An Image is Worth 16×16 Tiles: A Parallel Image Processing Accelerator

Jieruei Chang
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
jieruei@mit.edu

Vivian Ye
Electrical Engineering and Computer Science
Massachusetts Institute of Technology
yevivian@mit.edu

***Abstract*—We present Parallel Engine for Efficient Perception (PEEP), a multicore processor specialized for computer vision workloads. Using four parallel processor cores, taking advantage of massively parallel combinational memory reads, and exploiting parallelizable properties of image processing tasks, PEEP is capable of a theoretical throughput of 10 million pixels per second. We implement this system on a Spartan-7 FPGA and evaluate its performance on classic computer vision pipelines.**

***Index Terms*—Digital systems, field programmable gate arrays, image processing, computer vision, parallel systems**

I. Introduction

Realtime computer vision sees key applications in autonomous vehicles, industrial quality control, and localizing a dog's meandering path across the living room. The vision pipelines needed to conduct these tasks often rely on a set of highly parallelizable computations with strong spatial locality. However, most traditional vision pipelines still rely largely on CPU computation. This motivates the design of a GPU-analoguous parallel system capable of efficiently exploiting the structure of standard image processing workloads.

We propose **PEEP** (Parallel Engine for Efficient Perception), a demonstration of such a system. It has three main optimizations that exploit properties of common image processing tasks:

- Many CV pipelines are embarrassingly parallel. This means we can process multiple pixels at once by dispatching them to separate processors.
- Many CV operations exhibit spatial locality; i.e., the output of computation on one pixel only depends on itself and its close neighbors. We can design *tilecaches* that enable fast reads to square regions of an input region, and preemptively load the next region while the previous is undergoing processing.
- Most CV pipelines revolve around a discrete number of operations, such as convolution and thresholding. We optimize our processing cores for these tasks; PEEP includes hardware support for such primitives.

We demonstrate that PEEP is capable of handling nontrivial, multi-stage image processing pipelines. We implement our system in hardware on a Spartan-7 FPGA to demonstrate its feasibility, and evaluate its performance relative to naive single-threaded implementations. On morphological erosion, PEEP demonstrates a 17x speedup relative to Python.

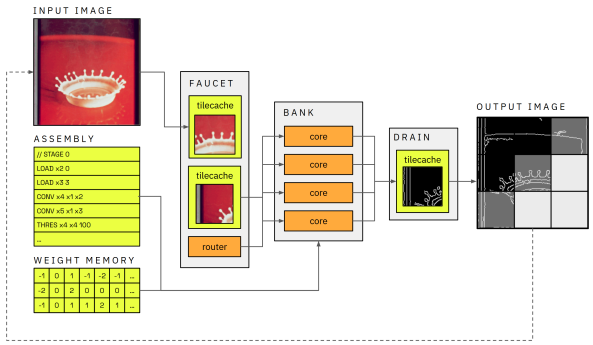


Fig. 1. High-level system flow.

II. System Architecture

PEEP integrates four processor cores, a tilecache and pixel coordination subsystem, SPI logic for reading and writing instructions and data, and interfaces to BRAM memory. A distribution system coordinates data distribution to multiple processors, each optimized for common image processing operations such as convolution, thresholding, and morphological filtering; a system of multiple tilecaches allows for fast reads and writes to 2D slices of images. The data flow through the system is shown in Fig. 1. A more detailed diagram of the full system architecture is shown in Fig. 2. We elaborate on the smaller subsystems afterwards.

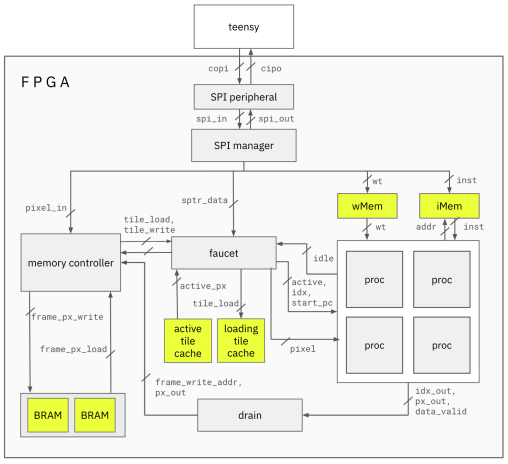


Fig. 2. High-level system architecture.

