Accelerator Module Report

Name - Sahasrajit Sarmasarkar Roll no- 160070010

October 2019

1 Introduction

An accelerator module was built which takes an input 32x32 image and a 4x4 kernel each of which built as a 16 bit unsigned integer. The dot products were computed on each of the first 29 rows and 29 columns with the 4x4 kernel. We could represent it as the following where p takes the values from 0 to 28 and q takes the values from

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

The image and kernel both are written on a shared memory by the external environment. After the image is written onto the shared memory the status register the shared memory is updated. The accelerator checks the status onto the shared memory through polling and once the status is updated by the external environment, the accelerator fetches the image and kernel from memory onto its internal storage and starts its dot product computation.

After the computation is done, it overwrites the original image on the shared memory location and updates the status register in shared memory location. The environment also checks the status register through polling and once computation is done it fetches the resultant 29x29 image and then once again writes the new set of image and kernel.

The accelerator is initially in idle mode which starts its operations once data is written on accelerator pipe.

2 Block Diagram with functionality

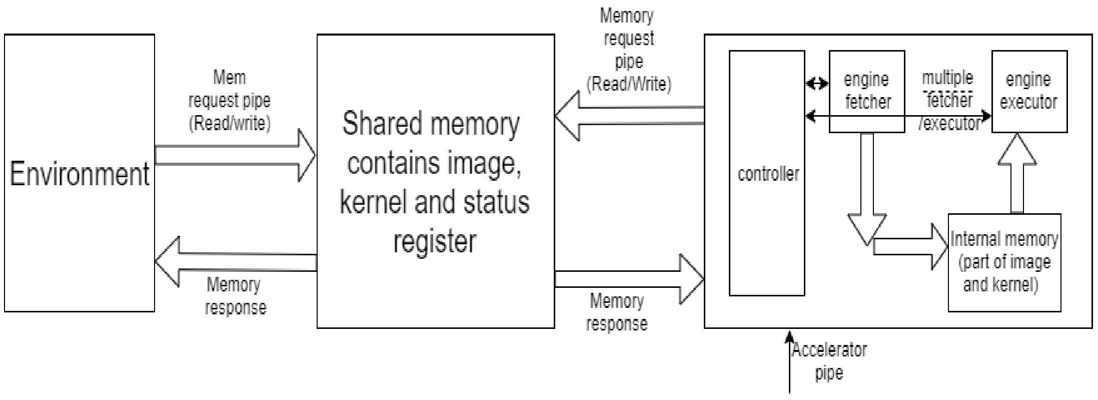
Environment
Mem request pipe (Read/write)
Memory response
Shared memory contains image, kernel and status register
Memory request pipe (Read/Write)
Memory response
controller
engine fetcher
multiple fetcher /executor
engine executor
Internal memory (part of image and kernel)
Accelerator pipe

Figure 1: Block Diagram

2.1 Memory Access

For writing/reading onto the memory, blocking pipes are used.

Whenever a read request is sent onto the memory read pipe with desired address, the shared memory block sends the required data back through the memory response pipe.

However whenever a write request is sent onto the memory no data is written onto the memory response pipe.

Both reading and writing is done pixel by pixel where pixel position is to be provided.

This is true for both the environment and accelerator.

2.2 Accelerator module

This module consists of multiple engines - namely controller and few fetch and execute engines.

The controller controls the sequence of actions in accelerator module where as the fetch modules request data from shared memory and update the internal memory of fetched image whereas the execute engines works on the image in internal memory to compute dot products and send write requests to shared memory through pipes.

3 Methodology used

Since there is a total memory limit of 4kB we can't fetch the whole image at once and store it in internal storage as size of image itself is 2 kB. So initially the first 29 rows were fetched and stored in internal storage after which the next last 3 rows were fetched and stored in 6x16 grid of internal storage, thus internal storage of 1856 bytes was used.

