Project Report

CS6023-GPU Programming

25 Nov 2018

Weight Sharing on CNN

Mohammed Khandwawala **EE16B117**

Índice

1. Introduction

This project tries to take advantage of the mechanism discussed in the paper "Incremental Network Quantization: Towards Lossless CNN's With Low-Precision Weights"to speed convolution. The methodology discussed in this paper takes trained weights of the model and converts them into discrete standard values. Using this fact to decrease the number of operation to get speed improvement. *ALL THE CODE TESTING AND TIMING AND PROFILING IS DONE ON NVIDIA 940MX*.

1.1. Incremental Network Quantization Mechanism

INQ converts the weights in trained network to quantized weights (weights are in powers of 2). A threshold is set and the weights above that threshold are quantized whereas other weights are retrained. The un-quantized weights above a certain threshold are quantized and the remaining weights are trained again. This Process is repeated until quantization is complete. The b in the equation below represents the value of number of bits needed to represent weighs.

$$n2 = n1 + \frac{1-2^{(b-1)}}{2}$$

n2 and n1 are the starting and the ending value of the the power of weights where n1 >n2 .

$$s = max(abs(\mathbf{W}_l))$$

s is chosen from the weights vector as shown by the above equation. From which then n1 is calculated.

$$n1 = floor(log_2(\frac{4s}{3}))$$

b is the parameter representing number of bits to store the weights.

$$P_l = \{\pm 2^{n1}, \cdot\ \cdot, \pm 2^{n2}, 0\}$$

P_l vector shows all the unique weights possible.

2. Problem Statement

We can use quantized weights generated by the algorithm mentioned in this paper to speed up the inference process of a network. Thereby the problem statement is inspecting the scope and finding ways to speed up the convolution if the kernel has quantized values only.

2.1. Convolution on GPU

For convolution on GPU, tiling can be used. Tiling can be used in two ways in a GPU since there are multiple calls to the same element in the input matrix for every output element.

- Input stationary Tiling: Each thread loads one element of the input tile and may calculate one or more output element.
- Output Stationary tiling: Each Thread computes one output element.

Input stationary convolution has control diversion in computing the output whereas Output stationary tiling has control diversion in the loading of the input tile. For this project weight sharing method is tried in both ways.

2.2. Ways for Improvement

Since the kernel can only take values from the set of values in P_l we can reduce the number of multiplications in the convolution process by adding the input matrix values that get multiplied to the same weight and the multiply the weight to the sum, instead of multiplying to each value. Saving the number of multiplications could potentially give the speedup.

3. Implementations

To gain speed various versions of the above two methods were tried to accommodate weight sharing in the convolution process. To test the correctness of the output, basic versions of both input stationary tiling and output stationary tiling, along with a serial code have also been implemented . All the versions of these codes support strided convolution.

3.1. Parameters

For all the codes the parameters remain the same (unless specified separately). Each code has some constraints on these parameters specified independently.

- **W** *input matrix size*
- **OW** *output matrix size*
- **n1** *parameter from INQ*
- **n2** *parameter from INQ*
- **b** *n1 - n2 + 1*
- **T** *kernel size*
- **D** *Number of Channels*
- **N** *Number of kernels*
- **STRIDE_LENGTH** *stride length*

For all the timing data, n1 was set to 3 and n2 was set to 1.

3.2. Output Stationary Tiling

Using the standard strided convolution, Output stationary tiling was modified since this method has regular access patterns in the input tile memory. Code version 3 is based on this. For version 5, an alternative method is used to save number of operations and is discussed below.

3.2.1. Version 3

In this version, instead of the kernel values, only its power along with its sign is stored in the kernel matrix. Since each thread computes one output element, each thread maintains the sum of the input matrix values that are going to be multiplied to the same weight in an array. This array is stored in the thread registers. Indices of the array have one to one mapping to the power of the weight so the all values in the array can added with appropriate bit shift (equivalent to multiplying to the weights that are in powers of 2). Here threads access contiguous memory blocks in DRAM while loading the input matrix tile. Version 6 is just a slight modification of the same but uses loop unrolling which helps if the value $(n1 - n2)$ is small.

