Effective Performance Modeling and Domain-Specific Compiler Optimization of CNNs for GPUs

Yufan Xu
yf.xu@utah.edu
University of Utah
Salt Lake City, Utah, USA

Qiwei Yuan
joshua.yuan@utah.edu
University of Utah
Salt Lake City, Utah, USA

Erik Curtis Barton
u0882558@utah.edu
University of Utah
Salt Lake City, Utah, USA

Rui Li
lirui@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

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

Aravind Sukumaran-Rajam
aravind_sr@outlook.com
Meta Platforms
Menlo Park, California, USA

ABSTRACT

The Convolutional Neural Network (CNN) kernel is a fundamental building block for deep learning, which dominates the computational cost of deep learning pipelines for image analysis. The synthesis of high-performance GPU kernels for CNNs is thus of considerable interest. The current state-of-the-art in optimizing CNN kernels is auto-tuning search using AutoTVM/Ansor, which has been shown to achieve higher performance than vendor libraries as well as polyhedral compilers. A primary reason for the failure of general-purpose optimizing compilers to deliver high-performance code for key kernels like CNN is the challenge of accurate performance modeling to enable effective choice among alternative transformations and/or parameter values such as tile sizes. In this paper we ask if a domain-specific compiler that is customized for the important CNN kernel can be more effective. Our results show that it can be very effective, enabling even higher performance of the generated GPU code for CNNs than auto-tuning with TVM/Ansor. Further, we demonstrate the effectiveness of a performance modeling approach that integrates analytical modeling of data movement volume with machine learning for offline training, enabling much more rapid code optimization than the approach of TVM/Ansor that is based on online construction of a machine learning model to guide auto-tuning search.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; *Neural networks*; • **Software and its engineering** → *Compilers*; **Software performance**.

KEYWORDS

CNN, GPU, Design space exploration, Tile size optimization, Performance modeling

ACM Reference Format:
Yufan Xu, Qiwei Yuan, Erik Curtis Barton, Rui Li, P. Sadayappan, and Aravind Sukumaran-Rajam . 2022. Effective Performance Modeling and Domain-Specific Compiler Optimization of CNNs for GPUs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 8–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569674

1 INTRODUCTION

Optimizing the Convolutional Neural Network (CNN) computation for GPUs is a very challenging task. A huge combination of choices exist for parameters determining the shape/size of thread blocks[1], work distribution among warps in a thread block, size/shape of register tiles at each thread, and tile loop permutations in the thread code. Due to the fact that the convolutional neural network stages dominate the compute time of DNN pipelines for image analysis applications, there have been numerous efforts to optimize the CNN kernel. These efforts can be classified into three broad categories:
Auto-tuning frameworks: Auto-tuners such as AutoTVM [2, 3], now integrated with the Ansor[18] optimizer, search in a large space of possible code variants. The search is guided by use of a dynamically constructed Machine Learning performance model trained with timing data from actual execution of many code variants on the target GPU platform during the auto-tuning search.
Polyhedral Compilers: Since the CNN computation is a purely affine loop computation, it is amenable to optimization by polyhedral compilers like PPCG [16], Tiramisu[1] and TensorComprehensions [15]. However, it has been shown that AutoTVM [2] and Ansor[18] generate much higher performing GPU code for CNN than Polyhedral GPU compilers. A primary reason is that polyhedral compilers to date have only been able to optimize code using linear objective functions and tilesize optimization is an inherently nonlinear optimization problem.
Vendor Libraries: Vendor libraries like cuDNN [4] are engineered by GPU experts to be well matched to low-level architectural features and therefore achieve extremely high performance close to machine peak for sufficiently large problem sizes. However, for the actual sizes/shapes of convolution stages found in practically used CNN image processing pipelines like ResNet [5] and Yolo

[1]We use Nvidia CUDA terminology in this paper, but the concepts are directly applicable to other GPUs and GPU programming models like HIP/ROCm, OpenCL, SyCL etc.

[13], the achieved performance is often lower than that achieved by optimized GPU code synthesized by TVM/Ansor.

For a given CNN pipeline, the size of each convolutional kernel is generally fixed and known at compile time. Hence, it is possible to create customized kernels optimized for the specific sizes of tensors for each stage in a CNN pipeline (as is done by auto-tuning frameworks like TVM/Ansor), providing an advantage over a library like cuDNN that cannot use layer-specific optimized code for each stage in a DNN pipeline.

This paper seeks answers to the following two questions:

(1) Although state-of-the-art auto-tuning search via TVM/Ansor has been shown to achieve substantially higher performance for CNN on GPUs than polyhedral compiler optimization [2, 18], can a domain-specific approach to compiler optimization do better than state-of-the-art auto-tuning search?
(2) While auto-tuning search with TVM/Ansor can generate GPU CNN kernels with higher performance than state-of-the-art vendor libraries like cuDNN, it generally requires a very large number of compile/execute trials to do so, in the order of a thousand or more, thereby needing several hours of time to optimize a DNN pipeline. Can a domain-specific approach enable much faster generation of GPU CNN kernels than Ansor/TVM?

This paper provides an affirmative answer to both the above questions:

Code Quality (Performance): With respect to the first question, we show that although the design space of all possible GPU implementations of the CNN kernel is explosively large, we can use domain-specific analytical modeling to significantly prune the huge design space. Guided by the analysis, we devise two GPU CNN kernel schemas with parametric tile sizes. Further, by imposition of capacity constraints from registers and scratchpad memory, the number of pruned kernel configurations is typically no more than a few thousand for representative CNN kernels - we evaluated the CNN kernels from three DNN pipelines: ResNet-18 [5], Yolo9000 [14], and DefoNet [17]. Thus, the total number of pruned configurations to consider is of the same order of the number of compile/execute trials typically needed for auto-tuning search with TVM/Ansor. We demonstrate consistently superior performance of the generated CNN kernels over both cuDNN and TVM/Ansor. Details are presented in **Sec. 3**.

Code Optimization Time: With respect to the second question, we develop an *off-line* regression model for the predicted execution time for any set of tile sizes for any given set of CNN parameters (sizes of activations and kernel weights) for each target GPU platform. In addition to the standard features that would be used to build a machine learning model to predict execution time of. code version (tile sizes, tensor sizes, total number of operations, etc.), we also include additional features from analytical modeling of the volume of data movement between global and shared memory and between shared memory and registers. The regression model is then used to rapidly evaluate a *Top-n* set of configurations via actual compile/execute on the target GPU platform. We demonstrate that the number of evaluations needed to achieve comparable performance with our domain-specific optimization approach is significantly lower than needed by TVM/Ansor. Details are presented in **Sec. 4**.

The key contributions of the paper are as follows:

- It presents the first evidence (to our knowledge) that an approach based on analytical performance modeling can surpass achieved performance for GPU CNN kernels than that achievable via the state-of-the-art TVM/Ansor auto-tuning search framework.
- It demonstrates the effectiveness of a new approach to performance modeling that includes analytically derived performance metrics among the input features used for training a machine learning regression model.
- The effectiveness of the domain-specific optimization approach is demonstrated on two GPUs (Nvidia Volta V100 and 2080 Ti) by synthesizing higher performing GPU CNN kernels than the state-of-the-art vendor library cuDNN and the state-of-the-art auto-tuning compiler framework TVM/Ansor, for the convolution stages of three image-processing CNN pipelines (ResNet18[5], Yolo9000[14], and DefoNet [17]). Further, we demonstrate the ability to generate code of comparable/better performance using vastly fewer compile/execute trials than TVM/Ansor.

