High-performance Architecture Aware Sparse Convolutional Neural Networks for GPUs

Lizhi Xiang
University of Utah
Salt Lake City, Utah, USA
xianglizhi456@gmail.com

P.Sadayappan
University of Utah
Salt Lake City, Utah, USA
saday@cs.utah.edu

Aravind Sukumaran-Rajam
Meta Platforms
San Francisco, California, USA
aravind_sr@outlook.com

ABSTRACT

Convolutional Neural Networks (CNN) are used to analyze data with spatial/temporal structure. In recent years, CNN's popularity has increased exponentially by virtue of its accuracy and applicability. Due to its massive deployment scale, especially in the automotive industry, image analytics, and portable devices, even fractional improvement in performance and power consumption can lead to enormous savings. In this work, we focus on exploiting the sparsity of feature maps and reducing the required number of computations and data movement, leading to improved performance. Compared to kernel sparsity, where the sparsity structure is known apriori, the feature map sparsity is only known during runtime, making this a challenging optimization problem, especially for GPUs. In this paper, we develop a GPU-friendly Sparse CNN framework capable of handling feature map sparsity. The efficacy of our approach is demonstrated by comparing the performance of our implementation with the state-of-the-art implementations. Our approach can also be extended to support upcoming techniques such as feature map pruning and submanifold sparse convolutional Networks.

ACM Reference Format:
Lizhi Xiang, P.Sadayappan, and Aravind Sukumaran-Rajam. 2022. High-performance Architecture Aware Sparse Convolutional Neural Networks for GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 8–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3559009.3569667

1 INTRODUCTION

Convolutional Neural Networks (CNNs) are widely used in many domains [3, 7, 10, 14, 25, 29, 30, 33, 35]. The improvement of performance of popular CNN pipelines is of significant interest. The exploitation of sparsity in the CNN computation is one approach to reducing the number of executed operations, where only non-zero elements (or elements with values above some threshold) are explicitly stored and used in the computations. However, just reducing the number of executed arithmetic operations is not sufficient to reduce execution time.

For example, while it is common to achieve well over 90% of machine peak performance for dense matrix/tensor computations, the highest performing Top500 entry for the HPCG (High-Performance Conjugate Gradient) benchmark, a key algorithm in scientific computing that is dominated by sparse matrix computations, achieves only 3.6% of machine peak in recent Top500 results (June 2022 [20]). Very effective sparse data structures and architecture-conscious design and implementation of the sparse kernels are essential for achieving lower execution time than the dense versions currently used in DNN pipelines.

The exploitation of sparsity in CNNs has been the subject of many research efforts, but the majority of them have focused on *kernel sparsity* [8, 9, 12, 15, 16, 18, 21, 23, 32], where kernel weights that are below some threshold are pruned away, retaining only a fraction of the weights. A challenge with exploiting kernel sparsity is the trade-off between sparsity and accuracy – by pruning away more kernel weights, the number of operations can be reduced, but model accuracy may suffer. In this paper, we focus on the much less addressed option of exploiting sparsity in feature maps [6, 19]. A significant challenge with this option is that feature-map sparsity is *dynamic*, i.e., unknown for each feature-map in a pipeline until it is produced. This is in contrast to exploiting kernel sparsity because kernel weights stay unchanged once trained and are repeatedly used for each sample processed through the DNN pipeline, allowing for expensive one-time off-line analysis and optimization of the sparse representation. However, a significant benefit in exploiting feature-map sparsity is that there is no trade-off with accuracy - it just turns out that a significant fraction of pixels in feature-maps at intermediate layers in image processing pipelines are zero because of the ReLU operators commonly used in these pipelines. Fig. 1 shows the structure of a block in two popular

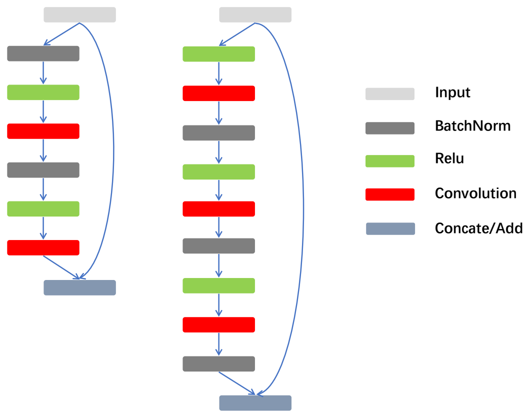

Figure 1: Block structure of DenseNet (left) and ResNet (right)

image-processing pipelines, DenseNet [13] and ResNet [11]. These

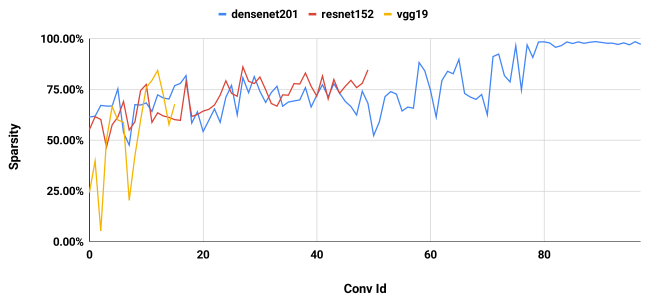

Figure 2: Sparsity as a function of convolution id (DenseNet 201, Resnet152, VGG19)

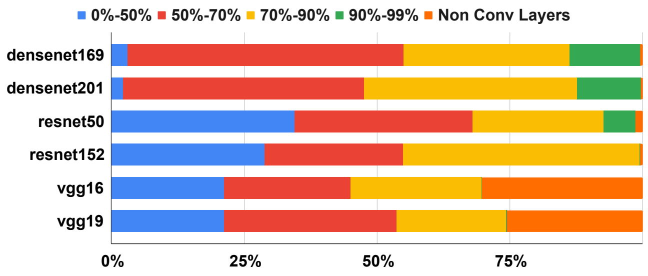

Figure 3: Ratio of execution time: sparse vs. dense layers

DNN pipelines have a sequence of many such blocks, with the sizes of kernels and feature-maps changing through the pipeline. The total execution time is dominated by the *Convolution* layers. It may be seen that between a pair of successive convolution operators, there is always a non-linear ReLU operator, which performs an element-wise operation of "rectifying" the values – convert it to a zero if negative or just pass it on unchanged if positive. The ReLU operator is the reason why a significant fraction of elements in an input feature-map for a convolution operation can be zero. Fig. 2 shows the fraction of zero elements in feature-maps at the input of each convolution layer for three DNN pipelines: DenseNet201, Resnet152, and Vgg19. (1x1 convolutions are omitted, since they consume significantly less time compared to other convolution layers). It may be seen that the sparsity (fraction of zero elements in the feature-maps) is over 50% for the majority of layers for all three DNN pipelines, and often above 75%. Fig. 3 shows the measured distribution of execution time attributable to different layers as a function of feature-map sparsity on a 2080Ti GPU using cuDNN library. The blue bars at the left represent the total fraction of time spent in the convolution layers with under 50% sparsity (i.e., with the relatively dense feature-maps) and can be seen to be only a small fraction of total time, especially with Densenet. Also notable in this figure is the dominance of the time taken by the convolutional layers relative to the other non-convolutional layers (the orange bars at the right), especially for Densenet and Resnet.

Key New Direction: A fundamental challenge in achieving high-performance while exploiting feature-map sparsity is that we do not know until run-time which elements of a feature-map will be zero or non-zero. The efficient dynamic creation and use of compact sparse data structures for the feature-maps poses many challenges, including controlling overheads of dynamic memory management, the overheads of synchronization/coordination among parallel threads to collectively generate the compact sparse representation for a sparse output feature-map, and efficient execution of the sparse convolution operation using it as an input feature-map at a following layer. To overcome these challenges, we develop a novel sparse tensor representation called COO*, along with associated kernels to perform sparse convolution operations.

The main design ideas are summarized below:

Space-Time Trade-off: We design a sparse tensor representation COO* (**Fig** 6) which can accommodate an arbitrary number of nonzero elements, including the extreme case of 0% sparsity. By doing so, we completely eliminate the overheads of dynamic determination of space requirements and dynamic memory management of such space. Although maximal space is allocated to accommodate any dynamic sparsity, the actual volume of data movement will only correspond to the actual non-zero elements. Sparse tensor representations generally use much less space than dense representation. While COO* storage requires slightly more storage than a dense representation, the performance benefits outweigh the slight increase in storage cost. Moreover, the feature map space is generally reused across multiple layers, whereas the kernel weights are kept stationary. Hence the kernel tensor requires much more storage space than feature maps. Therefore, a slight increase in feature map space does not create any practical memory constraint.

