Minimalist GPU on an FPGA
Final Report

Laker Newhouse
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA, USA
lakern@mit.edu

Hanfei Cui
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
hfcui@mit.edu

***Abstract*—We present a design and implementation for a minimalist GPU on an FPGA. Our GPU features 16x parallelism with 16-bit fixed point operations, capable of rendering the Mandelbrot fractal with 16 colors at a resolution of 102,400 pixels in 0.37 seconds. Our GPU is driven by a general-purpose, programmable controller using a custom instruction set architecture, making it possible to program a variety of linear algebra operations, such as matrix multiplication, in an assembly-style language. Our lightweight compiler automatically converts this program into machine code for the GPU.**

Code available at https://github.com/Arongil/FPGA-GPU

1. Introduction

Parallel architectures with high throughput are becoming increasingly important. GPUs (graphics processing units) are specialized hardware that can perform parallel operations for rendering imagery. They can also be used to speed up training and inference of machine learning models. Our goal for this project was to design and implement a minimalist GPU on an FPGA. We achieved the following goals from our project checklist:

1) Design an ISA to control the GPU
2) Multiply 2×2 matrices
3) Render the Mandelbrot set
4) Scale up to 16x parallelism

2. GPU Design

Our GPU has three core components:

1) Controller
2) Main Memory
3) FMA Blocks

The controller executes assembly-style programs with control flow to orchestrate parallelism across the GPU. The controller sends commands to the main memory, telling it when to push data to the fused multiply-add (FMA) blocks for computation. In parallel, the FMA blocks then perform the fused multiply-add operation $A \times B + C$. The results are sent to an output buffer, which waits until it receives three consecutive outputs from each FMA. Then the output buffer flushes data back into main memory, where the controller can use it for the next operation. In order to render the Mandelbrot fractal on hardware, we additionally implement a dual frame buffer for seamless transitions across frames.

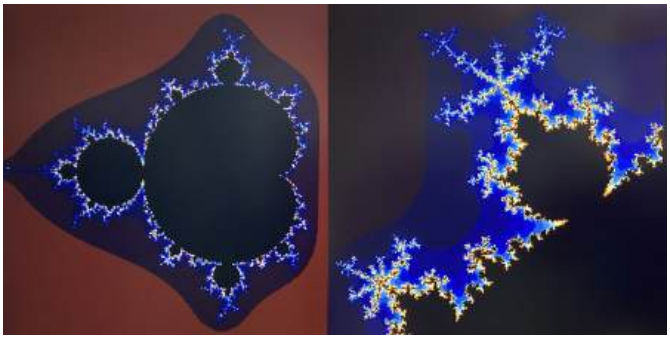

Figure 1. Mandelbrot fractal rendered with our GPU at two levels of zoom.

See Figure 3 for a full block diagram of our design.

2.1. Controller: Programming the GPU

We designed a custom ISA (instruction set architecture) for our GPU, along with a lightweight pre-processor and assembler to convert human-readable, assembly-style program files into machine code that the GPU can read. To execute program logic, the controller module acts as a finite state machine to run ISA commands like a small CPU, with 16 private registers of 16 bits at its disposal.

For example, the following instructions describe a loop:

```
xor 0 0      # zero out register 0
addi 1 0 7   # set register 1 to 7
addi 0 0 1   # increment register 0
bge 1 0      # check 7 >= register 0
jump 2       # if so, loop back
end          # finish when loop done
```

The lightweight pre-processer and assembler would translate these instructions into the machine code below:

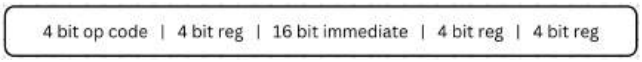

Figure 2. Our ISA supports 32 bit instructions with an op code (4 bits), three registers (4 bits each), and an immediate value (16 bits).

```
0x20000000
0x31000700
0x30000100
0x41000000
0x50000200
0x10000000
0x00000000
```

The `bge` and `jump` commands, used above, introduce control flow based on register values, making the controller Turing-complete. Other ISA commands such as `load` and `write` send data to the FMAs in a single-instruction multiple data (SIMD) paradigm, enabling parallelism.