2 BACKGROUND AND OVERVIEW OF PAPER

A GPU kernel for computing the convolution operation at a specific CNN layer in a pipeline is generally implemented as a collection of loops executed by each thread, so that the set of parallel threads collectively execute all the needed elementary operations. GPUs use a two-level thread hierarchy where a collection of threads forms a thread-block and a collection of thread-blocks forms a grid. The total number of threads in the grid collectively execute all operations needed to produce all elements of the result tensor. Each thread can be responsible for many output elements; conversely a collection of threads might partially contribute to an output element. There are a combinatorially explosive number of possible ways of mapping the total work to be performed onto a set of threads and the sequential order in which each thread executes the operations it is responsible for. The full space of design choices may be considered to be the set of all possible valid GPU programs that compute the given CNN computation.

The task of designing a high-performance GPU kernel for a given CNN stage may be viewed as a design-space exploration problem with an extremely large space of possible designs. AutoTVM [2] is a very effective framework that performs an auto-tuning search in the large space of possible loop configurations and tile sizes, guided by a dynamically generated ML model. The overall optimization process proceeds as a sequence of outer steps. A progressively refined ML model is used at each step, to rapidly evaluate a large number of candidate code configuration choices (tile size parameters and loop permutations) and a batch of the best predicted configurations are synthesized as CUDA codes, compiled by the Nvidia nvcc compiler and executed on the target GPU platform. The measured set of execution times from the current step are used as samples to further train the ML performance model. The best configuration from the current step is also recorded if it improves on the best time achieved so far. As the outer steps proceed, the best achieved time tends to improve for a while and then plateau. Typically, several hundreds to a few thousands of code versions are generated, compiled and executed and the best of all tested configurations is used as the optimized output kernel. Recently, the Ansor [18] optimizer was incorporated into the AutoTVM framework as the *AutoScheduler* option, and has been shown to generate better performing code than the original AutoTVM search strategy[2]. In this paper, we use the TVM AutoScheduler in all experimental comparisons with our developed kernels, referring to it as TVM/Ansor.

AutoTVM and Ansor do not use analytical performance models but use a data-driven ML model as the basis for their search through a design space of code configurations to generate, compile, and execute in the auto-tuning process. Several questions of interest in this study are:

- Is there potential to do even better than the impressive performance results achieved by AutoTVM/Ansor for generating high-performance GPU kernels for CNN pipeline stages?
- Although analytical performance models used in polyhedral optimizing compilers have not so far been shown to be competitive with TVM/Ansor [2], can a domain-specific approach based on analytical modeling be more effective?
- For scenarios where the excessive auto-tuning time of TVM/Ansor (typically many hours to generate high-performance GPU CNN kernels for a full DNN pipeline) is unsatisfactory, can any alternate performance modeling approach be developed that can achieve much faster results for GPU kernel optimization for CNN compared to that taken by TVM/Ansor?

We proceed as follows to address the above questions. First, consider the full space of all possible GPU programs for computing the convolution operations for a specified CNN stage. The possible code versions that are explored by TVM/Ansor are clearly only a very small sub-space of the full space of all possible CUDA programs for a given CNN stage, guided by a dynamically constructed machine learning model using data from compiling and executing a set of code configurations. Instead, we use analytical modeling as a means of pruning the design space of code configurations.

We start with a generic multi-dimensional nested tile loop structure corresponding to the GPU thread hierarchy and then use analytical reasoning about data movement to concretize sub-set code schemas from the generic schema by pruning away many sub-optimal choices with respect to data movement. By use of analytical reasoning, we further prune the design space to eliminate configurations that violate capacity constraints on available register counts and scratchpad shared-memory capacity, leaving only a manageable set of a few thousand configurations that we can exhaustively enumerate. We present the design and experimental evaluation in Sec. 3.

We then develop an effective performance model for rapid selection of a small subset of the pruned space of code configurations to compile and execute on the target platform. We develop an off-line ML regression model to predict execution time of a code configuration by generating a large training set of code configurations for a collection of random CNN problem sizes and tile sizes. In addition to using the tensor extents and tile sizes along the different iteration-space dimensions as input features for developing the regression model, we also include analytically derived features such as the estimated data movement volume between global memory and shared memory, and between shared-memory and registers.

Table 1: Description of Notation

	Description		Description
N	Batch size	T_x	# threads in x dim
X	Width of Output	W_x	# warps in x dim
Y	Height of Output	B_x	# threadblocks in x dim
F_x	Width of Kernel/Filter	R^x	Register Tile Size
F_y	Height of Kernel/Filter	T^x	Warp Tile Size
K	# Output Channels	W^x	Threadblock Tile Size
C	# Input Channels	B^x	Grid Tile Size

We present details of the performance modeling and experimental evaluation in Sec. 4.

3 GPU KERNEL DESIGN AND ANALYTICAL PRUNING OF SEARCH SPACE

In this section, we present the design of our GPU CNN kernels, where analytical modeling is used to significantly prune the very large design space. Table 1 shows the parameter naming convention we use in this paper. As elaborated later in this section, we use multi-level tiling along each of the iteration-space dimensions of the CNN computation. The levels of tiling correspond to the multiple levels of parallelism in the GPU thread hierarchy: each thread can have a register-level tile along each iteration-space dimension, a set of threads form a warp, a set of warps form a thread-block, and a set of thread-blocks constitute the grid of threads that performs the entire computation. Consider the horizontal spatial dimension along an image, denoted by X, the width of the output image. With multi-level tiling, there exists a tile at each level of tiling that corresponds to that dimension x, with the product of tile-loop extents at all levels equalling the full extent X. At any given level, the number of tiles at that level is denoted by a subscripted tile parameter, while the cumulative tile size is denoted with a superscripted tile parameter. The register-level tile size along x is denoted R^x. A group of T_x threads in a warp span equal distinct extents along that iteration-space dimension, giving a cumulative tile size that is denoted $T^x = T_x \times R^x$. Similarly, at the next level of tiling, W_x denotes the number of warps that span distinct extents along the x-dimension of the iteration space, while $W^x = W_x \times T_x \times R^x$ denotes the cumulative tile size at that level.

3.1 Design Overview

$$Out[n,k,y,x] = \sum_{c,fx,fy} In[n,c,y{+}fy,x{+}fx] \times Ker[k,c,fy,fx] \quad (1)$$

Equation 1 shows simplified code for a convolutional operator (stride/dilation are not shown, but the optimization approach also applies to all strided/dilated kernels; stride-2 kernels are included in the experimental evaluation). Tiling is a fundamental technique to improve data reuse. At a high level, the generation of an optimized GPU kernel for this loop code may be systematically modeled as a multi-level tiling, with tiles at different levels representing a partitioning of the 7D iteration space in a nested fashion onto the multi-level parallel GPU execution model. A CUDA program is organized as a collection of independently executing thread-blocks, with each thread block being comprised of a number of warps, the basic unit of hardware scheduling where a group of 32 threads form a warp. Our objective is to effectively map the CNN computation to the GPU thread hierarchy, minimizing data movement through the memory hierarchy.