III. Faucet

Faucet is responsible for orchestrating data movement between the memory system and the processor bank, ensuring that processors receive a continuous stream of pixels while new tiles are prefetched in the background. Conceptually, Faucet acts as a scheduler and double-buffered streaming controller: it walks through the interior pixels of each tilecache, dispatches them to processors in a round-robin fashion when processors are idle, and simultaneously loads the next tilecache from memory so that computation and memory access overlap.

A. Tilecache Double Buffering

To hide the latency of memory accesses, Faucet maintains two tilecaches: an active tilecache that processors read from, and a loading tilecache that is populated with pixel data for the next region of the frame. As pixels arrive from the memory controller, Faucet stores them sequentially into the loading tilecache and tracks the fill level via an internal load index. Once the tilecache is completely filled, the module marks it as valid. Only when (1) all non-border pixels of the active tilecache have been processed, (2) all processors are idle, and (3) the loading tilecache is full, does Faucet swap the roles of the two tilecaches. This swap is synchronized with updates to the tilecache base address, causing the processor bank to immediately begin operating on the next spatial region of the input frame.

Additionally, Faucet handles pixel read requests from the processors. This is done combinationally to minimize cycle latency. Each processor has a single read-address line into the active tilecache; Faucet fetches the nine pixels in a 3×3 neighborhood around the given address.

B. Start Pointer Management

Most image processing pipelines consist of multiple stages; the output of one stage is needed as input into the next. Consequently, the assembly code is divided into multiple stages, each of which has an entry point (an assembly instruction address) stored in a small FIFO inside Faucet. This start pointer is sent to processors upon activation, and is incremented to the next stage when the entire frame has finished processing.

C. Processor Dispatching

Pixel dispatching proceeds in a round-robin manner. Faucet keeps track of a pointer to the next processor to attempt activation, and a pointer identifying the next pixel within the tilecache interior. Each cycle, if the tilecache is ready and the addressed processor is not busy, Faucet activates that processor, provides the pixel index, and directs it to start at the correct assembly instruction address. The pixel index logic is structured to visit pixels in raster order while skipping tile borders, since most image processing operations require only interior pixels to avoid incomplete neighborhoods.

D. Memory Address Generation

Faucet also generates the global memory addresses needed for tilecache loading. As it fills the loading tilecache, it computes the appropriate global pixel address by combining the tilecache's base row and column with the tilecache load row and load column. After each complete tilecache load, the base address moves in a way that ensures successive tilecaches overlap appropriately for full image coverage.

IV. Data Storage and Memory

A. Instruction Memory

Instructions reside in BRAM since arbitrary combinational reads are unnecessary in the instruction fetch. However, a BRAM provides only two read ports. We therefore duplicate the instruction memory to supply enough throughput for the four processors, and set one port per BRAM to be a dual read/write port in order to write instructions to memory in the initial loading stage; this is acceptable since reads and writes never occur at the same time.

B. Weight Memory

Static convolution weights are stored in a small data memory (`wMem`) in a similar manner to the tilecache. It is built out of registers to ensure that it is combinationally accessible by the processing cores.

C. Frame Memory

Full input and output images are stored on BRAM. Similar to how we maintain both an active and a loading tilecache, we maintain two BRAMs. One BRAM acts as an input frame buffer where its data is loaded into tilecaches. The other BRAM is an output frame buffer, storing results from the processing cores. In the case of a multistage image processing pipeline, the BRAMs are swapped between stages.

V. Processor

A. Design.

Each processor contains a program counter, a register bank, and branching logic. It reads assembly instructions from a global instruction memory. The register bank for each processor contains eight registers, each being 18 bits wide. To read image data, the processing cores perform combinational reads from the global active input tilecache; to read convolution weights, the cores have access to the global static data memory `wMem`. The processor begins computing upon receiving a signal from Faucet. The full processor architecture is shown in Fig. 3.