The controller reads commands in from an instruction buffer implemented as a dual-port BRAM. Because BRAM reads take two cycles, the controller executes one command every two cycles. In the future, we could use prefetching to execute one instruction per cycle.

The pre-processor presents three main quality-of-life features. First, it allows the user to use comments (delimited by `#`) and newlines freely in the program. Second, it can an convert immediate number, written like `-1.25f`, into its corresponding 16-bit fixed point number, such as `0xFB00`. Third, the pre-processor makes it easier to use jump commands. The problem is that `jump` takes an immediate line number, but line numbers are not stable as the program changes over time. Our solution is to allow the programmer to place jump markers, for example `[Jump Marker 123]`, in the comment after a line in the program. Then, anywhere in the program, `jump (((123)))` is processed to substitute the correct immediate value to jump to the marked line.

See Table 1 for a full list of commands in our ISA. In the sections below, we describe command implementations at a high level. We refer the reader to our code for full details.

2.2. Memory: Input Buffer and Output Buffer

The essential problem that memory must address is storing intermediate values and allowing a control flow to move data between the computation units. For this storage, we chose not to use a BRAM because BRAMs are not compatible with wide SIMD instructions, only allowing at most two read and write addresses at once. Instead, we store two registers of width $768 = 3 \cdot 16 \cdot 16$ bits, called the input buffer and output buffer.

The input buffer stores A, B, and C values for each FMA. The output buffer captures FMA output values. In particular, we chose to make the output buffer store three previous outputs from FMAs. These values persist until each FMA has outputted three new values. This persistence is important so that we can access previous results for multiple new computations, as the input buffer is reset with new data every cycle. Thus, in order to use both old results and new inputs, we sandwich the FMA blocks between two buffers.

There are two ways to put data into the input buffer. One way is to load values using `load`. The `load` command sets the A, B, or C values of all FMAs at once. It takes a register and an immediate `diff`, setting FMA values to the register value plus `fma_id` $*$ `diff`. The other way, using `loadb` (load from buffer), is to push previously computed results into the input buffer. This case is particularly useful when computing the Mandelbrot set. We developed a novel shuffle functionality to be able to rearrange previous results to set up the next computations. Using one shuffle, we can permute the previous three outputs from each FMA, optionally multiplying each number by 2 or -1.

In this way, the FMA inputs come directly from the program at first, but then FMA outputs can be cycled through the output buffer back into the input buffer to build new computations. We do not have extra delay from BRAMs because all values are stored in registers. In fact, as a design decision, we decided we needed to not rely on BRAM because it is unable to support parallel computation.

The memory module is a state machine that executes the instruction sent from the controller every other cycle. It is important that memory not execute instructions during the controller's off cycle, or else FMAs can spew out two duplicate values, disrupting the output buffer.

2.3. FMA Blocks: Fixed Point Arithmetic

We implement fused multiply-add (FMA) blocks using 16-bit signed fixed-point numbers with 2^{-10} fractional precision. The FMA block is designed to support chained addition, such as in dot products. To do so, the module stores internal registers for its A, B, and C values so that it can reuse values. A flag called `replace_c` controls whether the FMA reuses its C value when memory writes new data, or whether it uses the new C value. Another flag called `output_can_be_valid` controls whether the FMA will output its result. To implement chained additions, we can hold `replace_c` high for the first addition then put it low, while holding `output_can_be_valid` low until the last addition when we put it high.

To multiply two fixed-point numbers, we can treat them as regular integers and then shift to the right by the number of decimal places, which here is 10. To compute the regular multiplication, the FMA performs a full precision 32-bit multiplication `mult` combinatorially. Every cycle, if its compute flag is high, it sets its output register to `mult >> 10`. To keep track of signedness, the FMA casts its input wires with `$signed`. To prevent a leaky interface, the FMA casts its output logic using `$unsigned`.

