ECE 350 Final Project - GPU

Fletch Rydell

I. Overall Design and Specifications

My project's goal was to augment the baseline ECE 350 CPU with a special-purpose graphics unit, with the ability to render semi-realistic 3D scenes as well as realtime graphics. To do this, I created a separate "GPU"[1] core for the ECE 350 CPU.

A. Goals and Non-goals

Because the above is fairly broad, I target two more specific goals for the GPU:

- Render a scene in real time (>20fps) with some interactive component. This will be referred to as the "game," although enjoyable gameplay is not a major focus of the project.
- Render a somewhat realistic 3D scene, including specular and diffuse lighting as well as reflections. The focus here is on render quality rather than latency, so real time animation is an explicit nongoal

While additional scenes are possible to showcase specific functionality, these two serve as the benchmark for what can be considered a "useful" GPU.

B. Project Components

The overall design can be isolated into 4 separate components. For simplicity, data flows linearly through these components in one direction, and they are listed here in this dataflow order:

1. **Input unit**: this is the simplest component, buffering input from the PS2-interface keyboard and exposing it to the CPU through a MMIO interface.
2. **CPU core**: this is the CPU core as designed throughout the course. It processes input for interactive demos and sets parameters (via registers, see Section II.D) for the GPU to control the render. The program run by the CPU is hard-coded to a ROM during synthesis/implementation.
3. **GPU core**: this is a 16-wide half-precision SIMD core that does the graphics processing (as the name implies). Its program (also hard-coded at synthesis/implementation) is referred to as a "shader," and runs once per pixel, writing resulting RGB colors to the framebuffer.
4. **Framebuffer/Output**: this generates the required VGA sync signals (as done in Lab 6) and sends the color data for each pixel over the built-in VGA port at the correct time. Color data is stored in a large RAM (12 bits per pixel), exposing a write port for the GPU to set pixel colors after computing them.

In terms of overall inputs and outputs, the project receives keyboard input via the built-in PS2 interface and outputs via the built-in VGA port. The on-board LEDs and switches are used for debugging, but are not considered part of the overall project interface.

[1] It's arguably more similar to an array processor employing predicated SIMD and one SIMT-like feature than a real GPU with multiple independent processors, but it is a Unit for Processing Graphics, so…

II. GPU Core Architecture[2]

The main component of this project is a 16-wide half-precision-float vector-based Graphics Processing Unit accompanying the CPU. This section details its unique architecture, scheduling, and integration with the rest of the system.

A. ISA

The ISA for our GPU core is described in the table below. All instructions are 20-bit, and most are "simple" (R-type) instructions with the following format:

Opcode	Destination	Source 1	Source 2	Unused
`insn[19:16]`	`insn[15:12]`	`insn[11:7]`	`insn[6:2]`	`insn[1:0]`

There are 4 exceptions, each of which has a unique format described in the table below. The table lists all currently-supported instructions, along with a brief description of their functionality:

Opcode	Instruction	Description
`0000`	`add DEST, SRC1, SRC2`	Adds two registers, setting DEST = SRC1 + SRC2
`0001`	`sub DEST, SRC1, SRC2`	Subtracts two registers, setting DEST = SRC1 - SRC2
`0010`	`mul DEST, SRC1, SRC2`	Multiplies two registers, setting DEST = SRC1 * SRC2
`0011`	`div DEST, SRC1, SRC2`	Divides two registers, setting DEST = SRC1 / SRC2
`0100`	`floor DEST, SRC1`	Floors a register to the nearest integer, ignoring the second source. Inputs in the range $[-0.00048828125, -0]$ are incorrectly floored to -0 rather than -1.
`0101`	`abs DEST, SRC1`	Takes the absolute value of a register, ignoring the second source
`0110`	`sqrt DEST, SRC1`	Takes the square root of a register, ignoring the second source
`0111`	`atan DEST, SRC1`	Computes $\frac{1}{\pi}\tan^{-1}(\mathrm{SRC1})$, ignoring the second source. Uses a 2048-entry lookup table, so the last 4 bits of mantissa are truncated (table entries correspond to the average of the 16 values mapping to the entry).
`1000`	`cmov DEST, SRC1, SRC2`	Sets DEST = SRC1 if SRC2 < 0, otherwise does nothing
`1010`	`done`	Signals that the output pixel color is set in registers 17, 18, and 19, stopping the program. This logically may occur as early as fetch in the 4-stage pipeline, so 3 no-ops should be added after the last write to one of these registers.
`1101`	`bltz SRC1, DESTPC`	Branches to PC = DESTPC if SRC1 < 0. There are several restrictions, described in Section II.E. The value of DESTPC is only 11 bits, the first 4 of which are `insn[15:12]` and the latter 7 are `insn[6:0]`.