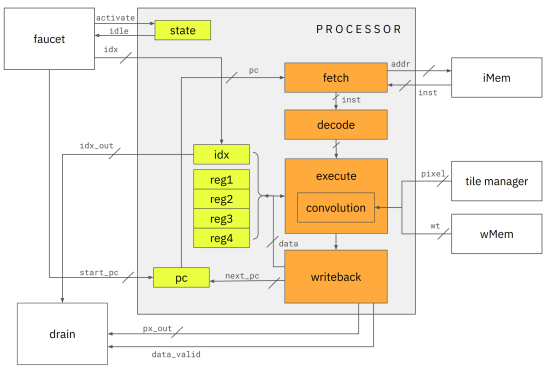


Fig. 3. Processor core diagram.

B. Instruction Set.

PEEP's processor cores use a custom instruction set very loosely based on the RISC-V architecture. Each machine code instruction is 24 bits, of the form

$$[\text{opcode} : 4][\text{rd} : 3][\text{rs} : 3][\text{rt} : 3][\text{imm} : 11].$$

The instruction encodings are as follows:

```
1       0x0: LOAD   rd, __, __, imm
2       0x1: ADDI   rd, rs, __, imm
3       0x2: ADD    rd, rs, rt,  __
4       0x3: MUL    rd, rs, rt,  __
5       0x4: SRA    rd, rs, rt,  __
6       0x5: THRES  rd, rs, __, imm
7       0x6: BGE    __, rs, rt, imm
8       0x7: CONV   rd, rs, rt,  __
9       0x8: MIN    rd, rs, rt,  __
10      0x9: MAX    rd, rs, rt,  __
11      0xA: ANDI   rd, rs, __, imm
12      0xF: RETURN __, rs, __,  __
```

THRES is a thresholding operation that sets register rd to rs if rs $\geq$ imm, and 0 otherwise. CONV computes a 3x3 convolution on the current pixel, which is specified by Faucet. CONV interprets the value in register rs as the top left index of a 3×3 kernel in the tilecache surrounding the current pixel, and the value in the register rt as the corresponding top left index in the weight memory. The processor issues a request for these nine pixel and nine weight values, and receives them combinationally from the tilecache and wMem registers. The multiplication and addition of the convolution is then computed, with the result written to register rd.

C. Implementation.

The cores are fully-pipelined four-stage processors. The first stage, FETCH_READ, reads out the current instruction from the instruction BRAM. This instruction is passed to DECODE, which determines the correct operation and fetches the correct register values so that EXECUTE can perform the correct calculation. Finally, these values are written back to the register bank in WRITEBACK, which also initiates the reading of the next instruction from the BRAM. The next instruction request is sent here as our BRAM has a two-cycle latency in retrieving instructions.

Bypassing logic handles transmission of intermediate results from the writeback and execute stages to the decode stage. The processor predicts that branches are not taken by default, and uses annulment logic to cancel inflight operations during a misprediction. Due to the design of the tilecaches and memories, we never have to perform fetch or data stalls.

The CONV operation, due to its heft, is split across two of the stages. Nine parallel multiplications are conducted in the EXECUTE stage, and the results are passed to and added together in WRITEBACK, and this final result is then written to a register.

Upon reaching a RET instruction, the processor returns the value in the specified register and halts computation. A valid signal is asserted, signaling to the drain and memory controller that data can be read from the processor.

VI. Drain

The processor cores are unaware of the actual global location of the pixel they are computing; they are only aware of the local index inside a tilecache. Drain's job is to convert these local indices into global frame addresses to write to the output frame buffer. It listens for valid high signals from each of the processing cores and routes associated global frame addresses to the memory controller.

VII. Memory Controller

The memory controller is in charge of the input and output frame buffers. It makes BRAM requests to provide data for the loading tilecache. The memory controller is also in charge of writing processed values to the output frame buffer. It receives pixel outputs from processing cores and frame memory addresses from *Drain*, and contains four FIFOs, one for each processing core. Each cycle, the memory controller checks for valid outputs from each of the processors and if valid, pushes the value into the corresponding FIFO. Then, the memory controller writes, in a round robin fashion, data from one of the FIFOs into the output BRAM.