Fig. 1(a) shows a 6D loop code for the convolution computation (we omit the batch index in our discussion, allowing us to focus on the more challenging case of performance optimization where batch-size=1). Fig. 1(b) shows an abstract multi-level tiled form of the CNN code, with a tile loop-band for each level in the GPU thread hierarchy: the grid level, thread block level, warp level, and thread-level (register) tiling. At the innermost loop-band, we use two variants, the *RS Kernel* shown in Fig. 1(c), and the *S Kernel*, shown in Fig. 1(d).

Of the 6 loops in Fig. 1(a), three represent reduction loops, corresponding to a summation over those dimensions, as seen in Eq. 1: C, F_x, and F_y. The remaining three loops, over X, Y, and K, are parallel loops that collectively loop over the computation to produce the $X \times Y \times K$ output tensor elements. Since F_x and F_y are generally small prime numbers (usually 1, 3, 5, or 7) and cannot be factored, tiling of those loops is not considered and they only appear in the innermost loop band (the RS or S kernel). The loop dimensions corresponding to the parallel loops, X, Y, and K are potentially tiled at every level in the hierarchy. The previously described naming convention for tiles is summarized in Fig. 1(e). Consider the tiling of the output channel dimension of total extent K. At the innermost level, each thread has a register-level tile size R^k, where the base R represents the register level of tiling, and the superscript k denotes the register tile extent. We require the tile size R^k to be a factor of the total extent along that iteration space extent, K. At the next level, the set of threads in a warp (32) are factored into a product $T_x \times T_y \times T_k = 32$, where T_x, T_y, T_k represent the number of iterations of the corresponding tile loops at the warp-level. Along the output channel dimension k, each thread traverses a range of R^k, and the set of 32 threads in a warp collectively traverse a tile of span $R^k \times T_k$, denoted as T^k. Similarly, at the next higher level in the thread hierarchy, a collection of warps forms a thread-block. The number of warps along the parallel loop dimensions at the thread-block level are denoted W_x, W_y, W_k, with the tile extent spanned by all threads in a thread block along the k dimension being $W^k = W_k \times T^k = W_k \times T_k \times R^k$. At the innermost tile band corresponding to register tiling, two alternate GPU kernels are considered (for reasons discussed later in this section), the *RS Kernel* and the *S Kernel*. These choices, based on domain-specific analysis, combined with capacity-based pruning reduce the total number of distinct loop configurations to a few hundreds to thousands, which can be exhaustively enumerated (by compiling and execution on the target GPU) at a comparable cost to that taken by TVM/Ansor via its online auto-tuning search with a dynamically constructed ML model. Further, as we demonstrate in Sec. 4, we can dramatically reduce the time for optimized code synthesis by developing an *off-line* hybrid analytical-ML performance model.

3.2 Design Details

Pruning Register Tiles for Input Channel: Parallelizing the reduction loop over the input channel dimension (c) will result in multiple threads contributing to the same output point, which necessitates atomic updates. In order to avoid this, we do not parallelize the C dimension at the thread/warp level.

It is desirable to maximize the reuse of output elements, especially since each contribution to the output results in a read-write operation. Hence we place a slice of the output tensor of size $R^k \times R^y \times R^x$ in registers, abstracted in Fig. 1 as O_{reg}.

The tile size for each loop affects the data movement and resource usage, which in turn affects thread occupancy (concurrency). We do not tile the C loop at the register level. Our reasoning is as follows. The data movement at the register level per thread can be expressed as $(F_x \times F_y \times R^k + (R^x + F_x - 1) \times (R^y + F_y - 1))$ and the data register capacity requirement can be expressed as $(R^c \times R^x \times R^y + R^c \times (R^x + F_x - 1) \times (R^y + F_y - 1)$. It may be seen that R^c appears only in the capacity expression but does not appear in the data movement expression, which implies that the choice of R^c does not directly affect the data movement but affects the capacity requirement. Based on the above analysis, any value of register-level tiling of R^c that is larger than 1 is clearly sub-optimal, since it will force smaller values for other register-tile sizes, which all appear in the denominator for the data volume expression. We therefore do not need to include any register-tile loop for C.

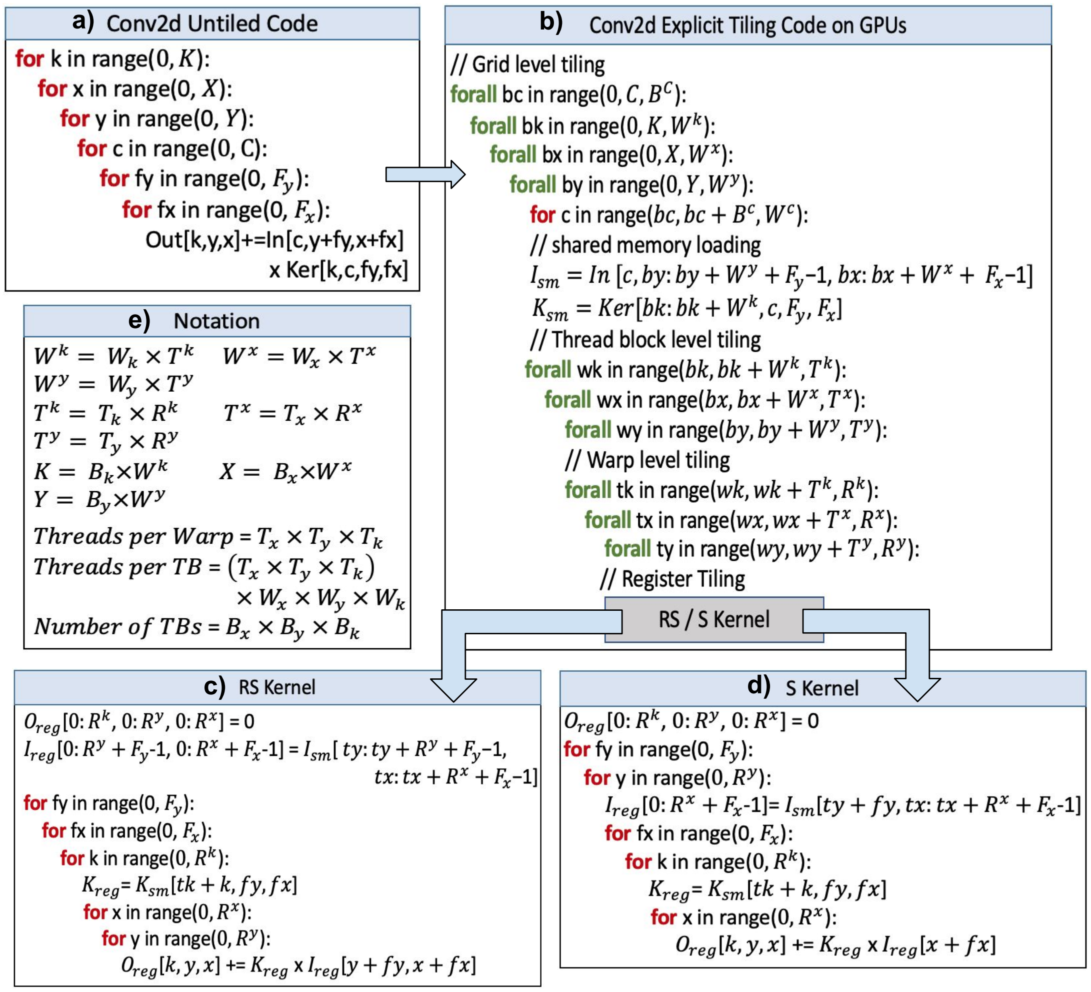
Design space pruning via capacity constraints: We first discuss the RS-kernel, which holds an output tile of size $R^k \times R^y \times R^x$ stationary in the registers per thread. This design allows read-write reuse of $F_x \times F_y \times C$ per output element. Similarly, each thread keeps an input slice of dimension $(R^x + F_x - 1) \times (R^y + F_y - 1)$. Each input element (except boundary elements) achieves a register level reuse of $R^k \times F_y \times F_x$.