[2]The rubric asks for processor modifications. Here, CPU modifications are limited to the MMIO interface for input and GPU registers, so we instead discuss the GPU's architecture, which technically could be considered an extreme modification to the processor.

1110	`setx COUNT`	Sets the X register (for looping) to COUNT, so an upcoming loop repeats COUNT times (for COUNT+1 total executions).
1111	`loop DESTPC`	Decrements the X register and sets PC = DESTPC if the original value is at least 1. Together with `setx`, this allows constant-iteration loops to be easily implemented.

B. Pipeline

The GPU's pipeline is somewhat similar to the standard 5-stage pipeline, but with the memory stage omitted because the GPU *does not* have any ability to access memory (due to 16-ported global memory being hard to synthesize). This consists of the following stages:

- Fetch: load the instruction from the current PC. Unlike the traditional pipeline, a few instructions are actually completed in this stage: `setx`, `loop`, and `done`. These instructions do not interact with registers in the standard way, so they complete in Fetch to avoid unnecessary stalling (note that this allows loops to be done with no stalls). The only non-scalar (i.e. scaling with SIMD width) part of this stage is the quasi-SIMT branching optimization described in Section II.E.
- Decode: load the two inputs from the register file. The MSB of the source determines which register file is loaded from (see Section II.D). Bypassing also occurs at this stage, with bypassed operands being latched into execute to avoid adding additional overhead to the compute-heavy execute stage.
- Execute: computes the result from the input operands. Most simple instructions use Vivado's provided floating point IP, with the exception of `abs`, `floor`, `atan`, and `cmov`. The `cmov` and `bltz` instructions simply check an input sign bit, either setting the destination register to $16 or setting the minimum executed PC as described in Section II.E, respectively. This stage is entirely vector, with all logic being duplicated for each SIMD vector element.
- Writeback: writes the result from execute to the destination register as long as it is nonzero. Note that, as described in Section II.D, only registers $17-31 are writeable by the GPU core, so the MSB of the destination register name is always asserted.

C. Scheduling

The GPU continuously renders the image, scanning through 16 pixels at a time. Each execution of the shader program on 16 pixels is referred to as a "dispatch." Each dispatch consists of a horizontal line of 16 pixels, and the x- and y- coordinates of the dispatch are considered to be those of the leftmost pixel in the dispatch. The x-coordinate ranges from −320 to +319 left-to-right, and the y-coordinate ranges from −240 to 239 top-to-bottom.

When the `done` instruction is seen, the scheduling logic extracts the color values from the output registers 17-19 and writes them to the framebuffer as described in Section II.F. On the next clock cycle, the GPU is reset to PC 0 and the next dispatch begins. Registers are not reset, so programs cannot take advantage of any initial register value without setting it on the previous dispatch.

D. Regfile

The GPU register file logically includes 32 registers divided into 2 groups: register 0-15 are scalar register and 16-31 are vector registers. Because all GPU instructions act on vectors, only the latter are writable by the GPU (with the exception of 16, see next paragraph). The scalar registers are instead exposed through the GPU interface to be written by the CPU, allowing parameters of the render (time, positions, colors) to be controlled.

Registers 0, 1, 2, and 16 contain special values and are non-writable. Register $0 is the constant 0, $1 holds the current dispatch's y-coordinate, and $2 holds the current dispatch's leftmost x-coordinate. (Vector) register $16 holds the index of the vector for each element (i.e. the numbers 0 through 15). This allows the expression `add $rd, $2, $16` to compute each pixel's x-coordinate in `$rd`.

The 16 scalar registers are stored in a central register file to allow writes from the CPU. The 16 vector registers, on the other hand, are distributed element-wise, with each execution core holding its own register file.

