GRIM: A General, Real-Time Deep Learning Inference Framework for Mobile Devices based on Fine-Grained Structured Weight Sparsity

Wei Niu[†],Zhengang Li[†], Xiaolong Ma, Peiyan Dong, Gang Zhou, Xuehai Qian, Xue Lin, Yanzhi Wang, Bin Ren

Abstract—It is appealing but challenging to achieve real-time deep neural network (DNN) inference on mobile devices, because even the powerful modern mobile devices are considered as "resource-constrained" when executing large-scale DNNs. It necessitates the sparse model inference via weight pruning, i.e., DNN weight sparsity, and it is desirable to design a new DNN weight sparsity scheme that can facilitate real-time inference on mobile devices while preserving a high sparse model accuracy. This paper designs a novel mobile inference acceleration framework GRIM that is General to both convolutional neural networks (CNNs) and recurrent neural networks (RNNs) and that achieves Real-time execution and high accuracy, leveraging fine-grained structured sparse model Inference and compiler optimizations for Mobiles. We start by proposing a new fine-grained structured sparsity scheme through the Block-based Column-Row (BCR) pruning. Based on this new fine-grained structured sparsity, our GRIM framework consists of two parts: (a) the compiler optimization and code generation for real-time mobile inference; and (b) the BCR pruning optimizations for determining pruning hyperparameters and performing weight pruning. We compare GRIM with Alibaba MNN, TVM, TensorFlow-Lite, a sparse implementation based on CSR, PatDNN, and ESE (a representative FPGA inference acceleration framework for RNNs), and achieve up to $14.08\times$ speedup.

Index Terms—Compiler optimization, Model compression, Real-time Inference, Deep neural networks, Mobile computing.

1 Introduction

Deep learning, as one of the most powerful machine learning techniques, has achieved extraordinary performance in computer vision and surveillance, speech recognition and natural language processing, healthcare and disease diagnosis, etc. The two major categories of deep neural network (DNN) models are convolutional neural networks (CNNs) [1], [2] and recurrent neural networks (RNNs) [3], [4] with unique model structures and application domains.

Due to the high efficiency and reliability, low cost, small footprint, and reprogrammability, mobile devices have been pervasively used for wireless access points, wearable electronics, robotics, autonomous driving, etc. If equipped with deep learning, mobile devices will achieve comprehensive functionality and better performance, further enabling their broader applications [5], [6], [7], [8], [9], [10].

It is appealing to support on-device inference by mobile devices, however, achieving efficient inference with real-time performance is still a challenging task. Because even the powerful modern mobile devices with high-end CPUs and GPUs are considered as ″resource constrained″ when executing the computation-extensive and memory-hungry state-of-the-art DNN models. Take the image classification task on ImageNet dataset [11] with VGG-16 model [12] as an example. The VGG-16 model has a size of 528 MB and sufficient learning capacity and therefore is used as one of the major pre-trained models in transfer learning [13]. We executed the VGG-16 model on an embedded GPU (Adreno 640) with 16-bit floating-point for weights/intermediate results by using two representative mobile inference acceleration frameworks – TensorFlow-Lite (TFLite) [14] and TVM [15]. The end-to-end inference time is 307 ms and 221 ms per frame on TFLite and TVM, respectively. However, the real-time performance typically requires 30 frames/sec i.e., 33ms/frame.

Complementary to those mobile inference acceleration approaches, DNN model compression techniques provide another possibility to efficient on-device inference. Two main-stream model compression techniques are weight pruning [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36] and weight quantization [37], [38], [39], [40], [41]. Weight pruning enjoys the great flexibility of various *DNN weight sparsity schemes* and has achieved very high pruning rate and accuracy. On the other hand, weight quantization is less supported in mobile devices especially mobile GPUs. Therefore, this paper leverages weight pruning as the main model compression technique, while we use 16-bit floating-point representation throughout the paper.

In this paper, we implement efficient DNN inference on mobile devices aiming for both real-time performance and high accuracy. The difficulty of achieving real-time

- *W. Niu, G. Zhou and B. Ren are with the Department of Computer Science, William & Mary, Williamsburg, VA, 23185.*
- *Z. Li, X. Ma, P. Dong, X. Lin and Y. Wang are with the Department of Electrical & Computer Engineering, Northeastern University, Boston, MA, 02115.*
- *X. Qian is with the Department of Computer Science, University of Southern California, Los Angeles, CA, 90089.*
- [†] *These authors contributed equally.*

Manuscript received October, 2020; revised XXX, XXX.

DNN inference on mobile devices necessitates the *sparse model inference* via the weight pruning techniques, i.e., DNN weight sparsity. However, the majority of the DNN weight sparsity schemes i.e., the non-structured sparsity and the structured sparsity are (i) incompatible with the data parallel executions on the computing systems and (ii) suffering from significant accuracy loss, respectively.

Recently, the pattern-based weight pruning techniques [34], [35] provide a novel weight sparsity scheme, i.e., *the fine-grained structured sparsity*. It can be considered as enabling a certain level of flexibility in the previous (coarse-grained) structured sparsity, thus simultaneously boosting the accuracy of the structured sparsity and facilitating real-time on-device inference. Furthermore, the work of PatDNN [42] leverages the pattern-based weight pruning techniques [35] to implement fine-grained structured sparse DNN models and performs compiler optimizations to achieve real-time mobile inference. PatDNN is the state-of-the-art mobile inference acceleration framework.

In this paper, we design a novel mobile inference acceleration framework GRIM that is General to both convolutional neural networks and recurrent neural networks and that achieves Real-time performance and high accuracy leveraging fine-grained structured sparse model Inference and compiler optimizations for Mobiles. We start by proposing a new fine-grained structured sparsity scheme through the Block-based Column-Row (BCR) pruning techniques, which works for both CNNs and RNNs. Specifically, for any weight matrices in CNNs and RNNs, we first partition it into a number of *weight blocks* and then apply independent column pruning and row pruning to each block. Please note that the operations in CONV layers can be transferred into the general matrix multiplication (GEMM) routine [43] and therefore we can obtain the corresponding matrix format for filters in a CONV layer. Our BCR pruning can result in a new fine-grained structured weight sparsity that enjoys the high accuracy as the non-structured sparsity and the regularity as the course-grained structured sparsity to facilitate real-time mobile inference while overcoming their shortcomings.

Based on the new fine-grained structured sparsity scheme, our GRIM framework consists of two parts: (a) the compiler optimizations of execution code generation for real-time mobile inference; and (b) the BCR pruning optimizations for determining pruning hyperparameters and performing weight pruning. Particularly, compared with PatDNN, GRIM's new compiler optimizations depend on the newly proposed BCR pruning that is generally applicable to both CNNs and RNNs, thus requiring different designs and implementations. For example, PatDNN targets CONV computations mainly without efficient support to the fully connected (FC) layers, another kind of computation kernels in neural networks (both CNNs and RNNs). GRIM unifies both CONV and FC operations by converting CONV into GEMM (im2col), supporting CNN, RNN, and potentially the latest transformer-based models (e.g. MobileBERT and GPT-2). Correspondingly, GRIM requires a new weight compression format, a different computation reorder strategy, and a different set of tuning parameters. Moreover, GRIM includes a more flexible/declarative Domain Specific Language (DSL) to assist finding the best-suited block configuration offline, thus saving training time. In contrast, PatDNN needs to consume more epochs to find the patterns during training.

We summarize the contributions of this paper as:

- We propose a new fine-grained structured sparsity scheme through the BCR pruning technique that is general to both CNNs and RNNs, achieves high accuracy for the sparse DNN model inference and facilitates real-time execution.
- We present a set of new compiler techniques to generate optimized execution codes of the sparse DNN inference implemented by the proposed BCR pruning, including a layerwise Intermediate Representation (IR) with the associated Domain Specific Language (DSL), matrix reordering, a compact model storage format, register-level load redundancy elimination, and an auto-tuning module.
- We provide systematic optimizations of the BCR pruning. We use a decoupling strategy to reduce the pruning hyperparameter search space and our hyperparameter optimizations incorporate mobile testing with compiler optimizations. For performing the pruning, we formulate the BCR pruning problem and provide an ADMM-based solution.
- We design the whole GRIM framework for mobile devices to realize real-time, end-to-end inference performance supporting both CNNs and RNNs. GRIM is the first unified mobile inference acceleration framework for both CNNs and RNNs.

For CNNs, we compare GRIM with Alibaba Mobile Neural Network (MNN) [44], TVM [15], TensorFlow-Lite (FTLite) [14], a sparse DNN inference implementation based on CSR format [45], and PatDNN [42], across various datasets, neural network models, and CPU/GPU excutions, achieving speedups in the end-to-end inference time up to $6.84\times$, $7.09\times$, $14.08\times$, $5.81\times$, and $2.11\times$, respectively. Since our GRIM is the first mobile inference acceleration framework for RNNs, we compare with ESE [46], a representative FPGA inference acceleration framework for RNNs. We achieve comparable end-to-end inference time (around 81 us by GRIM and 82 us by ESE) with significantly higher energy efficiency ($38\times$) compared with ESE.

2 Background on Sparse DNNs

We use Figure 1 to illustrate existing DNN weight sparsity schemes. We use grey color to represent the pruned weights. We start by Figure 1 (a) showing the weight tensors in a convolutional (CONV) layer. For Figure 1 (b)$\sim$(d) the CONV weight tensors are transferred into the GEMM matrix format.

Figure 1 (b) is the non-structured sparsity by the irregular weight pruning techniques [20], [24], [25], which prune weights at arbitrary locations. The irregular pruning can achieve a high pruning rate, but the non-structured sparsity is not compatible with data-parallel executions on the computing systems.