Multiple threads in a warp are distributed along the output channel dimension and hence do not require any intra warp reduction operations. So, the warp shape is T_x, T_y, T_k. The remaining threads in a thread block are split into W_x, W_y, W_k, representing a slice of X, Y and K dimension resulting in a thread block of shape $(T_x \times W_x, T_y \times W_y, T_k \times W_k)$. The tile sizes are selected such that the total number of registers in equation 2 required by each thread block to hold input-output operands for computation, is less than the maximum number of registers per thread block (hardware limit).

$$((R^x \times R^y \times R^k) + (R^y + F_y - 1) + (R^x + F_x - 1) + 1) \times (T_x \times T_y \times T_k) \quad (2)$$

Impact of Thread Occupancy: S Kernel: One drawback of the RS-Kernel is the high register capacity requirement, which can adversely affect thread occupancy. The high register usage for input register tiles can result in low thread occupancy, which can result in reduced performance. Hence, we also consider an alternate version of the RS-kernel called the S-kernel (Listing 1 d)). In contrast to the RS-kernel's register usage in equation 2, the S-kernel design only holds a 1D input slice of size $(R^x + F_x - 1)$ in registers, which reduce the register usage for a single thread and potentially have a chance to increase occupancy. Like the RS kernel, each thread holds an output tile of size $R^x \times R^y \times R^k$ stationary in the registers. An additional register is used to hold the kernel element. The tile sizes are selected such that the total number of registers in equation 3 required by each thread block to hold input-output operands for computation, is less than the maximum number of registers per thread block (hardware limit).

$$((R^x \times R^y \times R^k) + (R^x + F_x - 1) + 1) \times (T_x \times T_y \times T_k) \quad (3)$$



a) Conv2d Untiled Code

```
for k in range(0, $K$):
  for x in range(0, $X$):
    for y in range(0, $Y$):
      for c in range(0, C):
        for fy in range(0, $F_y$):
          for fx in range(0, $F_x$):
            Out[k,y,x]+=In[c,y+fy,x+fx]
                       x Ker[k,c,fy,fx]
```

b) Conv2d Explicit Tiling Code on GPUs

```
// Grid level tiling
forall bc in range(0, $C$, $B^C$):
  forall bk in range(0, $K$, $W^k$):
    forall bx in range(0, $X$, $W^x$):
      forall by in range(0, $Y$, $W^y$):
        for c in range($bc$, $bc + B^c$, $W^c$):
        // shared memory loading
        $I_{sm} = In\,[c, by\!: by + W^y + F_y - 1, bx\!: bx + W^x + F_x - 1]$
        $K_{sm} = Ker[bk\!: bk + W^k, c, F_y, F_x]$
        // Thread block level tiling
        forall wk in range($bk$, $bk + W^k$, $T^k$):
          forall wx in range($bx$, $bx + W^x$, $T^x$):
            forall wy in range($by$, $by + W^y$, $T^y$):
            // Warp level tiling
            forall tk in range($wk$, $wk + T^k$, $R^k$):
              forall tx in range($wx$, $wx + T^x$, $R^x$):
                forall ty in range($wy$, $wy + T^y$, $R^y$):
                // Register Tiling
                RS / S Kernel
```

e) Notation

$W^k = W_k \times T^k \quad W^x = W_x \times T^x$
$W^y = W_y \times T^y$
$T^k = T_k \times R^k \quad T^x = T_x \times R^x$
$T^y = T_y \times R^y$
$K = B_k \times W^k \quad X = B_x \times W^x$
$Y = B_y \times W^y$
$Threads\ per\ Warp = T_x \times T_y \times T_k$
$Threads\ per\ TB = (T_x \times T_y \times T_k) \times W_x \times W_y \times W_k$
$Number\ of\ TBs = B_x \times B_y \times B_k$

c) RS Kernel

```
$O_{reg}[0\!: R^k, 0\!: R^y, 0\!: R^x] = 0$
$I_{reg}[0\!: R^y + F_y - 1, 0\!: R^x + F_x - 1] = I_{sm}[\,ty\!: ty + R^y + F_y - 1,$
                                        $tx\!: tx + R^x + F_x - 1]$
for fy in range(0, $F_y$):
  for fx in range(0, $F_x$):
    for k in range(0, $R^k$):
      $K_{reg} = K_{sm}[tk + k, fy, fx]$
      for x in range(0, $R^x$):
        for y in range(0, $R^y$):
          $O_{reg}[k, y, x]$ += $K_{reg}$ x $I_{reg}[y + fy, x + fx]$
```

d) S Kernel

```
$O_{reg}[0\!: R^k, 0\!: R^y, 0\!: R^x] = 0$
for fy in range(0, $F_y$):
  for y in range(0, $R^y$):
    $I_{reg}[0\!: R^x + F_x - 1] = I_{sm}[ty + fy, tx\!: tx + R^x + F_x - 1]$
    for fx in range(0, $F_x$):
      for k in range(0, $R^k$):
        $K_{reg} = K_{sm}[tk + k, fy, fx]$
        for x in range(0, $R^x$):
          $O_{reg}[k, y, x]$ += $K_{reg}$ x $I_{reg}[x + fx]$
```

Figure 1: Abstracted Structure of Design Space of CNN Kernels for GPU

Tail effect and Synchronizations: Reduction Parallelism along Input Channels: Tail effects (also called wave effects) refer to load imbalance in GPU execution that can occur when the total number of thread blocks is not much larger than the number of SMs (Streaming Multiprocessors) in a GPU. For example, if the number of thread blocks is 80 and the GPU has 64 SMs, a first *wave* of 64 thread blocks will keep all SMs busy, but the remaining 16 thread blocks will only keep one fourth of the SMs occupied during the second wave. This is often the case for later stages of DNN pipelines where X and Y are small and C and K are large. For such CNN stages, it may be desirable to utilize reduction parallelism across the input channel dimension c. By splitting C across multiple thread blocks, we can increase the total number of thread blocks. This implies increased global memory traffic of output elements and the need for atomic operations. Nevertheless, in some cases, the gains from alleviating tail effects can overcome the cost of splitting C. The top figure of Fig. 2 shows details of the RS-kernel mapping, and the bottom one shows the S-kernel design.

