Dear Editors and Reviewers:

Thank you for your letter and for the reviewers’ comments concerning our manuscript entitled “**Optimizing Depthwise Separable Convolution Operations on GPUs” (ID: TPDS-2021-01-0032)**. Those comments are all valuable and very helpful for revising and improving our paper, as well as being important guiding significance to our research. We have studied the comments carefully and have made corrections which we hope will be met with approval. The main corrections in the paper and responses to the reviewer’s comments are as following:

**Responses to Reviewers:**

To Reviewer 1:

1. **Comment:**

*In Section 3.1.2, an optimized implementation is presented but with no source or reference to its origin. Is it an implementation used in earlier works?*

**Response:**

Yes, we use this optimized implementation in our previous work to demonstrate the ineffectiveness of dynamic indexing in shuffle instructions. We add a reference to our previous work for the optimized implementation and cite it at the beginning of Section 3.1.2 in the revised manuscript.

1. **Comment:**

*In section 4.3.1, authors mention extraR is determined through an off-line method. Does this mean its empirically derived since the CUDA compiler always uses such additional registers for a given platform?*

**Response:**

Yes, *extraR* is empirically determined because CUDA compiler always allocates more registers than derived register counts (*Rresult + Roperand + Rtmp* in Formula 1 of Section 4.3.1) for our kernels. Extra registers are usually used to store temporary variables that are necessary to fully utilize GPU arithmetic pipelines. CUDA compiler averagely allocates 40 extra registers for our kernels on both 2080Ti and Xavier, hence we set *extraR* = 40. We add a line below Formula 1 in Section 4.3.1 of the revised manuscript to show the value of *extraR*.

1. **Comment:**

*Typo: In section 6.1.1 and later, change Implicit and Precomp to either all caps or capitalize as they are CuDNN modes.*

**Response:**

Thank you for your suggestion. We have changed all occurrences of cuDNN algorithms, including IMPLICIT, PRECOMP, GEMM, FFT, TILING, WINOGRAD and NONFUSED, in figures and texts to all cpas in the revised manuscript.

1. **Comment:**

*Authors mention that CuDNN Implicit gives the best performance for Depth-wise convolution and hence they compare their results to it over other CuDNN algorithms. It either needs to be shown or mentioned if other works show that. Same applies to choosing CuDNN Precomp for INT8 for point-wise convolution.*

**Response:**

We appreciate your comments and suggestions.

In our experiments, we tested all algorithms of cuDNN on both platforms with FP32 and INT8. Results show that IMPLICIT and PRECOMP obtain best performance among all algorithms for all test cases.

1. For depthwise convolutions with FP32, IMPLICIT is faster than PRECOMP in all test cases on both platforms. Therefore, we choose to compare our approach with IMPLICIT.
2. For depthwise convolutions with INT8, IMPLICIT and PRECOMP are over 10 times slower than our approach in all test cases on both platforms, hence we do not show the results.
3. For pointwise convolutions with FP32, IMPLICIT is faster than PRECOMP in 152 out of 180 test cases on 2080Ti and 147 out of 180 test cases on Xavier, hence we choose to compare our approach to IMPLICIT.
4. For pointwise convolutions with INT8, only IMPLICIT and PRECOMP are supported in cuDNN. PRECOMP is faster than IMPLICIT in 180 out of 180 test cases on 2080Ti and 127 out of 180 test cases on Xavier, hence we take IMPLICIT as the baseline and compare our approach with PRECOMP.

We have added illustrations on how we choose the algorithm to compare with in Section 6.1.1 and the second paragraph of Section 6.2.1 in the revised manuscript.

1. **Comment:**

*Typo: In section 6.1.2, "improves \*over\* cuDNN implicit by...."*

**Response:**

Thank you for your comments and suggestions. We have proofread the manuscript and corrected all typos in the revised manuscript.

1. **Comment:**

*For better understanding of the readers, it would be better to choose a single baseline across experiments. Instead of choosing MobileNet's depth-wise kernel for depth-wise convolutions and cuDNN GEMM as baseline for point-wise convolutions.*

**Response:**

We appreciate your comments and suggestions.