Figure 1 (c) is a coarse-grained structured sparsity scheme by the filter pruning techniques [18], [19], [22] and Figure 1 (d) is a coarse-grained structured sparsity scheme by the column pruning techniques [26], [31]. Filter pruning by the name prunes whole filters from a layer. (Please note that some references mention channel pruning [19], which

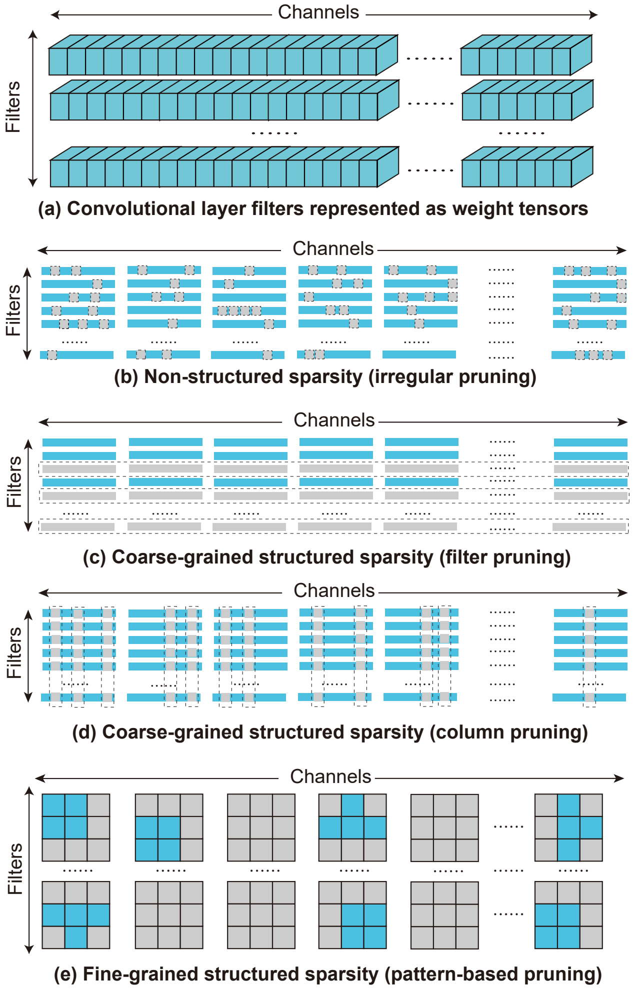

Channels
Filters
(a) Convolutional layer filters represented as weight tensors
Channels
Filters
(b) Non-structured sparsity (irregular pruning)
Channels
Filters
(c) Coarse-grained structured sparsity (filter pruning)
Channels
Filters
(d) Coarse-grained structured sparsity (column pruning)
Channels
Filters
(e) Fine-grained structured sparsity (pattern-based pruning)

Fig. 1: **Weight sparsity schemes by different pruning techniques.**

by the name prunes some channels completely from the filters. Essentially channel pruning is equivalent to filter pruning, because if some filters are pruned in a layer, it makes the corresponding channels of next layer invalid.) Column pruning prunes weights for all filters in a layer, at the same locations. The coarse-grained structured sparsity preserves regularity on the sparse models, but suffer from significant accuracy loss.

Figure 1 (e) shows a fine-grained structured sparsity scheme by the pattern-based pruning techniques [34], [35], [42], which are a combination of *kernel pattern pruning* and *connectivity pruning*. In kernel pattern pruning, for each kernel in a filter, a fixed number of weights are pruned, and the remaining weights form specific kernel patterns. The example in Figure 1 (e) is defined as 4-entry kernel pattern pruning, since every kernel reserves 4 non-zero weights out of the original 3×3 kernels. The connectivity pruning cuts the connections between some input and output channels, which is equivalent to removing corresponding kernels. Note that the pattern-based pruning is not based on the GEMM matrix format. More details about the differences between this fine-grained structured sparsity scheme by the pattern-based pruning and that by our BCR pruning are provided in Section 7.

3 The New Fine-Grained Structured Sparsity and GRIM Overview

3.1 Unified View of CNN/RNN Computation

The layerwise computations of CNN include CONV layer computations with different kernel sizes, mostly 3×3 and 1×1 kernels (larger kernels such as 5×5 ones may be utilized for input layer), and FC layer computations which are essentially matrix-vector multiplications. On the other hand, computations in RNNs (e.g., LSTM or GRU) are mostly FC layers (matrix-vector multiplications). It is well known that the CONV in DNNs is commonly transformed into GEMM, i.e., the multiplication of a weight matrix and an input matrix. GEMM is commonly utilized in DNN acceleration frameworks [14], [15]. In this way, all computation types in CNN and RNN can be unified as matrix-vector or matrix-matrix multiplication and will be treated in a unified manner through the fine-grained structured sparsity by BCR pruning.

3.2 Motivation of Fine-Grained BCR Pruning

From a survey of recent research works, we have reached the following conclusions: (i) non-structured sparsity has the advantage of high pruning rate but is typically not compatible with the data parallel executions on the computing systems; (ii) coarse-grained structured sparsity facilitates inference acceleration but is often subject to accuracy degradation. The accuracy degradation in coarse-grained structured sparsity is especially significant for RNNs. When a whole row or column in a weight matrix (input, state-transition, or output matrix) of RNN is pruned, it assumes that a whole input or output entry is not used at all time steps. This is easy to cause intolerable accuracy loss. As a result, it is desirable to design a fine-grained structured sparsity scheme possessing more flexibility (and thus higher accuracy) while still maintaining regularity (for facilitating inference acceleration).

We propose BCR pruning to achieve this goal, which applies to different computation layers in CNNs and RNNs. For a weight matrix in GEMM or FC layer computation, we divide it into $n \times m$ blocks with equal size. We apply independent row and column pruning on each block, with potentially different pruning rates (number of pruned rows/columns) in each block, to ensure high flexibility. The remaining weights in each block still form a full matrix. The illustrative example is shown in Figure 2. At first glance, BCR pruning is a tradeoff between the most flexible non-structured pruning and the most rigid structured pruning that prunes whole rows/columns. It becomes the former with block size 1-by-1 and becomes the latter with block size the same as the whole weight matrix. We will see in the following that BCR pruning is beyond a mere tradeoff, from both accuracy (pruning rate) and inference acceleration perspectives, especially with the aid of compiler optimizations.

From the accuracy perspective, we observe that BCR pruning obtains a significant accuracy enhancement (under the same pruning rate) compared with the most coarse-grained structured pruning that eliminates whole rows/-columns, even with a small number of blocks. This is validated in various datasets under the same (ADMM-based)

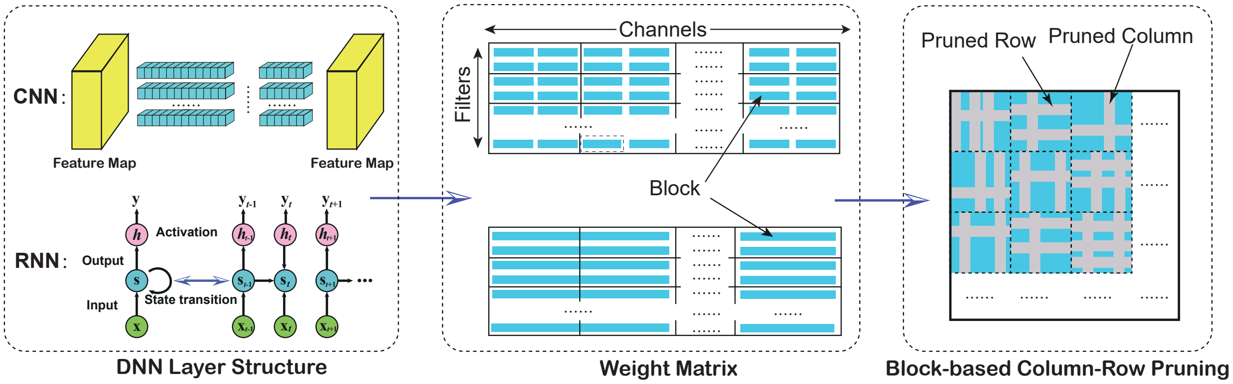


Fig. 2: **Illustration of the proposed BCR pruning.**

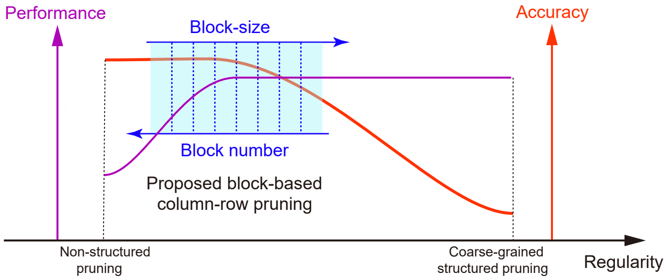


Fig. 3: **Relation of accuracy and performance with regularity.**

pruning algorithm. With a moderate 8 - 256 number of blocks in weight matrix, BCR pruning's accuracy can be similar or even surpass non-structured pruning under the same pruning rate. This is because non-structured pruning has a large search space, and it often takes too long time to converge to a desirable solution. This accuracy phenomenon is illustrated conceptually in Figure 3.

Here is an example to quantitatively compare the searching space of non-structure pruning and BCR pruning. Consider pruning a layer with 64 filters and 64 channels with a 2× pruning rate. The total number of possible combinations of non-structured pruning positions is C(4096, 2048), extremely large. In contrast, if using BCR pruning with a block size of 4×16 and only considering the column pruning, the total number of combinations is C(1024, 512). If using filter pruning, the number of combinations is C(64, 32). The above comparison shows that the total number of BCR pruning candidate combinations is within a proper range, neither too large like non-structured pruning, nor too small like coarse-grained structured pruning. Although Reinforcement Learning can somehow alleviate the impact of large search space for non-structured pruning, the sampling action in DRL is generated in a randomized manner, and it is challenging to make satisfied decisions for high pruning rates. Thus, we claim that controlling the search space for weight pruning is still necessary.

From the inference acceleration perspective, with a moderate 8 - 256 number of blocks in weight matrix, the inference acceleration performance on a mobile device can be close to the coarse-grained structured pruning, far better than the non-structured one. The most important reason is that the remaining parallelism in each block (after pruning) is still much higher than that in a mobile CPU/GPU. Taking a 1024×1024 weight matrix as an example. Suppose 64 blocks are utilized and a further 8× BCR pruning rate is adopted, the average number of remaining weights per block is 2,048. These 2,048 weights form a weight matrix that is still large enough for parallelization on mobile CPU/GPU. Moreover, the overhead in column/row index storage, input and output transition, etc. can be effectively reduced through code optimization capability of compiler, and load balancing can be maintained. As a result, with the help of compiler, the inference performance can be guaranteed under fine-grained BCR pruning.

In summary, the conceptual Figure 3 shows that BCR pruning is "beyond a mere tradeoff" of non-structured and the most coarse-grained structured pruning. Rather, it can achieve the best of both schemes, i.e., both high accuracy (pruning rate) and high inference acceleration performance, under a compiler-assisted acceleration framework.

3.3 Overview of the GRIM Framework

Figure 4 illustrates the overview of our end-to-end GRIM acceleration framework, which consists of two major parts: (1) an execution code generation stage with the compiler-based optimizations enabled by our BCR pruning (Section 4). This part assists inference acceleration with a given BCR pruned DNN (CNN or RNN) model and is performed offline; and (2) an optimization framework to determine the block size (for each layer) and other hyperparameters, and perform BCR pruning accordingly (Section 5). This part is performed during the training phase.

