Spatial Separable Convolutional Neural Networks Parallelization and Accelerations

Zhiyuan Ning[1*], Shuangge Wang[2*], Viktor K. Prasanna[2]
[1]Department of Computer Science, University of Southern California
[2]Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California
Email: {zhiyuann, larrywan, prasanna}@usc.edu

***Abstract*—In recent years, convolutional neural network (CNN) has been proven very useful in artificial intelligence and machine learning. In 2017, a group of researchers at Google, developed a novel CNN model named MobileNets, and it has demonstrated significant improvement on inference run time with respect to its counterparts with little compromise on accuracy. MobileNets adopted a spatial separable convolution technique, dividing the convolution computation into two separate processes, the depthwise separable convolution and the pointwise separable convolution. Although there are available software implementation of parallel spatial separable convolution, there is little research focusing on how to optimize the parallelization strategy, which is obvious as we will present some of the shortcomings of the current designs. Firstly, since most GPUs have a thread limit for each kernel, one convolution layer is usually sliced into separate kernels to be processed in parallel. Unfortunately, the current slicing technique creates unbalanced kernel size, and some processes finish earlier than others do, so they go into idle before synchronizing, amounting to huge hardware resource waste. Secondly, the major bottleneck for spatial separable convolution is the pointwise convolution, which is difficult to accelerate by leveraging the power of shared memory due to its dimensions. In this paper, we will introduce two approaches to further optimize the current parallel designs and to address the two problems mentioned above. We first propose to introduce another kernel to better reach kernel dimensional homogeneity, which significantly reduces idle time. Then, utilizing constant memory, we can reduce thee data communication time of pointwise convolution by a large margin.**

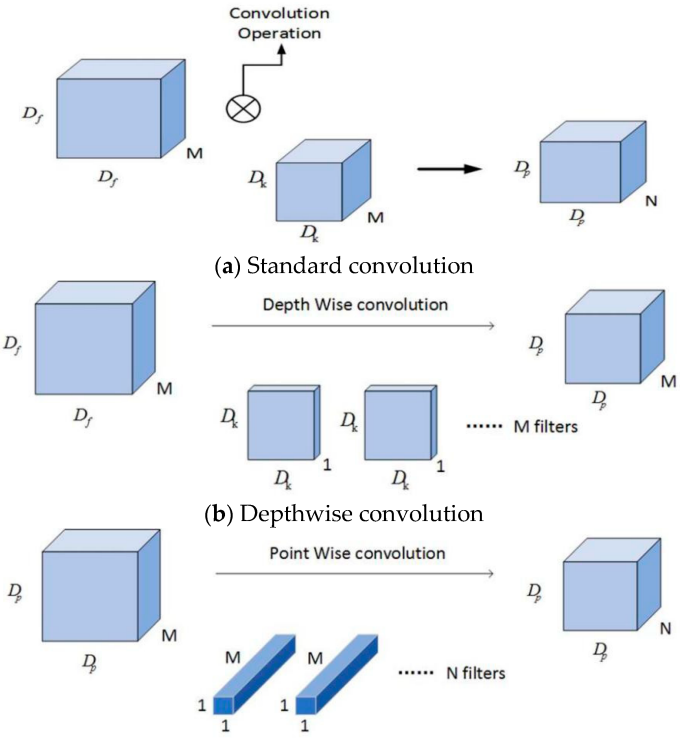
***Index Terms*—Convolutional Neural Network, MobileNets, Spatial Seperable Convolution, Parallel Computing, CUDA**

I. Introduction

In this work, we focus on a specific novel type of CNN model, MobileNets developed by Howard et al. [1]–[4]. This CNN model has significantly less parameters compared to its counterparts like VGGnets with surprisingly small compromise on top-one and top-five error percentage. Therefore, given the minute accurate disparity, a huge decrease in parameters could improve the run time of inferencing. MobileNets has demonstrated great potential in the application of object detection, face attributes, finegrain classification, and landmark recognition.

Besides the forward and backward propagations, there are two specific chunks of code that could be paralleled to optimize performance. The first chunk of code is the multiplication of input from previous layer with the weight of each neuron edge. The second chunk of the code is the convolution layers.

*Both authors contributed equally to this work.



(a) Standard convolution

(b) Depthwise convolution

(c) Pointwise convolution

Fig. 1. Spatial Separable Convolution

However, this approach to CNN has a different types of convolution that's separated into two parts, the depthwise separable convolution and the pointwise separable convolution.