3.2.2. Version 5

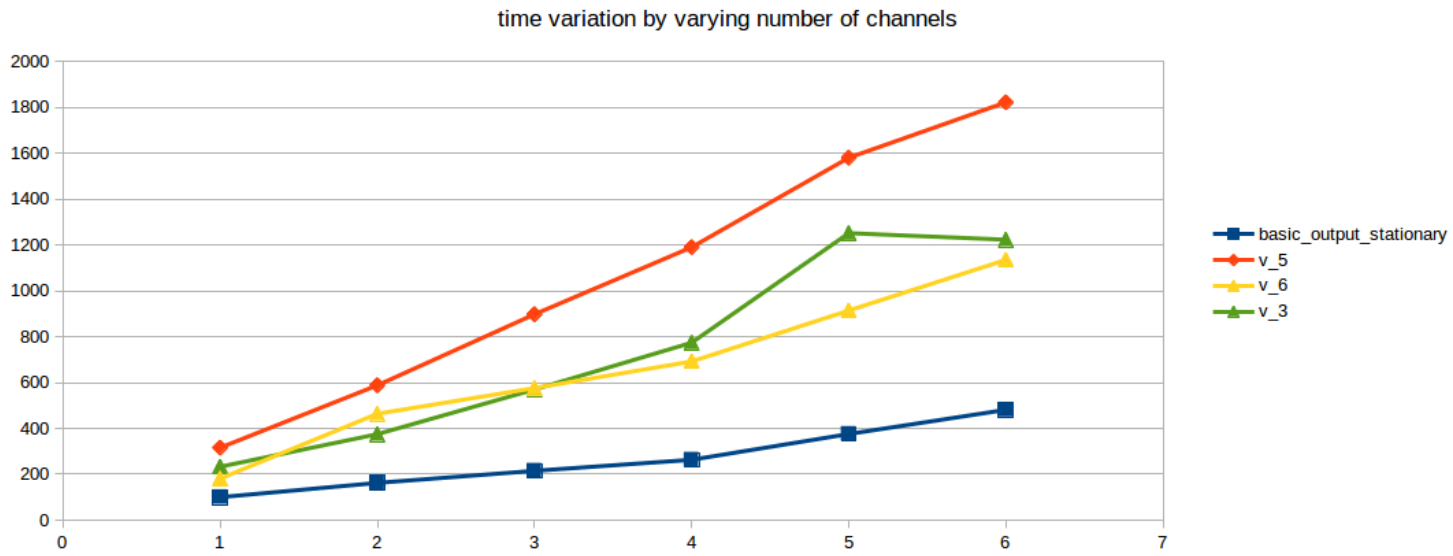
For this version, the kernel stores the power (in this implementation the sign is ignored but can be added), and the index of the weight in the normal kernel, to save its original position after this is preprocessed. Now each kernel (there are stacked N kernels each individually sorted) is sorted according to the weight and the position of the changed weights is stored. Now using this kernel in the basic code, separate sums for each output element can be maintained. All the elements of input matrix getting multiplied to the same weight are stored in this sum. This partial sum is added up by multiplying to by the respective weights to obtain the output. But this method saves multiplications at the cost of 1: Extra memory accesses to the global memory to load the kernel matrix, which now also has positions so three more integer values need to be loaded, and 2: also adds control divergence in the basic code to implement the weight sharing logic.

