for "C".

vga

This module uses the 25.125 MHz clock output from the PLL to implement the accurate VGA timings necessary to drive and animate a display output. The outputs of this module include VGA vertical and horizontal synchronization signals, a signal driven high when a VGA device will be ready to receive pixel data (*valid* signal), and clock which is pulsed high on the last pixel of each frame. This last clock permits frame-dependent counters which may increment synchronously with new frames to dynamically change and animate output signals. This module outputs the horizontal and vertical coordinates of the current pixel as two 10-bit unsigned values.

pattern_gen

This module specifies the output colors within the VGA signal based on the outputs from the VGA module. When the *valid* signal is low, indicating that the monitor is not ready to receive graphical data, this module drives the color data lines in the VGA signal low. When the *valid* signal is high, the output is dependent on the coordinates of the current output pixel output from the vga module. Breaking the visible screen down into coordinate-delineated regions. For example, between the horizontal pixel coordinates of 240 and 294 and between the vertical pixel coordinates of 0 and 54 the output color will be the 6-bit truncated value of the first element of matrix "C". Since the VGA output is limited to 6-bit color, any values of an element of "C" greater than 63 will appear identical.

Results

State

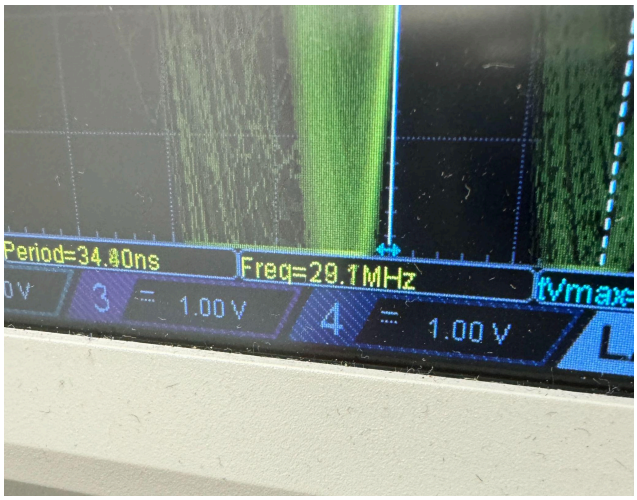
The accelerated matrix multiplier successfully communicates two 3×3 input matrices of 8-bit unsigned numbers from a microcontroller to the FPGA, multiplies them, and displays them and the product matrix using RGB values on a VGA display. For the Pico, we have both a Micropython and an embedded C version, but the embedded C performs significantly better. We exceeded our initial goal of building accelerated matrix multiplication by adding a visual representation of the matrices.

Despite allowing 8-bit unsigned numbers, the visual representation of the matrix uses the least significant 6 bits of each number due to limitations of VGA. These 6 bits encode RGB values, where black represents decimal 0 (000000), all the way up to white representing decimal 63 (111111).

This project serves as a proof-of-concept for accelerating matrix multiplication of larger matrices, which is a fundamental tool for data science and machine learning. Even with a relatively small 3×3 matrix, the FPGA performs significantly better than the equivalent software solution, so for large-scale projects, having accelerated multiplication is crucial.

Timing

Our controller clock frequency appears to be ~20MHz with the embedded C solution:



We could not adjust the scope to smoothly display the square wave from the controller clock, but the frequency stayed around 20-30MHz.

When using the Micropython solution, however, the clock is limited to ~40KHz due to Python function execution overhead. By simply using the built-in Timer library, we are limited by Python's overhead with function callbacks. We have tried to reduce the time within the function itself, but it seems that past 20KHz, the clock speeds slow down significantly for no apparent reason. When we simply switched the code to embedded C programming, we saw marked improvements upon the run time and easily exceeded 1MHz with no further improvements. We believe that the many layers of abstraction present in MicroPython significantly slows down the max clock frequency.

Using the faster 20MHz benchmark, it takes 145 total clock cycles to send both input matrices to the FPGA, and the multiplication on the FPGA occurs instantaneously, meaning 145 total clock cycles are required for the matrix multiplication.

$$145\ cycles\ \times \frac{1}{20000000}\ sec/cycle\ =\ 7.25\ \times\ 10^{-6} sec\ =\ 7250\ ns$$

For a comparison metric, we have a Python script which times the multiplication of two 3×3 matrices using the same hardcoded matrix multiplication method that we use in VHDL. Hardcoding was the fastest solution in Python–nested for loops took about twice as long. This software multiplication takes on average around $1.5\ \times 10^{-5}\ sec\ =\ 15000\ ns$

So even for small matrices with relatively small values, the FPGA allows for significantly accelerated matrix multiplication versus the fastest software solution.