Tiling and trade-off between load-imbalance and data movement: Tiling is used to address the problem of minimizing synchronization/coordination overheads among parallel threads. A performance model is developed to choose tile sizes that judiciously balance the trade-off between increasing load-imbalance as tile size is increased and increase in data movement from global memory as tilesize is decreased.(**Sec** 3.5)

Architecture-conscious microkernel design: Customized kernels using the COO* data representation are designed for convolution and ReLU operators, taking into consideration various key factors in optimizing performance on GPUs (**Sec** 3.4) .

The paper makes several contributions:

i) It develops a novel sparse representation (COO*) for efficient processing of sparse tensors whose sparsity structure can only be dynamically known at runtime.

ii) It develops optimized kernels for tiled execution of CNN image processing pipelines where feature-map sparsity is effectively exploited, even when the sparsity factor is as low as 50%.

iii) It demonstrates significant end-to-end performance on several DNN image-processing pipelines (ResNet, Vgg, DenseNet) over dense baselines using the state-of-the-art cuDNN library.

2 RELATED WORK

Convolutions can be implemented using different high-level algorithms such as direct convolution, Winograd, and FFT methods. Liu et al. [22] propose using the Winograd algorithm to accelerate convolutions with sparse kernels.

Several existing works have attempted to increase the efficiency of convolutions by exploiting the sparsity structure of kernels. The Tiramisu Compiler [2] is a polyhedral model-based compiler for sparse and dense data-parallel algorithms. The user has to specify the required optimizations manually, and the performance of the framework is entirely dependent on this specification. The distribution includes an optimization specification for convolutions with sparse kernel on CPUs; however, it does not include sparse feature map optimization configurations for CPUs or GPUs.

The sparsity of the kernels and thereby performance, can be increased using weight pruning [8, 9, 12, 15, 16, 18, 21, 23, 32]. Park et al. [26] propose an efficient sparse-dense multiplication based solution for sparse convolution. The kernel is treated as a sparse matrix/tensor, and the feature map is considered as a dense matrix/tensor. A guided pruning based scheme is used to make the kernel sparse. In [4] Chen proposes a GPU based scheme for convolutions with sparse kernels. Graham et al. [6] propose a sparse convolutional neural network framework that tries to increase the feature map's sparsity using pruning. Since pruning may affect the accuracy, they retrain their model using the pruned sparse feature maps. In contrast to these approaches, the approach we develop exploits sparsity in the input feature maps and therefore has no negative impact on accuracy.

In [34] Xu developed a feature map-based sparse convolution scheme on GPUs, which also tries to make use of the sparse feature map to accelerate the convolution. However, their method outperforms the dense version only when the feature map is at least 90% sparse, and the convolution feature map dimensions are small. Their approach is based on an SpMV formulation, which also makes the prepossessing expensive.

In [19], Liang et al. proposed dynamic feature map pruning. At runtime, they check whether all the values corresponding to a given feature map channel is less than or equal to a given threshold $(\epsilon) \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$. If so, the entire channel is removed. However, the benefits are only applicable if the entire channel can be removed. This is also demonstrated in their experiments (only 1.6% of all channels were pruned for Resnet50). Moreover, since they rely on cuDNN, a new input tensor and kernel must be constructed to reflect the channel removal, introducing additional overhead. End-to-end experiments on CNN pipelines were not demonstrated.

In some cases, such as handwriting recognition, the input image itself is sparse. Since each output is computed based on its receptive field on input (e.g., 3× 3 kernel), after some convolution layers, the entire feature map will become relatively dense. In [21], Lui et al. proposed submanifold sparse convolution to maintain the spatial sparsity. This work only exploits layer-wise sparsity – the computation and associated data-movement can only be removed if all the input elements in a given feature layer is zero.

In many applications, prior knowledge can be used to reason about the sparsity structure. This information can be used to mask unnecessary computations. Ren et al. [27] propose a masking based sparse-blocks convolution network. In comparison, our approach *does not rely on any prior knowledge of sparsity structure* and does not require sparse blocks.

Open source frameworks such as Tensorflow [1, 17] also support many sparse machine learning primitives. However, they do not support sparse convolutions. Theano [31] has support for convolutions with sparse feature maps. However, it only supports single-channel feature maps. The significant difference between these prior efforts and our work in this paper is the ability to handle arbitrary dynamic sparsity in feature maps.

3 CUSNN: SPARSE CONVOLUTION NEURAL NETWORK

3.1 Overview

The fundamental objective of this work is to design a high-performance sparse convolution framework, which accepts a fully trained CNN network, analyzes the feature-map sparsity based on activations, and produces a high-performance GPU CNN network, where our high-performance kernels replace convolution layers with high sparsity.

Table 1: Notation summary.

B	batch size	ρ	density
C	# i/p channels	N	# o/p channels
H	image height	W	image width
R	stencil height	S	stencil width
$\mathcal{I}$	input tensor	$\mathcal{O}$	output tensor
$\mathcal{K}$	kernel tensor	nnz	# non-zeros

Figure 4 provides a high-level illustration of the cuSNN approach. It involves two phases (i) an offline code generation phase and (ii) an online adaptive kernel selection phase. In the offline phase (shown at the top of the figure), the network summary (topology, layer sizes, types, etc.) of a CNN network as taken as an input and customized sparse kernel code versions are generated for each CNN layer, with distinct versions for different sparsity thresholds. As described later (section 3.5), different register-level tile sizes are used for different sparsity levels in the input feature maps, based on an analytical cost model. Further, the code versions are executed and the minimum level of sparsity is determined for which the execution time of the sparse kernel is less than the cuDNN kernel's execution time for dense convolution at that CNN layer. The offline processing is only performed once per CNN layer.

In the online phase that is executed during use of cuSNN for inference, each CNN stage is executed using either one of the customized sparse code versions (marked code cache in the figure) or the cuDNN library kernel, based on a fast dynamic selector that determines the sparsity fraction of the input feature map. We note that the sparse/dense dynamic selector is very lightweight and can be integrated into the previous layer's RELU operator by only adding a zeros counter when it scans the output tensor. If the input feature map is not above the minimum needed sparsity threshold for that layer, the cuDNN library kernel is invoked using a standard dense representation of the input feature map. Otherwise, a sparse representation is created in a new COO* format for the input feature map (details in section 3.4), which is then processed by a customized sparse kernel variant best suited for the specific sparsity fraction of the input feature map. (The dense to the sparse COO* representation overhead is included in our evaluation.)

For the rest of the paper we use 'B' to represent the batch, 'N' to represent the output channels, 'H' and 'W' to represent the image height and width, 'R' and 'S' to represent the kernel height and width, and 'C' to represent the input features. Tensor $\mathcal{I}$ represents

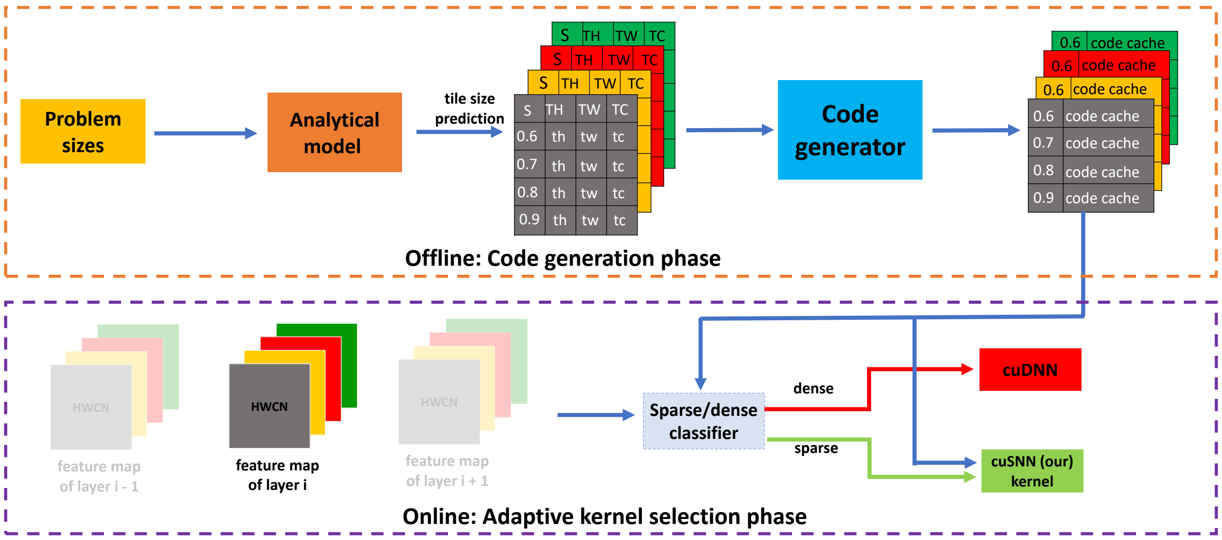

Figure 4: Overview of cuSNN framework

the input feature map, tensor $\mathcal{K}$ represents the kernel (weight), and tensor $\mathcal{O}$ represents the output feature map. ρ represents the sparsity of the input tensor ($\mathcal{I}$). The dense convolution using the above notation can be expressed as:

$$\mathcal{O}(b,n,h,w) = \sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1}\mathcal{I}(b,c,h+r,w+s)\times\mathcal{K}(n,c,r,s)$$

3.2 Challenges

Achieving high performance for sparse computations on GPUs is a challenging task. Compared to other architectures such as CPUs, GPUs have less circuitry dedicated to optimizing control flow (e.g., branch prediction) and lower per-thread cache capacity. Hence, careful software design is vital to achieving high performance on GPUs. Data movement, load-imbalance, and the number of atomic operations are critical factors that determine sparse computations' performance on GPUs.