In a multistage image processing pipeline, the memory controller also manages the switching of the two BRAMs. When all pixels in the existing input BRAM have finished processing, Faucet sends a signal to the memory controller to flip the two BRAMs, preparing PEEP for the next stage of the image processing pipeline.

VIII. Communication

PEEP needs to be capable of bidirectional communication. It needs to load instructions and data, and subsequently output the results of its computation. We implement this with SPI as it is the only simple protocol with sufficiently high bandwidth.

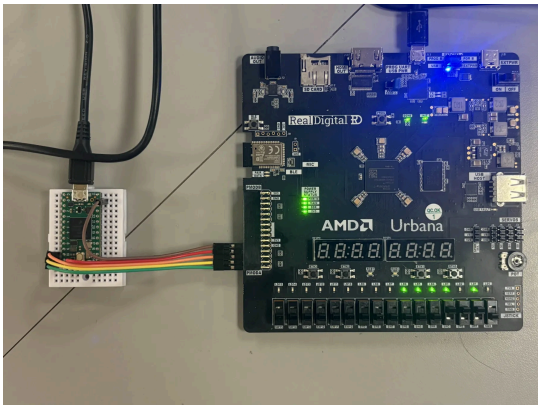


Fig. 4. SPI Hardware Setup.

A. Hardware.

In order to interface the FPGA with a laptop computer over SPI, we use a Teensy 4.0 microcontroller as an intermediary.

The Teensy communicates over an emulated serial link on 480 MHz USB 2.0 with the laptop computer, which it then translates to SPI lines hooked up to the PMODB pins on the FPGA. An image of the hardware setup is shown in Fig. 4.

B. SPI Peripheral.

The SPI peripheral is fully compliant SPI with MSB-first SPI mode 0, in order to interface with the default Teensy SPI controller. Multiple synchronizer stages are placed in front of all SPI input signals to the FPGA in order to account for metastability. Tests show that communication is reliable up to 20MHz, though in practice the SPI is clocked at 5MHz to maximize stability.

C. SPI Manager.

The raw data received over SPI must be routed to four different paths: the assembly instructions, the starting instruction pointers, the weight or data memory, and finally the image data. The state machine cycles through the SPI $\rightarrow$ FPGA paths in order, switching when it receives a sentinel value (`0x42`) indicating the end of that section. Notably, the SPI $\rightarrow$ FPGA paths have different data widths, so we need to assemble the incoming bytes into larger words before writing to the corresponding FIFOs.

IX. Pipelines

A. Gaussian Blurring

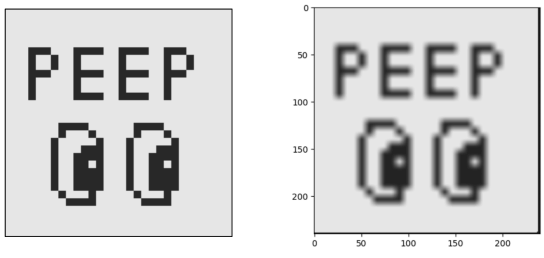
To test basic image processing functionality, we implement Gaussian blurring on PEEP. Successful implementation of this pipeline meets our commitment goal. Gaussian consists of the following assembly program:

```
1  ; stage 0
2  conv x2, x1, x0
3  load x3, 4
4  sra x2, x2, x3
5  return x2
```

The weight memory stores the Gaussian filter

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}.$$

PEEP produces the images shown in Fig. 5 and Fig. 6. Its output (after shaving the edges of the image) perfectly matches OpenCV's implementation with a 3x3 kernel.



(a) (b)

Fig. 5. PEEP processes input image (a) into the blurred output (b).