We instantiate 16 FMA blocks for 16x parallelism, wiring up all FMA outputs to the single output buffer. We assign each FMA a unique `fma_id` from 0 to 15. The SIMD instruction `LOAD` can use `fma_id` to load a different value into each FMA with a single instruction. We prefer this

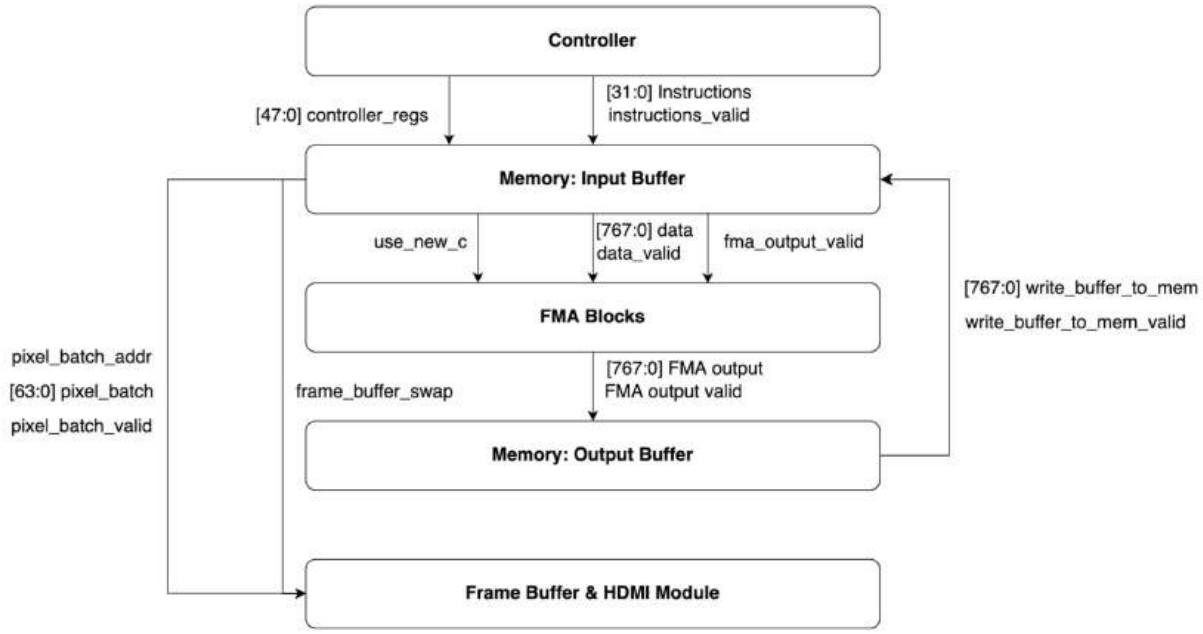

Figure 3. Full block diagram of the GPU.

approach to the alternative, which is to load data into one FMA per instruction, defeating the purpose of parallelism.

With the controller, memory, and FMA modules wired together, the GPU is ready to use. In the next section, we present a case study to better understand the way our GPU can be used in practice.

2.4. Computing the Mandelbrot Fractal

The Mandelbrot fractal is defined on the complex plane. We overlay the complex plane on our display, associating each pixel (x, y) with a complex number

$$z_0 = \frac{x - x_{\text{min}}}{S} + \frac{i(y - y_{\text{min}})}{S'},$$

for some scale factors $S, S' > 0$ and window boundaries $x_{\text{min}}, y_{\text{min}}$. The Mandelbrot iterative relation is defined as

$$z_{n+1} = z_n^2 + z_0.$$

A complex number z_0 is defined to lie inside the Mandelbrot set if $|z_n| < 2$ as $n \to \infty$. The most common tractable approximation is the escape-time algorithm, which defines z_0 as inside the Mandelbrot set if $|z_N| < 2$ for some large integer N. We use N between 64 and 128.

In order to perform complex number arithmetic on the FPGA, we break down a complex number into its real and imaginary components, $z_0 = x_0 + iy_0$, where we represent x_0 and y_0 as 16-bit fixed point numbers. We iterate using the following relations:

$$x_{n+1} = x_n^2 - y_n^2 + x_0,$$
$$y_{n+1} = 2x_n y_n + y_0.$$

If $x_N^2 + y_N^2 \geq 4$, we record that pixel as outside the Mandelbrot set. To add color to the fractal, one common technique is to track how many iterations it takes before a pixel diverges. If a pixel takes longer to diverge, we color it a brighter color along a gradient. If a pixel is inside the Mandelbrot set, we color it black. These colors give rise to the lightning-like patterns visible in Figure 1.