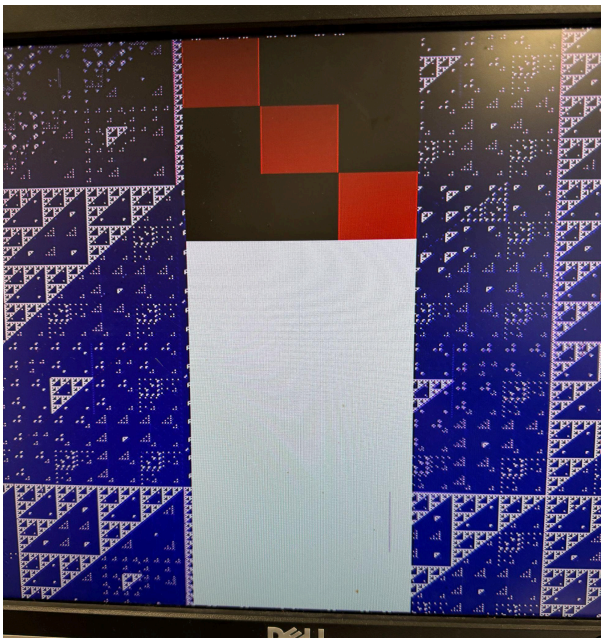
Known Bugs

The controller clock signal does not appear as a discrete square wave from the C solution (it does for the microPython solution). Despite this, data transmission still occurs without error so the FPGA must be interpreting the clock signal correctly still, it just makes measuring the frequency a bit more difficult as it frequently varies between 5-30MHz on the scope.

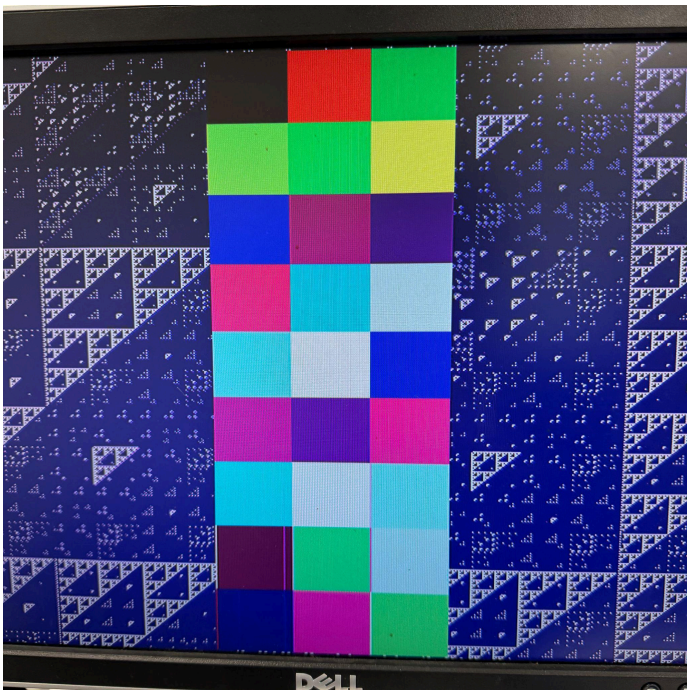
Occasionally, one line of pixels on the VGA display will be the wrong color. These single pixel lines correspond to a single bit of an input number. We initially thought this was an error with our multiplication algorithm, but later realized that the lines can be eliminated by adjusting the "pixel timer" and other display settings on the VGA monitor, so we believe it is just an issue with interactions between the VGA display and the FPGA.