At the high-level, GRIM represents the DNN models as computational graphs with a set of associated optimizations like TVM [15]. Based on this optimized baseline and by leveraging our BCR pruning, this work focuses on proposing a layerwise Intermediate Representation (and a Domain Specific Language) for each DNN layer, and designing multiple optimization and code generation techniques. Our proposed optimizations include an efficient CONV to matrix multiplication transformation (i.e., `Im2col` for CNN only), matrix reordering, a compact model storage format, register-level load redundancy elimination, and an optimized auto-tuning. These optimizations are general, applicable for both CNNs and RNNs (and associated computation types),

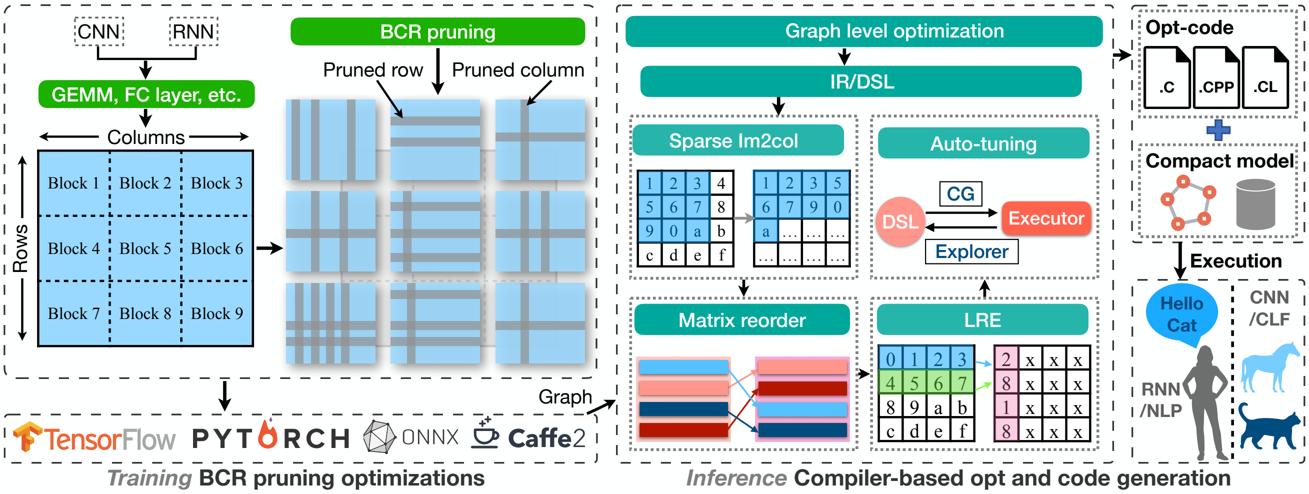


Fig. 4: **GRIM overview.**

working for both CPUs and GPUs on mobile devices. The optimized RNN and CNN models with BCR pruning can be used for various real-time workloads like natural language processing, computer vision, and video processing.

4 Compiler Optimizations

GRIM employs a compiler-based framework to generate optimized DNN inference code on mobile devices. At the high-level, this framework represents DNN models as computational graphs like TVM [15] with all optimizations summarized in Table 5. Based on this optimized baseline, this section focuses on the optimizations performed on each DNN layer and enabled by BCR pruning only.

It is worth noting that although GRIM shares similar optimization objectives with PatDNN (i.e., addressing performance challenges in pruned DNN executions: *thread divergence* and *load imbalance* among threads, *redundant memory access*, and *unnecessary zero storage*), its new optimization techniques depend on the new BCR pruning that is generally applicable to both CNNs and RNNs, thus requiring very different designs and implementations comparing to PatDNN. Moreover, GRIM introduces a new DSL to improve DNN programming productivity. Figure 5 shows an overview and a simplified code transformation and generation example of the GRIM compiler.

4.1 DSL and Compiler-based Framework

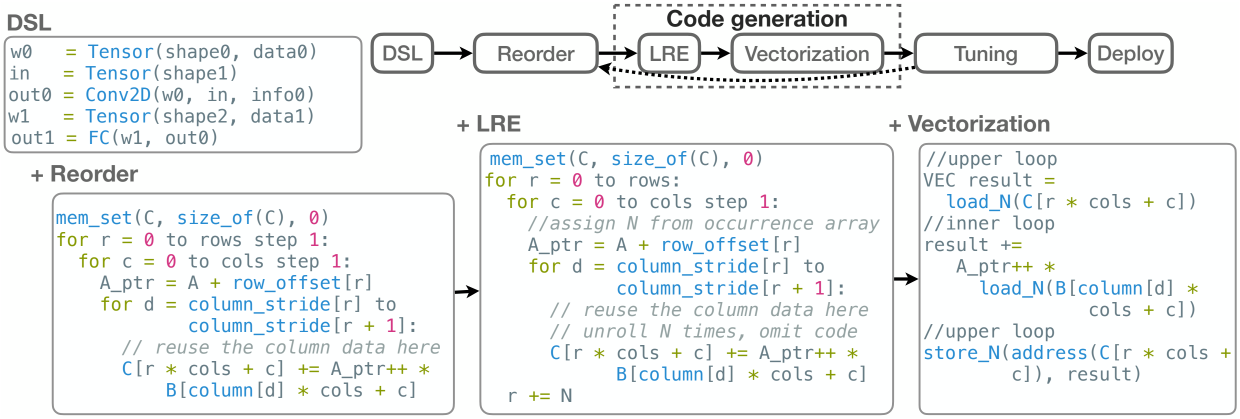
DNN models contain layers with varied computations, such as CONV, FC, pooling, etc. GRIM offers a high-level Domain Specific Language (DSL) to specify the functionality (e.g., CONV or FC), input (e.g., model, image, and intermediate results), output (e.g., intermediate and final results), and a layerwise Intermediate Representation (IR) with BCR pruning information. The input and output are in the form of tensors with different shapes. GRIM's DSL also provides a `Tensor` function for users to create matrices (or tensors).

Essentially, this DSL is equivalent to the computational graph (i.e., DSL is another high-level set of functions to model the data-flow of DNN models) and they can convert to each other conveniently. DSL offers users the flexibility of using existing DNNs or creating new DNNs, improving the programmability (or productivity) in DNN programming. If a DNN already exists, GRIM transforms it into an optimized computational graph and translates this graph to DSL. Otherwise, the user writes the model code in our DSL, translates it back to a computational graph, performs high-level optimizations, and re-generates the optimized DSL code.

Figure 5 shows a DSL example with two connected layers, `Conv2D` and `FC`. `Conv2D` takes a model tensor (`w0`) with the shape of `shape0` and data of `data0` and an input feature map (`in`) with the shape of `shape1`, and generates a result tensor (`out0`). Next, `FC` takes a model tensor (`w1`) with the shape of `shape2` and data of `data1` and previous `Conv2D` output, and generates a new result tensor (`out1`).

GRIM compiler translates DSL to low-level C++ (on CPU) and OpenCL code (on GPU) and optimizes the low-level code with a set of BCR pruning enabled optimizations, such as matrix reorder, compact data storage, load redundancy elimination, configuration parameters auto-tuning, and vectorization (as Figure 5). The generated code is deployed on mobile devices.

Layerwise IR: The key design of our DSL is *prune-aware*. It allows integrating BCR pruning information to the kernel computation by a layerwise IR (e.g., `info` in the DSL example in Figure 5). This IR provides the compiler necessary information to perform the subsequent BCR pruning-based code optimization. Figure 6 shows more details of this IR. It is an `FC` layer (`full_cnt1`) from `vgg16`, and this IR is for `CPU` optimization. It mainly consists of three aspects of information: block information (e.g., `block_size` and `layout`), tuning information (e.g., `unroll` factor, and `tiling` size), and other basic information (e.g., `strides`). This design is general, potential to support more advanced pruning and to represent other sparsity information for further performance optimization.



DSL

```
w0   = Tensor(shape0, data0)
in   = Tensor(shape1)
out0 = Conv2D(w0, in, info0)
w1   = Tensor(shape2, data1)
out1 = FC(w1, out0)
```

+ Reorder

```
mem_set(C, size_of(C), 0)
for r = 0 to rows step 1:
  for c = 0 to cols step 1:
    A_ptr = A + row_offset[r]
    for d = column_stride[r] to
            column_stride[r + 1]:
      // reuse the column data here
      C[r ∗ cols + c] += A_ptr++ ∗
          B[column[d] ∗ cols + c]
```

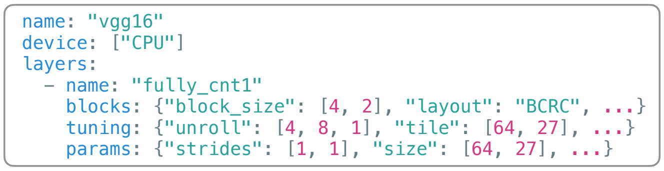
+ LRE

```
mem_set(C, size_of(C), 0)
for r = 0 to rows:
  for c = 0 to cols step 1:
    //assign N from occurrence array
    A_ptr = A + row_offset[r]
    for d = column_stride[r] to
            column_stride[r + 1]:
      // reuse the column data here
      // unroll N times, omit code
      C[r ∗ cols + c] += A_ptr++ ∗
            B[column[d] ∗ cols + c]
  r += N
```

+ Vectorization

```
//upper loop
VEC result =
  load_N(C[r ∗ cols + c])
//inner loop
result +=
   A_ptr++ ∗
     load_N(B[column[d] ∗
               cols + c])
//upper loop
store_N(address(C[r ∗ cols +
        c]), result)
```

Fig. 5: **GRIM's compiler-based optimization and code generation flow:** compiler takes both `DSL` and layerwise IR (as an example in Figure 6) to generate low-level C/C++ and OpenCL. This low-level code is further optimized with matrix reordering and our BCRC compact model storage (`+Reorder`), the register-level load redundancy elimination (`+LRE`), and other optimizations like vectorization (`+Vectorization`). Finally, the code is further tuned by the auto-tuning module and deployed on mobile devices.



```
name: "vgg16"
device: ["CPU"]
layers:
  - name: "fully_cnt1"
    blocks: {"block_size": [4, 2], "layout": "BCRC", ...}
    tuning: {"unroll": [4, 8, 1], "tile": [64, 27], ...}
    params: {"strides": [1, 1], "size": [64, 27], ...}
```

Fig. 6: **A layerwise IR example.**

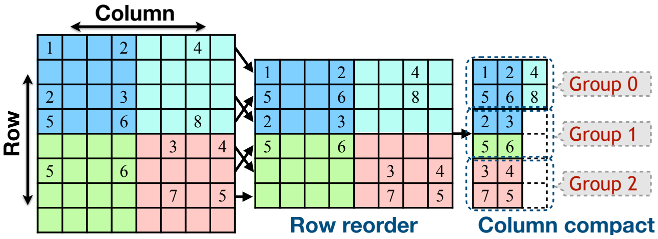


Fig. 7: **Matrix reordering.**

4.2 Matrix Reordering

BCR pruning partitions the weight matrix of a whole layer into blocks with different pruning configurations. Without any further optimization, it will encounter the well-known challenges for sparse matrix multiplications, i.e., heavy control-flows within each thread, load imbalance among multiple threads, and irregular memory access. Although there are many existing efforts on sparse matrix multiplications [45], [47], they cannot leverage the optimization opportunities offered by BCR pruning.

To address this issue, we propose a matrix reorder method based on BCR pruning. Our later evaluation demonstrates that this kind of compression and acceleration *co-design* significantly outperforms existing general sparse matrix multiplication optimizations that do not take the pruning characteristic into account.

Figure 7 illustrates the basic idea of matrix reorder. Because BCR pruning removes certain whole columns and rows of weights within a block, the remaining weights only appear in other rows and columns with a certain degree of regularity. Based on this insight, matrix reorder first reorders the rows (e.g., filters in CNN) by arranging the ones with the same or similar patterns together. Next, it compacts the weights in the column direction (e.g., kernels in CNN). At last, the rows with the same or similar computations are grouped together.

Figure 7 shows a simplified example with only three groups and two rows in each group. Real CNN and RNN models usually have tens of groups with hundreds of rows in each group. Each group is processed by all threads in parallel, and each thread is in charge of multiple continuous rows. Thus, the computation divergence among these threads is significantly reduced.