We use a lookup table with 16 colors, indexed by `iters >> 3` for `iters` ranging from 0 to 127, or `iters >> 2` for `iters` ranging from 0 to 63. Each pixel requires 4 bits to store its iterations in the frame buffer's BRAM. With a 320 by 320 display, the dual frame buffer uses 819,200 bits, or 102,400 KB. To save space, we bitshift `iters` down and make the 16^{th} entry of the color gradient black. As a result, points that diverge in the sixteenth part of the range of `iters` are incorrectly counted as inside the Mandelbrot set. Fixing this imprecision would require storing another bit per pixel.

2.5. Parallelizing Mandelbrot on our GPU

Let us dive into the specific way we use our ISA to perform the Mandelbrot computations in parallel.

We parallelize along the y-axis, computing iterations until divergence for 16 pixels at once. In our assembly-style program, we use registers, branches, and jumps to create two outer for loops iterating over all pixels in the screen.

Before we enter these loops, we set registers for `width` (320), `height` (320), `x_min` (-1.875), `y_min` (-1.25), `dx` (0.0078125), `dy` (0.0078125), `16*dy` (0.125), and `max_iters` (63). We additionally set registers for `x_val` (initialized at `x_min`) and `y_val` (initialized at `y_min`).

Instruction	Syntax	Description
NOP	nop	Do nothing
END	end	End execution
PAUSE	pause	Pause execution until user continues (btn[1])
XOR	xor a_reg b_reg	XOR a_reg and b_reg, place result in a_reg
ADD	add a_reg b_reg c_reg	Add b_reg and c_reg, place result in a_reg
ADDI	addi a_reg b_reg imm	Add b_reg and immediate imm, place result in a_reg
BGE	bge a_reg b_reg	Set special register compare_reg to 1 if a_reg $\geq$ b_reg
JUMP	jump line_num	Jump to instruction at line_num if compare_reg is 1
LOADI	loadi reg_a imm	Load immediate imm into memory at word index reg_a
LOAD	load abc b_reg diff	Load b_reg_value + fma_id * diff into each FMA's A, B, or C value
LOADB	loadb shuffle1 shuffle2 shuffle3	Shuffle each FMA's last three outputs and load into its A, B, and C values
WRITE	write replace_c output_result	Tell FMA blocks to compute, optionally chaining old C values or not outputting
OR	or iters	Store iters for each FMA if its C value is greater than 4 (Mandelbrot-specific)
SENDITERS	senditers a_reg	Send iterations to the dual frame buffer (Mandelbrot-specific)
FBSWAP	fbswap	Tell the dual frame buffer that the current frame is ready to use

TABLE 1. FULL INSTRUCTION SET ARCHITECTURE FOR THE GPU, CONSISTING OF 15 INSTRUCTIONS.

Once we are inside the two loops, we want to iterate sixteen pixels in parallel according to the Mandelbrot formula. Recall that memory's output buffer stores the previous three FMA outputs before flushing its contents into memory's write buffer. For each FMA, we maintain a useful invariant for the three values in the output buffer:

1) x_i
2) y_i
3) $x_{i-1} \cdot x_{i-1} + y_{i-1} \cdot y_{i-1}$

To set up the invariant, before we enter the inner iterations loop, we use LOAD commands to set all FMA C values to x_0. All pixels are at the same x_0 value because we parallelize over columns, so we use a diff value of 0. Then we WRITE. But because each FMA should compute a separate y_0 value in parallel, next we call LOAD with y_0 and a diff of dy. We write WRITE again. Lastly, for the magnitude, we compute $x_0^2 + y_0^2$. To do so, we set all FMA A values to x_0, set all FMA B values to x_0, set all FMA C values to 0, then WRITE with replace_c but not output_result. Chaining from the previous result, we set all FMA A and B values to y_0 with diff of dy, then WRITE without replace_c but with output_result.

