EE789 Mid-semester Problem Description

Madhav P. Desai

August 30, 2019

1 Overview

You will be designing an accelerator which will perform dot-products of a data (image) vector, with a kernel. Suppose there is an image with pixels $\{x_{i,j} : 0 \leq i < 32, 0 \leq j < 32\}$. Further, assume that you are given a kernel which is a 4x4 image $\{k_{i,j} : 0 \leq i < 4, 0 \leq j < 4\}$. Assume that the pixels and kernel values are coded as 16-bit unsigned integers. Ignore overflows.

Let p take on values $0, 1, \ldots, 28$, and q take on values $0, 1, \ldots, 28$. The accelerator is supposed to compute the following numbers:

$$u_{p,q} = \sum_{i=0}^{i=3} \sum_{j=0}^{3} x_{(p+i),(q+j)} \times k_{i,j} \tag{1}$$

The overall scheme can be visualized as shown in Figure 1.

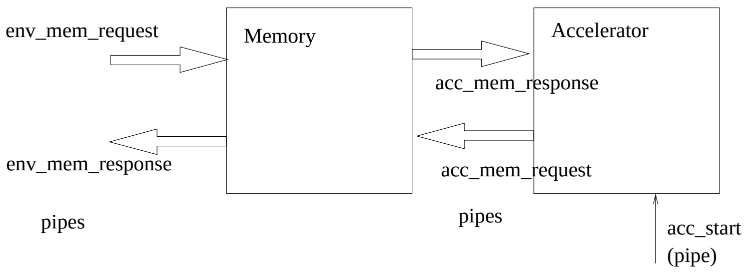

Figure 1: Accelerator Scheme

2 Accelerator Behaviour

The accelerator will go through the following sequence:

1. Attempt to read from the start pipe. This will block until a value is written from the environment into the start pipe.
2. After successfully reading from the start pipe, the accelerator will read a command descriptor from a fixed location in the shared memory. The command descriptor should specify the 16 values for the kernel, and a command counter.
3. The accelerator will execute the command, calculate the u values and write them back to memory.
4. After the accelerator has finished, it should write a status into a status word, which can then be used by the environment to pull out the u-values. Note that the command word can itself have a status bit which can be set by the accelerator.
5. The accelerator will look for the next command (command counter is checked to see if a new command is available), and execute it as shown above... etc.

The total memory that you are allowed to use is 4kB (That is 2kB for the image and an additional 2kB for the command descriptor etc). Note that the u-values can be written into the image memory itself. You can decide the memory layout.

3 Design Decisions

You will have to make several design decisions:

- Memory layout.
  - data-width, address-width.
  - coding of command, status registers.
  - location of command, status registers.
  - location and layout of incoming image.

    - location and layout of outgoing u-values.

- Parallelization scheme.
    - Use of parallelism is **mandatory**. However, first construct and verify the core before going for parallelism.

4 Evaluation scheme

You will submit a report of your work. Evaluation will based on the report and a viva. The evaluation scheme is

- Quality of design decisions (25%).
- Status of implementation.
    - Initial documentation (10%).
    - Aa implementation and verification (20%).
    - VHDL implementation and verification (20%).
- Viva and demonstration (25%).

5 Suggested implementation for the memory

To implement the memory, you can use the following pattern (I have shown it for an address width of 8 and data-width of 32, but you can decide your configuration). Note that memAccessDaemon and accMemAccessDaemon are independent threads.

```
$pipeline $depth 7 $module [accessMem]
  $in (read_write_bar: $uint<1> addr: $uint<8> write_data: $uint<32>)
  $out (read_data: $uint<32>)
{
  $storage mem_array: $array[256] $of $uint<32>

  $guard (read_write_bar)   t_read_data := mem_array[addr]
  $guard (~read_write_bar)  mem_array[addr] := write_data
  read_data := ($mux read_write_bar t_read_data 0)
```

```
}


// bits [63] = read_write_bar
//      [62:40] = unused.
//      [39:32] = write-address.
//      [31:0] =  write-data
$pipe env_mem_request: $uint<64> $depth 2
// read data.
$pipe env_mem_response: $uint<32> $depth 2
$module [memAccessDaemon]
   $in () $out () $is
{
   $branchblock[loop] {
       $dopipeline $depth 7 $buffering 2 $fullrate
          cmd := env_mem_request
          $volatile $split (cmd 1 23 8 32)
                  (rwbar unused addr wdata)
          $call accessMem (rwbar addr wdata) (rdata)
          env_mem_response := rdata
       $while 1
   }
}

// bits [63] = read_write_bar
//      [62:40] = unused.
//      [39:32] = write-address.
//      [31:0] =  write-data
$pipe acc_mem_request: $uint<64> $depth 2
// read data.
$pipe acc_mem_response: $uint<32> $depth 2
$module [accMemAccessDaemon]
   $in () $out () $is
{
   $branchblock[loop] {
       $dopipeline $depth 7 $buffering 2 $fullrate
          cmd := acc_mem_request
          $volatile $split (cmd 1 23 8 32)
```

```
                (rwbar unused addr wdata)
        $call accessMem (rwbar addr wdata) (rdata)
        acc_mem_response := rdata
     $while 1
  }
}
```