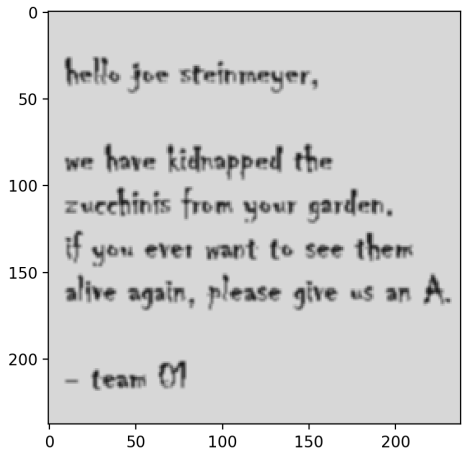
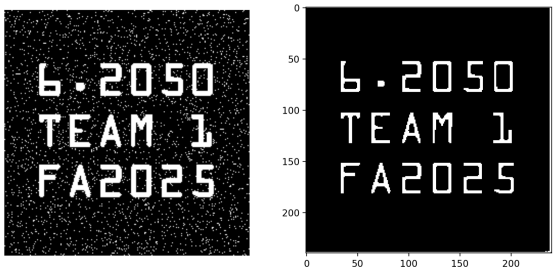


Fig. 6. Sample readout of Gaussian Blur pipeline from FPGA.

B. Erosion

Erosion is a fundamental operation in morphological image processing. Its effect is to reduce the shapes contained in a binary image by stripping their border pixels. To do so, we convolve a structuring element across the image and zero out pixels where the element is not entirely inside a shape. An example is shown in Fig. 7.



(a) (b)

Fig. 7. (a) Noisy input cleaned by FPGA with one iteration of erosion, producing output image (b).

Oftentimes, we want to run an erosion operation multiple times (for example to remove some small shapes, which may be noise). This means we can take advantage of PEEP's multistage functionality. We program PEEP to perform more iterations of erosion with several start pointers commanding it to start at the beginning of the erosion pipeline.

The assembly to run three iterations of erosion is as follows:

```
1  ; stage 0
2  ; stage 1
3  ; stage 2
4  conv x2, x1, x0
5  load x4, 3
6  sra x2, x2, x4
7  ret x2
```

The result of this pipeline is shown in Fig. 8. As the number of iterations increases, we see shapes in the binary image become thinner and then disappear.

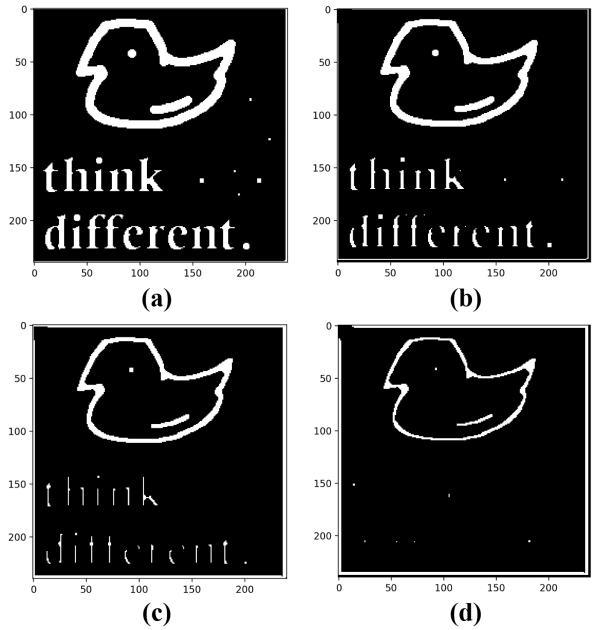


Fig. 8. Output of erosion pipeline from FPGA with (a) one, (b) two, (c) three, and (d) four stages.

C. Sobel Edge Detection

Our processing cores support signed weight values and their corresponding computations. As an example of a pipeline involving signed weights, we implement a simple Sobel-x filter aiming to detect vertical edges in a given image. The assembly program for Sobel is as follows:

```
1  ; stage 0
2  conv x2 x1 x0
3  addi x2 x2 128
4  load x3, 3
5  sra x2 x2 x3
6  ret x2
```

The weight memory stores the kernel

$$\begin{pmatrix} -2 & 0 & 2 \\ -1 & 0 & 1 \\ -2 & 0 & 2 \end{pmatrix}.$$

We show the results of the Sobel computation in Fig. 9.