We have changed the baseline of depthwise convolutions to cuDNN GEMM. The updated result is shown in Fig. 9 and speedups have been updated in Section 6.1.2 in the revised manuscript. For pointwise convolutions with INT8, only IMPLICIT and PRECOMP are supported in cuDNN and PRECOMP performs better than IMPLICIT, therefore we take IMPLICIT as the baseline and compare our approach with PRECOMP. We have added illustrations about the choice of the baseline for depthwise and pointwise convolutions in Section 6.1.1 and the second paragraph of Section 6.2.1 in the revised manuscript.

1. **Comment:**

*In the work, authors haven't mentioned what tile sizes or parameter values were chosen by their search-based algorithm. Assuming the results are shown for best parameter values, but a general discussion of which values were chosen for what case would be beneficial to the readers.*

**Response:**

We appreciate your comments and suggestions.

We have updated Table 2 in the revised manuscript to show the values of *WarpH*, *WarpW*, *Blocknum* and *Cnum*, and take parameter values generated for 2080Ti as examples to demonstrate the behavior of our dynamic tile size scheme.

Normally, for a given convolution layer configuration, when the width of the logical layout of output (Fig. 8 in the revised manuscript) is small, our scheme tends to choose a small value for *Cnum*. The reason is that small *Cnum* can generate more warps to saturate GPU. On the other hand, our scheme tends to choose a large value for *Cnum* to reduce the number of warps because there are already enough warps to saturate GPU.

An exception is the layer configuration CONV9 with *IN* = 1 and its parameter tuple is (*WarpH*, *WarpW*, *Blocknum*, *Cnum*) = (12, 2, 2, 32). The width of the logical layout of CONV9 is small, hence we would like to search for a small *Cnum*. However, in this case, *Cnum* = 32 is large. The reason is that our scheme finds that different values of *Cnum* produce similar GPU utilization, then it tries to find a value for *Cnum* to maximize the arithmetic intensity (AI, Formula 4 in Section 4.3.2 in the revised manuscript). In this case, *Cnum* = 32 will maximize AI (the relationship of AI and *Cnum* is detailed in Section 4.2.2 in the revised manuscript). We have added the discussions of parameter values in the third paragraph of Section 6.2.1 in the revised manuscript.

1. **Comment:**

*CuDNN's implementation is lagging behind in terms of performance for DSCs. Therefore, DNN platforms such as Tensorflow (TF) have their own implementations for DSCs (*[*https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/depthwise\_conv\_op\_gpu.h*](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/depthwise_conv_op_gpu.h)*). It would be better to see results for the presented optimizations for this work against the TF's implementation or some other state-of-the-art DNN platform like Pytorch (*[*https://github.com/pytorch/pytorch/pull/22302*](https://github.com/pytorch/pytorch/pull/22302)*).*

**Response:**

We appreciate your comments and suggestions.

PyTorch and TensorFlow implement depthwise convolutions with FP32. Therefore, we use both implementations to conduct experiments on both platforms. The results are shown in Fig. 9 in the revised manuscript. The baseline implementation is cuDNN GEMM. PyTorch achieves an average speedup of 1.6x and 30.1x for the 3x3 filter on 2080Ti and Xavier respectively, and 1.9x and 19.1x for the 5x5 filter on 2080Ti and Xavier respectively. TensorFlow achieves an average speedup of 1.8x and 34.6x for the 3x3 filter on 2080Ti and Xavier respectively, and 2.2x and 25.3x for the 5x5 filter on 2080Ti and Xavier respectively. We can see that TensorFlow is faster than PyTorch because TensorFlow uses a simple block strategy to chooses different blocks sizes for different convolution layers. However, both PyTorch and TensorFlow do not optimize on memory performance which is import for depthwise convolution. In summary, compared with TensorFlow, our approach achieves an average speedup of 1.5x and 1.6x for the 3x3 filter on 2080Ti and Xavier respectively, and 2.2x and 1.7x for the 5x5 filter on 2080Ti and Xavier respectively. We have added the results in the paragraph **FP32 implementation** of Section 6.1.2 in the revised manuscript.