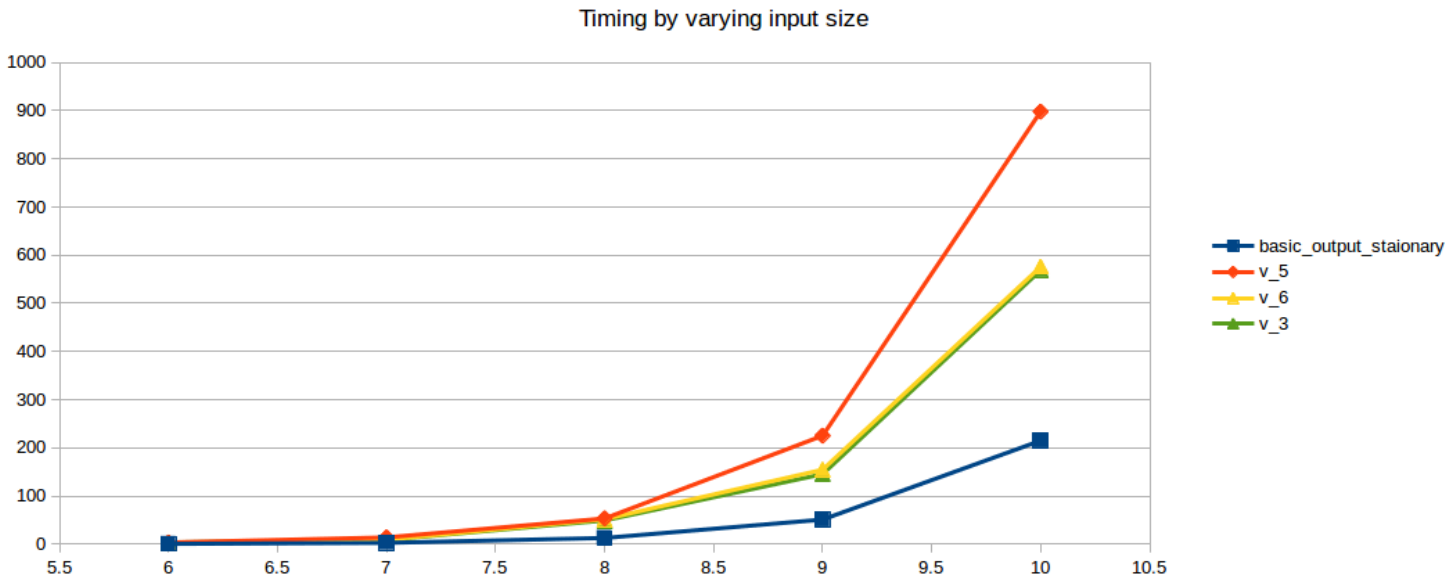
3.3. Timing Analysis

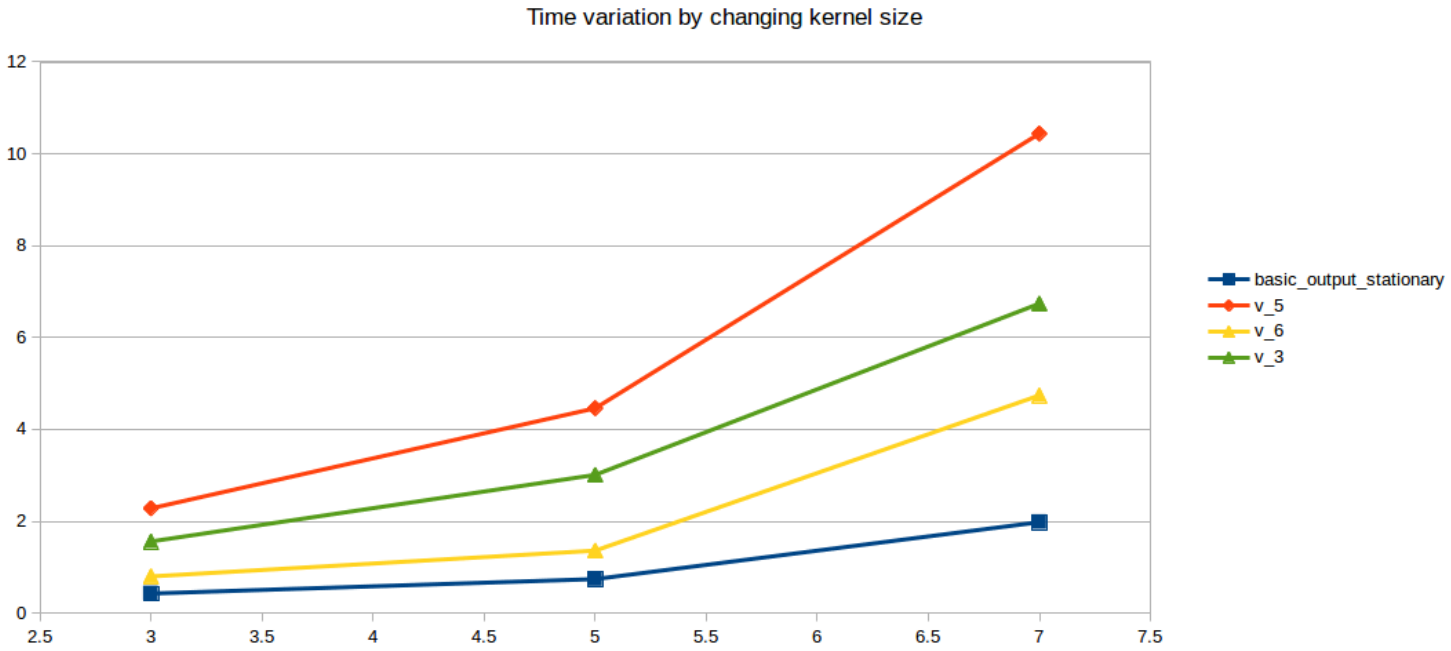
For the Timing Analysis, the set of parameters (unless specified) are-
Stride length is kept 1 for all the tests as higher stride length reduces the the number of operations resulting in less saving.