4.3 Compact Model Storage (BCRC)

After the matrix reorder, GRIM stores the model in a compact format by leveraging the BCR pruning, called BCRC (i.e. Blocked Column-Row Compact) format. BCRC aims to avoid zero-weights storage as CSR with an even better compression ratio by adopting a hierarchical index structure to remove redundant column indices generated by BCR pruning. BCRC helps to save the scarce memory-bandwidth of mobile devices.

Figure 8 shows a simplified example of BCRC. The original matrix with BCR pruning (left-hand side) is transformed into a compact matrix by reordering (middle) and then stored in BCRC (right-hand side). BCRC consists of six arrays: `reorder`, `row offset`, `occurrence`, `column stride`, `compact column` and `weights`:

- `Reorder array` denotes a mapping between the row id in the original matrix and the one in the reordered matrix. For example, the number 0 and 3 (in reorder array[0] and [1]) denote that the row_0 and row_3 in the original matrix are placed in the 0 and 1 rows, respectively, after the reorder.
- `Row offset array` denotes the offset of each row when the reordered matrix is linearized into a 1-d array (i.e. `weights array`). For example, the 0 and 3 (in row offset array[0] and [1]) mean that the row_0 and row_1 in the

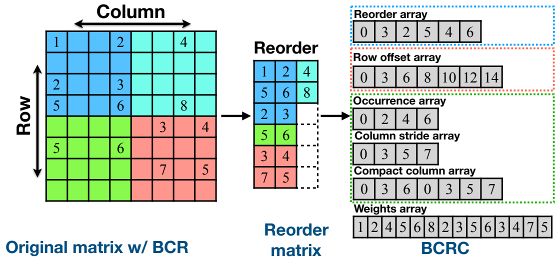


Fig. 8: **BCRC compact storage.**

reordered matrix start from index 0 and 3, respectively, in the 1-d `weights array`.

- The key advantage of BCRC over CSR is to use a more compact way to store the column index based on the observation that *multiple rows may share the same column index due to the BCR pruning*. It uses three arrays to achieve this: `occurrence`, `column stride` and `compact column`. Here is the basic idea. `Compact column array` stores the column index of each row in the reordered matrix. The `column stride array` denotes the offset of the column index in each row. For example, the 0 and 3 (in column stride array[0] and [1]) mean that the first row in reordered matrix has the column index [0, 3, 6] (i.e. from compact column array [0] to $[2(i.e., 3-1)]$). If two rows share the same column index, `compact column array` only stores once. The `occurrence array` is used to specify which rows have the same column index. For example, the first two numbers [0, 2] (in occurrence array [0] and [1]) show row_0 and row_1 have the same column index [0, 3, 6].
- `Weights array` is to store the matrix weights in a linearized 1-d array.

The low-level code starts to support computations on BCRC from `+Reorder` in Figure 5.

4.4 Register Load Redundancy Elimination

Poor memory performance caused by the irregular and redundant memory access is another key bottleneck of efficient DNN execution. GRIM employs two further optimizations to address this challenge: (1) matrix tiling (with the best tiling size decided by auto-tuning) to improve the load/store efficiency from memory to register, and (2) register load redundancy elimination (LRE) to reduce the number of register loads. This section focuses on the latter because of its novelty.

Figure 9 shows a register-level RLE example, in which both [1,4] and [5,8] (i.e. the first two rows) in the kernel matrix require the first and the last rows of the input feature map. Thus, the first and last rows of the input feature map could be loaded into the register once and reused by the first two rows of the kernel matrix. GRIM achieves this by a proper loop unrolling transformation (as shown in Figure 5, `+LRE`), because this LRE opportunity is decided by the kernel matrix that is already known during the compilation time.

It is worth noting that although it is easy to implement this LRE for dense models, it is challenging (even not possible) for randomly pruned models. Our BCR pruning re-enables LRE, showing the benefit of a model compression and compiler optimization co-design.

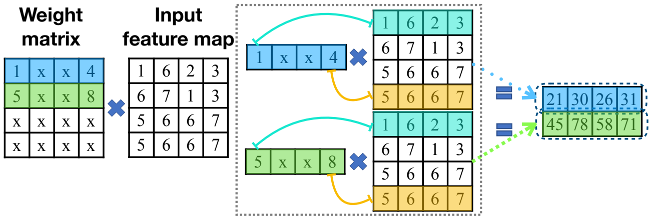


Fig. 9: **Register level LRE.**

4.5 Auto-Tuning and Other Optimizations

GRIM also includes some other optimizations that improve execution performance obviously:

Auto-tuning. DNN execution usually involves many configurable performance parameters, such as the data placement on GPU heterogeneous memory, matrix tiling sizes, loop unrolling factors, etc. Tuning them manually is tedious and error-prone. GRIM thus includes an auto-tuning module based on Genetic Algorithm to explore them automatically. In particular, after BCR pruning, different model kernels have varied sizes and shapes that require different tiling shapes and thread block settings. GRIM employs this auto-tuning module to extensively explore the best configurations for all DNN kernels. Comparing to existing auto-tuning approaches in TVM, GRIM's auto-tuning exploits better parallelism because its foundation, Genetic Algorithm allows starting parameter search with initializing an arbitrary number of chromosomes. GRIM's auto-tuning is more efficient.

Vectorization. GRIM also vectorizes CPU and GPU code automatically with ARM NEON and OpenCL, respectively. CPU and GPU have different (and limited) numbers of vector registers. To fully utilize them while minimizing the register spilling, GRIM carefully designs another level of loop unrolling to pack more computations together. Combining this optimization with the regularity given by BCR pruning and matrix reorder, GRIM generates more efficient vector codes comparing to other DNN acceleration frameworks.

Computation Transformation. GRIM transforms CONV to sparse matrix multiplication, which requires to convert CONV weights to a GEMM-based matrix format (i.e, the step of `Im2col` in Figure 4). `Im2col` is memory-bound as it only reads weights and expands them to a larger matrix. GRIM optimizes `Im2col` by skipping the matrix row during expanding, when a certain weight column is completely pruned.

5 BCR Pruning Optimizations

In this section, we present the BCR pruning techniques to cooperate with the compiler optimizations. Besides performing BCR pruning itself, we need to optimize the block size (for each layer) and also other hyperparameters (such as the pruning rate for each layer). The search space of all the hyperparameters is huge, therefore we propose a *decoupling strategy* of the hyperparameter space to reduce the problem complexity. It is based on the following two observations. First, generally, the sparse DNN accuracy is higher when the block size is smaller. Second, the mobile inference acceleration relates to the block size (and thus the number of blocks) and is independent of actual weight values. Therefore, we decouple block size optimization from other hyperparameter optimizations. More specifically, we perform mobile testing with the compiler optimizations to evaluate inference acceleration performance at different block sizes and select the smallest block size such that the inference acceleration performance degradation (compared with pruning whole rows/columns under the same pruning rate) is within a predefined threshold. This step is *independent* of DNN training or actual BCR pruning and should run much faster. The underlying principle is that the derived block size will likely provide the highest accuracy while satisfying the inference acceleration performance requirement. More elaborations about the decoupled optimizations are provided in the following.

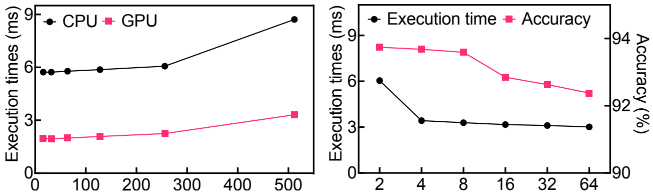
5.1 Block Size Optimization

Block size affects accuracy because a small block size results in a larger search space that mitigates the accuracy loss without changing pruning rate. The granularity of pruning increases with the block size growing. For example, under the same pruning rate, non-structured pruning can achieve higher accuracy than coarse-grained structure pruning. As the block size increases, the BCR pruning with increasing regularity becomes increasingly similar to the coarse-grained structure pruning, while as the block size decreases, the BCR pruning with less regularity becomes increasingly similar to a non-structured pruning.

```
1  def synthesize(origin_layer, rate, size):
2    # Copy the layer information, e.g. Weight shape
3    layer = copy_layer(origin_layer)
4    # Randomly generate weights
5    generate_random_weight(layer, rate, size)
6    return layer
7
8  # <Algorithm Entry>: find optimal block size with a
       pruning_rate
9  def find_opt_blk(layer, rate, block_sizes, device):
10   opt_size = -1
11   opt_latency = INFINITE_MAX
12   # Traverse different block size
13   for size in block_sizes:
14     # Synthesize a new layer
15     syn_layer = synthesize(layer, rate, size)
16     # Run synthesized layer on mobile device and get
        latency
17     latency = run_layer(syn_layer, device)
18     # if improvement of latency is less than a
       threshold, then stop
19     if opt_latency / latency < threshold:
20       break
21     # Record a more optimal block size
22     opt_latency = latency
23     opt_size = size
24   # Return the records
25   return opt_size
```

Listing 1: Block size optimization

The block size optimization is based on offline mobile testing with the compiler optimizations. Its goal is to select the smallest block size for each layer such that the inference acceleration performance degradation is within a tolerable range. Different layers may have different properties, e.g. the size of feature maps and convolution kernels. The runtime performance of different layers may vary as different properties result in different cache performance and computation patterns. Therefore, GRIM requires to study the relationship between layer size (and layer structure) and desirable block size and identifies the preferred block size for each layer. We evaluate mobile CPU/GPU in a layerwise manner, *using synthesized BCR pruning strategies with a pruning rate for each target layer*, and then select the desirable block size.



(a) 1024 × 1024 matrix. (b) Whole VGG-16.

Fig. 10: **(a) The CPU and GPU execution time (y-axis) for a single weight matrix as changing the number of blocks (x-axis); (b) The CPU execution time (left y-axis) and accuracy (right y-axis) for VGG-16 on CIFAR-10 as changing the block size.**

List 1 shows our detailed block size optimization algorithm. The objective of this algorithm is to find the optimal block size for each target layer under certain pruning rates. It consists of three steps. The first step accepts a layer, a pruning rate, a candidate block size set, and a device (mobile CPU or GPU) as input for the optimal block size testing (Listing 1 line 9). The block size set consists of a group of width and height pairs that can divide the layer's widths and heights. The second step generates a synthesized layer with features (e.g. CONV kernel size, weight size, and stride size, except the real weight numbers) identical to the original layer. This step also generates random weights that satisfy the pruning rate and block size requirements (Listing 1 lines 1 to 6) to improve the algorithm efficiency by avoiding training weights. The key insight is that the pruning ratio rather than the specific location of non-zero weights impacts more on the latency of a certain CONV layer with our BCR pruning that already guarantees certain regularity. The third step runs this synthesized layer on a mobile device to test its latency. If the latency improvement of this synthesized layer exceeds a threshold, our algorithm sets this block size as local optimal; otherwise it terminates with returning the last local optimal block size (Listing 1 lines 19 to 25). This algorithm executes on a host (PC) machine except the third step (*run_layer*) that runs on mobile devices to test latency. The whole process performs offline efficiently, independent of training/pruning. For example, for VGG-16 (on ImageNet dataset), this process takes less than 1 hours.