The GPU memory hierarchy consists of global memory (DRAM), shared L2 cache, a programmable thread block private cache called shared-memory, L1 cache (in some architectures, the shared-memory and L1 cache use the same physical hardware), and thread private registers. Accessing global memory is very expensive, and the capacity of the shared L2 cache is tiny compared to the total number of cores (1.3 KB/core in Nvidia GTX 2080 Ti). Hence, utilizing the shared memory/L1 cache and registers is pivotal to achieving good performance. Since all threads share shared-memory/L1 cache in a thread block, it is beneficial to keep intra thread block shared objects in shared memory. Registers are the fastest entities in the memory hierarchy. While sparse convolutions can have a high degree of data reuse, exploiting sparsity is not trivial. The irregular memory access pattern poses a formidable challenge for data reuse. For example, current GPU compilers won't place typical data structures, even dense data structures such as arrays, in registers when they are associated with sparse computations (due to non-uniform accesses, unknown loop bounds, etc.). Hence meticulous sparsity aware and GPU architecture conscious microkernel design is required to achieve good performance.

Tiling is one way to achieve good data reuse and is employed in many dense computations. However, in sparse convolutions, tiling can lead to severe thread idling. The GPU thread hierarchy consists of a group of threads called warps (32 in current architectures), which are aggregated to form thread blocks, which in turn are aggregated to form the grid. All threads in a warp should execute in a lock-step manner (modern GPUs have some relaxation for this criterion). Consider a design where each warp process a tile of the input tensor, and different threads within a warp process different points within the tile. If the number of elements in a tile is much smaller than the number of threads in a warp, the majority of the threads will be idle – even though the overall processing of the tile is not yet done. In summary, *tiling helps to achieve good data reuse for this scheme but causes load imbalance and thread idling*. Weighing the interplay between these factors is vital in a practical design. A performance model that can quickly determine effective tiles sizes can aid the code generation and provide a high-level cost metric to external tools.

Efficient data representation is another factor that affects data movement. Typical convolutional networks have a chain of convolution layers separated by filtering functions (e.g., ReLU), where the output of one layer feeds the next layer's input. Hence, to avoid data format conversion costs, the output format (including data layout) must be the same as the input, unless different convolutional primitives are used for different convolutional layers. Hence we need to design data structures that are both read and write friendly. Moreover, in practice, a sparse layer could be followed by a dense layer and vice-versa. We need to develop data structures that reduce data movement as well as support efficient dense/sparse format conversion.

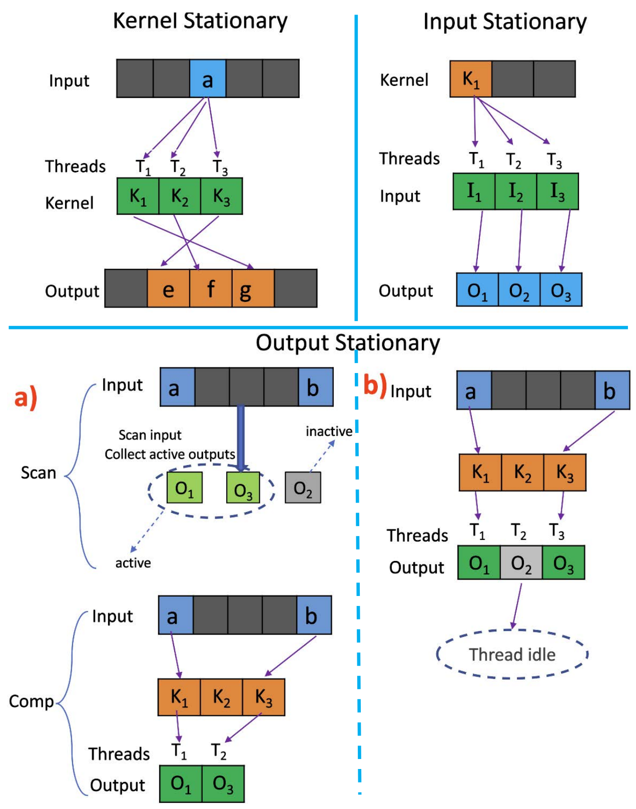

Figure 5: Design exploration

3.3 Design Exploration

To achieve good performance, we have to get maximal reuse of all tensors. However, for ease of understanding, this subsection explains the design considerations assuming that the objective is to maximize the reuse of a single tensor. Note that the actual design, described in section 3.4, tries to get maximal reuse across all tensors and considers load-balancing and thread-idling.

One strategy is to maximize the reuse of kernel elements. Different kernel elements can be distributed across different threads; each thread can load the kernel element(s) in its responsibility to its register(s) (or shared-memory) once and make all the corresponding contributions to the output tensor. This kernel stationary strategy is illustrated at the top left in Fig. 5. However, this approach has two fundamental problems: (i) resource utilization and (ii) synchronizations/atomic operations. If the kernel size is small, distributing it across multiple thread blocks will not achieve the required amount of parallelism in GPUs, resulting in under-utilization of the resources (a form of load-imbalance). Moreover, the partial results generated by different threads may contribute to the same output location. Hence, expensive atomic operations and/or synchronizations are required.

Similarly, maximal reuse of input tensor can be achieved by loading the corresponding into shared-memory (or registers). Such a input stationary strategy is illustrated at the top right in Fig. 5. Like the maximal kernel reuse case, the contributions from the same input spans across multiple output locations; Hence sophisticated designs must be explored to minimize or eliminate the cost of atomic operations and/or synchronizations.

Targeting maximal reuse of the output tensor is another possibility. Each thread can load an output element once to its register (or shared-memory), compute all contributions and write it out. The major challenge here is that the information regarding whether an output element is active or is not known a priori and is determined entirely by the input sparsity structure. One way to overcome this challenge is to perform an initial scan of the input sparsity structure to determine the output structure and distribute the threads only among the active output elements. Output Stationary (a) in Figure 5 depicts this scheme. The other way is to assign a thread to each output element, irrespective of whether it is active or not, and ignore computations to inactive elements. Output Stationary (b) in Figure 5 depicts this scheme. The former suffers from the initial scan cost, and the latter suffers from wasted resources as many threads would be idle, especially if the sparsity is high.

Achieving good performance requires sufficient reuse of all tensors and not just one. Every operation requires one operand from each of the three tensors; even fetching a single operand from global memory per operation will expose memory latency and thereby drastically impact performance. Any approach that targets the reuse of multiple tensors should simultaneously handle the challenges associated with the reuse of each tensor described earlier. Our sparse kernel design, described in detail in the next subsection, uses an output stationary strategy at the register level, an input stationary strategy at the shared-memory level, as well as sufficient register-level reuse for each global-memory load for the weights. The design also ensures good intra-warp, intra-thread block, and inter-thread-block load balance.

3.4 cuSNN kernel design

Data Structure:

The format in which the sparse tensors are kept plays a significant role in determining the runtime efficiency. Several prior works have studied efficient sparse formats in the context of graph primitives and machine learning primitives such as recommender systems. Most of these works assume that the sparse matrix structure is either known apriori or the same sparse matrix is used multiples times. If the structure is known apriori, then expensive offline decisions can determine which format is the best. If the structure is not known apriori, but the same sparse matrix is used multiple times (e.g., Pagerank), the new format's performance can amortize the one-time format conversion cost. Both these approaches are not suitable for the input/output feature maps whose structure is neither known apriori nor is used multiple times. In addition, the data structure should allow quick conversion between the sparse and dense data formats.