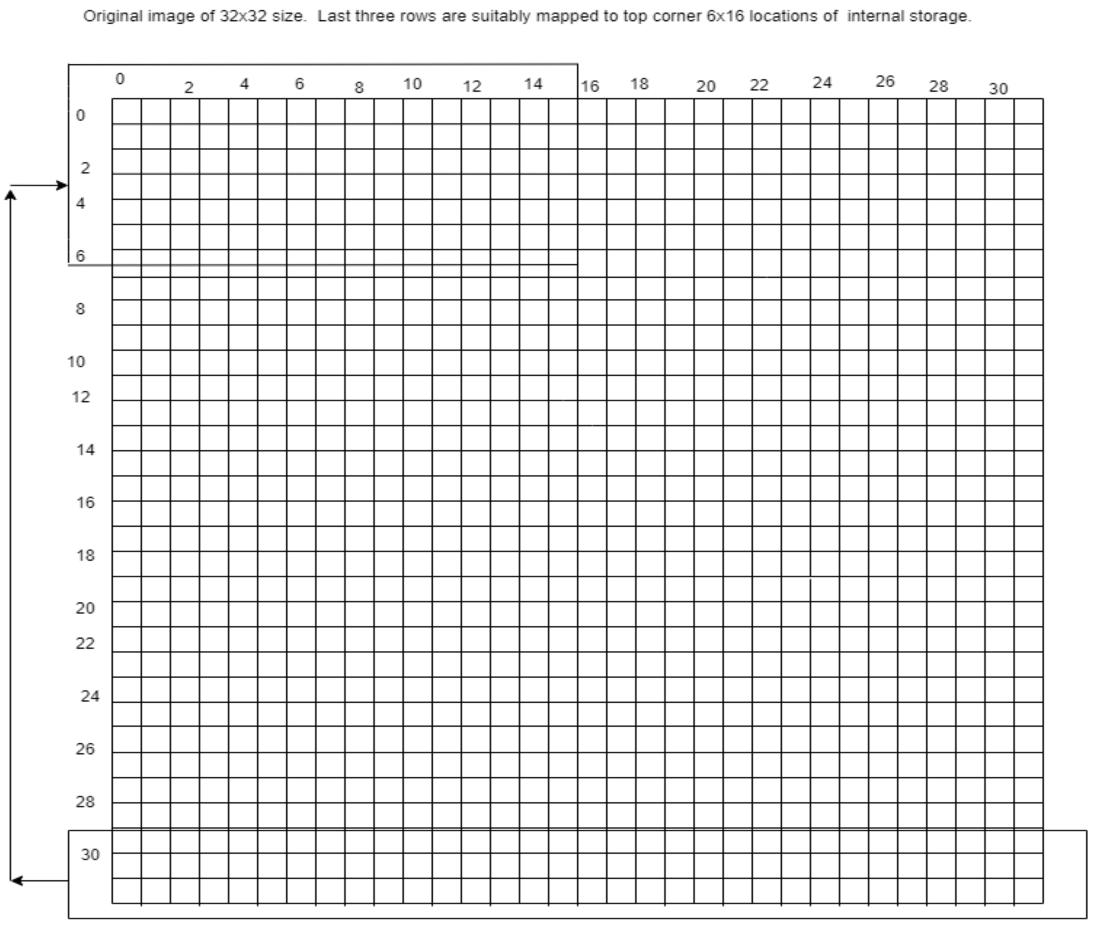
Original image of 32x32 size. Last three rows are suitably mapped to top corner 6x16 locations of internal storage.
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30

Figure 2: Storing image internally

Since internal memory is being re-used, the rows 29-31 cannot be fetched as long as all the elements in top 6x16 positions have not been re-used.

Thus to make things simple, we fetch and compute the last three rows only after fetching and computation on all rows except last three rows have been done.

We have tried to explore parallelism between fetching and execution i.e. we try to compute the resultant image at required position whenever all the sixteen required pixels are available.

Thus to denote whether data is up-dated with new image in shared memory or not, we use another bit corresponding to each pixel which denotes whether recent image is fetched or not. Before starting the operation it is initialised to zero and is changed to 1 as and when required data is fetched.

This required another 928 bytes.

We also tried to have multiple fetch engines and multiple execute engines to enable further parallelism.

4 Parallelism explored

So the following versions were tried. However the loops and modules were pipelined wherever possible to depth 7.

4.1 No parallelism

So here we fetch the 29x32 image first post which we try to compute the dot product at each pixel for rows 0-25 which is immediately written to memory through pipes.