II. Spatial Separable Convolution

What sets MobileNets apart from its counterparts is its separation of the convolution process into the depthwise separable convolution and the pointwise speparable convolution. Fig. 1 is a visual representation of the spatial separable convolution, and we will elaborate the algorithm more specifically below.

A. Depthwise Separable Convolution

Let's assume that our input is an RGB image. Therefore, our input image has three channels, hence $M = 3$. For each channel, we introduce a filter by the size of $D_k \times D_k \times 1$ to detected some desired type of geometric shape (e.g. edges). The k^{th} channel of the input image proceeds to convolution

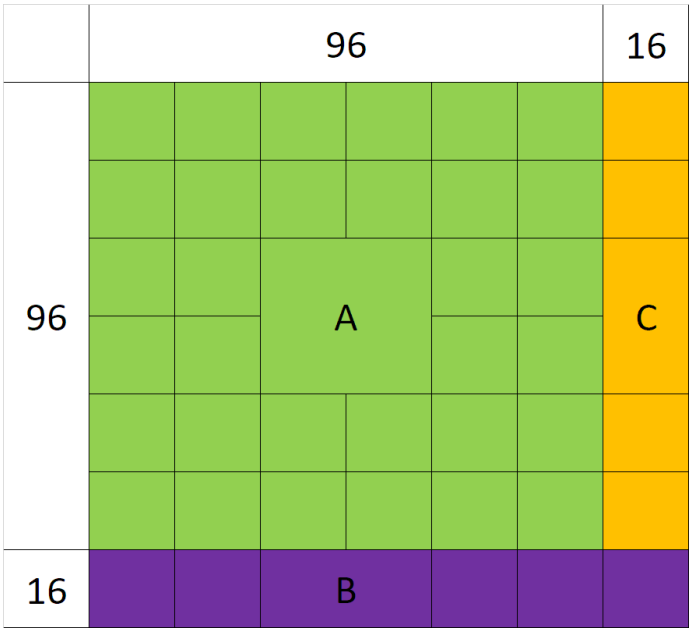


Fig. 2. 3 Kernels

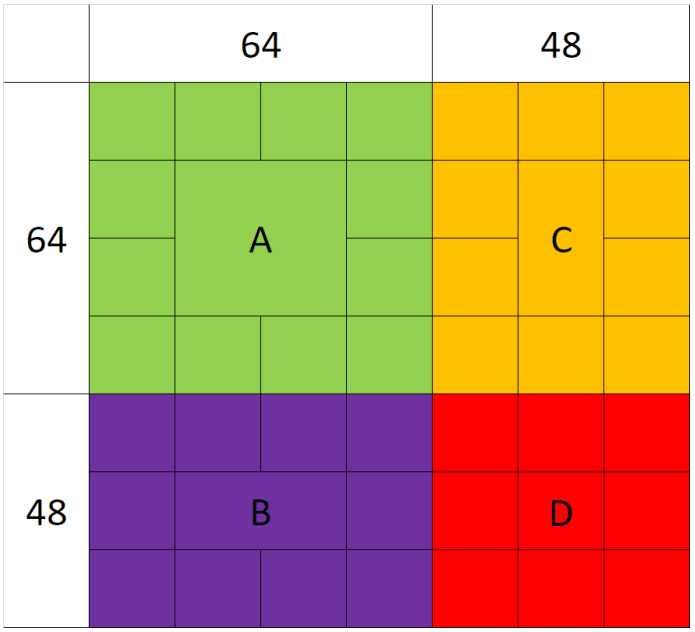


Fig. 3. 4 Kernels

with the k^{th} filter, producing an intermediate image the size of $D_p \times D_p \times M$.

B. Pointwise Separable Convolution

Based on the desired number of channels of the output image, we introduce N filters the size of $1 \times 1 \times M$ in this part of the convolution. All channels of the intermediate image proceeds to convolution with the k^{th} filter, producing the k^{th} channel of the output image. It is fairly easy to realize that spatial separable convolution produces the output image with the same dimension as the traditional convolution approach.

III. Problem Formulation

A. Problem I

Fig. 2 is the original design of each kernel of the first convolution layer. As we can see, the dimension of each kernel is highly unbalanced, so some kernels finishes early before the others do. From Fig. 7 we can see that, in all convolutional layers, the run time for each kernel (part A, B, and C) are drastically different, which creates a lot of idle time for some nodes.

B. Problem II

According to Fig. 6, we can realize that pointwise convolution takes more thah 90% of the run time, which makes pointwise convolution the bottleneck of spatial separable convolution. This is because pointwise convolution, due to the dimension of the filters (see Fig. 4), has low shared memory usage.