The left part of Figure 10 shows an illustrative example using a 1024 × 1024 weight matrix, under 10× BCR pruning rate. As the block number increases, the execution time remains stable before it reaches 256, and increases dramatically after that. The right part of Figure 10 shows the execution time and accuracy trend with changing block size for VGG-16 trained on CIFAR-10. The x-axis shows the first dimension of the block size, and the second dimension is fixed as 16. With increasing block size, the execution time drops quickly at first until reaching a relatively stable level (around 3 ms), and the inference accuracy drops slowly at first and quickly after a point. Therefore, it is possible for us to find a specific block size (e.g., 4×16) that yields (near-)optimal execution time without compromising accuracy.

5.2 BCR Pruning using ADMM

Based on the derived block size (number) for each layer, we will perform BCR pruning along with the optimizations of the remaining key hyperparameters: target pruning rate for each layer. We adopt state-of-the-art weight pruning algorithm using ADMM (Alternating Direction Methods of Multipliers) and generalize to BCR pruning, for two reasons: (i) It achieves (one of) the highest weight pruning rates satisfying accuracy constraint [25], [26], [32]. (ii) The ADMM-based solution framework, when generalized to BCR pruning, can automatically determine the desirable column and row pruning rates for each block given a predefined pruning rate for a whole weight matrix (for a specific layer).

BCR Pruning Problem Formulation and ADMM-based Solution: For an N-layer DNN of interest, let $\mathbf{W}_i$ and $\mathbf{b}_i$ denote the weights and biases of the i-th layer respectively. We minimize the loss function associated with the DNN model, subject to the specific fine-grained structured sparsity constraints (columns and rows in a block are pruned) on the weights in the corresponding layers, i.e.,

$$\begin{aligned} &\underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} && f\big(\{\mathbf{W}_i\}_{i=1}^N, \{\mathbf{b}_i\}_{i=1}^N\big), \\ &\text{subject to} && \mathbf{W}_i \in \mathbf{S}_i,\ i = 1, \ldots, N \end{aligned} \tag{1}$$

where $\mathbf{S}_i$ is the set of $\mathbf{W}_i$ with the sparsity constraint α_i.

Fine-grained structured sparsity by BCR: Consider the weight matrix of the i-th DNN layer divided into $n \times m$ blocks. The constraint on the weight matrix is that, the ratio of the total number of zero weights in all blocks to the total number of weights is no less than α_i (the sparsity constraint). And the zero weights form whole columns and rows.

Corresponding to every set $\mathbf{S}_i$, $i = 1, \ldots, N$, we define the indicator function $g_i(\mathbf{W}_i) = \begin{cases} 0 & \text{if } \mathbf{W}_i \in \mathbf{S}_i, \\ +\infty & \text{otherwise.} \end{cases}$. Problem (1) with constraint cannot be solved directly by classic stochastic gradient descent (SGD) methods [48] as original DNN training. However, the ADMM regularization can reforge and separate the problem, then solve them iteratively [49], [50]. First, we reformulate the problem (1) as:

$$\begin{aligned} &\underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} && f\big(\{\mathbf{W}_i\}_{i=1}^N, \{\mathbf{b}_i\}_{i=1}^N\big) + \sum_{i=1}^N g_i(\mathbf{Z}_i), \\ &\text{subject to} && \mathbf{W}_i = \mathbf{Z}_i,\ i = 1, \ldots, N, \end{aligned} \tag{2}$$

where $\mathbf{Z}_i$ is an auxiliary variable. Then, with formation of augmented Lagrangian [51], the problem (2) can be decomposed into two subproblems (3) and (4),

$$\underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} \quad f\big(\{\mathbf{W}_i\}_{i=1}^N, \{\mathbf{b}_i\}_{i=1}^N\big) + \sum_{i=1}^N \frac{\rho_i}{2} \|\mathbf{W}_i - \mathbf{Z}_i^t + \mathbf{U}_i^t\|_F^2, \tag{3}$$

$$\underset{\{\mathbf{Z}_i\}}{\text{minimize}} \quad \sum_{i=1}^N g_i(\mathbf{Z}_i) + \sum_{i=1}^N \frac{\rho_i}{2} \|\mathbf{W}_i^{t+1} - \mathbf{Z}_i + \mathbf{U}_i^t\|_F^2, \tag{4}$$

where $\mathbf{U}_i$ denotes dual variable and t is the iteration index, and we update $\mathbf{U}_i$ in each iteration by $\mathbf{U}_i^t := \mathbf{U}_i^{t-1} + \mathbf{W}_i^t - \mathbf{Z}_i^t$. These two will be iteratively solved until convergence.

The first subproblem can be solved by classic SGD. For the second subproblem, the solution is given by

$$\mathbf{Z}_i^{t+1} = \prod_{\mathbf{S}_i} (\mathbf{W}_i^{t+1} + \mathbf{U}_i^t), \tag{5}$$

where $\prod_{\mathbf{S}_i}(\star)$ is the Euclidean projection to $\mathbf{S}_i$, thereby guarantees weight matrices are subjected to *the fine-grained structured sparsity by BCR*.

Whole layer pruning rates are the hyperparameters in the ADMM-based solution framework. We use a straightforward, uniform target pruning rate for all layers in the DNN. This is shown as a valid hyperparameter setting for overall acceleration. More sophisticated hyperparameter determination procedure is possible and is orthogonal to this work.

6 Evaluation

This section evaluates GRIM by comparing it with TVM [15], TFLITE [14], MNN [44], an optimized sparse matrix implementation (CSR) based on CSR [45], and PatDNN [42].

6.1 Methodology

Evaluation Objective. Our evaluation has four objectives: (1) proving BCR pruning results in both high pruning rates and accuracy by comparing it with several state-of-the-art model compression efforts; (2) demonstrating GRIM runs faster than state-of-the-art end-to-end DNN execution frameworks, achieving real-time execution of mainstream DNNs on mobile devices without accuracy compromise; (3) studying the performance impact of GRIM's major compiler optimizations and the underlying reasons for the performance gains; (4) validating GRIM's good portability by comparing it with other frameworks on two other mobile devices.

Models and Datasets. GRIM is evaluated on three mainstream CNNs, VGG-16 (VGG), ResNet-18 (RNT) and Mobile-Net-V2 (MBNT). They are trained and tested on two datasets, CIFAR-10 and ImageNet. Here, we use 4×16 as the block size. When pruning and retraining models, the initial learning rate is 1e$-$2 for CIFAR-10, and it is reduced to 1e$-$3 for ImageNet. The learning rate is fixed for pruning, while adjusted in retraining with a scheduler following the cosine function. Besides, the numbers of epochs for pruning of all models on CIFAR-10 and ImageNet are fixed to 400 and 100, respectively, and for retraining are 300 and 100, respectively. For all models' pruning, the penalty factor (ρ) increases exponentially from 1e$-$4 to 1e$-$1. All training are conducted on NVIDIA Titan RTX GPUs with Ubuntu operating system and the PyTorch 1.3 framework with CUDA 10.1. For training cost, take VGG-16 as an example. On CIFAR-10 dataset, pruning (with ADMM optimization) and retraining (with normal DNN training setting) consume 22 seconds/epoch and 20 seconds/epoch on average with one Titan RTX GPU, respectively. On ImageNet dataset, pruning and retraining consume 42 minutes/epoch and 38 minutes/epoch on average with four Titan RTX GPUs, respectively. GRIM is also evaluated on a popular GRU RNN model that is widely used in previous studies [46], [52], [53]. GRU contains 2 GRU layers and about 9.6M parameters. GRU is trained and tested on the TIMIT dataset [54] that is commonly used for evaluating automatic speech recognition systems.

Testbed and Evaluation Setup. Our evaluations are conducted on a cell phone, Samsung Galaxy S10 with the latest Qualcomm Snapdragon 855 that consists of a Qualcomm Kryo 485 Octa-core CPU and a Qualcomm Adreno 640 GPU. The portability is tested on a Xiaomi POCOPHONE F1 phone with a Qualcomm Snapdragon 845 that consists of a Kryo 385 Octa-core CPU and an Adreno 630 GPU, and an Honor Magic 2 phone with a Kirin 980 that consists of an ARM Octa-core CPU and a Mali-G76 GPU. All experiments run 50 times on varied inputs with 8 threads on CPU, and all pipelines on GPU. Multiple runs do not vary severely, so we only report the average execution time for readability. We tune all runs to their best configurations, e.g., we apply Winograd optimization [55] for all dense runs, and use 16-bit float point for all GPU runs.

6.2 Accuracy Report

CIFAR-10. Table 1 shows the BCR pruning results for CIFAR-10 dataset. We use pre-trained VGG-16, ResNet-18 and MobileNet-V2 networks as our starting points to perform BCR pruning separately. For VGG-16, the original accuracy of the pre-trained model is 93.5%. Compared to the original model, BCR pruning achieves up to 50.5× pruning rate without any accuracy degradation. We further extend the pruning rate to 71.3× and get only 0.4% accuracy loss. For ResNet-18, when the pruning rate is 24.4×, BCR pruning achieves lossless 94.1% accuracy. When the pruning rate extends to 27.0×, the accuracy degradation is still negligible. For MobileNet-V2, BCR pruning achieves 9× pruning rate with minor accuracy loss compared to the original model (94.5%). Considering MobileNet-V2 is already a compact network, this weight pruning result is still prominent. To conduct an apple-to-apple comparison among DCP [28], PatDNN [42], and GRIM. We also evaluate the accuracy of ResNet-18 under a pruning rate of 27.0×. DCP, PatDNN, and GRIM yield an accuracy of 68.2%, 92.1%, and 93.9%, respectively.

ImageNet. Table 2 shows the BCR pruning results of VGG-16, ResNet-18 and MobileNet-V2 on ImageNet dataset. For VGG, the original model has top-5 accuracy as 91.7%, and BCR pruning achieves 8× pruning rate with no accuracy loss for top-5. When the pruning rate reaches 12×, the top-5 accuracy is 90.8%. For ResNet-18, the accuracy degradation is negligible when the pruning rate is 4×. For MobileNet-V2, BCR pruning achieves 2× pruning rate with 0.7% top-5 accuracy degradation.