One possibility is to keep the image in dense format and to check if the value is non-zero. Such an approach can reduce the number of floating-point operations required, but not data movement. Note that each input feature map element has $R * S * N$ uses, and hence during convolution, the zero values could be unnecessarily read multiple times from global memory. It could also lead to thread-idling as some threads may not have work. Another possibility is to extend the standard Coordinate format (COO) to represent the

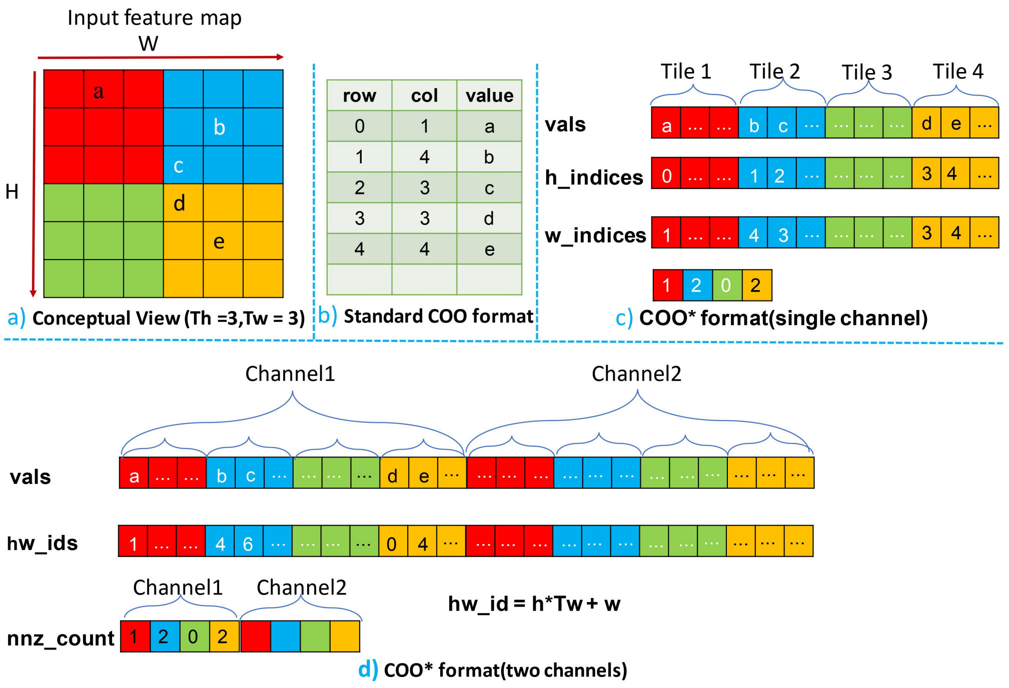

Figure 6: Data representation of sparse feature maps

sparse tensor. Standard COO format represent the matrix as a set of triples (h,w,x), where x is a non-zero entry in the matrix and h,w represent the row and column indices(See Figure 6 (b)). Similar to the row-pointer in CSR format, each channel will have a start pointer, which points to the first non-zero element in that channel. The 'value', 'x', and 'y' coordinates are kept in the COO format. While this format seems reasonable, it suffers from high format conversion overhead. To convert elements from the dense format to sparse format, we have to place non-zeros in the first channel contiguously. This is followed by placing non-zeros in the second channel and then non-zeros in the third channel and so. In order to do this, we need to find all the non-zero elements first and then compute its prefix sum to figure out where each channel should begin. This requires two passes over the data, which increases the data movement and requires synchronization, which exposes latency.

A simple but powerful extension to the COO format can solve the latter issue – the COO* format that we propose in this paper. Our format is motivated by the fact that the ultimate goal of sparse representation for CNNs is to remove unnecessary computation and reduce global memory data movement. In COO* format, each channel is allocated memory, assuming that it is fully dense; this removes the need for prefix computation to find the starting location and does not require expensive synchronization. However, we only store the number of non-zero values and the values themselves. The remaining space is left unused. A count array (*nnz_count*) keeps track of the number of nnz elements in each channel. Figure 6 (c) shows the proposed format corresponding to Figure 6 (a) which only has one channel. Figure 6 (d) shows how multiple channels can be represented using our format. COO* representation requires two major buffers, the first buffer is to store the non-zero elements. Assuming that the data type is float (4 bytes), the storage requirement for storing non-zero elements is given below:

$$ceil(\frac{H}{TH}) \times ceil(\frac{W}{TW}) \times ceil(\frac{C}{TC}) \times (TH \times TW \times TC) \times 4. \quad (1)$$

The second buffer is the index_buffer which indicates the positions of the non-zeros. The indexing buffer compress the (h,w) coordinator into one linear index. Hence, it takes same amount of words compare with the nnz_buffer. However, we can use char (1 byte) as the data type for the index_buffer. The memory required for index_buffer in bytes is shown below:

$$ceil(\frac{H}{TH}) \times ceil(\frac{W}{TW}) \times ceil(\frac{C}{TC}) \times (TH \times TW \times TC). \quad (2)$$

If we ignore the ceiling operation in Eq 1, COO* sparse data representation takes 1.25X memory space compare with dense format tensor. With this format, each thread block (or warp) can read all values corresponding to a particular channel from the dense format and save them in the sparse format very efficiently (format conversion friendly). Since the number of non-zeros is known apriori, the total work can be efficiently distributed between threads. Moreover, during convolution, each thread block (or warp) can read all non-zero values corresponding to a particular channel from COO* format efficiently. Furthermore, during convolution, the zero elements (unused space) are not read from the global memory, thereby reducing data movement. The COO* format also allows us to keep the sparse tensor in tiled form. Algorithm 1 shows the dense to sparse feature map conversion algorithm. Essentially, for each element in the input tensor, we decompose its co-ordinates to corresponding intra-tile iterators (th, tw, tc) and inter-tile iterators(h, w, c), where th, tw, tc are the tile sizes along H, W, and C, respectively. For each non-zero element, we update the *len[tileId]*, which corresponds to the number of non-zero elements in that tile. The value returned by the *atomicAdd* operation (old value at *len[tileId]*) is used as the index of the current non-zero element within the tile. Once the index is identified, we write the co-ordinate information and values in the corresponding arrays. *The significant difference between this algorithm and typical formats like COO is in computing the output write location.* Consider unordered COO format, which uses a global counter used to determine the write location. This global counter is incremented to add a new element, and the updated counter value (or the previous value) is used to determine the write location. The key here is that any element can be written in any position in the array and hence is not directly compatible with tiling. While it is not impossible to support tiling in COO format, as discussed earlier the cost of prefix sum required to determine the write location will overpower any practical benefit. As shown in Algorithm 1, figuring out the write location in our format is straightforward. We can efficiently determine the *tileid* for a given element, and the element can only be placed in that tile. A counter maintains the active number of elements in each tile.

Algorithm 1: Dense to COO* format conversion

```
Input: Dense feature map ($\mathcal{I}$)
Output: Sparse feature map ($\mathcal{SI}$) in COO* representation
parameter: thread block size (BlkSize),thread
           blocks(NumBlks)
index = blockId * BlkSize + threadId
while $index < C*H*W$ do
    v = $\mathcal{I}$[index]
    if $v! = 0$ then
        c = index / (H*W)
        h = (index - c * H * W)/W
        w = (index - c * H * W) - h*W
        th = h - floor($\frac{h}{TH}$)*TH
        tw = w - floor($\frac{w}{TW}$)*TW
        tc = c - floor($\frac{c}{TC}$)*TC
        tileId = $\frac{c}{TC} * \frac{H}{TH} * \frac{W}{TW} + \frac{h}{TH} * \frac{W}{TW} + \frac{w}{TW}$
        writePos = atomicAdd(len[tileId],1)
        $\mathcal{SI}$.val[tileId*TH*TW*TC+writePos] = v
        $\mathcal{SI}$.h[tileId*TH*TW*TC+writePos] = h
        $\mathcal{SI}$.w[tileId*TH*TW*TC+writePos] = w
    end
    index += NumBlks * BlkSize
end
```

Kernel:

Our scheme targets good reuse of all the three tensors (input, output, and kernel) either from shared-memory/L1 cache or registers. We tile the sparse input feature map along all three dimensions (H, W, and C). We employ uniform tiling on the iteration space resulting in tiles of shape $TH * TW * TC$. Each such tile is then mapped to a thread block. Each thread will iterate over the R, S, and TC dimensions sequentially. Different threads are mapped to different output channels. Each thread does the following computation:

$$O(n, th, tw) = \sum_{c=0}^{TC-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \mathcal{I}(c, th - r, tw - s) \times \mathcal{K}(n, c, r, s)$$