For depthwise convolutions with INT8, PyTorch and TensorFlow use the same code as FP32 and the performance is poor. In summary, our approach is over 10x faster than PyTorch and TensorFlow. We have added the illustration in the paragraph **INT8 implementation** of Section 6.1.2 in the revised manuscript.

For pointwise convolutions, PyTorch and TensorFlow use cuDNN implementations which have been tested on both platforms for both data types. We have added the illustration in the first paragraph of Section 6.2.1 in the revised manuscript.

1. **Comment:**

*Author haven't mentioned in this work that the techniques presented for optimizing depth-wise convolution is inspired from their previous work from CLUSTER 2020.*

**Response:**

We appreciate your comments and suggestions. We have added citations to our previous work in the fifth paragraph of Section 1, the first paragraph of Section 3 and the first paragraph of Section 4 in the revised manuscript.

1. **Comment:**

*Another reference that presents a technique for optimizing convolutions on GPU: A versatile software systolic execution model for GPU memory-bound kernels (SC '19)*

**Response:**

Thank you for recommending the paper *A versatile software systolic execution model for GPU memory-bound kernels (SC '19).* In the following discussions, we use the abbreviation SSAM (Software Systolic Array Execution Model) to refer the SC’19 paper. We carefully read the paper and learnt some insights about the optimization methods in SSAM. Though both SSAM and our methods optimize convolutions along the width and height dimensions, the details are different.

1. **Optimization along the width dimension:** Assume the filter size is 5\*5. In SSAM, 32 threads of a warp will load a 5\*32 patch of the input and slide the 5\*5 filter over the patch to generate 28 output elements. For a 72-channel 56\*56 output, SSAM needs 72\*56\*56/28 = 8064 warps. Our column reuse method ensures that each thread generates one output element, therefore our method needs 72\*56\*56/32 = 7056 warps. We can see that our approach is more efficient than SSAM.
2. **Optimization along the height dimension:** To reduce the number of warps, both SSAM and our approach let one thread produce a column of output elements. Two adjacent output elements of the same column will share some input elements. To avoid reloading shared input elements, SSAM loads all needed input elements into registers, while our approach loads one input element each time and calculate all output elements that dependent on the loaded input element. Thus, our approach uses less registers and can execute more warps on one Streaming Multiprocessor (SM) concurrently.

We have added the discussions at the end of Section 7 in the revised manuscript.

To Reviewer 2:

1. **Comment:**

*Reading the introduction, it does not appear like an extension of a conference paper. I suggest adding a line or two saying what is new in the manuscript compared to conference paper in introduction itself.*

**Response:**

We appreciate your comments and suggestions. We have added citations to our previous work in the fifth paragraph of Section 1, the first paragraph of Section 3 and the first paragraph of Section 4 in the revised manuscript.

1. **Comment:**

*It appears that only square input sizes are being used for convolution. In image processing, the images are more likely to be rectangular.*

**Response:**

We appreciate your comments and suggestions.

Our optimization techniques are not limited to square input sizes. For depthwise convolutions, we first divide the output into sub-blocks and each warp process one sub-block. The height and width of the input will affect the height and width of the output, which determines the number of sub-blocks. Our column reuse and row reuse techniques are applied on the sub-block and will not be affected by the number of sub-blocks. For pointwise convolutions, we divide the output into sub-blocks based on the logical view of the output (Fig. 8 in the revised manuscript). One dimension of the logical view is the product of the number, the height and the width of the input. Therefore, square or non-square input sizes will not affect the size of this dimension and also will not affect our optimization techniques.

As depthwise and pointwise convolutions are commonly used in Convolutional Neural Networks (CNNs), we take convolution layer configurations from CNNs and all of these configurations are square input sizes.

1. **Comment:**

*Please explain a bit more on the performance aspects when batch size increases so that it can be contrasted with cuDNN.*

**Response:**

We appreciate your comments and suggestions.