IV. Hypothesis

In this section, we propose two hypothesis to address the problems mentioned in the previous section.

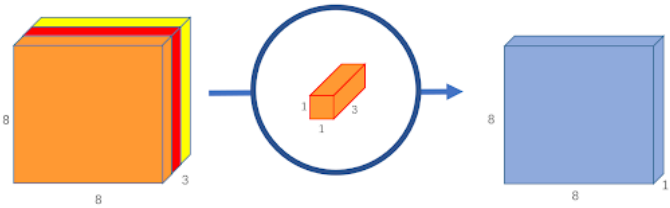


Fig. 4. Pointwise Dimension

A. Hypothesis I

First, we propose to add another kernel and changing the size of each kernel to reduce the variance of each kernel's dimensions. We propose that increasing the kernel number and achieving a optimal of homogeneity of kernel dimensions could increase the parallel efficiency.

B. Hypothesis II

Therefore, we propose that the adoption of constant memory (see Fig. 5) in 1D convolution (pointwise) convolution will address thee problem that pointwise convolution have low shared memory usage and increase the parallel efficiency.

V. Experimental Setup

A. Assumptions

Since Yanhui Wang and Tingyu Zhang have already presented their approach to depthwise convolution using shared memory, we decided that, for our test case, we will use one full convolution layer and two pointwise convolution layer. It is also worth noting that the hypothesis I, if valid, could be universally applied to both full and depthwise convolution.

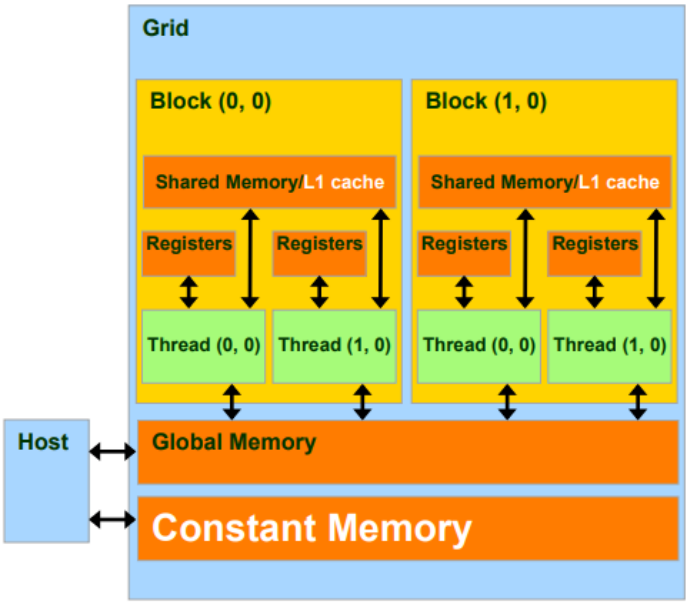


Fig. 5. Constant Memory

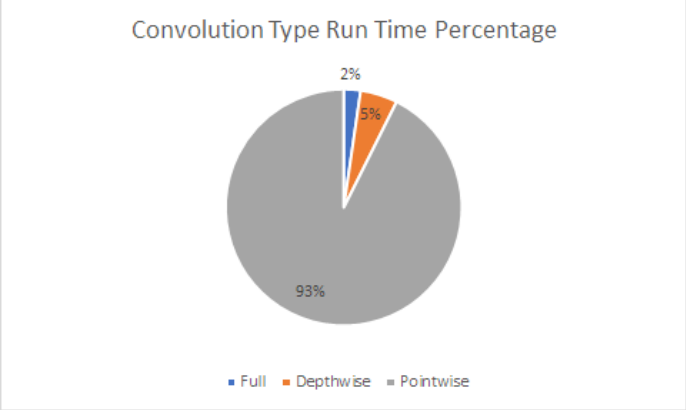


Fig. 6. Convolution Type Run Time Percentage

Even though MobileNets have 29 convolution layers, for the sake of simplicity and immediate results, we will try to improve only two layers, layer 1 for the full convolution and layer 3 for the pointwise convolution.

In this project, we will only be focusing on inferencing because we have been provided with a set of images and weight layers, which is tuned to be functional and robust. In fact, one prominent characteristic of MobileNets is that it's extremely fast and lightweight while inferencing, and we plan to optimize its inference run time even further.

It is also important to note that our baseline is not a serial program but a parallel program already. Therefore, the observed improvement, if any, might be miniscule. Also, our calculated speedup is based on the already parallel baseline.