If the number of output channels (N) is higher than the thread block size, we will use thread coarsening along the output channel dimension. I.e., each thread will process $\frac{N}{BlkSize}$ output channels sequentially. The threads in a thread block will read the input tile to shared memory in a coalesced manner. The kernel elements are streamed to shared-memory and are double-buffered. Each input gets full reuse from the shared-memory. For each output element, the total number of expected intra-tile reuses is $\rho * TC * R * S$. In order to maximize the performance and minimize data movement, we need to keep the partial output as close to the cores as possible. Shared-memory is one option, but its bandwidth is less than registers. Moreover, in modern GPUs, the shared-memory capacity is less than the collective register capacity. Using a smaller memory will force us to choose smaller tile sizes, which will result in reduced data reuse. Hence it is vital to keep the output elements in registers. In our scheme, each thread will hold a buffer of size $TH * TW$ in registers, which are used to store the intermediate output elements. In addition to data reuse, there are several benefits to this strategy. One significant advantage is that we get perfect intra warp level load-balance, inter warp level load-balance, zero warp diverge, and zero write collisions. We also achieve good input coalescing and output coalescing.

```
//Input: Sparse input feature map I, Dense kernel K
//Output: Sparse output feature map O
shared input_val[TH*TW*TC], input_h[TH*TW*TC],
       input_w[TH*TW*TC]
float temp_result[TH][TW], kernel[R][S]
unsigned int tile_id = blockId%(H/TH * W/TW)
unsigned int batch_id = blockId/(H/TH * W/TW)
unsigned int output_n = threadIdx.x
//copy kernel elements from global to shared
copy(input_val,input_h,input_w,gInput_val,...)
syncthreads() //synchronize all threads in a thread block
for c = 0 to TC:
  copy(kernel,gKernel,n)
  for (v,h,w) in (input_val,input_h,input_w):
    for y_out = max(0,h-R+1) to min(h,H)
      for x_out = max(0,w-S+1) to min(w,W)
        y_kernel = y - y_out
        x_kernel = x - x_out
        result = v * kernel[y_kernel][x_kernel]
        temp_result[y_out*TW+x_out] += result result
// Write the output back to memory
for th to TH:
  for tw to TW:
    y = tile_id/(W/TW)*TH+th
    x = tile_id%(W/TW)*TW+tw
    O[batch_id*H*W*N+y*W*N+x*N+n]+=temp_result[th*TW+tw]
```

Listing 1: Simplified psuedocode corresponding to our approach

Listing 1 shows the simplified pseudocode corresponding to our sparse feature map-dense kernel approach. Figure 7 illustrates the data tiling, shared-memory usage and work division used in

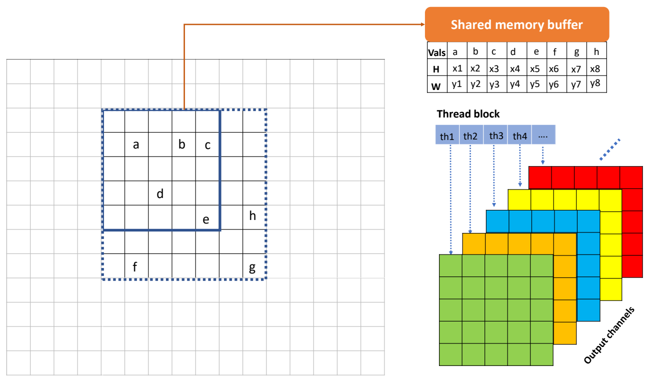

Figure 7: Illustration of Listing 1. Each tile includes a copy of the halo region.

Input data volume	$H * W * C * \rho$
Kernel data volume	$\frac{H}{TH} * \frac{W}{TW} * R * S * C * N$
Output data volume	$H * W * N * \frac{C}{TC}$
Total threads	$\frac{H}{TH} * \frac{W}{TW} * \frac{C}{TC} * BlkSize$
#Atomic operations	$H * W * N * \frac{C}{TC}$

Table 2: Parameters that affect performance and the corresponding values parametrized by tile-sizes.

Listing 1. We use `blockDim` to denote the number of threads in a thread block and `threadId` to denote the id of the thread within a thread block. Line 7 and 8 computes the batch id and channel id that a particular thread is responsible for. The `temp_result` is a temporary array to hold the output result. Loop in line 12 is the coarsening loop which iterates over the input channel. In line 13 we load the kernel elements from global memory and the loopnest in line performs the actual computations. Once all the computations are done, the results from `temp_result` are written back to memory (loopnest in line 22).

Micro-kernel: The key challenge in listing 1 is to ensure that the `temp_result` array, which corresponds to the output elements, stays in registers. Unfortunately, GPUs do not explicitly allow us to place data directly in registers, and ensuring that the compiler does the right data placement is a non-trivial task. In order to ensure that `temp_result` is placed in registers, all the access functions corresponding to it should be statically resolvable – all array iterators should have compile-time constant loop bounds, and the access function should be a non-parametric affine function of the array iterators or constants. A direct translation of the code in listing 1 wont place the `temp_result` in registers (the `y_out` and `x_out` array iterators that are used to access `temp_result` does not have constant loop bounds). Hence we carefully designed a micro-kernel that satisfies all the constraints that the compiler requires to place the `temp_result` array (output) elements in registers. We ensured that the compiler does the right job by checking the SASS code. This micro-kernel design achieves a 2x to 4x performance improvement over the original code.

Load Balance: As explained earlier, since we distribute the threads in a thread block across different output channels, all the threads in the same thread block have the same amount of work. Hence our strategy has good inter and intra thread block load balance. However, if the number of output channels is lower than the block dimension, we assign different warps to process different input feature map tiles to prevent thread idling. In order to achieve good load balance, all the threads in a thread-block collectively bring all input feature map elements corresponding to their share of tiles. We then compute a prefix sum of the elements across all the tiles and divide the work evenly. The cost of prefix operations is small compared to the performance gained from load balancing.

Code generation: We developed a python based code generator that emits CUDA code. Since the actual sparsity of the feature map is not known during the offline phase, we first generate a sparsity list that consists of numbers in the [0.4 to 0.9] range in steps of 0.05. Each entry in this list and the layer dimensions are fed to the analytical model, which predicts the best possible code configuration (tile sizes and other parameters). The predicted configuration is passed to the code generator, and the generated code is saved in the code cache to be used at runtime. On average, the time for this one-time pre-processing step is around 15ms per layer of the DNN pipeline, which includes the modeling and code generation time.

3.5 Tile size selection

Selecting the right tile-sizes at register level and shared-memory level is crucial in determining the data-reuse potential and occupancy and thus plays a big role in determining the performance.

Table 2 shows the relationships between data movement and tile size along with number of atomic operations. Apart from data movement and atomic operations, the tile-sizes also affect the achievable occupancy and concurrency. For example, as TH and TW increase, the data reuse of the kernel also increases which decrease the data movement volume for the kernel. However, this will also increase the register usage, ultimately reducing the occupancy(concurrency). Thus a code version with optimal data-movement may not be the best performing one. We also need to ensure that all GPU cores have some work assigned to them. A low TH and TW will increase the total number of thread blocks and thereby increase concurrency (at the cost of data movement). Tile size along the input channels (TC) also has a similar effect. A low TC enables more concurrency, but a high TC enables more output reuse and will reduce the total number of required atomic operations. In addition to tile size parameters, execution parameters such as block size also play a role in determining the performance.

A straight forward way to select the tile size is to do an exhaustive search. For a given convolution layer, we can obtain the average sparsity for this layer by feeding sample images to the network. We can using this information to an exhaustively try all possible code configurations to determine the best candidate. However such an approach is very expensive. Moreover, using average sparsity may degrade the performance of some images.

Instead, our approach is based on analytical cost modeling. We propose a three-stage model. First, we enumerate all possible configurations, which is the cross-product of all possible tile sizes and execution parameters. In stage 1, we eliminate all configurations with low occupancy (occupancy < 50%). We can estimate the occupancy based on the shared-memory and register level tile sizes.

In stage 2, we estimate the total data movement of each configuration based on the equations provided in Table 2. We select the top one third configurations with the least data movement. Stage 1 and 2 ensure that the selected configurations have low data movement and high occupancy. Stage 3 estimates the time of each configuration using our analytical model as follows. Let *num_blks* represent the number of thread blocks required to process a given input activation for a given tile size. It can be computed as

$$num\_blks = \frac{H}{TH} \times \frac{W}{TW} \times \frac{C}{TC} \tag{3}$$

Let *occupancy* represent the occupancy corresponding to a given code configuration and ρ represent the density.