TABLE 1: **BCR pruning with optimized block size vs. other pruning methods on CIFAR-10.**

Methods	Dense Accuracy	Sparse Accuracy	Conv Pruning Rate	Pruning Method
VGG-16				
Iterative Pruning [16] [24]	92.5%	92.2%	2.0×	Irregular
One Shot Pruning [24]	92.5%	92.4%	2.5×	Irregular
2PFPCE [29]	92.9%	92.8%	4.0×	Filter
Efficient ConvNet [21]	93.2%	93.4%	2.7×	Filter
2:4 Pattern [56]	93.5%	93.8%	35.7×	Irregular
2:4 Pattern [56]	93.5%	93.3%	71.3×	Irregular
PatDNN [42]	93.5%	93.7%	19.8×	Pattern-based
GRIM	**93.5%**	**93.8%**	**35.7**×	**BCR**
GRIM	**93.5%**	**93.6%**	**50.5**×	**BCR**
GRIM	**93.5%**	**93.1%**	**71.3**×	**BCR**
ResNet-18				
DCP [28]	88.9%	87.6%	2.0×	Filter
AMC [23]	90.5%	90.2%	2.0×	Filter
Variational Pruning [33]	92.0%	91.7%	1.6×	Filter
PatDNN [42]	94.1%	94.2%	16.0×	Pattern-based
GRIM	**94.1%**	**94.4%**	**22.9**×	**BCR**
GRIM	**94.1%**	**94.1%**	**24.4**×	**BCR**
GRIM	**94.1%**	**93.9%**	**27.0**×	**BCR**
MobileNet-V2				
DCP [28]	94.5%	94.7%	1.4×	Filter
GRIM	**94.5%**	**94.7%**	**6.0**×	**BCR**
GRIM	**94.5%**	**94.5%**	**7.2**×	**BCR**
GRIM	**94.5%**	**94.4%**	**9.0**×	**BCR**
GRIM	**94.5%**	**93.3%**	**11.9**×	**BCR**

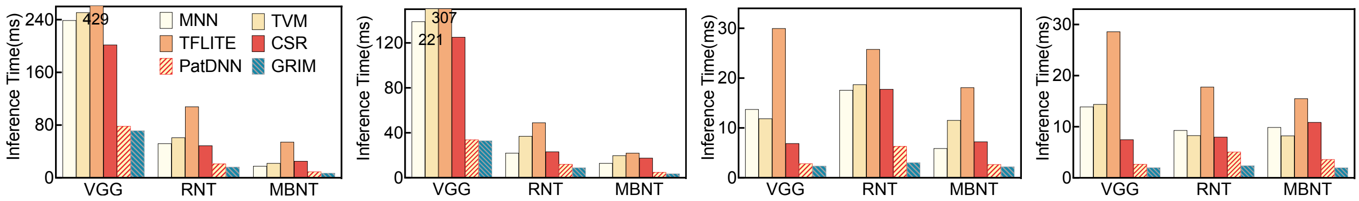
TABLE 2: **BCR pruning with optimized block size vs. other pruning methods on ImageNet.**

Methods	Dense Top 1/5 Accuracy	Sparse Top 1/5 Accuracy	Conv Pruning Rate	Pruning Method
VGG-16				
Decorrelation [27]	73.1%/*N/A*	73.2%/*N/A*	3.9×	Filter
APoZ [17]	*N/A*/88.4%	66.2/87.6%	2.0×	Filter
2:4 Pattern [56]	74.5%/91.7%	73.4%/91.0%	12.0×	Irregular
PatDNN [42]	*N/A*/91.7%	*N/A*/91.6%	8.0×	Pattern-based
GRIM	**74.5%/91.7%**	**74.4%/91.7%**	**3.0**×	**BCR**
GRIM	**74.5%/91.7%**	**74.1%/91.7%**	**8.0**×	**BCR**
GRIM	**74.5%/91.7%**	**73.1%/90.8%**	**12.0**×	**BCR**
ResNet-18				
Network Slimming [22]	68.9/88.7%	67.2/87.4%	1.4×	Filter
DCP [28]	69.6/88.9%	64.1/85.7%	3.3×	Filter
PatDNN [42]	69.9/89.1%	69.5/89.2%	4.0×	Pattern-based
GRIM	**69.9%/89.1%**	**69.6%/89.2%**	**4.0**×	**BCR**
GRIM	**69.9%/89.1%**	**68.4%/88.6%**	**6.0**×	**BCR**
GRIM	**69.9%/89.1%**	**67.2%/87.7%**	**8.0**×	**BCR**
MobileNet-V2				
AMC [23]	71.8%/*N/A*	70.8%/*N/A*	1.4×	Irregular
GRIM	**70.9%/90.4%**	**70.0%/89.7%**	**2.0**×	**BCR**

TABLE 3: **BCR pruning with optimized block size vs. other methods on TIMIT. PER is *phone error rate.***

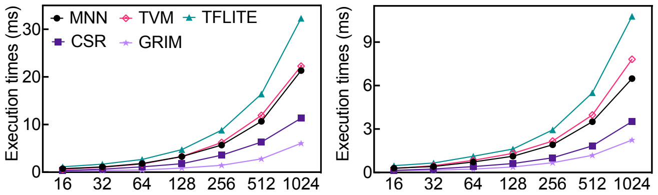
Methods	Dense PER	Sparse PER	Pruning Rate	Pruning Method
GRU				
ESE [46]	20.40%	20.70%	8.0×	Irregular
C-LSTM [52]	24.15%	24.57%	8.0×	Block-circulant
C-LSTM [52]	24.15%	25.48%	16.0×	Block-circulant
E-RNN [53]	20.02%	20.20%	8.0×	Block-circulant
GRIM	**18.8%**	**18.8%**	**10.0**×	**BCR**
GRIM	**18.8%**	**18.8%**	**19.5**×	**BCR**
GRIM	**18.8%**	**23.2%**	**103.8**×	**BCR**
GRIM	**18.8%**	**24.2%**	**245.5**×	**BCR**

TIMIT for RNN. Table 3 shows the BCR pruning results that are evaluated by phone error rate (PER) and pruning rate. We compare BCR pruning with other state-of-the-art



(a) ImageNet-CPU (b) ImageNet-GPU (c) CIFAR-10-CPU (d) CIFAR-10-GPU

Fig. 11: **Overall performance: x-axis are DNN models; y-axis is average DNN end-to-end inference time on a single input.**



(a) CPU exe time (ms). (b) GPU exe time (ms).

Fig. 12: **Matrix Multiply performance: x-axis is row/column size.**

methods, including ESE [46], C-LSTM [52] and E-RNN [53] on the same dataset TIMIT. When pruning rates are low (i.e., not higher than 20×), the BCR pruning guarantees no accuracy degradation, which outperforms ESE at 8× pruning rate and C-LSTM at both 8× and 16× pruning rates in terms of both pruning rate and accuracy. When pruning rates are high (such as 103.8×), BCR can maintain an admirable speech recognition performance, which means the BCR pruned model can even outperform C-LSTM regarding both pruning rate and accuracy. Moreover, the BCR method can well adapt to *ultra-high pruning rate* scenario, e.g., our model with 245× pruning rate can still maintain a comparable phone error rate (24.20%) to C-LSTM (24.15%).

6.3 Overall Execution Time Report

Figure 11 reports GRIM's CPU and GPU execution performance, and compares GRIM with MNN [44], TVM [15], TFLITE [14], CSR [45], and PatDNN [42] on three CNNs (VGG-16, ResNet-18, and MobileNet-V2) trained on two datasets (ImageNet and CIFAR-10), respectively[1]. These evaluations test and report the whole model execution time rather than the time of CONV layers only as PatDNN. GRIM outperforms other frameworks for all cases. On CPU, GRIM achieves 2.47× to 5.75×, 3.08× to 6.11×, 5.98× to 12.55×, 2.81× to 5.81×, and 1.09× to 2.06× speedup over MNN, TVM, TFLITE, CSR, and PatDNN, respectively. On GPU, GRIM achieves 2.46× to 6.84×, 3.08 to 7.09×, 5.47× to 14.08×, 2.59× to 5.41×, and 1.03× to 2.11× speedup over MNN, TVM, TFLITE, CSR, and PatDNN, respectively. Particularly, comparing to PatDNN, GRIM's BCR pruning is more flexible, not only working for 3 × 3 (or 5 × 5, etc.) `CONV` layers targeted by PatDNN but also leading to better pruning and proper optimizations for 1 × 1 `CONV` and varied `FC` layers that PatDNN cannot fully optimize. For the largest CNN (VGG) trained on the largest dataset (ImageNet), GRIM can complete the whole inference of a single input within 33 ms with our mobile GPU, meeting the industrial real-time standard (i.e., 30 frames/sec).

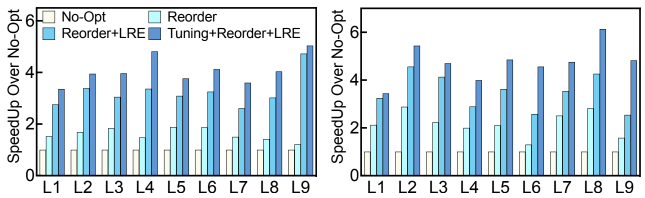
BCR pruning requires to convert all convolutions to GEMM through `Im2col`, which incurs overhead, particularly for large kernels. To validate GRIM's performance on large kernels, this part compares the performance of two kernel sizes, (3, 3) and (11, 11) under the same computation workload (by changing the number of channels) and a 10× pruning rate. The (3, 3) CONV layer achieves 4.5× speedup over TFLite, while (11, 11) achieves 3.3× speedup over TFLite. This is consistent with our intuition and proves that GRIM can still result in notable performance gains on larger kernels regardless of the overhead introduced by `Im2col`. Moreover, large kernels (> 5x5) usually occupy a small portion of the overall computation of models (e.g. 9.3% in AlexNet).

To further validate the advantages of GRIM, this part also compares it with another cutting-edge sparsification approach (named 2:4 pattern) proposed by NVIDIA [56]. 2:4 pattern can also be defined as a fine-grained structured pruning that is natively supported by the latest NVIDIA GPU architecture. This work implements 2:4 pattern pruning with the algorithm stated in the section 5.2. Table 1 and Table 2 shows that GRIM achieves comparable accuracy with 2:4 pattern pruning under the same pruning rate. Because neither mobile CPU or mobile GPU is equipped with dedicated hardware that supports 2:4 pattern pruning, this work uses the CSR baseline aforementioned to store the sparse data and performs the computation of 2:4 pattern. For VGG-16 on CIFAR-10 (with 71.3× pruning rate) and VGG-16 on ImageNet (with 12× pruning rate), GRIM achieves 2.6× and 3.1× speedup compared with this CSR-based NVIDIA 2:4 implementation on mobile CPU, respectively.

For GRU RNN, because the above mobile frameworks do not support end-to-end execution. We compare GRIM with them on matrix multiplication kernels with varied sizes.

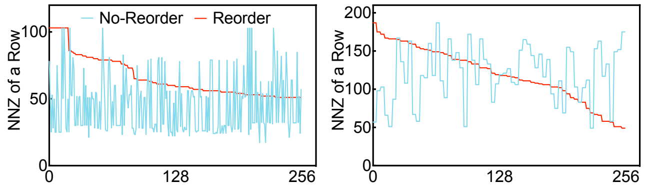
The weight matrix is pruned with a 10× pruning rate. Figure 12 reports the result. All frameworks' execution time increases as the matrix size grows. GRIM performs the best, with up to 2.3×, 4.3×, 6.1×, and 2.5× speedup over MNN, TVM, TFLITE, and CSR. PatDNN is not listed because it optimizes CONV directly without transforming to GEMM. GRIM completes GRU inference on Adreno 640 GPU within 81us (for sequence length of 1 and batch size of 32). We compare GRIM with a representative FPGA implementa-

[1] Sparse models w/ highest pruning rate in Table 1 to 3 are selected



(a) ImageNet-VGG16-CPU (b) ImageNet-VGG16-GPU

Fig. 13: **Speedup: Opt over No-Opt on different CONV layers of VGG.**



(a) RNN: a 1024 × 1024 layer (GRU). (b) CNN: 256×128 in/out channels (VGG).