We will be using CUDA, which stands for compute unified device architecture, a programming model by NVIDIA, to mobilize GPU power to accelerate the code chunks mentioned above.

B. Data Preparation

1) Layer 1: Below is the input and output dimensions of layer 1.

$$INPUT\_LAYER\_SIZE = 225 \times 225 \times 3$$
$$FIRST\_LAYER\_WEIGHT\_SIZE = 32 \times 3 \times 3 \times 3$$
$$FIRST\_LAYER\_OUTPUT\_SIZE = 114 \times 114 \times 32$$
$$FIRST\_LAYER\_CHANNELS = 32 \tag{1}$$

	Grid Size	Block Size
Part A	$< 32, 3, 3 >$	$< 32, 32 >$
Part B	$< 32, 7 >$	$< 16, 16 >$
Part C	$< 32, 6 >$	$< 16, 16 >$

Above is the original grid and block size for layer one, which will be modified to the one below to fit Fig. 3.

	Grid Size	Block Size
Part A	$< 32, 2, 2 >$	$< 32, 32 >$
Part B	$< 32, 2, 3 >$	$< 32, 16 >$
Part C	$< 32, 3, 2 >$	$< 16, 32 >$
Part D	$< 32, 3, 3 >$	$< 16, 16 >$

2) Layer 3:

$$INPUT\_LAYER\_SIZE = 112 \times 112 \times 32$$
$$FIRST\_LAYER\_WEIGHT\_SIZE = 64 \times 32$$
$$FIRST\_LAYER\_OUTPUT\_SIZE = 113 \times 113 \times 64$$
$$FIRST\_LAYER\_CHANNELS = 64 \tag{2}$$

3) Layer 5:

$$INPUT\_LAYER\_SIZE = 56 \times 56 \times 64$$
$$FIRST\_LAYER\_WEIGHT\_SIZE = 64 \times 128$$
$$FIRST\_LAYER\_OUTPUT\_SIZE = 58 \times 58 \times 128$$
$$FIRST\_LAYER\_CHANNELS = 128 \tag{3}$$

C. Benchmark Measurements

In this subsection, we discuss the adjustable parameters in our test benches so that we can report the benchmarks for optimal performance. The key measurements for our experiments will be the run time of each kernel, and we will report the speedup based on our measurements. We will refrain from adjusting the convolutional model of MobileNets as it is powerful, efficient, and accurate enough.

VI. Results

A. Kernel Dimensional Homogeneity

The following algorithm is the pseudocode of the first convolution layer part D of our improved design. The algorithms

for other kernels are similar. The complete improved CUDA implementation can be found here [1]

Algorithm 1 Improved Design: First Layer Part D

$product \leftarrow 0$
$outputOffset \leftarrow 115$
$stride \leftarrow 2$
$filter\_number \leftarrow blockIdx.x$
$output\_Position \leftarrow (filter\_number \cdot 114 \cdot 114) + (64 \cdot 114) + 64 + (blockIdx.y \cdot 16 \cdot 114) + (blockIdx.z \cdot 16) + (threadIdx.x \cdot 114) + (threadIdx.y)$
$weight\_Position \leftarrow filter\_number \cdot 27$
$input\_Position \leftarrow ((64 \cdot 64 \cdot 225) \cdot stride) + (blockIdx.y \cdot 16 \cdot stride) + (blockIdx.z \cdot 16 \cdot stride) + (threadIdx.x \cdot 225 \cdot stride) + (threadIdx.y \cdot stride)$
for $channel \leftarrow 0$ to 2 **do**
  for $row \leftarrow 0$ to 2 **do**
    $product \leftarrow product + ((Layer1\_Neurons\_GPU[(channel \cdot 225 \cdot 225) + input\_Position + (row \cdot 225)] \cdot Layer1\_Weights\_GPU[weight\_Position + (channel \cdot 3 \cdot 3) + (row \cdot 3)]) + (Layer1\_Neurons\_GPU[(channel \cdot 225 \cdot 225) + input\_Position + (row \cdot 225) + 1] \cdot Layer1\_Weights\_GPU[weight\_Position + (channel \cdot 3 \cdot 3) + (row \cdot 3) + 1]) + (Layer1\_Neurons_GPU[(channel \cdot 225 \cdot 225) + input\_Position + (row \cdot 225) + 2] \cdot Layer1\_Weights\_GPU[weight\_Position + (channel \cdot 3 \cdot 3) + (row \cdot 3) + 2]))$
  end for
end for
$Z \leftarrow \frac{product - Layer1\_Mean\_GPU[filter\_number]}{Layer1\_StanDev\_GPU[filter\_number]}$
$Z \leftarrow Z \cdot Layer1\_Gamma\_GPU[filter\_number] + Layer1\_Beta\_GPU[filter\_number]$
if $Z < 0$ **then**
  $Z \leftarrow 0$
end if
if $Z > 6$ **then**
  $Z \leftarrow 6$
end if
$Layer2\_Neurons\_GPU[output\_Position + outputOffset] \leftarrow Z$ =0