3.3 Experimental Evaluation

Next, we present performance data from experimental evaluation. We used the non-degenerate convolution layers (i.e., excluding 1x1 convolutions, which are essentially simpler matrix-matrix multiplication operations) in three CNN networks (Resnet18[6], Yolo9000[14] and DefoNet[17]) as the benchmarks for evaluation. The size parameters for these convolution layers are provided in Table 2.

We compare the performance of the best code version found by exhaustive search over feasible configurations of the RS-kernel and S-kernel (which we label CNNOpt) with both Nvidia's cuDNN[4] (v8.2) library (state-of-the-art CNN library) and optimized code from TVM/Ansor[18] (state-of-the-art auto-tuning framework). The input tensor data layout used for all experiments was NCHW.

The total number of configurations that remain in the design space after the analytical-modeling based pruning is shown in Table 3. We note that even the full space is only of the same order as the number of trials used with TVM/Ansor, but this can be very significantly pruned via use of a performance model, described in Section 4. We compare the best GEMM algorithm used by cuDNN, includ-

Table 2: Configurations of conv2d operators in DefoNet (Left), ResNet-18 (Middle) and Yolo-9000 (Right); K: # output channels; X,Y: output image width and height; C: #input channels; $F_{x,y}$ kernel size; batch size = 1; kernel stride = 1/2 (2 if marked with * after kernel name, 1 otherwise)

Layer	K	C	X,Y	$F_{x,y}$
D1	32	3	108	3
D2	32	32	108	3
D3	64	32	54	3
D4	64	64	54	3
D5	128	64	27	3
D6	128	128	27	3

Layer	K	C	X,Y	$F_{x,y}$
Y0	32	3	544	3
Y2	64	32	272	3
Y4	128	64	136	3
Y8	256	128	68	3
Y12	512	256	34	3
Y18	1024	512	17	3

Layer	K	C	X,Y	$F_{x,y}$
R1*	64	3	224	7
R2	64	64	56	3
R4*	128	64	56	3
R6	128	128	28	3
R7*	256	128	28	3
R9	256	256	14	3
R10*	512	512	14	3
R12	512	512	7	3

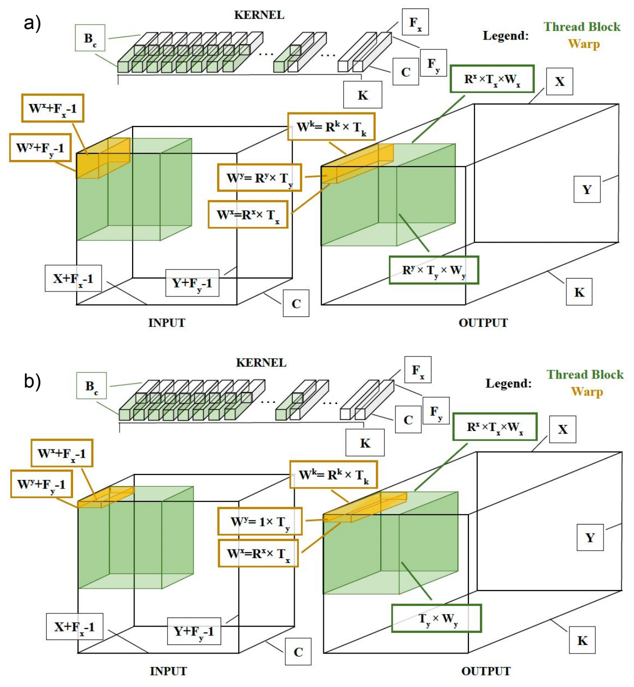


Figure 2: RS Kernel mapping in a), S Kernel mapping in b)

Table 3: Total Configurations in the Pruned Search Space

Layer	# Config.	Layer	# Config.
Defo1	842	yolo12	321
Defo2	2494	yolo18	52
Defo3	2300	res1*	423
Defo4	708	res2	2429
Defo5	708	res4*	1398
Defo6	708	res6	2161
yolo0	368	res7*	876
yolo2	321	res9	1017
yolo4	1387	res10*	150
yolo8	802	res12*	159

ing IMPLICIT_GEMM, GEMM, IMPLICIT_PRECOMP_GEMM [2].

For collecting TVM/Ansor's performance, we allowed the auto-scheduler to explore 2000 trials and used the best version found for each CNN benchmark.

The experiments were carried out on two Nvidia GPU systems: (i) Nvidia 2080 Ti GPU hosted on an AMD Ryzen Threadripper 3990X 64-Core CPU running Ubuntu 20.04, and (ii) Nvidia V100 GPU hosted on an Intel(R) Xeon(R) CPU E5-2680 v4 26 cores CPU running CentOS 7.8.2003. These machines represent different GPU generations (Turing and Volta) and use cases (non-enterprise and enterprise). The Nvidia 2080 Ti GPU has a peak performance of 14.2 Tflops for FP32 computation[10] and the V100 GPU has a 15.7 Tflops peak performance for FP32 computation[9]. We compiled our kernel, and the kernel generated by TVM/Ansor by using GCC 9.3 and NVCC 11.3 with flags "-O3 -arch=sm_xx -res-usage -lineinfo". The '-arch' parameter was set based on the machine architecture. We used sm_75 for the 2080Ti and sm_70 for the V100.

Table 4: Geometric mean of speed-up of CNNOpt over cuDNN and TVM/Ansor for conv2d layers in DefoNet, Resnet and Yolo

[V100]	DefoNet	ResNet	Yolo
vs cuDNN	1.89×	2.76×	1.08×
vs TVM/Ansor	1.41×	1.96×	1.48×
[2080Ti]	DefoNet	ResNet	Yolo
vs cuDNN	1.61×	2.32×	1.21×
vs TVM/Ansor	1.12×	1.42×	1.18×

For each benchmark, Nvidia Nsight(ncu)[11] was used for measurement of kernel execution time ncu –target-processes all –metrics gpu__time_duration –csv [executable]). Figure 3 compares the achieved speedup of the code generated by CNNOpt and cuDNN over the optimized code generated by TVM/Ansor. We also mark the absolute performance in teraflops (TFLOPS) for Ansor's best configuration as a reference (to improve readability, we do not also mark the absolute TFLOPs performance for CNNOpt and cuDNN in the figure). It may be seen that CNNOpt often achieves much higher performance than cuDNN and considerable speedup over TVM/Ansor on many of the benchmarks. The geometric means of speed-up across the benchmarks from the three networks are summarized in Table 4.

For completeness, we also include performance data for cuDNN using the Winograd algorithm for CNN instead of the standard algorithm. Table 5 lists the ratio of execution time of the CNNOpt

[2]cuDNN's Winograd implementation is faster in some cases, but we excluded those in the data presented here, so that all compared codes execute the same number of arithmetic operations. However, for completeness, performance data from cuDNN's Winograd implementation is also provided at the end of this section.