After this we fetch the last 3 rows and write it in top 6x16 positions in image and then compute the dot products for pixels in rows 26,27 and 28 which is written to shared memory through pipes.

4.2 Parallelism in fetching and execution

Here parallelism in fetching and execution is done. We use the status bits for each of the memory addresses The same algorithm as above is used but fetching and execution is done in parallel

The first 29 rows are fetched parallel to which execution of dot product of resultant image at pixel positions corresponding to rows 0-25 is done which is written to shared memory through pipes

Dot product computation for each pixel is done only when all 16 required pixels are available - else it waits.

Post this, the last three rows are fetched and dot product computation on rows 26-28 is done in parallel which is written to shared memory through pipes.

4.3 Parallelism through one fetching engine and two execution engines with fetching and execution not in parallel

Here we have one fetching engine which fetches the first 29 rows of pixels after which we have two engines working in parallel on two portions of the image. One working on first 13 rows and the other execution engine is working on next 13 rows and both could write in parallel onto the memory.

After this we fetch the last 3 rows and write it in top 6x16 positions in image and then compute the dot products for pixels in rows 26,27 and 28 which is written to shared memory through pipes.

4.4 Parallelism through one fetching engine and two execution engines with fetching and execution in parallel

The only difference her from that of the earlier case is that here fetching and execution is done in parallel.

Thus before starting any operation in parallel we fetch the overlapping zone i.e the row numbers 13,14 and 15 as these data might change after one engine has written data.

So we first fetch the rows 13,14 and 15 and update the corresponding status bits.

After this we fetch the remaining rows of 29x32 image and do execution simultaneously using two engines.

However note that we don't do consecutive fetching of all addresses row by row.We first fetch image pixel one in region 1, the next pixel in region2 (from row 16) and then the next again from region 1 and so on.

Post this, the last three rows are fetched and dot product computation on rows 26-28 is done in parallel which is written to shared memory through pipes.

4.5 Parallelism through one fetching engine and four execution engines with fetching and execution not in parallel

Here we have one fetching engine which fetches the first 29 rows of pixels after which we have four engines working in parallel on four portions of the image. One working on first 13 rows and 13 columns and the other execution engine is working on same 13 rows and subsequent 16 columns.

The next two engines work similarly on next 13 rows.