E. Control, Predication, and SIMT

My GPU supports 3 types of control: static looping, conditional moves, and "quasi-SIMT" forward conditional branches. Static looping support is supported by the `setx` and `loop` instructions described in Section II.A.

Conditional branching is more complex, as the vector nature of the core prevents traditional register-based branching due to the possibility of different elements in a dispatch diverging. This would traditionally be handled with predication (an "associative processor" in Flynn's taxonomy), but I implement a weaker form using conditional moves only. Rather than conditionally skipping a basic block or predicating away all instructions within it, shaders must use scratch registers for potentially-unwanted computation, conditionally moving results back into the desired location with `cmov`. Most conditional computing in my shaders utilizes this primitive.

My GPU does implement one conditional branch: `bltz` or branch less than zero. When an element conditionally branches based on this, it sets a "minimum executed PC" register local to its execute stage to the new program counter. Future instructions are then ignored until this program counter is reached, emulating a forward conditional branch.

As an optimization, the fetch stage looks at the minimum executed PC registers for each element and finds the minimum among them. If this value is greater than the current PC, we skip forward to this value, as no instructions prior to it are executed. This speeds up execution in two very common cases: if every element in a dispatch branches past a block, the whole dispatch "executes" (stalls) only a few instructions before skipping past as a whole, and if a branch is used to break out of a loop, the whole dispatch breaks out once all individual elements have done so.

Because this conditional branch logically causes each element to execute a different series of instructions, I call it "quasi-SIMT (Single Instruction Multiple Thread)," inspired by modern GPUs' approach toward branches of allowing each data value to logically follow an independent thread while using masking to resolve intra-dispatch conflicts. My implementation has several restrictions due to its simplicity:

- It only works on forward branches, more formally with the restriction that the control flow must satisfy the following constraints for any combination of taken/not-taken branches:
  - Control flow must at some point reach the destination PC
  - All instructions after reaching the destination PC must statically appear later than the destination (i.e. must have a higher PC)
  - All instructions executed prior to reaching the destination PC must statically appear earlier than the destination (i.e. must have a lower PC)
- No `done` or `setx` instructions may be skipped by such a branch. `loop` instructions may be skipped, but the previous restriction ensures skipping and executing such loops are equivalent when the branch occurs (as all instructions "inside" the loop will be predicated away).

- The `bltz` instruction only has 11 bits for the destination PC. A smart microarchitect would make the branch relative, but I didn't bother to while implementing it, so conditional branches become useless for long programs. Users should resolve this by writing shorter programs (if you choose to write a >2,000 instruction for this thing, conditional branch limitations are the *least* of your problems).