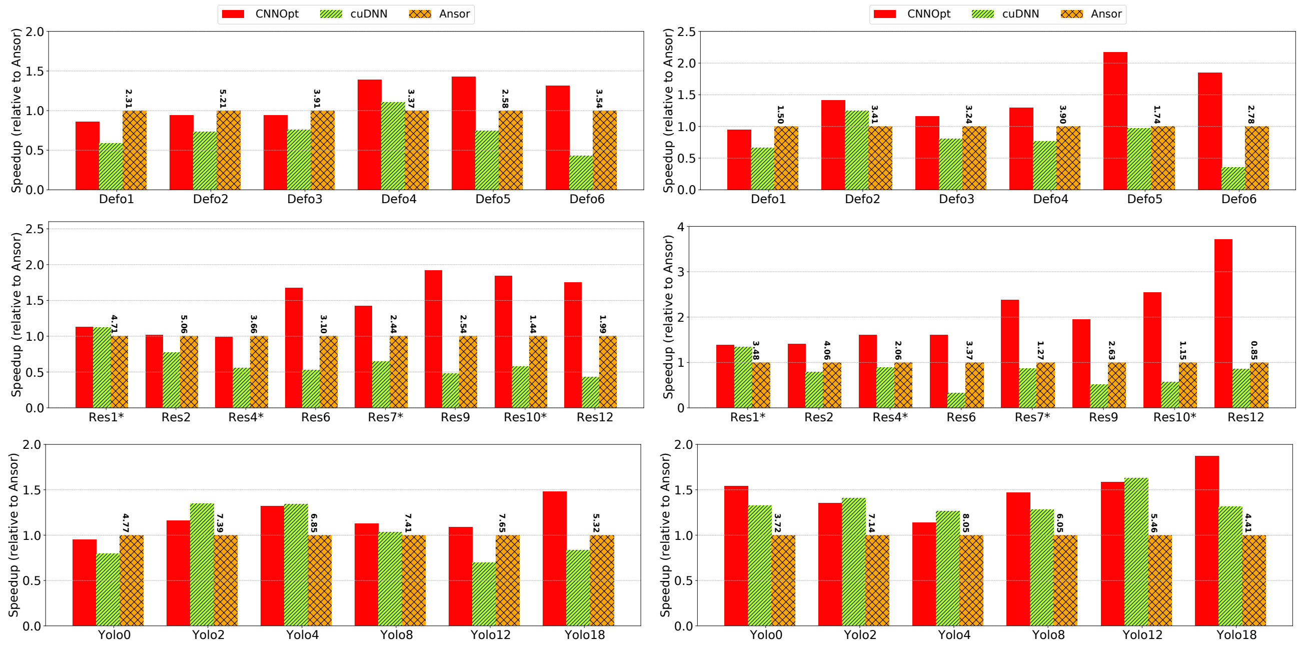


Figure 3: Speedup of CNNOpt (red) and cuDNN (green) relative to Ansor (brown), for DefoNet, Resnet-18 and Yolo-9000, on 2080Ti(Left) and V100 (Right); convolutions with Stride 2 are marked with *

kernel and cuDNN's winograd kernel, where applicable (i.e., excluding the stride-2 kernels where the Winograd algorithm is not appropriate). For some of the DNN layers, the cuDNN winograd algorithm achieves better performance than CNNOpt (values larger than 1). However, for the majority of stages, CNNOpt outperforms the Winograd algorithm. We note that our methodology can also be applied to optimize the Winograd algorithm, but it is beyond the scope of this paper.

Table 5: Relative performance of cuDNN winograd convolution against CNNOpt. CNNOpt is the performance baseline (1.0) in the table (the cuDNN Winograd algorithm does not support any stride 2 problem sizes).

Layer	2080	v100	Layer	2080	v100
Defo1	0.265	0.182	yolo12	1.237	1.321
Defo2	0.764	0.521	yolo18	0.659	0.721
Defo3	0.663	0.417	res1*	N/A	N/A
Defo4	1.509	1.359	res2	1.508	1.408
Defo5	0.617	0.382	res4*	N/A	N/A
Defo6	0.726	0.479	res6	0.689	0.469
yolo0	0.602	0.642	res7*	N/A	N/A
yolo2	1.635	1.624	res9	0.622	0.421
yolo4	1.698	1.736	res10*	N/A	N/A
yolo8	1.606	1.538	res12*	0.428	0.389

3.4 Profiling with Hardware Counters

In this section, we present experimental data collected from hardware counters, seeking to understand the factors contributing to improved performance of the best CNNOpt kernel over the best Ansor/TVM kernel for each benchmark. We used the Nvidia Nsight-Compute profiler to collect various metrics for all benchmarks on both target GPU platforms, for the CNNOpt kernel as well as the kernel code generated by Ansor (the best code versions in both cases). The profiled metrics included: L1 miss-count, L2 miss-count, global-memory transactions, shared-memory transactions and achieved occupancy. We evaluated the Pearson correlation coefficient for each of the measured metrics against the kernel execution time. As seen in Table 6, across both machines and both systems (CNNOpt and Ansor), there was a consistent and very strong positive correlation between execution time and the number of shared-memory transactions: 0.92, 0.863 for CNNOpt and 0.982, 0.978 for Ansor, on the 2080Ti and V100, respectively. Figure 4 plots the shared-memory transaction count versus kernel execution time across all the benchmarks, for CNNOpt and Ansor, for both devices. Our kernels generally incur fewer transactions between shared memory and registers, which appears to be the hardware metric most correlated with execution time. Table 7 lists the ratio of execution time of the CNNOpt kernel and Ansor kernel as well as the ratio of

Table 6: Pearson correlation coefficient between profiled metrics and execution time

Profiling Metric	CNNOpt		Ansor	
	2080	v100	2080	v100
SharedMem Transactions	**0.920**	**0.863**	**0.982**	**0.978**
GlobalMem Transactions	0.623	0.583	0.524	0.422
Achieved Occupancy	-0.185	0.396	0.174	0.369
L1 miss-count	0.640	0.543	0.714	0.162
L2 miss-count	0.480	0.323	0.366	0.387

Table 7: Ratio of execution times and shared-memory transaction counts between CNNOpt and Ansor

	2080Ti		V100	
Layer	TimeRatio	SM ratio	TimeRatio	SM ratio
R1*	1.06	1.08	0.92	0.41
R2	0.90	0.29	0.86	0.67
R4*	0.78	0.54	0.87	1.00
R6	0.70	0.29	0.73	0.56
R7*	0.68	0.29	0.64	0.31
R9	0.58	0.19	0.51	0.26
R10*	0.50	0.33	0.56	0.28
R12	0.69	0.46	0.70	0.29
Y0	1.04	0.86	1.14	1.07
Y2	0.93	0.52	1.04	0.88
Y4	0.98	0.40	0.80	0.94
Y8	0.87	0.68	1.05	0.86
Y12	0.84	0.93	0.94	1.69
Y18	0.66	0.16	0.73	0.38
D1	0.75	1.43	0.78	1.52
D2	0.91	0.84	0.99	0.94
D3	0.87	0.66	0.82	1.08
D4	0.78	0.39	0.87	0.54
D5	0.87	0.23	0.71	0.42
D6	0.85	0.81	0.65	0.76

measured shared memory transactions for the corresponding kernels. Except for some outliers, the CNNOpt kernel achieves a lower volume of shared-memory transactions than the kernel generated by Ansor.