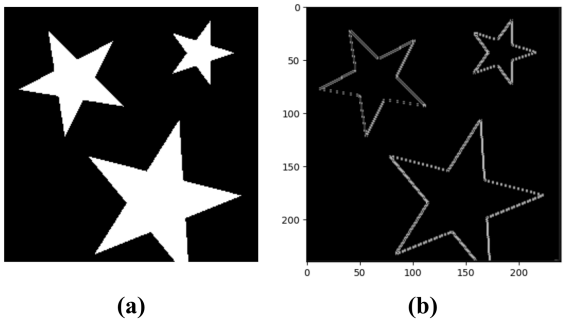


Fig. 9. Sobel edge detection pipeline from FPGA. Vertical edges from image (a) are highlighted in image (b).

D. Ordered Dithering

Dithering is used to convert a continuous-valued image into a binary or low-bit representation while preserving the perception of detail. Standard dithering techniques such as Floyd–Steinberg rely on error diffusion, which introduces strong data dependencies: each pixel's value depends on previously processed neighbors, preventing straightforward parallelization. Ordered dithering algorithms, which replace error propagation with a spatially fixed threshold pattern, do not have this limitation and can be parallelized efficiently. We implement an ordered dithering using the 2x2 Bayer map

$$\mathbf{M}_2 = \frac{1}{4}\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}. \tag{1}$$

Our assembly code below computes this using conditional logic based on the parities of row and column indices of each pixel. This serves as a demonstration of the branching capabilities of PEEP. Outputs are shown in Fig. 10 and Fig. 11.

```
1   ; stage 0
2   conv x2 x1 x0
3   load x6 x0 1
4   load x7 x0 4
5   sra x3 x1 x7
6
7   andi x3 x3 1 ; row parity
8   andi x4 x1 1 ; column parity
9
10  bge x3 x6 ROW_ODD
11  bge x4 x6 ROW_EVEN_COL_ODD
12  ROW_EVEN_COL_EVEN: load x5 x0 0
13    bge x0 x0 MUL_PIXEL
14    ROW_EVEN_COL_ODD: load x5 x0 2
15      bge x0 x0 MUL_PIXEL
16  ROW_ODD: bge x4 x6 ROW_ODD_COL_ODD
17    ROW_ODD_COL_EVEN: load x5 x0 3
18      bge x0 x0 MUL_PIXEL
19    ROW_ODD_COL_ODD: load x5 x0 1
20  MUL_PIXEL: mul x2 x2 x5
21    load x4 x0 7
22    sra x2 x2 x4
23    ret x2
```

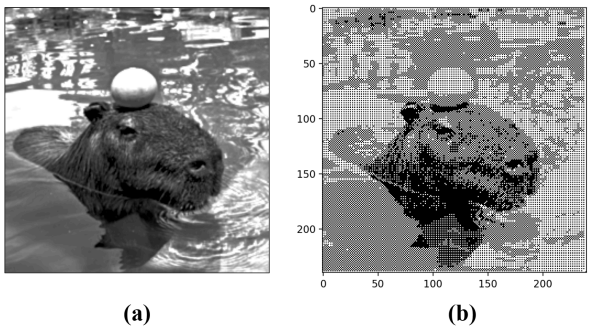
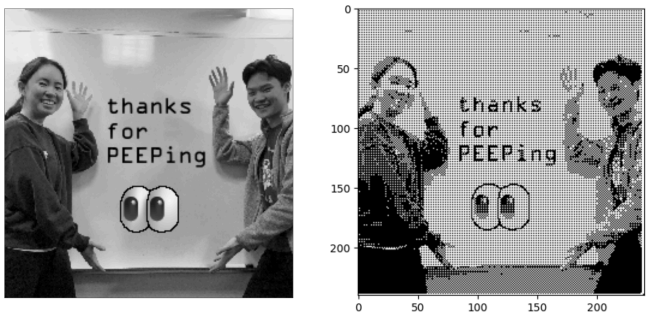


Fig. 10. Ordered dithering pipeline from FPGA. The input image (a) is processed to binary values (b) by a 2x2 Bayer map.