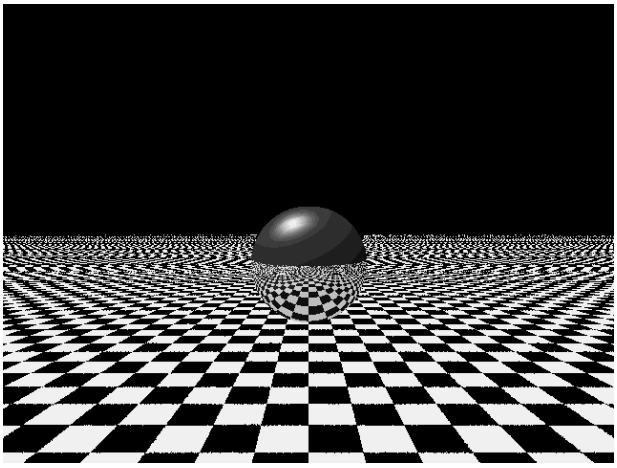
F. Color conversion / Dithering

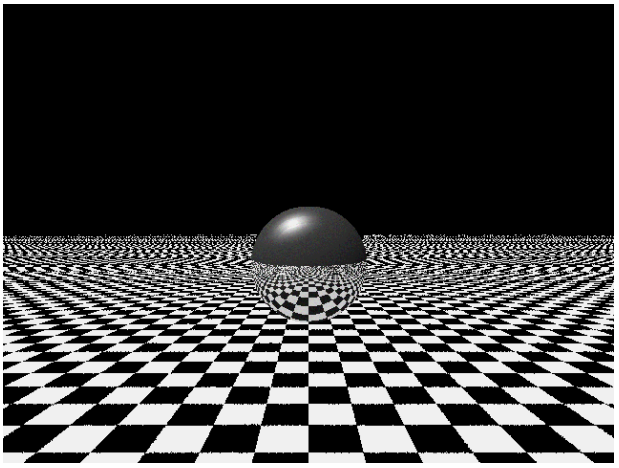
Once the shader program executes the `done` instruction, the 16-bit half-precision floating point values in registers 17, 18, and 19 are converted to three 4-bit VGA channels for compatibility with the Nexys A7 interface. To avoid color banding due to the lack of depth, randomized dithering is used, producing a smoother color gradient.

Color conversion begins with using Vivado IP blocks to convert each half-precision float value into an 8-bit signed integer. Then, the lower 3 bits are compared to a pseudo-random 3-bit number, and if they are greater, 1 is added to the upper 4 bits before outputting; otherwise, the upper 4 bits are taken as output. There are two exceptions to this algorithm: if the number is negative, 0 is output, and if it is greater than $\texttt{01111000}_2$, the maximum is output (ignoring the comparison to avoid oveflow).

The pseudo-random numbers are generated with a 144-bit LSFR (for 16 pixels in a dispatch each containing 3 channels needing 3 bits of randomness). Bits 143, 141, 139, and 136 (zero-indexed) are XORed to create a new LSB each cycle, in accordance with Ward and Molteno's table of maximum-cycle LSFRs.[3]

A comparison of non-dithered and dithered renderings is shown below:





III. Other Components and Integration

While the GPU core is the main component of this project, there are a few others, in both hardware and software, I wanted to document here.

A. Keyboard Interface

The only user input device is a PS/2-interface keyboard connected to the Nexys A7's onboard USB port. The PS/2 protocol is handled with the provided code from Lab 7, which presents 8-bit data words and a ready signal.

[3] https://datacipy.cz/lfsr_table.pdf
Archived Version: https://web.archive.org/web/20231031092924/https://datacipy.cz/lfsr_table.pdf

Some basic sequential logic translates these signals into a single register containing the currently-held key code. On a keypress, this register is set to the corresponding scancode. On release, keycode `0xF0` indicates that the following key was released, so our logic simply sets the register to 0 and ignores the subsequent word. This fails to correctly handle overlapping or simultaneous keypresses, but is sufficient for our purposes.

B. CPU MMIO

The CPU reads input and writes to GPU scalar registers through a basic MMIO interface. Since the standard ECE 350 CPU only uses 4096 words of RAM, addresses above `0x1000` are used to communicate with the rest of the system.

GPU registers 3-15 are mapped to addresses `0x1003` through `0x100f`. When a write is performed to this location, it is converted from a 32-bit integer assumed to be made up of 16 integer bits and 16 fraction bits to a half-precision float.[4] From the CPU side, this offers a reasonable compromise, with values from 0.015625 to 32768 representable at least as well as half-precision floats, smaller values being limited to $1.5 \cdot 10^{-5}$ absolute precision (greater than the 10 bits of *relative* floating point precision), and larger values being unrepresentable.

Memory address `0x1000` is mapped to the keyboard interface described above. Loads from this address get the currently held key (with 24 leading `0`s), and any store to this address clears the key, allowing an already-processed keypress to be subsequently ignored.

C. Output Module

The output module generates VGA sync and data signals to display the data in the framebuffer onscreen. Because the GPU processes 16 pixels simultaneously, the framebuffer also stores colors for 16 pixels in each word (for more details on the framebuffer, see Section IV.A). This enables all results from the dispatch to be written to the framebuffer in one cycle.

This design slightly complicates the VGA output module's logic. The provided VGA interface from Lab 6 produces x and y coordinates of the current pixel, as well as the necessary horizontal and vertical sync signals. Using these coordinates, the formula `(y << 5) + (y << 3) + (x >> 4) + 1` gives the address of the next upcoming block of 16 pixels in the framebuffer. Once the leftmost pixel in a dispatch is reached, the data at this location is stored into a shift register, which shifts 12 bits with each 25 MHz pixel clock cycle. The current pixel color is therefore always in the last 12 bits of this shift register, which are sent to the built-in VGA port.