4 PERFORMANCE MODELING FOR RAPID DESIGN SPACE EXPLORATION

In some usage scenarios, the time taken to generate the optimized code is a significant factor and it is desirable to reduce it from the many hours it takes to optimize all stages of a DNN pipeline using TVM/Ansor. In this section we develop a performance modeling approach for scenarios where the code optimization and code generation time are a concern. Our solution is to build an offline regression model that is trained with experimentally measured data for a number of CNN problem sizes and tiling configurations. During the online phase, where the actual CNN tensor sizes are known, the pre-trained performance model is rapidly queried to estimate the performance of each candidate configuration, which only takes a few seconds to execute in total. The *Top-n* candidates with the lowest predicted execution times are compiled and executed to determine the best candidate. This process is significantly faster than empirically assessing all candidate configurations, and as we show later in this section via experimental evaluation on all the benchmarks, the number of trials needed to achieve very high performance is very low.

4.1 Analytical Metrics in ML Performance Modeling

As mentioned in Section 3, the tile sizes affect several factors, such as data reuse, resource usage, concurrency, etc., and thus influence performance. A pure data movement-based analytical model is insufficient as it ignores interaction between many such factors, reducing its accuracy in predicting the best configurations. On the other hand, machine learning can capture the interaction between most factors that affect performance. However, a *black box* ML model based on standard features like the problem extents and tile sizes may not be able to capture aspects with complex relationships to input features such as data movement, occupancy, or the tail effect from waves. Hence, our approach relies on using an ML-Analytical hybrid model. The analytical model is used to estate data movement and occupancy, which are then included as additional features to train an ML model that captures the complex interaction between different features that affect performance.

Figure 5 provides a summary of the workflow for the performance modeling approach used with CNNOpt.

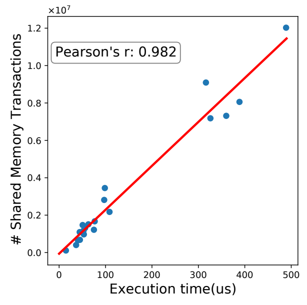
TVM/Ansor[18] uses an evolutional machine learning algorithm to generate code. The advantage of this approach is that it can find a reasonably good configuration since the search is done on a specific problem after evaluating some configurations. However, this approach has two main disadvantages (i) Portability – the model is built dynamically for a specific problem size and hence cannot be reused for other problem sizes in the future (ii) Code generation time – the internal auto-tuning/auto-scheduler has to empirically evaluate a large number of candidates (typically in 100's to 1000's) and hence is expensive. Hence, for a convolutional neural network with several distinct convolution layers, TVM/Ansor requires many hours to generate optimized kernel code.

Compared to Ansor, CNNOpt can perform a very fast prediction (in a few seconds) of the estimated execution time of all pruned configurations (Table 3) and can generate high-performance kernel code for the best predicted configurations in a short time. The left portion of the workflow in Figure 5 represents the offline training part (training set generation, calculating various input features, including analytical features, and training the model). The right part of the figure depicts the online inference for generating optimized kernel code for given CNN problem sizes. The details are explained below.

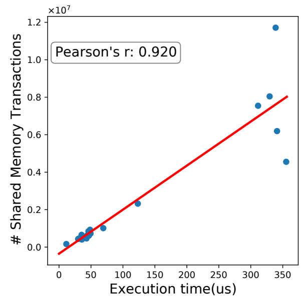
CNNOpt Model Construction: As mentioned earlier, our ML modeling approach augmented by analytical metrics can be summarized as two steps: **1)** Use the analytical model to estimate data movement at different memory levels and other factors such as occupancy and number of waves; **2)** Feed the input features (including analytical features) to a Random Forest Regressor in addition to the problem sizes and corresponding tile sizes. We choose Random Forest Regression method provided by the scikit-learn library[12] to build our offline model. The random forest methods only require a few samples for training and have relatively low training and prediction time.

Data Collection and Feature Selection:

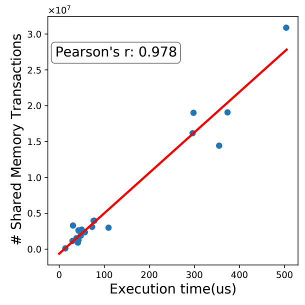
1) The problem sizes used for training the offline model were generated as follows: (i) X and Y were varied randomly in the range 3 to 1024 such that the maximum prime factor was smaller than or equal to 17; (ii) K was randomly selected in the range from 2^4 to 2^{10}; (iii) C was randomly selected from range 2^4 to 2^{10} and 3 that is a common input channel size for the first convolution layer in most of well-known CNN architectures; and (iv) Kernel/Filter shape is limited to 3, 5, and 7. These five values form the



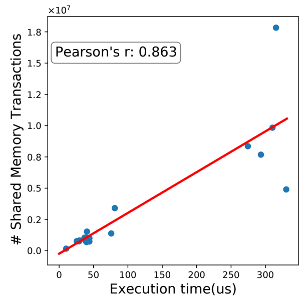
(a) Ansor, 2080Ti



(b) CNNOpt, 2080Ti



(c) Ansor, V100



(d) CNNOpt, V100

Figure 4: Correlation between Share-memory transactions and executiio time; 2080Ti (a and b) and V100 (c and d)

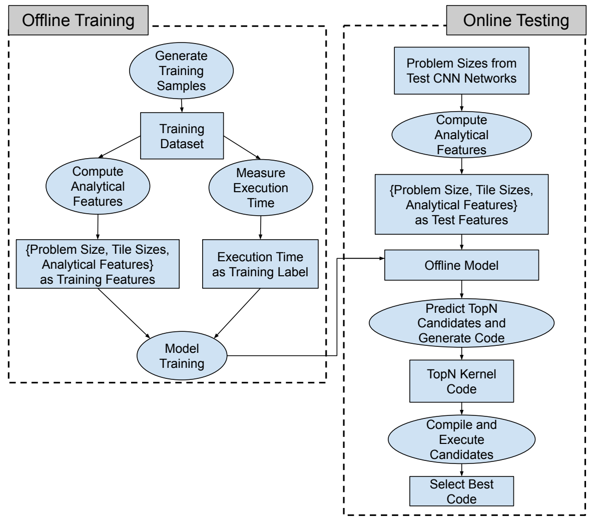

Offline Training
Generate Training Samples
Training Dataset
Compute Analytical Features
Measure Execution Time
{Problem Size, Tile Sizes, Analytical Features} as Training Features
Execution Time as Training Label
Model Training
Online Testing
Problem Sizes from Test CNN Networks
Compute Analytical Features
{Problem Size, Tile Sizes, Analytical Features} as Test Features
Offline Model
Predict TopN Candidates and Generate Code
TopN Kernel Code
Compile and Execute Candidates
Select Best Code

Figure 5: Workflow of CNNOpt. Rectangular nodes represent data and oval nodes processes.

first set of input features, called *{Problem Size}*, used for training. We developed two models separately for stride one and stride two kernels. **2)** The second group of training features are tile sizes in different tiling levels (described in the previous section 3). We consider all factors as possible tile sizes for each dimension, and we currently do not consider partial tile solutions. The tile-size selection search space consists of the cross-product of tile sizes corresponding to different dimensions. The tile-size combinations that do not obey the hardware resource capacity constraints are filtered out. We use *{Tile Sizes}* to denote this group. **3)** The last group of the training features is *{Analytical Features}* calculated by the analytical model. The analytical features we chose for training the hybrid model can be classified as follows: i) data movement-related: we estimate memory transactions at different memory levels(global-to-shared memory and shared-to-register); ii) concurrency related: we estimate occupancy based on resource usage of registers, shared memory and thread block shape; the number of concurrent thread blocks iii) SMs utilization related: We calculate the number of thread block waves to estimate the tail effect load imbalance. These three groups of features collectively form the training features, and the ground truth label $Time$ for each training sample is the actual execution time. We measure the running time on target GPUs for each training sample.