(a) **(b)**

Fig. 11. Another ordered dithering example with (a) input processed to (b) binary output.

X. Evaluation

A. Timing

We evaluated the erosion pipeline on a 240 × 240 image with 32 stages and timed the difference between the sending of the last SPI pixel command and the start of the valid SPI readout. This takes around 225 ms, so a single erosion stage takes 7 ms on the FPGA (in reality, it is even lower due to residual overhead from the SPI-Teensy-FPGA routing). In a naive Python implementation, this operation takes 3860 ms for 32 erosion stages, for an average of 120 ms per stage. PEEP is therefore about 17x faster than the naive implementation. OpenCV takes ~0.3 ms per frame, though this test was run on a computer with 4.0 GHz clock speed. Normalizing for clock speed, PEEP is ~3x faster.

B. Memory

Our current design uses 34 36 kilobit BRAMs and 4 DSPs. We can reduce our BRAM usage by moving the frame memory to SDRAM, as we will discuss in the next section.

C. Project Checklist

The commitment milestone for our project has stayed consistent throughout its development: using our tilecache system and BRAM image storage, we want to run a single stage image processing workload on a single processing core. We exceeded this goal on the Gaussian pipeline with four integrated processing cores computing in parallel.

Our goal milestone initially involved running a multistage image processing pipeline with our image stored on off-chip SDRAM memory. After a check-in with the teaching team, we moved the off-chip SDRAM requirement from our goal milestone to our stretch milestone. We thought it would be more interesting to focus on adding support of signed values in the processing cores and testing our system on additional computer vision pipelines like erosion or dithering. We achieved both of these goals.

Our stretch goal of using SDRAM to store frame memory has not been reached, but this would be a natural next step for our project. Additionally, we would like to build support for larger kernel sizes, as we are currently limited to 3× 3 kernels. These two additions to PEEP would allow us to process larger images and compute more complex computer vision workloads.

XI. Conclusion and Discussion

We build PEEP, a quadcore processor for computer vision pipelines. Its design takes advantage of both the FPGA's parallel nature and the parallel structure of image processing tasks (high spatial locality and SIMD paradigms). We demonstrate that PEEP is capable of handling multistage traditional image processing pipelines. We implement our system in hardware and evaluate its performance relative to naive single-threaded implementations, demonstrating up to a 17x speedup.

XII. Reflection

This project taught us a lot about processor and system design, as well as computer vision and image processing. Building a custom processing core was a very interesting task. We were able to make intentional choices about every part of the processor, from building the ISA to mapping out the unique pipeline stages.

The scaffolding of PEEP was a part of the project that we hadn't expected to be as challenging as it was. We underestimated the complexity of wiring together the different parts of our system (processing cores, SPI communication, frame memory). As a result, a large portion of our time was spent debugging the entire system after we had integrated the submodules. We had tested the individual parts separately, but in retrospect, this testing could have been more thorough and rigorous, which would have saved debugging effort.

Overall, we learned a lot about GPUs, processor design, system integration, and debugging in SystemVerilog. The process of designing a system from scratch was difficult but exciting, and the iterative process of building, testing, and debugging was an extremely valuable learning experience.

XIII. Code

Our team's code can be found at https://github.mit.edu/6205F25/fa25-6205-team01.

XIV. Contribution Statement

Jieruei built the SPI system, processor dispatching infrastructure, tilecache logic, ISA, assembler, and image processing pipelines. Vivian performed validation and proof-of-concept testing for instruction and pixel storage, built the signed optimized processor cores, implemented the memory controller, and edited the final video. All authors contributed equally to project conception, testing, debugging, and writing of each section of the manuscript.

Acknowledgment

We thank Joe Steinmeyer and Hasan Zeki Yildiz for both their low-level technical guidance and high-level advice throughout the development of this project.