Diagrammatically can be shown as

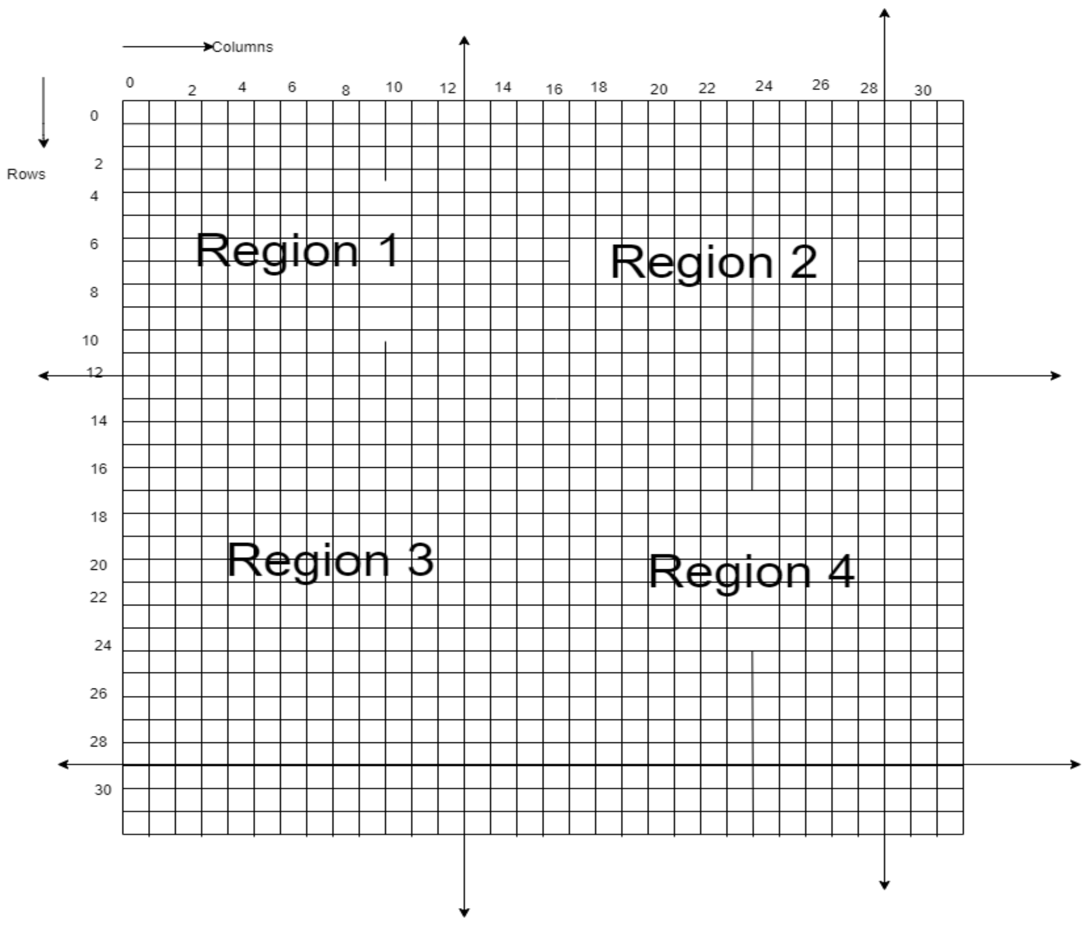

Figure 3: Regions of operation of each engine

After this we fetch the last 3 rows and write it in top 6x16 positions in image and then compute the dot products for pixels in rows 26,27 and 28 which is written to shared memory through pipes.

4.6 Parallelism through one fetching engine and four execution engines with fetching and execution in parallel

The only difference here from that of the earlier case is that here fetching and execution is done in parallel.

Thus before starting any operation in parallel we fetch the overlapping zone i.e the row numbers 13,14 and 15 and column numbers 13,14 and 15 (shown in figure as region 5 and region 6) as these data might change after one engine has written data which might be used by the other engine in computation.