D. Assembler

To support the new GPU ISA, I wrote a basic assembler in Python. It deals with the strange format of simple instructions (i.e. 4-bit destinations and 5-bit sources), but does not handle branches and jumps.[5]

Instead, the assembler outputs labels along with the generated binary, along with annotations at each branch and jump. This makes it fairly easily to manually set branch targets. Not ideal, and I've planning on fixing it for weeks, but nothing's more permanant than a temporary fix!

[4]Technically, values are converted to single-precision floats and then half-precision floats. This is because the Vivado IP Catalog fixed-to-float converter doesn't allow conversion to half-precision floats. I know, I'm as upset about it as you! That's *dozens* of LUTs I'll never get back!

[5]Let's pretend it's for performance reasons: I can't afford a *second pass* when assemblying these 30-100 instruction files! That's *dozens* of microseconds I'll never get back!

IV. Challenges

As with any project, there were dozens of medium-to-large challenges I faced in this project. Unfortunately, most were small typos or similarly silly screw-ups that aren't worth describing here. I instead describe two of my more interesting challenges and solutions.

A. BRAM Limitations

The Artix-7 FPGA includes 4,860 Kbits of block RAM.[6] Because my framebuffer is made up of $640 \cdot 480 \cdot 12 = 3,686,400$ bits of data, I initially assumed RAM usage would not be a problem.

Unfortunately, the FPGA BRAM is divided into 135 36K BRAM blocks, each containing 32768 bits of data (the 36K advertised capacity includes parity disabled here) addressable in several configurations: 1x32K, 2x16K, 4x8K, and so on, where NxMK means a depth of MK and a word size of N bits. We needed a framebuffer with $\frac{640 \cdot 480}{16} = 19200$ words each of size $16 \cdot 12 = 192$ bits.

Because the depth of this RAM is greater than 16K, the synthesized configuration uses the 32K configuration with each bit taking a full BRAM block. This requires 192 BRAM blocks in total, more than our chip has.

With this understanding of the BRAM allocation mechanism, I realized that the framebuffer could be divided into two parts: a 16384-entry main buffer and a 2816-entry secondary one. This requires only 96 BRAM blocks for the main buffer (each 2x16K), and 24 for the secondary (each 8x4K). The only downside (and reason this isn't synthesized automatically) is the extra mux for reads and combinational write-enable logic it requires.

With this change, 120 BRAM blocks are used for the framebuffer. With the addition of `atan` lookup tables and the CPU and GPU register files, a total of 130.5 BRAM blocks are utilized, about 97% of the FPGA capacity.

B. Debugging

Testing methodology is discussed more in Section V, but this challenge concerns bugs that escaped testing and were not apparent until flashing the design onto the FPGA. This happened fairly often, as even 10 frames of rendering represents millions of GPU instructions to step through, and glitches with user input occur at the 0.1-1.0 second level, not in a 0.00001 second simulation.

Because there was is so much state to keep track of, and re-implementing a design with different debug probes or outputted signals on LEDs takes about 10 minutes, my debugging logic gradually developed into the following:

```
assign LED[15:0] = SW[4:0] == 5'd0 ?
      SW[15] == 0 ? regfile_scalar[SW[8:5]] : {8'd0, kb_data}
      : SW[15] == 0 ?
        cpu_rs1 == SW[4:0] ? cpu_regA[15:0]
          : cpu_rs2 == SW[4:0] ? cpu_regB[15:0]
          : 16'd0
        : cpu_rs1 == SW[4:0] ?
          cpu_regA[31:16]
          : cpu_rs2 == SW[4:0] ? cpu_regB[31:16]
          : 16'd0;
```

[6] See https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

Beautiful, isn't it? Scalar GPU registers are easily accessed by the last 5 switches, and viewing the contents of the CPU register file (which only has 2 read ports due to the structural Verilog implementation for Checkpoint 2) is accomplished with persistance-of-vision!

V. Testing

Testing this project mostly consisted of rigorous testing of each component part, as outlined in the following subsections.