Photos/Videos

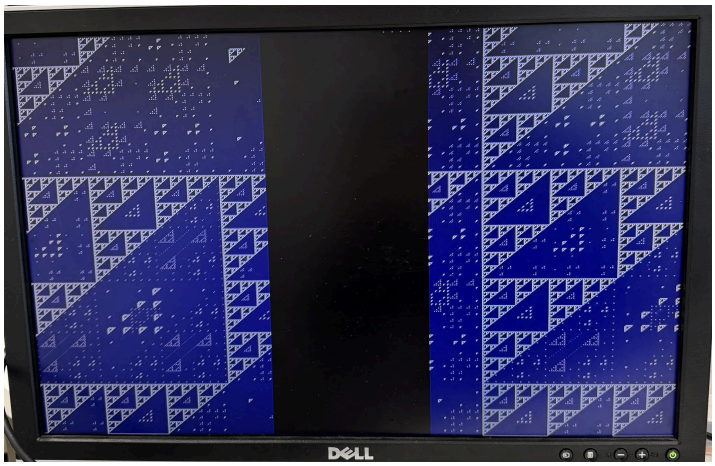
This video first demonstrates a column shift, visualized by the one white square changing position from the input matrix A to the result. Next, it shows the identity matrix times the identity matrix, just producing the identity matrix as a result.



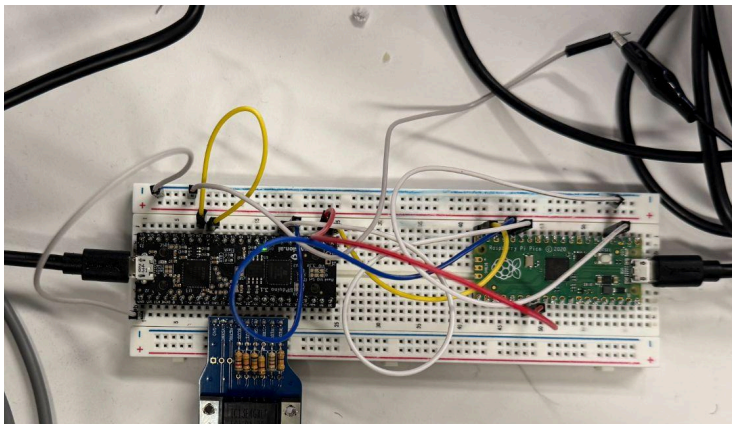
Identity times a "max value" matrix with all decimal 255 (equivalent to decimal 63 on the visual display), producing the "max value" matrix as expected.



Two matrices of random values 0-63, just to demonstrate the different colors. The pixel errors in the result are described in the "Bugs" section above.



The uninitialized VGA display, prior to the Pico passing over any input matrices to the FPGA. The background Sierpinski triangles are just for visual pleasure.



Top-down view of our circuit.

Reflection

In general, the project went very well. We got the accelerated multiplication matrix working as we had hoped, and we even added the ability to visualize the results on the monitor. Most importantly, we were able to shatter the computational time of the same operation done in simple coding languages. One issue we encountered was viewing the clock signal from the C solution–with the Python solution,

we could see a nice square wave clock on the oscilloscope, but there were no discrete waves visible from the C clock. Despite that, we did get frequency readings around 20MHz and the clock successfully sent data serially to the FPGA, so serial data transmission between the Pico and an FPGA was a big achievement of this project. If we had to do the project again, we would not attempt to make a microPython solution for the Pico. We spent a huge amount of time trying to use microPython because it is no longer maintained for PyCharm, so we had to read outdated documentation and use older Python syntax. In the end, even once it was functional our Python solution was much slower than the embedded C solution due to Python's many abstractions.

Some potential future improvements on the project could include displaying the numbers themselves as opposed to mapping them to RGB color values. Additionally, it would be cool to implement dynamic comparisons between the computer's compute time and the hardware's compute time, displaying them on the screen as well. If we hadn't encountered so many set-up/communication issues, we may be able to incorporate matrix multiplication into an interactive game, small machine learning operations, solving systems of equations, etc. Furthermore, we could improve upon the communication set-up to speed up the calculations even more. Instead of communicating serially, we could make use of all of the pins and transmit the data in parallel, potentially decreasing the runtime by over 8 times. We were also recommended to use an ESP32 (quite late into the project), since it had less communication issues, timer problems, and more robust documentation (and TA support). Another improvement we could make is to switch out the Raspberry Pi Pico for an ESP32 for enhanced performance.

Work Division

Ha

- Setting up the PyCharm environment, testing and debugging the microPython code.
- Debugging the VHDL code for the serial communication between the microcontroller and the FPGA.

Jack

- Setting up PyCharm environment and building microPython solution
- Building embedded C solution and compiling with Pico SDK

Camran

- Implemented the vga and pattern_gen modules, as well as the matrix_mult module logic

Michael

- Debugging and troubleshooting VHDL serial communication via shift register; MicroPython pin ON/OFF communication; PiPICO's slow clock frequency output
- Testing intermediate outputs via LEDs and oscilloscopes