Let $max\_ths, comp\_thr$ represent the maximum number of simultaneous threads allowed by the GPU hardware and theoretical compute throughput of the GPU respectively. Let *waves* represent the number of computational waves required for the kernel.

$$waves = ceil((max\_ths \times occupancy)/(num\_blks \times blk\_dim)) \tag{4}$$

Let *blks_wave* represent the number of thread blocks can be run concurrently in each wave.

$$blks\_wave = (max\_ths \times occupancy)/blk\_dim \tag{5}$$

Let $flops\_blk$ to represent the total number of floating point operations inside each thread block and $flops\_wave$ to represent the number of flops in a full wave.

$$flops\_blk = 2 \times TH \times TW \times TC \times N \times \rho \tag{6}$$

$$flops\_wave = blks\_wave \times flops\_blk \tag{7}$$

Let *time_wave* to represent the estimated runtime for each wave.

$$time\_wave = flops\_wave/(comp\_thr \times occupancy) \tag{8}$$

The execution time of a kernel is determined by the time at which last wave finish execution. This can be computed as

$$estimated\_time = time\_wave \times waves \tag{9}$$

We compute the estimated time for all configurations remaining after state two and select the configuration with the lowest predicted execution time.

3.6 Discussion

We believe our approach can be extended to current and upcoming architectures. (i) Most architectures share a common cache, and each core has a private cache. Hence our micro-kernel design principles are directly applicable. (ii) We can also extend the cost model to support new architects as the basic principles remain the same. (iii) COO* format can also be used in spatial array architectures.

One advantage of the COO* format on these architectures is that the PEs do not have to communicate to determine the output location. Without this format, the PEs would initially have to determine the number of non-zeros each PE will generate, communicate with other PEs to determine the output location, and then perform the actual computation. Such a scheme will significantly impact the synchronization and communication costs.

We also believe that our approach can be beneficial for graph applications. An example of a pertinent graph application is GNN (Graph Neural Networks). SpMM (Sparse-dense Matrix Multiplication) is the most time consuming kernel in GNNs. If we set R=S=1 for the sparse convolution kernel developed in this paper, we essentially have an SpMM operation. With GNNs, the randomized sampling of the edges of the graph results in dynamic sparsity patterns similar to the dynamic sparsity of the input feature maps handled in this PACT submission.

4 EXPERIMENTAL EVALUATION

4.1 Software and Hardware Configuration

We use two machines for benchmarking i) Nvidia GTX 2080 Ti machine(68 SMs, 11 GB running Ubuntu 20.04 LTS and, ii) Nvidia Volta V100(84 SMs, 32GB) machine running Ubuntu 18.04.4 LTS. We used CUDA 11.0 paired with cuDNN 8.0.2 on the 2080Ti machine and CUDA 10.2 paired with cuDNN 7.6.5 on the V100 machine. These machines represent two different GPU architectures (Volta and Turing) and are designed for two different user groups (enterprise, consumer).

4.2 Networks and Dataset

We used two DenseNet[13] networks (169, 201), two ResNet[11] networks(50, 152) for evaluation , and two VGG[28] networks (16, 19) all of which are widely used. We extracted the kernel weights from the corresponding pre-trained network in Tensorflow. We used the images from the ImageNet [5] database for evaluations. We analyzed the sparsity structure and time fraction by from 100000 images from the traning set of Imagenet[5] (Figure 2 and Figure 3). Note that all the input images are **dense**. For reporting performance, we randomly selected 10000 images from the test set of Imagenet [5]. Note that the set of images used to analyze the sparsity structure and the set of images used for testing are disjoint.

4.3 Effectiveness of Analytical Modeling

Before presenting experimental data comparing performance of cuSNN against cuDNN, we first assess the effectiveness of our approach to selection of code version via analytical modeling. For each input activation at a given layer, our objective is to select the best possible configuration - either dense execution or one among the set of pre-compiled sparse kernels corresponding to different register-tile sizes. We compare our analytical modeling approach for this selection with two *exhaustive* search approaches:

Offline Exhaustive Search: In figure 8, we compare the execution time (average over 100000 test images) per convolution layer for the choice of code version made by our analytical modeling versus one based on offline exhaustive search. This search was done by evaluating the performance of all code configurations on a set of sample images and then selecting the configuration with minimal

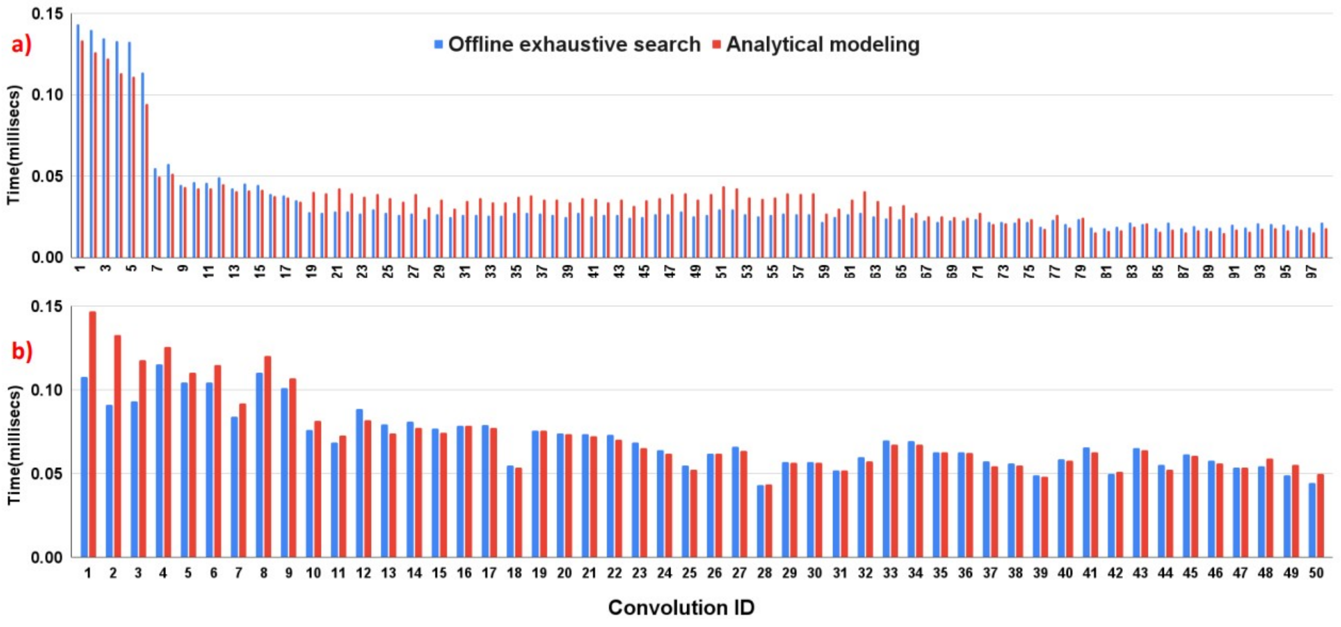

Figure 8: Run-time comparison: offline exhaustive search vs. modeling V100 for (a) DenseNet 201 and (b) ResNet 152