W 1024	**n1** 3	**n2** 1
b 3	**T** 3	**D** 4
	N 128	

The time difference increases as the input matrix size is increased, in either dimension or number of channels. Increasing the size of kernel also leads to an increase in kernel execution time.







3.3.1. Inference

The plots above show the timing variations due to variation in parameters. Blue line represents basic code. V_x indicates the version number. As the kernel size is increased, the difference in the timing, from the basic version to the modified version increases. The reason being that it is getting memory bound due to decrease in computational intensity. For each output element $(D*D*T-b)$ multiplications are saved. Instead, additional memory calls and control divergence introduced have a dominating effect and results in overall slowdown in performance.

For version 3 & 6

- Number of multiplications saved per output element: $D \times T \times T - b$
- Number of extra Memory (Shared) accesses (over normal): $T \times T \times D + b$
- Number of extra Memory (Global) accesses (over normal): D

For version 5

- Number of multiplications saved per output element: $D \times T \times T - b$
- Number of extra Memory accesses (over normal): $3 \times T \times T \times D + 3 \times D + b$
- Number of extra Memory accesses (over normal): $3 \times D$

3.4. Input Stationary Tiling

Weight sharing, as implemented with output stationary, was implemented on input stationary tiling as well to see the difference in the performance with the output stationary tiling. A basic version of the code is implemented and compared with the code implementing weight sharing.

3.4.1. Version 1

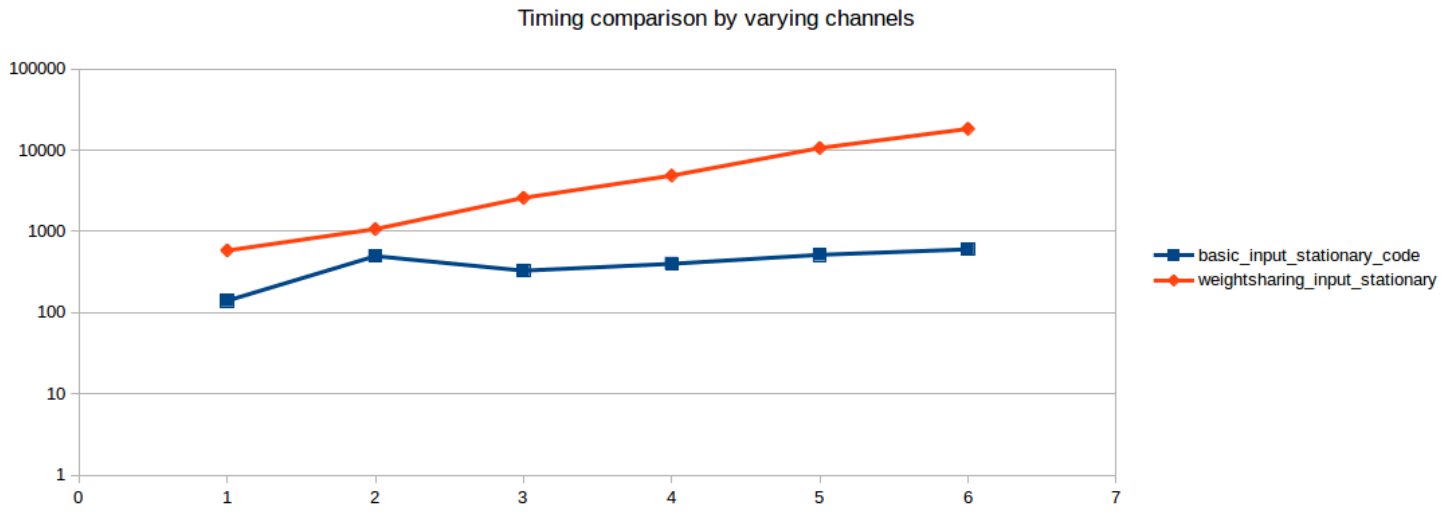
This version is similar to Version 3 of output stationary tiling, instead of the kernel values. In this method, its power along with sign is stored in the kernel matrix. Since each thread computes one output element, each one maintains the sum of the input matrix values that are to be multiplied to the same weight, in an array stored in the thread register. Index of the array can be directly related to the power of the weight (weights are powers of 2, thus the indices are related to the logarithms of the weight) so the all values in the array can added with appropriate bit shift (equivalent to multiplying to the weights). Here Memory accesses are regular to the input matrix tile. Version 2 is just a slight modification of the same but uses constant memory to store the weights to avoid loading the weights for each thread. Values of the weights are stored in the constant memory, i.e. the sign and exponent are not stored separately. Version 2 performance did not show any difference in the performance from Version 1 even though it replaced shared memory call to one constant memory call which is slower.