With the invariant in place, we set a register called iters to 0 and enter the inner iterations loop for max_iters steps. Because the invariant means x_i and y_i are stored in the output buffer, we can leverage the LOADB command to shuffle values into place as we need them. In particular, LOADB allows two special cases in addition to permuting: it can multiply by 2 or -1. These are useful for computing $x_i^2 - y_i^2$ and $2x_iy_i$. In fact, our current instruction set provides no other way to manipulate x_i and y_i across all FMAs except using the shuffle functionality of LOADB.

Inside the inner iterations loop, we compute x_{i+1}, y_{i+1}, and then $x_i \cdot x_i + y_i \cdot y_i$ in order to maintain the invariant.

1) We compute $x_{i+1} = x_i^2 - y_i^2 + x_0$ with two rounds of FMAs. First, we use LOADB to load $x_i, x_i, 0$ into FMA A, B, and C values, followed by LOAD to place x_0 into the FMA C value. We WRITE with replace_c but without output_result. Second, we use LOADB to load $-y_i, y_i, 0$ into FMA A, B, and C values, then WRITE without replace_c but with output_result.
2) We compute $y_{i+1} = 2x_iy_i + y_0$ with one round of FMAs. We use LOADB to load $2x_i, y_i, 0$ into FMA A, B, and C values, followed by LOAD with diff of dy to place the correct y_0 into each FMA C value. We WRITE with replace_c and output_result.
3) We compute $x_i \cdot x_i + y_i \cdot y_i$ with two rounds of FMAs. First, we use LOADB to load $x_i, x_i, 0$ into FMA A, B, and C values. We WRITE with replace_c but without output_result. Second, we use LOADB to load $y_i, y_i, 0$ into FMA A, B, and C values, then WRITE without replace_c but with output_result.

With the square magnitudes $x_i \cdot x_i + y_i \cdot y_i$ in hand, we call the special instruction OR to check whether any square magnitudes are greater than 4. For any that are, the instruction stores iters, the iteration at which the point diverged. One nuance is that our fixed-point numbers will overflow because points that diverge continue to diverge more and more. To address the overflow, for each pixel, we only store iters if we have not yet stored iters for that pixel before. One optimization would be to break out of the loop if every pixel has already diverged. To do so, OR would need to output a single bit back to the controller.

At the end of the iterations for loop, we call SENDITERS to send the sixteen computed iteration counts to the frame buffer. We add 16 to y_counter and 16*dy to y_val, because we have completed 16 pixels in parallel.

Once we finish both outer loops, FBSWAP tells the frame buffer that the current frame is ready to display. We then pause to allow the user to zoom in by pressing btn[1]. When the user presses the button, we zoom in by adding a small value to x_min (0.16409) and y_min (0.07591), subtracting the smallest value we can represent with our fixed-point numbers from dx and dy (0.00098), and then jumping back to the start of the two outer loops. We chose these particular numbers to zoom in around a starfish valley centered at $z = -0.563 - 0.643i$, as seen in Figure 1.

3. Design Evaluation

Latency and Throughput We have low latency per cycle, but the controller only process a command every two cycles. This bottleneck lowers the throughput of our system by half. We chose to make the controller a finite state machine to make it easier to reason about, although it would be possible to increase throughput by executing multiple instructions at once. Our design does not require pipelining because the FMA computation already fits within one clock cycle. A future improvement would be implement prefetching so that the controller can process one command every cycle or even process several commands at once.

BRAM and DSP Usage We utilized 37.33% of available BRAM and 15.00% of available DSPs, as well as 37.03% of LUTs and 7.18% of slice registers. Our FMA blocks are logic-hungry because we demand single-cycle reads and writes from the input and output buffers. Using BRAMs for the input and output buffers would reduce LUT usage, but it would defeat the purpose of parallelism because BRAMs only support at most two reads and writes at once. Similarly, all our BRAM utilization comes from our frame buffer prior to HDMI. We already store a lean representation of color in four bits, so we do not believe we can reduce BRAM usage. In terms of congestion, although we have a few very long registers (768 bits for all FMA inputs), every triplet of words (48 bits) acts independently. As a result, Vivado is able to synthesize lines that do not need to stay physically close, making congestion not a concern.