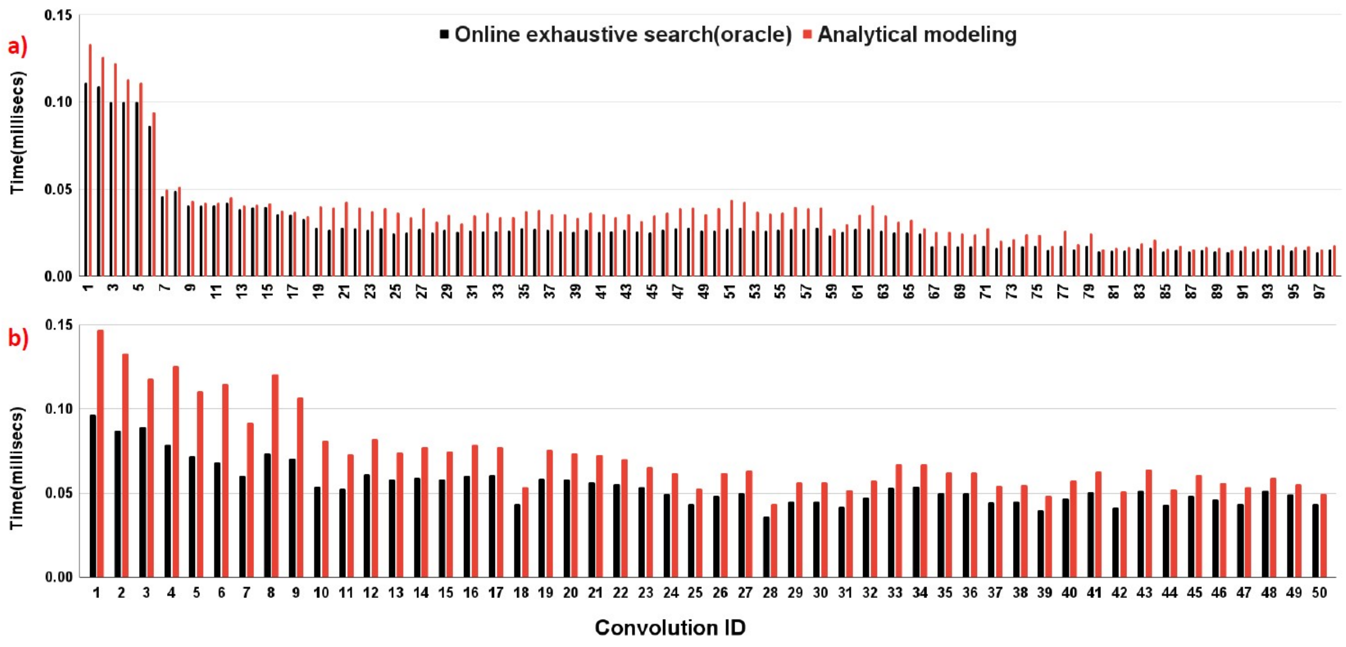

Figure 9: Run-time comparison: online exhaustive search (oracle) vs. modeling V100 for (a) DenseNet 201 and (b) ResNet 152

average cost across the images. Clearly, there is no guarantee that this on-average best configuration is the best for a given test image. Indeed, in some cases, the average execution time using the configuration from offline exhaustive search is worse than that from our analytical modeling. Overall, our model is better or comparable to selection via offline exhaustive search. The same performance characteristics hold for the remaining networks. The trends on the 2080Ti machine are similar and omitted for brevity.

Online Exhaustive Search: Figure 9 compares our modeling against dynamic online exhaustive search (oracle). In this test, for each test image, we measure the performance of all configurations by actually executing all of them and select the best-performing

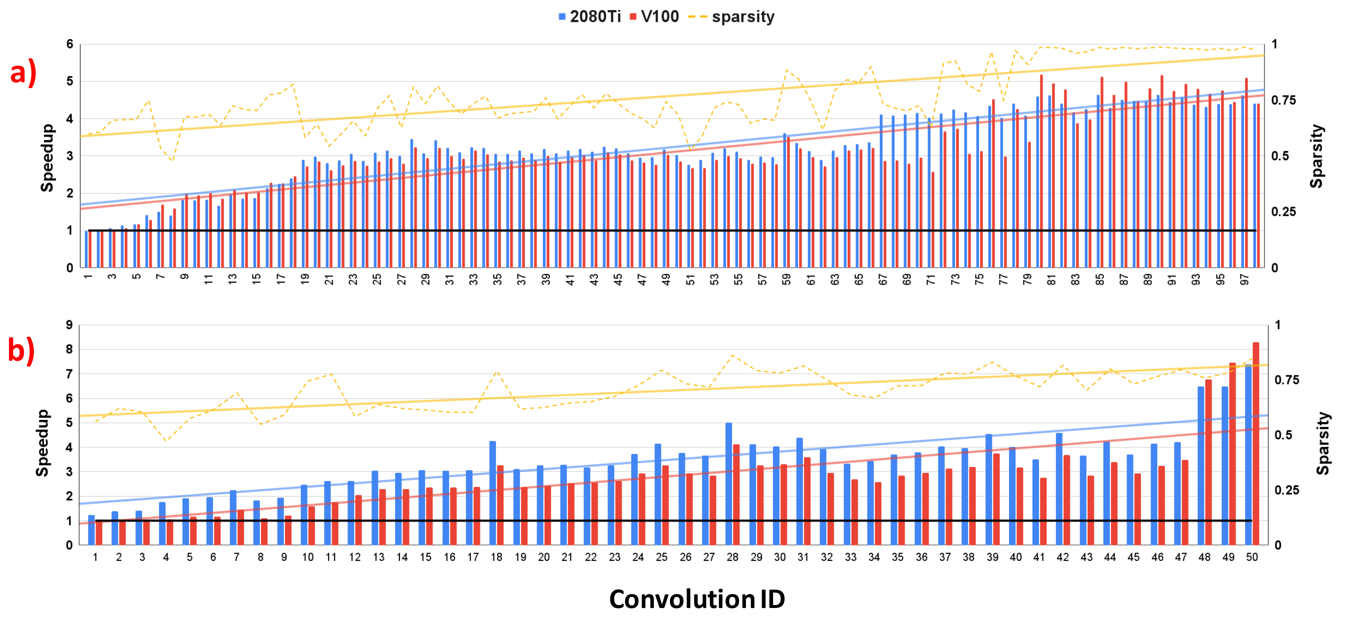

Figure 10: Runtime comparison (cuSNN vs cuDNN): DenseNet-201 (a) on 2080 Ti (average speedup is 3.23) and V100 (average speedup is 3.11); ResNet-152 (b) on 2080 Ti (average speedup is 3.5) and V100 (average speedup is 2.83)

configuration per image. Online exhaustive search is of course not practical for the real inference use case, but serves as a lower bound on performance if a perfect oracle could select the best code configuration for each image. Figure 9 shows that the loss of performance over the ideal choice by using our analytical modeling approach is low for most layers.

4.4 Performance Comparison with cuDNN

In this subsection, we compare cuSNN against cuDNN, the state-of-the-art convolution library from Nvidia. We use FP32 datatype for all experiments.

Evaluation Metric: We use the execution time of GPU kernel(s) as the evaluation metric. We also compare the energy consumption of different approaches. We measure the execution time using cuda events and energy using the NVML API [24]. cuDNN requires a warmup run, however, for fairness, we don't include the warmup time in our experiments. Any *overhead* such as dense-sparse conversion associated with our approach is also *included* in the measurement.

Layer wise evaluation: First, we demonstrate the effectiveness of our approach by comparing the run-time of individual layers. Figure 10 shows our layer-wise speedup over cuDNN on different networks on different machines with the layers' sparsity overlapped (also shown in Fig.2). The X-axis shows the layer id, the left Y-axis shows the speedup and the right Y-axis shows the sparsity. The dotted yellow line shows the sparsity across all convolution layers for DenseNet-201(a) and ResNet-152(b). The solid yellow line shows the sparsity trend for all convolution layers, the solid blue line shows the speedup trend on the 2080Ti machine, and the solid red lines show the speedup trend on the v100 machine. The solid black line corresponds to a speedup of one. The speedup has a similar trend as the sparsity, increasing towards the later layers. On the 2080Ti GPU, we achieve better speedup in the middle layer since cuDNN performance is low in these layers. The other networks follow a similar trend. Figure 11 shows the effect of different batch sizes (V100 results are omitted for brevity).

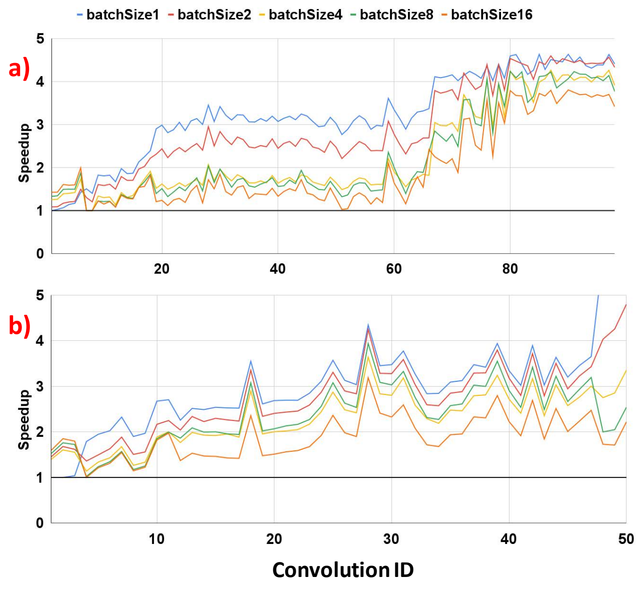

Figure 11: Speedup for different batch sizes for (a) Densenet 201 and (b) Resnet 152: 2080 Ti

Whole-Network Evaluation: In order to show that our sparse convolution is impactful, we present the full network evaluation of