Diagrammatically can be shown as

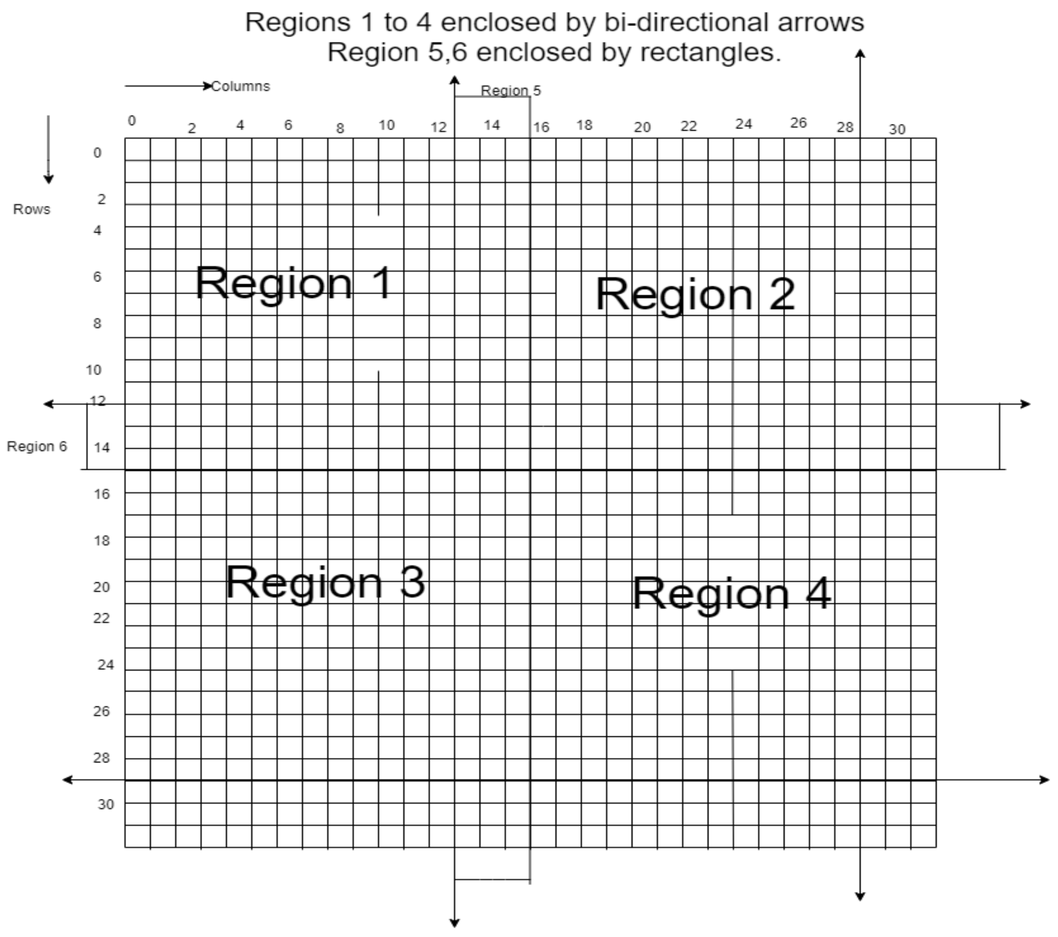

Figure 4: Regions of operation of different engine

So we first fetch the rows and columns 13,14 and 15 and update the corresponding status bits.

After this we fetch the remaining rows of 29x32 image and do execution simultaneously using four engines each operating on region 1,2,3 and 4 respectively.

However note that we don't do consecutive fetching of all addresses row by row.We first fetch image pixel one in region 1, the next pixel in region2 (from row 16) ,third from region 3 , fourth from region 4 and then the next again from region 1 and so on.This ensures no engine would be sitting idle for most of time initially.

Post this, the last three rows are fetched and dot product computation on rows 26-28 is done in parallel which is written to shared memory through pipes.

4.7 Parallelism using two fetch engines and two execution engines

Here again we fetch the image pixels corresponding to rows 13, 14 and 15.

Post this we divide it into two regions namely one from rows 0-12 and the other from rows 13-28

After this we have two fetch engines each picking up pixels from one region and updating the status bit. However to synchronise the two,both the parallel fetching units do reading from shared memory through a common module which is called by both. Simultaneously, the execution engines also work on both these regions.

After all these operations have finished the last three rows are fetched and dot product computation on rows 26-28 is done in parallel which is written to shared memory through pipes.

4.8 Parallelism using four fetch engines and four execution engines

Here again we fetch the image pixels corresponding to rows 13, 14 and 15 and columns 13,14 and 15.

Post this we divide it into four regions as the one in figure 4

After this we have four fetch engines each picking up pixels from one region and updating the status bit. However to synchronise the two,both the parallel fetching units do reading from shared memory through a common module which is called by both the fetching engines. Simultaneously, the execution engines also work on both these regions.

After all these operations have finished the last three rows are fetched and dot product computation on rows 26-28 is done in parallel which is written to shared memory through pipes.

5 Verification and results obtained

Testing was don through a proper c-testbench which was playing the role of environment and generation 16-bit data using random generator function.

This thread writes the data onto the shared memory multiple times and waits for accelerator to complete. Once accelerator updates its status, the thread fetches the 29x29 resultant image and compares with expected data. Errors if any are printed.

Tried upto 10 such operations in one simulation using the c- testbench. Each operation consists of the whole process described above. Also tried to note down the resources and time taken for different levels of parallelism as described in previous section.

No data mismatch was observed in any of the different levels of parallelism described indicating that the algorithm and hardware generated is mostly correct.

Note that total memory used in less than 4kB in all cases.

The results are described in table below.

In the time taken, time is given for five such operations in one simulation.

Parallelism used	Time taken	FF used	Time per operation
3.1	2.34 ms	1678	0.468 ms
3.2	2.45 ms	2475	0.49 ms
3.3	2.3 ms	2194	0.46 ms
3.4	2.57 ms	3290	0.51 ms
3.5	2.306 ms	3473	0.461 ms
3.6	3.45 ms	4434	0.69 ms
3.7	2.3 ms	3229	0.46 ms
3.8	2.77 ms	4744	0.55 ms

Thus we see that parallelism in section 3.7 gives the fastest operation.