A. GPU Core Testing

The main issue with testing the GPU core is that Icarus Verilog does not support the specialized Vivado Floating Point IP blocks. So, I wrote a "stub" floating point unit (execute stage) that writes all inputs to a file and reads outputs from another. By repeatedly running a test in Icarus and using a basic C program to compute the correct floating point unit outputs, the correct inputs and outputs would eventually stabilize (with number of runs bounded by the number of executed instructions). While not ideal, this served as a good way to validate the overall structure of the GPU core itself.

B. Output and Scheduling Testing

After the basic GPU core was complete, I integrated it with the output/framebuffer module and wrote the scheduling logic. Testing this involved Vivado Behavioral Simulation of a basic program generating a gradient. Once the output was correct, the gradient appeared on the VGA monitor, validating scheduling logic.

C. CPU Integration Tests

Once the CPU was integrated, simulating one end-to-end test became very difficult. Luckily, a basic test program displaying different GPU scalar registers made it clear that all registers were correctly writable through the MMIO interface.

D. GPU Program Emulation

For individual shader programs for demos, I wrote an emulator (in Python, with numpy providing passable performance) of the GPU ISA, allowing programs to be tested without synthesizing/implementing a new design. This allowed programming to happen before the GPU was complete, sped up iterations when tweaking programs, and provided a playground to try out and measure performance improvements (such as forward branching).

E. Timing Validation

Beginning with the initial GPU core and continuing through full integration, I ran Vivado timing simulations after major changes to ensure clock constraints were met. Unfortunately, the use of a single-cycle execute stage limited the clock speed to 25 MHz. The Vivado IP floating-point division was the limiting factor.

VI. Assembly Programs / Demos

I wrote 3 demos to showcase the generality and functionality of my GPU. The first is a basic game, the second a properly-lit raymarched 3D scene, and the third a basic Mandelbrot set fractal explorer. A photo of the 3D scene can be found in Section II.F, and the other two are shown below.

A. Tunnel Running Game

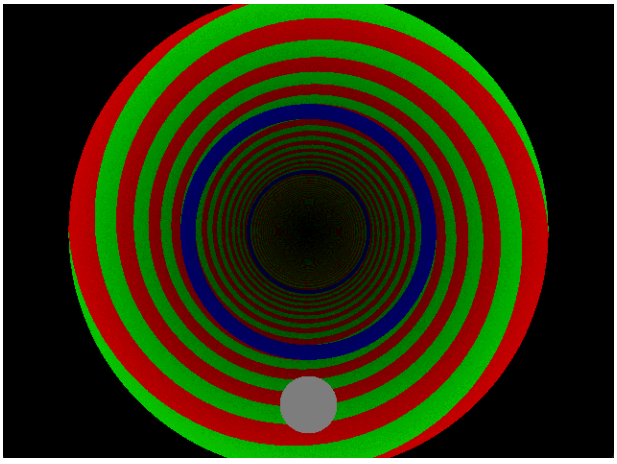
In this demo, a 3D-looking red/green spiral tunnel is rendered by the GPU, with blue rings periodically advancing along the z-axis. The player is represented with a gray circle, and the goal is to jump over each blue ring. The CPU assembly program runs a basic startup animation, randomly generates ring locations with a Linear Congruential Generator with ZX11 parameters, detects keyboard input to trigger player jump, simulates gravity while the player is in the air, and checks if the player fails to jump over a ring and resets the game on a loss. The GPU renders the tunnel with basic trigonometry and overlays the player on top.