Timing We ran on the HDMI clock (74.25 MHz, period 13.468 ns). After Vivado optimized our design, we had a Worst Negative Slack (WNS) of 1.031 ns and Worst Hold Slack (WHS) of 0.018 ns. To improve timing, one step would be to pipeline the FMA's combinatorial multiplication. Another step would be to pipeline the FMA output registers so that Vivado would not need to synthesize them using logics because of the single-cycle write and read constraint. We ran preliminary experiments using 44-bit fixed point numbers, but these did not meet timing. We have not tried using 32 FMAs, but our FPGA has the necessary resources and it is likely we would meet timing.

Deliverables We began with two main goals: (1) our GPU should be able to multiply large matrices and (2) our GPU should be able to render the Mandelbrot fractal with 16x parallelism, a gradient of colors, and at more than 1 frame per second. At the end, our GPU is indeed able to multiply matrices and render the Mandelbrot fractal. The Mandelbrot frame rate is 3 FPS (depending of course on `max_iters`, which is by default 63). Because we implemented an instruction set, our GPU is general-purpose and can do more than we have tested on it so far. Thus, for matrix multiplication we fully met our goals. For Mandelbrot, we met all our goals except to reach 60 FPS. But we are still very happy with the frame rate as is, because the Mandelbrot set is computationally intensive. With 10 FLOPs required per iteration per pixel, rendering one frame requires an overall 64.5M FLOPs (for `max_iters` of 63). We believe that further optimizations as outlined above, along with short circuiting out of the inner loop when all pixels have already diverged, would be sufficient to reach around 30 FPS.

4. Team Member Contributions

Early in the project, Hanfei extensively researched real-world GPU designs and read documentation. Hanfei set up several meetings with Darren for input on our block diagram. Laker set up a meeting with graduate student Andrew Feldman to consult on GPU design.

Laker and Hanfei jointly designed the FMA module. Laker developed the controller as a finite state machine and wrote the Python compiler from assembly-style language into machine code. Hanfei developed the two versions of memory module, one without FSMs and one as an FSM, the latter being used as the foundation of the current memory module. Hanfei implemented the commands `LOADI`, `SENDL`, `LOADB`, `WRITEB` and testbenched them. Hanfei also wrote the HDMI module. Laker implemented the memory commands `LOAD`, the new `LOADB`, and `WRITE`.

Hanfei and Laker jointly designed the shuffle functionality in the input buffer, which was crucial for Mandelbrot. Laker wrote the assembly-style program that computes the Mandelbrot set. Additionally, Laker debugged extensively in simulation. When we encountered bugs on hardware, Laker built a system to step through the assembly program with checkpoints, including the ability to inspect any register in the program by using switches on the board. In effect, the debugging setup created a testbench on hardware. Laker also wrote the program extension that allows zooming in.

Hanfei created the the top level diagram for the final report. He wrote sections 2.2, 3, 4, and 5. He also created Table 1 and Figure 3. Laker created Figure 1 and Figure 2. He wrote sections 1, 2.1, 2.3, and 2.4. Hanfei and Laker collaborated on section 2.5 and both proofread the report.

5. Retrospective

Testbench, testbench, testbench! Due to our testbenching, when we compiled our program and flashed it on to hardware, there were only a couple minor bugs, and within 24 hours of our first compile, we had a Mandelbrot display that was almost correct. However, we spent too much time in the initial design and implementation stage trying to make our design general purpose. When we began to implement the Mandelbrot fractal, we ran into several roadblocks that actually provided a direction for what new features to implement, expanding both the generality of our GPU and its specific usefulness for Mandelbrot. In hindsight, we should have put our design on hardware sooner. Creating a hardware debugging system was critical to reduce debugging from $O(\infty)$ to $O(n)$.

Acknowledgments

Thank you to the many colleagues and friends who advised us on the architecture of GPUs, including Professor Steinmeyer, Darren Lim, Kailas Kohler, Joseph Feld, and Andrew Feldman.