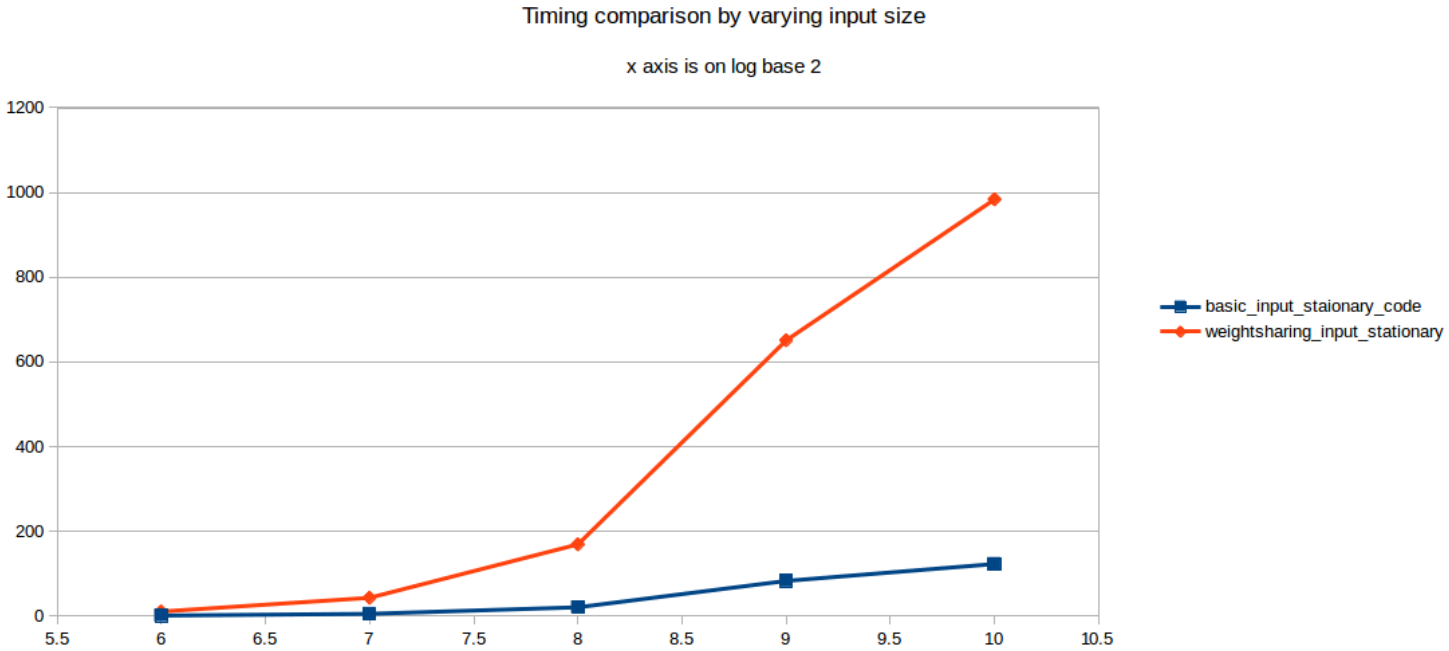
3.5. Timing Analysis

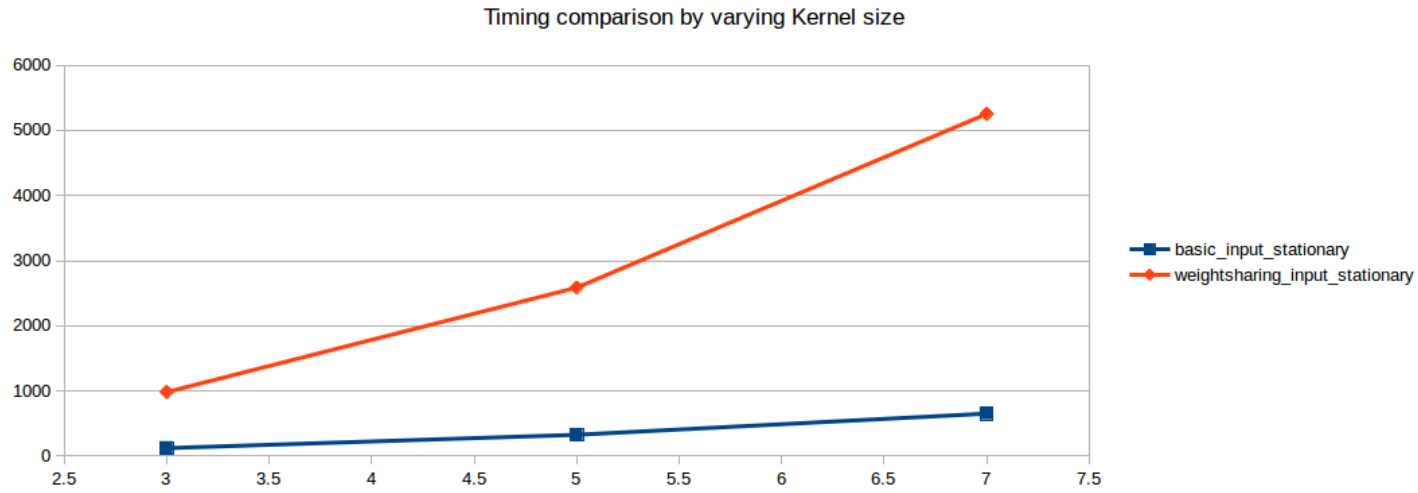
For the Timing Analysis, the set of parameters (unless specified) are as follows:
As earlier, stride length is kept at 1.

W 1024	**n1** 3	**n2** 1
b 3	**T** 3	**D** 4
	N 128	

The time difference increases as the input matrix size is increased, either in dimensions or in channels (for channels y axis scale is logarithmic). Increasing the size of kernel also leads to a slow down much worse than that in output stationary tiling.







3.5.1. Inference

Input Stationary Tiling has a greater slowdown as compared to the basic code as it is more memory bound and has more memory global accesses for loading the kernel, which are needed to load the exponent and the sign. and extra shared memory calls to compare the weights . This results in less computational intensity as the memory calls increased but operations per bytes decreased.

For version 1

- Number of multiplications saved per output element: $D \times T \times T - b$
- Number of extra Memory(shared) accesses (over normal): $T \times T \times D + b$
- Number of extra Memory(global) accesses (over normal): D

4. Conclusion

The use of Incremental Network Quantities in convolution did not result in any speed up. The different tests (as evidenced by the attached plots) varying different parameters suggest that the reduced operation cost is over-dominated by memory calls and introduced control divergence. Testing the basic code by removing multiplications (keeping the number of memory calls same i.e. calling the kernel value in the convolution loop but not multiplying) in the convolution loop gave upto 40 % speedup for large input matrices. This proves the scope for speedup in kernel execution time. Implementation of weight sharing avoiding access memory calls and control divergence

would be ideal to exploit this speed up.

5. Learning Outcome

This project helped understand the implementation side of the concepts covered in class. The project provided an exposure towards working on a problem from in its entirety, i.e. right from reading related work to get a grasp of developments in the domain, to designing and implementing the idea and profiling and debugging the complicated code for the same. The problem statement was challenging and open-ended leaving room for creativity and discretion.