As the batch size increases, more warps are generated and GPU utilization increases, which leads to different consequences for depthwise and pointwise convolutions.

1. **For depthwise convolutions:** From Fig. 9 in the revised manuscript we can observe that speedups of our approach over cuDNN IMPLICIT fluctuate in a small range as batch size increases. The reason is that depthwise convolution is memory bound and our column and row reuse techniques focus on optimizing memory performance of depthwise convolutions, thus high GPU utilization contributes to neither our approach nor cuDNN IMPLICIT.
2. **For pointwise convolutions:** From Fig. 11 and Fig. 12 in the revised manuscript, we can observe that as batch size increases, speedups of our approach over cuDNN IMPLICIT and cuDNN PRECOMP tend to become lower. The reason is that pointwise convolution is more sensitive to GPU utilization. As batch size increases, generated warps can saturate GPU, thus cuDNN IMPLICIT and PRECOMP can fully utilize GPU and our dynamic blocking size scheme will lose its advantage.

We have added the discussions at the end of Section 6.1.3 and Section 6.2.3 in the revised manuscript.

1. **Comment:**

*I suggest to use ImageNet. Currently, only MobileNet is being used for experiments.*

**Response:**

We appreciate your comments and suggestions.

In our experiments, we use ImageNet dataset as input to MobileNet for training and inferencing. We also conducted experiments on a new model EfficientNet-B0, which also uses ImageNet dataset as input. The results of EfficientNet-B0 are shown in Table 3 and Table 4 in the revised manuscript. For EfficientNet-B0 with FP32, our approach improves the performance of inference by 14.4% and 12.3% on average compared to cuDNN IMPLICIT on 2080Ti and Xavier, respectively. For EfficientNet-B0 with INT8, we obtain 9.9% and 9.6% improvements on average over cuDNN PRECOMP on 2080Ti and Xavier, respectively. Our approach reduces the training time of EfficientNet-B0 by 7.2% on average compared to cuDNN IMPLICIT on 2080Ti. We have added illustrations of EfficientNet-B0 in Section 6.3.2 in the revised manuscript.

1. **Comment:**

There are many typos in the paper.

**Response:**

Thank you for your comments and suggestions. We have proofread the manuscript and corrected all typos in the revised manuscript.

To Reviewer 3:

1. **Comment:**

*\* The paper is well written and I enjoyed reading the technical details of the algorithms presented. The explanation of how the column and row reuse helped the depthwise convolutions was interesting to read and the as shown in the results helped in the performance of the algorithm on NVIDIA GPUs.*

*\* The idea of converting dynamic indices to static indices for better register usage was a neat novel idea that hopefully can be replicated by others in their pursuit for GPU optimization. It would have been even better if there were profiling results comparing the number of registers used to confirm the theory.*

*\* The dynamic tile sizing to improve GPU utilization and hide the memory traffic from GPU global memory has also improved performance of pointwise convolutions.*

*\* It would be have been nice to compare the results of the authors implementation with cuDNN for bigger batch sizes. Is there a tipping point where the cuDNN implementation overtakes the authors implementation ?*

*\* Profiling results showing the impact of the implementation on memory footprint of the applications, gpu utilization and register utilization would add great value to the paper.*

**Response:**

We appreciate your comments and suggestions.

Yes, there is a tipping point where the cuDNN implementation overtakes our implementation.

For depthwise convolutions, we hardly observe this tipping point.

For pointwise convolutions, the tipping point is *IN = 128* (batch size is 128). When the batch size is small, the computation workload of pointwise convolutions is insufficient to saturate GPU and cuDNN implementations adopt a fixed tile size scheme, which results in GPU underutilization. We instead propose a dynamic tile size scheme to generate more warps to saturate GPU for pointwise convolutions with small computation workload. When the batch size increases to 128, the computation workload of pointwise convolutions is sufficient to saturate GPU, thus cuDNN implementations will not be affected by the GPU underutilization. With the help of assemble level optimizations, cuDNN implementations suppress our implementations.