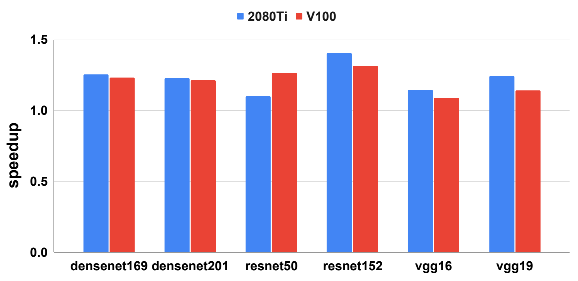

Figure 12: Full network evaluation on 2080 Ti and V100 GPUs using analytical model for tile size selection

different networks and show the overall performance improvement (includes all non-convolution as well as convolution layers). We constructed the six networks using our cuSNN kernels and compare them with cuDNN kernels. For all non convolution layers such as batchnorm, fully connected layer, etc. and for dense convolutions we used the cuDNN implementation. Figure 12 shows the corresponding results use analytical modeling tile-size selection. The X-axis shows the network name, and the y-axis shows the speedup. On the 2080Ti machine, cuSNN achieved 1.24X speedup on average for the two DenseNet architectures, 1.26X speedup on average for Resnet, and 1.2X speedup on average for VGG. On the V100 machine, the corresponding speedup are 1.23X,1.3X and 1.12X. For offline exhaustive search tile-size selection, the speedup on 2080Ti is (1.32X, 1.26X, 1.27X), and V100 is (1.18X, 1.32X, 1.18X), these are also on average speedup for each two DenseNet, Restnet and VGG. We note that our average speedup for whole-network evaluation is lower than that reported earlier for layer-wise evaluation. Since we only optimize the convolution layers and other layers such as BatchNorm, Pooling, and FullyConnected layers take a non-trivial amount of time, the overall speedup is reduced.

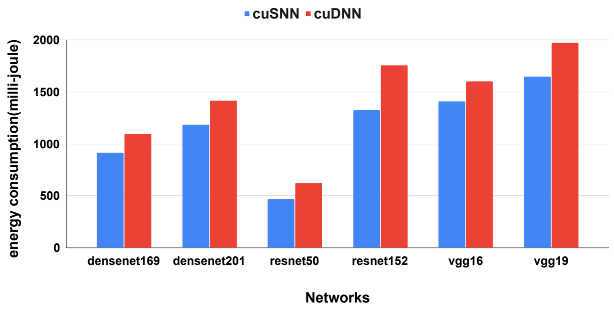

Figure 13: Energy consumption comparsion of cuSNN and cuDNN across the six networks on V100 machine

Energy evaluation: Figure 13 compares the total energy consumption characteristics of different implementations on the V100 machine. The X-axis shows the networks we evaluated. The Y-axis shows the energy consumed in millijoules by each framework(cuDNN vs. cuSNN). On average, cuSNN consumes 19% less energy when compared with cuDNN across the six networks.

5 CONCLUSION

In this paper, we designed and developed a high-performance GPU implementation for sparse feature map convolutions. Our design considers various performance bottlenecks in GPUs and successfully mitigates them. The result section shows that, on real benchmarks with real datasets, our approach achieves substantial speedup over the state-of-the-art implementations. We believe that our design's efficacy can be adapted to other machine learning and graph computation primitives.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback that helped us improve the paper. The support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged. We are also very thankful for the use of the RI2 compute cluster in the CSE Department at the Ohio State University. Work at the University of Utah was supported in part by the National Science Foundation through awards 2018016.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
[2] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205, 2019.
[3] K. Behrendt, L. Novak, and R. Botros. A deep learning approach to traffic lights: Detection, tracking, and classification. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1370–1377, 2017.
[4] Xuhao Chen. Escort: Efficient sparse convolutional neural networks on gpus. *CoRR*, abs/1802.10280, 2018.
[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.
[6] Benjamin Graham and Laurens van der Maaten. Submanifold sparse convolutional networks. *CoRR*, abs/1706.01307, 2017.
[7] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, 2013.
[8] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.
[9] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1135–1143. Curran Associates, Inc., 2015.
[10] P. Harár, R. Burget, and M. K. Dutta. Speech emotion recognition with deep learning. In *2017 4th International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 137–140, 2017.
[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
[12] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.
[14] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando A. Mujica, Adam Coates, and Andrew Y. Ng. An empirical evaluation of deep learning on highway driving. *CoRR*, abs/1504.01716, 2015.

[15] Yani Ioannou, Duncan P. Robertson, Roberto Cipolla, and Antonio Criminisi. Deep roots: Improving CNN efficiency with hierarchical filter groups. *CoRR*, abs/1605.06489, 2016.

[16] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *ArXiv*, abs/1405.3866, 2014.

[17] F. Kjolstad, S. Chou, D. Lugato, S. Kamil, and S. Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 943–948, 2017.

[18] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[19] Tailin Liang, Lei Wang, Shaobo Shi, and John Glossner. Dynamic runtime feature map pruning. *arXiv preprint arXiv:1812.09922*, 2018.

[20] Top 500 list. Top 500: HPCG - june 2022. https://www.top500.org/lists/hpcg/2022/06/, 2022.

[21] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[22] Xingyu Liu, Jeff Pool, Song Han, and William J. Dally. Efficient sparse-winograd convolutional neural networks. *CoRR*, abs/1802.06367, 2018.

[23] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[24] Nvidia. Nvidia management library. https://developer.nvidia.com/nvidia-management-library-nvml.

[25] D. W. Otter, J. R. Medina, and J. K. Kalita. A survey of the usages of deep learning for natural language processing. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2020.

[26] Jongsoo Park, Sheng R. Li, Wei Wen, Hai Li, Yiran Chen, and Pradeep Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *CoRR*, abs/1608.01409, 2016.

[27] Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. Sbnet: Sparse blocks network for fast inference. *CoRR*, abs/1801.02108, 2018.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] K. Sirinukunwattana, S. E. A. Raza, Y. Tsang, D. R. J. Snead, I. A. Cree, and N. M. Rajpoot. Locality sensitive deep learning for detection and classification of nuclei in routine colon cancer histology images. *IEEE Transactions on Medical Imaging*, 35(5):1196–1206, 2016.

[30] Wenqing Sun, Bin Zheng, and Wei Qian. Computer aided lung cancer diagnosis with deep learning algorithms. In Georgia D. Tourassi and Samuel G. Armato III, editors, *Medical Imaging 2016: Computer-Aided Diagnosis*, volume 9785, pages 241 – 248. International Society for Optics and Photonics, SPIE, 2016.

[31] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[32] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 2082–2090, Red Hook, NY, USA, 2016. Curran Associates Inc.

[33] Tao Xu, Han Zhang, Xiaolei Huang, Shaoting Zhang, and Dimitris N. Metaxas. Multimodal deep learning for cervical dysplasia diagnosis. In Sebastien Ourselin, Leo Joskowicz, Mert R. Sabuncu, Gozde Unal, and William Wells, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2016*, pages 115–123, Cham, 2016. Springer International Publishing.

[34] Weizhi Xu, Shengyu Fan, Hui Yu, and Xin Fu. Accelerating convolutional neural network by exploiting sparsity on gpus. *arXiv preprint arXiv:1909.09927*, 2019.

[35] T. Young, D. Hazarika, S. Poria, and E. Cambria. Recent trends in deep learning based natural language processing [review article]. *IEEE Computational Intelligence Magazine*, 13(3):55–75, 2018.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the scripts and data required to reproduce the experimental results in the PACT 2022 paper titled "High-performance Architecture Aware Sparse ConvolutionalNeural Networks for GPUs". The git repository contains:

- The cuSNN generated source code for sparse convolution layers;
- The cuSNN generated code for end2end network evaluation;
- The scripts to run layer wise performance comparison and end2end inference time comparison(cuSNN vs cuDNN) ;

A.2 Artifact check-list (meta-information)

- **Run-time environment:**
  CUDA: 10.2(v100), CUDA:11.0(2080Ti), cuDNN:7.6.5(v100), cuDNN: 8.0.2(2080Ti), Linux platform such as Ubuntu or CentOS.
- **Hardware:** Nvidia 2080Ti or Nvidia V100.
- **How much disk space required (approximately)?:** > 50 GB.
- **How much time is needed to complete experiments (approximately)?:** Should be less than 1 hour.
- **Publicly available?:** Yes

A.3 Installation and Run

Clone the repository (recursively):
https://github.com/aGoodCat/PACT-CUSNN.git
See README in our repo for instructions to build and run:

A.4 Evaluation and Expected Result

The run-time of individual layers(densenet201, resnet152) comparison(cuSNN vs cuDNN). Entire network evaluation for densenet(121, 169, 201), resnet(101, 152) using sparse convolution kernel code generated by cuSNN analytical modeling.