Fig. 14: **Matrix reorder: x-axis is row id.**

TABLE 4: **VGG CONV layers characteristics.**

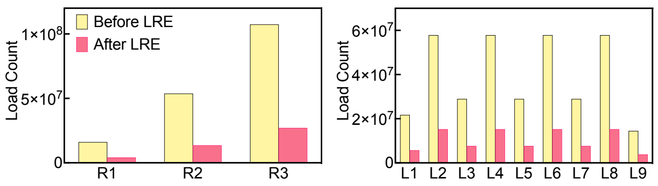
Name	Filter shape	Name	Filter shape	Name	Filter shape
L1	[64,3,3,3]	L4	[128,128,3,3]	L7	[512,256,3,3]
L2	[64,64,3,3]	L5	[256,128,3,3]	L8	[512,512,3,3]
L3	[128,64,3,3]	L6	[256,256,3,3]	L9	[512,512,3,3]

tion, ESE [16]. GRIM can even slightly outperform ESE[2]. Specifically, GRIM can achieve significantly higher energy efficiency (38×) comparing with ESE.

6.4 Performance Optimizations Break-down

GRIM's superior performance mainly comes from two major resources. First, it has a fully optimized dense baseline, which is already 1.1× to 1.6× faster than TVM and MNN (extra optimizations are shown in Table 5). Second, the flexible BCR pruning compresses the overall computation by 4× to 20×. However, this computation reduction cannot transform to performance gains directly without further compiler optimizations due to the computation and memory access irregularity. This can be proved by CSR's performance in Figure 11. Because CSR cannot leverage our compiler optimization, although its computation is almost equivalent to GRIM's, a significant performance gap still exists between CSR and GRIM. This part carefully studies the impact of GRIM's compiler optimizations. Please notice that these optimizations are *only enabled by BCR pruning*.

Figure 13 shows the performance improvement given by each optimization for VGG (on ImageNet)[3]. The x-axis denotes the layers in VGG, and more detailed information is shown in Table 4. This result uses the DNN execution code on BCR pruned models without any optimization (`No-Opt`) as the evaluation baseline. On CPU, matrix reorder (`Reorder`) brings 1.21× to 1.88× speedup, register-level load redundancy elimination brings extra 1.11× to 3.51× speedup, and auto-tuning brings additional 0.31× to 1.45× speedup. On GPU, these numbers are 1.30× to 2.88×, 0.89× to 1.90×, and 0.19× to 2.28×, respectively. Matrix reorder optimization yields more benefits on GPU than CPU, because GPU has more threads and hence is more sensitive to thread divergence and load imbalance. We next characterize matrix reorder, load redundancy elimination, and compact storage optimizations to explain why they work. Auto-tuning and other optimizations are not further explained because their effects are more straightforward.



(a) GRU(RNN) (b) VGG(CNN)

Fig. 15: **Register load counts before and after LRE. (R1 to R3 in RNN are layers of GRU with different matrix sizes,** 152×1024, 512×1024, 1024×1024**. CNN uses CONV layers from VGG.)**

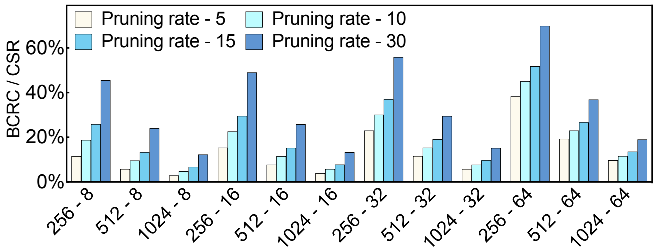


Fig. 16: **Extra data overhead comparison: BCRC/CSR with varied matrix sizes (x-axis) and pruning rates.**

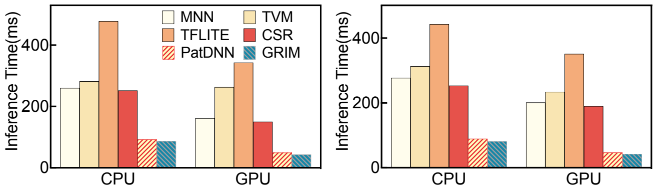
Effect of Matrix Reorder. Figure 14 shows the number of non-zero weights (nnz) in each row for an RNN FC layer and a CNN CONV layer, respectively. Only the first 256 rows are plotted for readability. The nnz distribution is very random before matrix reorder (`No-Reorder`), incurring significant thread divergence and load imbalance if these rows are processed by different threads. This distribution becomes much more regular after reorder (`Reorder`). The rows with similar nnzs can be grouped together and each group can be processed by all threads simultaneously to minimize thread divergence and load imbalance.

Effect of LRE. Figure 15 reports the register load counts before and after the load redundancy elimination for multiple layers with different matrix sizes from both GRU (RNN) and VGG (CNN). It shows the number of register loads is significantly reduced with LRE optimization. This explains why LRE yields so obvious performance gains even after the traditional data locality optimizations like tiling.

BCRC VS CSR. Figure 16 shows the extra data storage overhead (i.e., the data size other than non-zero weights) for both BCRC and CSR with varied matrix sizes and pruning rates. BCRC saves 61.7% to 97.1%, 54.9% to 95.2%, 48.3% to 93.3%, and 30.1% to 87.7% extra data over CSR for different pruning rates. This results in up to 48.5%, 47.6%, 46.6%, and

[2] ESE completes GRU with around 82us

[3] RNN and other CNN results are omitted, very similar to VGG.



(a) Snapdragon 845 (b) Kirin 980

Fig. 17: **Portability evaluation with VGG on ImageNet.**

43.8% overall data reduction.

6.5 Portability Evaluation

We also run GRIM on two other phones to validate its portability. We got very similar performance comparison results as above, which are omitted due to the space limitation. Figure 17 reports the performance comparison of VGG (the most complex/largest DNN in our evaluation) between GRIM and other frameworks. On both platforms, GRIM outperforms others for both CPU and GPU, proving GRIM's good performance portability. GRIM's design and optimization are general, not specific to any brand or type of mobile device. GRIM is also less sensitive to resource constraints because of its high pruning rate, so its performance is stable on other mobile devices with even weaker computation power and smaller memory capabilities (e.g., Raspberry Pi).

7 Related Work

There have been many efforts on DNN acceleration frameworks on mobiles, like DeepEar [57], MCDNN [58], DeepMon [59], DeepSense [60], DeepCache [61], etc. Among those TFLite [14], TVM [15], Alibaba MNN [44], and PatDNN [42] are four state-of-the-art end-to-end acceleration frameworks for mobiles. Please note that those four do not or only partially support RNNs. Also, these prior work do not utilize sparse DNN model inference, except for PatDNN. Table 5 compares the major optimizations in TFLite, TVM, and MNN with GRIM. Basically, TFLite, TVM, and MNN are optimized for dense DNNs.

Table 6 compares GRIM with PatDNN [42], an effort sharing the most similarities with GRIM. PatDNN does not support RNNs, because the pattern-based pruning it uses results in a fine-grained structured sparsity that only applies to weight tensors of the CONV layers. So next we will further explain the differences between them on CNNs. The fine-grained structured sparsity scheme by our BCR pruning achieves a higher pruning rate than PatDNN without accuracy loss. Because (i) our fine-grained structured sparsity scheme has higher flexibility since PatDNN performs pruning only within CONV filter kernels, while our BCR pruning is performed within in a desirable block size, which does not necessarily follow the CONV filter kernels; (ii) GRIM uses a systematic optimization approach to determine the hyperparameters for pruning, while PatDNN uses a heuristic approach to select pattern candidates. From the compiler optimizations, our GRIM introduces a new Domain Specific Language to offer users more flexibility of using existing DNNs or creating new ones, thus improving the programmability and productivity in DNN programming. Furthermore, our compiler optimizations are fully customized for our BCR pruning techniques. Lastly, PatDNN and GRIM are complementary to each other and can be combined.

There are some other efforts that explore model compression to accelerate DNN executions including [47], DeftNN [62], SCNN [63], and AdaDeep [64]. However, they either require new hardware support, or need a trade-off between inference acceleration performance and accuracy, or do not target mobile platforms. Their focuses are different from GRIM.

TABLE 5: **DNN acceleration frameworks on mobile.**

DNNs	Optimization Knobs	TFLite	TVM	MNN	**GRIM**
Dense	Parameters auto-tuning	N	Y	N	**Y**
	CPU/GPU support	Y	Y	Y	**Y**
	Half-floating support	Y	Y	Y	**Y**
	Computation graph opt.	$Y^!$	Y^*	$Y^!$	$\mathbf{Y}^{**}$
	Tensor optimization	$Y^!$	$Y^\dagger$	$Y^!$	$\mathbf{Y}^{\dagger\dagger}$
	RNN opt support	Y^p	N	N	**Y**
Sparse	Sparse DNN model support	N	N	N	**Y**
	BCR pruning	N	N	N	**Y**
	Matrix reordering	N	N	N	**Y**
	Opt. sparse kernel code gen	N	N	N	**Y**
	Auto-tuning sparse models	N	N	N	**Y**

* Operator fusion, constant folding, static memory plan, data layout transform

** Besides above in *, operation replacement

† Scheduling, nested parallelism, tensorization, explicit memory latency hiding

†† Besides above in †, dense kernel reordering, SIMD operation optimization

! Similar optimizations as TVM, but less advanced

p Latest version (partially) supports some RNN executions w/o RNN-specific opts.

TABLE 6: **Comparison between GRIM and PatDNN.**

Approach	Generality	Granularity	Pruning rate	Prune Strategy	DSL
PatDNN	CNN	Fine-grain	Low	Empirical	N
GRIM	CNN&RNN	Fine-grain	High	Systematical	Y

8 Conclusion

This paper presents a novel mobile inference acceleration framework GRIM that is general to both CNNs and RNNs and that achieves real-time performance and high accuracy, leveraging fine-grained structured sparse model inference and compiler optimizations for mobile devices. We begin with design of a new fine-grained structured sparsity scheme through the BCR pruning techniques. Our GRIM framework consists of two parts: (a) the compiler optimizations of execution code generation for real-time mobile inference; and (b) the BCR pruning optimizations for determining pruning hyperparameters and performing weight pruning. For CNNs, we compare with Alibaba MNN, TVM, TensorFlow-Lite, a sparse DNN inference implementation based on CSR format, and PatDNN, achieving significant speedups in the end-to-end inference time. For RNNs, we compare with ESE. GRIM achieves comparable end-to-end inference time with significantly higher energy efficiency. GRIM also has the potential to support other neural networks (e.g, transformers). The extremely deep nature of transformer models (e.g., BERT and its variants, and GPT, etc) requires more careful designs and optimizations of

GRIM. This serves as a promising direction of our future work. Particularly, our on-going study shows that GRIM can achieve a $1.8\times$ pruning rate on BERT-base models without notable accuracy loss.

Acknowledgements

This research is partially funded by National Science Foundation CCF-2047516 (CAREER), CCF-1901378 and CCF-1937500, Army Research Office/Army Research Laboratory via grant W911NF-20-1-0167 (YIP) to Northeastern University, and Jeffress Trust Awards in Interdisciplinary Research to William & Mary. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, Army Research Office, or Thomas F. and Kate Miller Jeffress Memorial Trust.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder–decoder approaches," in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014, pp. 103–111.