The performance table for each kernel and each convolution type of the original design is shown in Fig. 7, and Fig. 8 shows that part A of the first full convolution takes 77% percent of the run time.

Via our improved design from Fig. 3, we achieve a run time for each kernel with less variance (see Fig. 9), meaning that the GPU experience less idle. Not only that, each of our

[1] An improved CUDA implementation: https://github.com/shuanggewang/spatial-separable-convolutional-neural-networks/tree/main

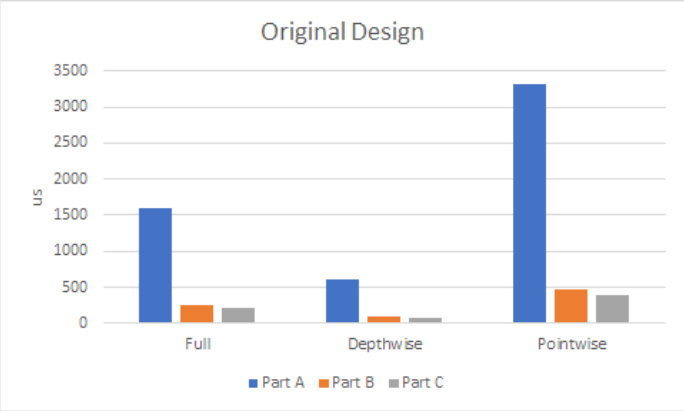


Fig. 7. Original Design Run Time

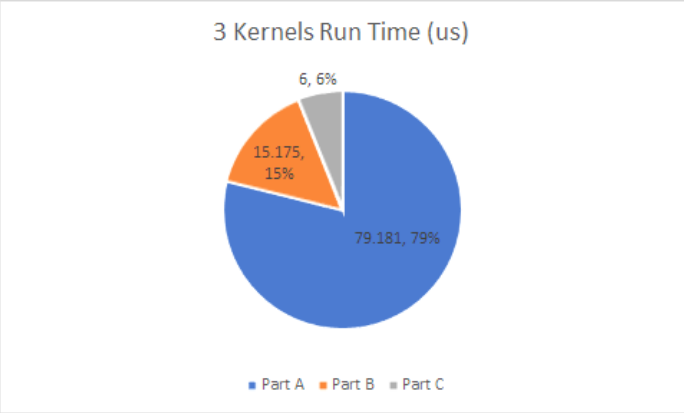


Fig. 8. 3 Kernels Run Time Percentage

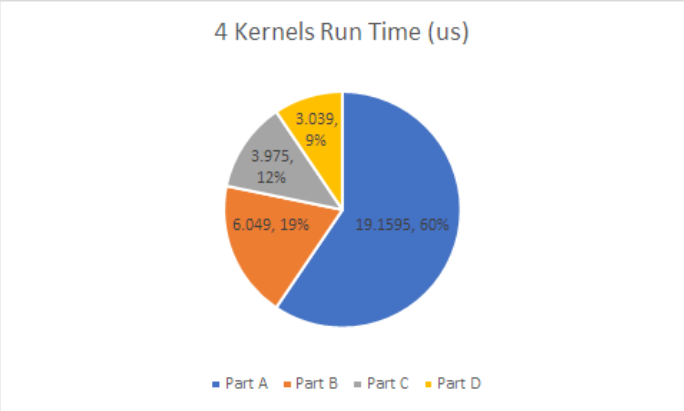


Fig. 9. 4 Kernels Run Time Percentage

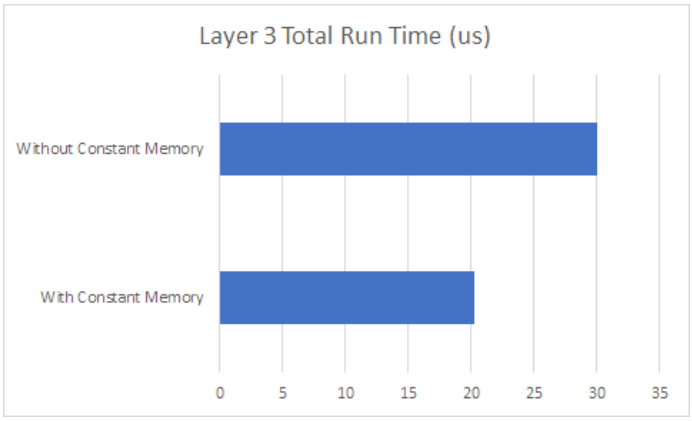


Fig. 10. Layer 3

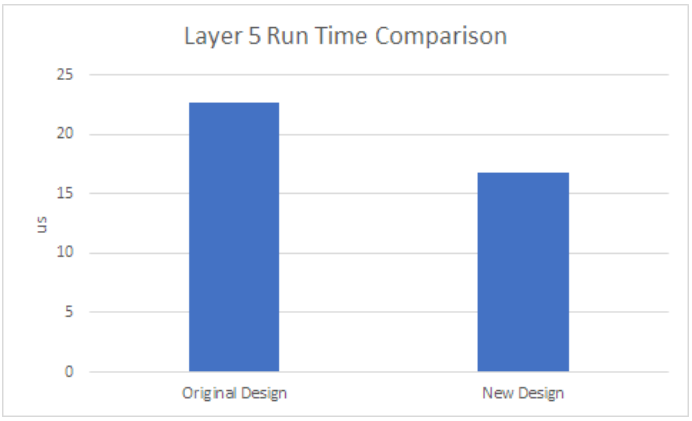


Fig. 11. Layer 5

kernel runs faster than the original design does. Therefore, our algorithm has a speed up of:

$$\begin{aligned} \text{Speedup} &= \frac{\max_{i \in A,B,C} \text{Run Time of } i}{\max_{j \in A,B,C,D} \text{Run Time of } j} \\ &= \frac{79.181}{19.1595} \\ &= 413\% \end{aligned} \tag{4}$$

B. Constant Memory

By adopting constant memory, we achieve a substantial improvement of run time in pointwise convolution, although the speedup is not as substantial as the one we observed in full convolution, which coincides our conviction that pointwise convolution has low memory reuse and therefore is difficult to optimize.

```
__constant__ double Weights[32*64];
```

Above is the dimension of constant memory. According to Fig. 10, the speedup of layer 3 using constant memory is:

$$\begin{aligned} \text{Speedup} &= \frac{30.083}{20.287} \\ &= 148\% \end{aligned} \tag{5}$$

Since the size of constant memory is fairly small, usually 24K, we have to take into the homogeneity of X, Y, and Z value of a matrix when designing the kernels. In layer 5, we re-separate the layer to four matrices with a more homogeneous dimension. Below is dimension of constant memory.

```