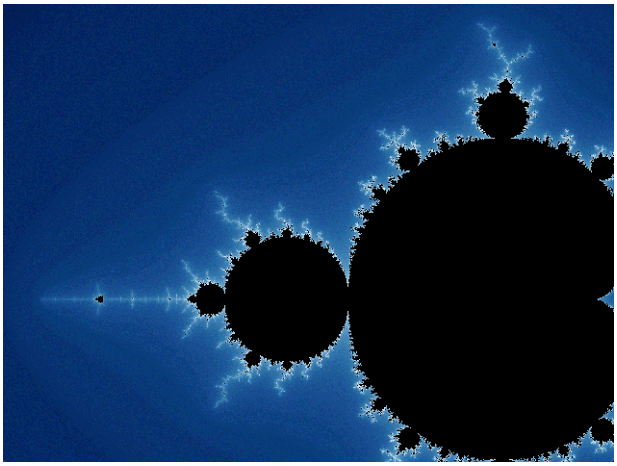
B. Raymarching Sphere Demo

In this demo, the GPU uses a basic raymarching loop to render a sphere sitting on a checkerboard floor. While the intersection of a ray with the sphere could be evaluated with a closed-form expression, raymarching was used to provide an extensible template for more complex shapes. Once the raymarching is complete, a ray hitting the sphere calculates the reflected ray to add Phong lighting along with basic diffuse lighting based on the sphere normal. The reflected ray is then cast to the floor to reflect the checkerboard floor onto the sphere. If the ray misses the sphere, the reflection is skipped with a forward branch, so the floor/sky color is computed for the original ray. Unfortunately, this method causes inconsistent rounding in the checkerboard floor, as rays missing the sphere use their new position to calculate the floor color, which will vary as the sphere moves. The CPU program just initializes GPU registers and moves the sphere's vertical position gradually up and down.

C. Fractal Explorer

To showcase the maximum floating-point throughput of the GPU as well as better use of color, I wrote a basic Mandelbrot set renderer for the GPU. Iteration count is interpolated with a linear function of distance of the escaping point from the origin, somewhat reducing banding (a log function is needed for a smooth iteration count). This iteration count is translated to a color along a white to blue gradient and output to the screen. Additionally, the corresponding CPU program allows for navigation and zooming with the keyboard.





VII. Possible Improvements

Given that this was a one-person 3–4 week project, there are several improvements I would like to have made but simply did not have time for (both due to time constraints and significant wastes of time). This section details several of the most tempting improvements, from broad architecture changes to small incremental improvements.

A. Width

The total project uses only 35% of the available LUTs on the Nexys A7 FPGA. Increasing from 16-wide to 32-wide should therefore be feasible, which would provide a 100% speedup. Unfortunately, the `atan` lookup table currently is implemented in BRAM for each element, so increasing width would be limited by BRAM limitations. Switching the lookup table to distributed ROM and/or multi-porting it would allow for this optimization to be done fairly easily.

B. Execute Pipelining

I used asynchronous floating point operations to make the GPU pipeline easier to implement. Using clocked floating point operations would significantly increase possible clock frequency. Additionally, Vivado's Floating Point IP includes pipelined units, which would allow such frequency increases without changes to throughput. While this would significantly complicate the GPU pipeline, it would unlock significant performance improvements, as well as allowing more complex instructions (such as non-lookup-based trig).

C. Framebuffer Retrieval

Currently, data flows strictly one way through this system: from CPU to GPU to framebuffer. Enabling the framebuffer to be read by the GPU would allow for several interesting possibilities. At its most basic, cellular automata and/or sand games could be run. With some scheduling logic changes, a course-grained render could be followed up by a higher-resolution refinement pass. This would be a fairly simple way to add many new capabilities to the project without the complexity of full memory access.

D. Precision

In both the raymarching and fractal demos, floating point imprecision limited detail/zoom sooner than performance. This indicates that increasing to (or separately supporting) single-precision floating point could be worth the latency increase.

E. Left Edge

Currently, the first dispatch of each row of pixels does not render properly. I didn't notice this bug until the last few days of work, as the VGA monitor adapts to this and crops off the left edge. However, this would be an interesting (or painful) bug to investigate.

F. Assembler Improvements

My assembler is mostly human-powered, with no branch, label, or even `setx` suppport. This should have been fixed during the project (it would have saved far more time than it took), but it's easy to ignore infrastructure with a deadline approaching.

G. Memory

This is the largest difference between my project and any serious processor—wide memory access is one of the hardest problems in any vector processor or GPU, and I sidestepped it completely with my split

register file and simple demos. At the very least, implementing per-element scratchpad memory would be fairly easy to do and greatly beneficial (the raymarching demo was severely bottlenecked by the number of registers). More general memory would be difficult, but interesting. Alas, doing so would be more akin to starting again than improving this GPU; after all, it's hard to access memory when you don't even have integer instructions/memory!