[5] S. Bhattacharya and N. D. Lane, "From smart to deep: Robust activity recognition on smartwatches using deep learning," in *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. IEEE, 2016, pp. 1–6.

[6] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of the 15th International Conference on Information Processing in Sensor Networks*. IEEE Press, 2016, p. 23.

[7] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, "Squeezing deep learning into mobile and embedded devices," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017.

[8] K. Ota, M. S. Dao, V. Mezaris, and F. G. De Natale, "Deep learning for mobile multimedia: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 13, no. 3s, p. 34, 2017.

[9] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications Surveys & Tutorials*, 2019.

[10] Y. Deng, "Deep learning on mobile devices: a review," in *Mobile Multimedia/Image Processing, Security, and Applications 2019*, vol. 10993. International Society for Optics and Photonics, 2019, p. 109930A.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 248–255.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] L. Torrey and J. Shavlik, "Transfer learning," in *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI Global, 2010, pp. 242–264.

[14] https://www.tensorflow.org/mobile/tflite/.

[15] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[16] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 1135–1143.

[17] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv preprint arXiv:1607.03250*, 2016.

[18] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 2074–2082.

[19] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1398–1406.

[20] X. Dong, S. Chen, and S. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 4857–4867.

[21] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *International Conference on Learning Representations (ICLR)*, 2017.

[22] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2736–2744.

[23] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "Amc: Automl for model compression and acceleration on mobile devices," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 784–800.

[24] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *International Conference on Learning Representations (ICLR)*, 2018.

[25] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 184–199.

[26] T. Zhang, K. Zhang, S. Ye, J. Li, J. Tang, W. Wen, X. Lin, M. Fardad, and Y. Wang, "Adam-admm: A unified, systematic framework of structured weight pruning for dnns," *arXiv preprint arXiv:1807.11091*, vol. 2, p. 3, 2018.

[27] X. Zhu, W. Zhou, and H. Li, "Improving deep neural network sparsity through decorrelation regularization," in *Proceedings of International Joint Conferences on Artificial Intelligence (IJCAI)*, 2018, pp. 3264–3270.

[28] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2018, pp. 875–886.

[29] C. Min, A. Wang, Y. Chen, W. Xu, and X. Chen, "2pfpce: Two-phase filter pruning based on conditional entropy," *arXiv preprint arXiv:1809.02220*, 2018.

[30] W. Niu, X. Ma, Y. Wang, and B. Ren, "26ms inference time for resnet-50: Towards real-time execution of all dnns on smartphone," *arXiv preprint arXiv:1905.00571*, 2019.

[31] N. Liu, X. Ma, Z. Xu, Y. Wang, J. Tang, and J. Ye, "Autoslim: An automatic dnn structured pruning framework for ultra-high compression rates," *arXiv preprint arXiv:1907.03141*, 2019.

[32] A. Ren, T. Zhang, S. Ye, J. Li, W. Xu, X. Qian, X. Lin, and Y. Wang, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 925–938.

[33] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian, "Variational convolutional neural network pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2780–2789.

[34] M. Yang, M. Faraj, A. Hussein, and V. Gaudet, "Efficient hardware realization of convolutional neural networks using intra-kernel regular pruning," in *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE, 2018, pp. 180–185.

[35] X. Ma, F.-M. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren, and Y. Wang, "Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices," in *Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[36] P. Dong, S. Wang, W. Niu, C. Zhang, S. Lin, Z. Li, Y. Gong, B. Ren, X. Lin, and D. Tao, "Rtmobile: Beyond real-time mobile acceleration of rnns for speech recognition," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

[37] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 3123–3131.

[38] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2016, pp. 4107–4115.

[39] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2016, pp. 525–542.

[40] D. Lin, S. Talathi, and S. Annapureddy, “Fixed point quantization of deep convolutional networks,” in *International Conference on Machine Learning (ICML)*, 2016, pp. 2849–2858.

[41] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, “Extremely low bit neural network: Squeeze the last bit out with admm,” in *Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[42] W. Niu, X. Ma, S. Lin, S. Wang, X. Qian, X. Lin, Y. Wang, and B. Ren, “Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[43] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cudnn: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.

[44] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, C. Lyu, and Z. Wu, “Mnn: A universal and efficient inference engine,” in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., 2020, vol. 2, pp. 1–13.

[45] J. L. Greathouse, K. Knox, J. Poła, K. Varaganti, and M. Daga, “clsparse: A vendor-optimized open-source sparse blas library,” in *Proceedings of the 4th International Workshop on OpenCL*, 2016, pp. 1–4.

[46] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, “Ese: Efficient speech recognition engine with sparse lstm on fpga,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 75–84.

[47] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, “Sparse convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 806–814.

[48] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[49] M. Hong, Z.-Q. Luo, and M. Razaviyayn, “Convergence analysis of alternating direction method of multipliers for a family of nonconvex problems,” *SIAM Journal on Optimization*, vol. 26, no. 1, pp. 337–364, 2016.

[50] S. Liu, J. Chen, P.-Y. Chen, and A. Hero, “Zeroth-order online alternating direction method of multipliers: Convergence analysis and applications,” in *International Conference on Artificial Intelligence and Statistics*, 2018, pp. 288–297.

[51] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[52] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, “C-lstm: Enabling efficient lstm using structured compression techniques on fpgas,” in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2018, pp. 11–20.

[53] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang, “E-rnn: Design optimization for efficient recurrent neural networks in fpgas,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 69–80.

[54] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, N. L. Dahlgren, and V. Zue, “Timit acoustic-phonetic continuous speech corpus,” *Linguistic data consortium*, vol. 10, no. 5, p. 0, 1993.

[55] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021.

[56] https://developer.nvidia.com/blog/exploiting-ampere-structured-sparsity-with-cusparselt/.

[57] N. D. Lane, P. Georgiev, and L. Qendro, “Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning,” in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 283–294.

[58] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, “Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2016, pp. 123–136.

[59] L. N. Huynh, Y. Lee, and R. K. Balan, “Deepmon: Mobile gpu-based deep learning framework for continuous vision applications,” in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2017, pp. 82–95.

[60] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, “Deepsense: A unified deep learning framework for time-series mobile sensing data processing,” in *Proceedings of the 26th International Conference on World Wide Web*, 2017.

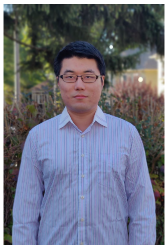
[61] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, “Deepcache: Principled cache for mobile deep vision,” in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. ACM, 2018, pp. 129–144.

[62] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, “Deftnn: Addressing bottlenecks for dnn execution on GPUs via synapse vector elimination and near-compute data fission,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 786–799.

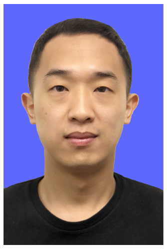
[63] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “Scnn: An accelerator for compressed-sparse convolutional neural networks,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.

[64] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, “On-demand deep model compression for mobile devices: A usage-driven model selection framework,” in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. ACM, 2018, pp. 389–400.

Wei Niu received the B.E. degree in software engineering from Beihang University, Beijing, China, in 2016. He is currently pursuing the Ph.D. degree under Professor Bin Ren in computer science, William & Mary, Williamsburg, VA, U.S. His current research interests include compiler optimizations for model compression of deep neural networks and computer architecture & heterogeneous computing and machine Learning & deep Learning.



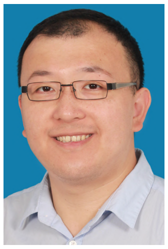
Zhengang Li received the B.E. degree in electronic information engineering from Zhejiang University, Zhejiang, China, in 2017. He is currently pursuing the Ph.D. degree under Professor Yanzhi Wang in computer engineering, Northeastern University, Boston, MA, U.S. His current research interests include model compression of deep neural networks, machine learning algorithm and computer vision.



Xiaolong Ma received the MS degree in Electrical Engineering in Syracuse University, Syracuse, U.S. He is currently a Ph.D. candidate in the Department of Electrical and Computer Engineering in Northeastern University (NEU), Boston, U.S., supervised by Professor Yanzhi Wang. His research interests include machine learning algorithm, high performance computing and computer vision.

Peiyan Dong received the B.E. degree in electronic and information engineering from Beijing Institute of Technology, Beijing, China, in 2016. She is currently pursuing the Ph.D. degree under Professor Yanzhi Wang in computer engineering, Northeastern University, Boston, MA, U.S. Her current research interests include Model Compression for Deep learning & Speech Recognition and Synthesis & Deep Learning.

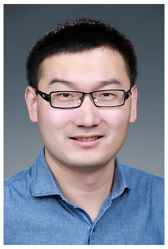
Dr. Gang Zhou is currently an Associate Professor in the Computer Science Department at William & Mary. He was Graduate Program Director from 2015 to 2017. He received his Ph.D. degree from the University of Virginia in 2007 under Professor John A. Stankovic. He has published more than 100 papers in the areas of wireless networks, sensor systems, internet of things, smart health, mobile and ubiquitous computing. There are over 7300 citations of his papers per Google Scholar. He serves or served on the Journal Editorial Board of (1) ACM Transactions on Sensor Networks, (2) IEEE Internet of Things, (3) Elsevier Computer Networks, and (4) Elsevier Smart Health. He served as NSF, NIH, and GENI proposal review panelists multiple times. He also received an award for his outstanding service to the IEEE Instrumentation and Measurement Society in 2008. He is a recipient of the Best Paper Award of IEEE ICNP 2010. He received NSF CAREER Award in 2013. He received a 2015 Plumeri Award for Faculty Excellence. He is a Senior Member of IEEE and ACM.



Xuehai Qian received the BSc degree from Beihang University, China, in 2004, the MSc degree from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2007, and the PhD degree from the Computer Science Department, University of Illinois at Urbana-Champaign, in 2013. Currently, he is an assistant professor with the Ming Hsieh Department of Electrical Engineering and the Department of Computer Science, University of Southern California. His research interests include the fields of computer architecture, architectural support for programming productivity, and correctness of parallel programs.

Xue Lin (S'12–M'16) received the B.S. degree from Tsinghua University, Beijing, China, in 2009, and the Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 2016, under the supervision of Prof. M. Pedram. She is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA. Her current research interests include high-performance computing and mobile cloud computing systems, near-threshold computing for low-power embedded systems, and machine learning and computing in cyber-physical systems. She has published about 50 papers in the above areas.

Yanzhi Wang (Member, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2009, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 2014. His research interests include energy-efficient and high-performance implementations of deep learning and artificial intelligence systems, emerging deep learning algorithms/systems, generative adversarial networks, and deep reinforcement learning.



Bin Ren received the B.S. degree from Beihang University, China, in 2006, and the M.S. and Ph.D. degrees from the Department of Computer Science and Engineering, Ohio State University, Columbus, Ohio, in 2013 and 2014, respectively. He was a post-doctoral research associate with the High-Performance Computing Group, Pacific Northwest National Laboratory. He is currently an assistant professor with the Department of Computer Science, William & Mary. His research interest includes software systems, specifically programming systems, compiler and programming language support for parallel computing, and high-performance machine learning/deep learning systems.