__constant__ double Weights_Five_B[64*32];
__constant__ double Weights_Five_C[64*32];
__constant__ double Weights_Five_D[64*32];
```

Fig. 11 shows that the speedup of layer 3 using constant memory with a different cut is:

$$\begin{aligned} \text{Speedup} &= \frac{22.659}{16.829} \\ &= 134\% \end{aligned} \tag{6}$$

VII. Conclusions

In this paper, we showed that, in full convolution layers, by reaching kernel dimensional homogeneity, the kernels experience less idle, accelerating the total run time. We also showed that constant memory, although circumstantial to a small shared memory size, could improve the memory accessing efficiency when memory reuse is low. Although our test cases are limited, these are heuristic conclusions that could be easily applied to other convolution layers without loss of generality. With constant memory, we achieve less data accessing time, and the program becomes more agile since we can adopt a conflation of constant and global memory.

Future directions of research could focus on what is the optimal number of kernels given a convolution layer and its corresponding slicing strategy. For instance, layer 1 could be further separated into nine kernels. Although more numbers of kernels generally gives less variance in kernel dimensions, each kernel has less registers amounting to an increase in communication overhead. Therefore, it is important to strike a balance between large kernel numbers and small communication overhead. Another potential direction of research is focusing on how to distribute the data to global memory and constant memory to achieve maximum efficiency.

REFERENCES

[1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[2] N. Lahoti, "Low power mobilenets acceleration in cuda and opencl," 2019.

[3] X. Chen, J. Chen, D. Z. Chen, and X. S. Hu, "Optimizing memory efficiency for convolution kernels on kepler gpus," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[4] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU technology conference, GTC*, vol. 10. San Jose, CA, 2010, p. 16.