A sample in the training dataset can be described as:

$$(ProblemSize, TileSizes, AnalyticFeatures) \longrightarrow Time \qquad (4)$$

Evaluation Metric: We use Loss of Performance(LoP) as the metric to compare a model's performance. Let $Tpred$ represent the execution time of the best version selected by a model. Let $Tbest$ represent the actual best configuration from exhaustive exploration of all alternatives. LoP is computed as:

$$LoP = \frac{Tpred - Tbest}{Tbest} \qquad (5)$$

Model Prediction: The right part of Figure 5 depicts CNNOpt online performance prediction. Once the offline model is built, for any query, we enumerate all possible tile sizes that do not violate capacity constraints, calculate the corresponding analytic features to form a test dataset. CNNOpt compiles and executes the *Top-n* configurations and selects the version achieving th ebest performance (the lowest execution time).

4.2 Comparison with TVM/Ansor

We compare the effectiveness of CNNOpt in producing a fast GPU kernel against TVM/Ansor. Figure 6 shows the results for CNNOpt *Top-n* prediction and Ansor. We use Ansor's best performance as the baseline, highlighted as a horizontal black dashed line at a value of 1.0 in the figures. We report on the best performance from running Ansor with 100 trials and 500 trials and the best performance from executing the top-10 and top-30 from the model predictions. We also include CNNOpt-best, the best among all configurations (the CNNOpt performance reported in Sec. 3. It may be seen that for the vast majority of the cases, just 10 compile/executes (the top-10 data) with CNNOpt achieves better performance than the best result from 2000 trials from TVM/Ansor. In contrast, with a hundred trials of Ansor (red bars), the performance achieved often has very significant performance loss.

The Table 8 shows the number of trials that Ansor and CNNOpt require to attain 95% of the performance of the best kernel found with exhaustive search. We set the 95% performance(5% LOP) threshold over the highest performance shown in section 3.3 that both Ansor and CNNOpt achieve with exhaustive searching (2000 trials

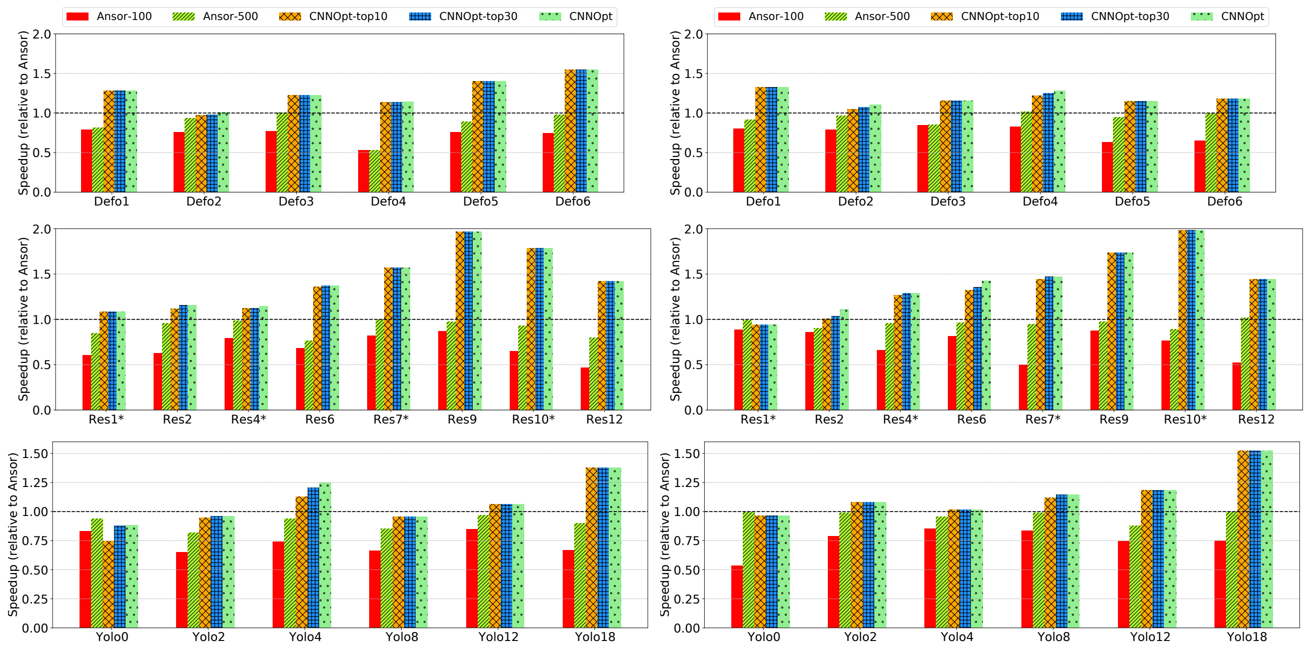


Figure 6: Speedup relative to Ansor-best (1.0), for Ansor with 100 and 500 trials, and CNNOpt Top-10/30/best; on 2080Ti (Left) and V100 (Right)

in case of Ansor). The smaller number of trials means shorter exploration time to find a fast kernel. Ansor takes on average over 600 trials to achieve a performance of 95% of its best kernel. In contrast, CNNOpt takes far fewer trials on both GPUs devices. In addition, we also compare CNNOpt with Ansor in the number of exploration trials to find the best configuration. Similar to the results for the 95% threshold, CNNOpt needs far fewer trials to produce the best kernel code in Table 8.

4.3 Effectiveness of Hybrid Modeling: Ablation Study

In this section, we perform an ablation study on the hybrid-ML modeling used by CNNOpt by asking whether a purely analytical modeling or a black-box ML model can achieve comparable effectiveness.

In assessing the relative effectiveness of different modeling approaches, we use loss-of-performance (LOP) in comparison to the best code version as the assessment criterion. We do so instead of metrics such as RMSE (Root Mean Square Error) because the model-predicted time is merely used to select a small subset of code versions for compiling/execution from a large design space. Thus, the ability to correctly discriminate between fast code versions and slow code versions is much more important than the closeness of the predicted times and the actual execution times of code versions. Further, even with respect to discriminatory ability, the accuracy of rank ordering among the fast code versions is much more important than the accuracy of rank ordering between slow versions, as long as slow code versions are consistently ranked worse than fast code versions.

Analytical Modeling: As explained in Section 3, data movement is a key factor that affects performance. Either the data movement between global memory and shared-memory or that between shared-memory and registers could be the bottleneck. We evaluated a pure analytical model seeking the configuration that minimizes the bandwidth-scaled data movement in the two levels of the memory hierarchy in GPUs. The data volume at each level is estimated using the analytical model described in Section 3, which is based on problem sizes and data reuse factors. As mentioned earlier, counting data movement in terms of individual elements will not reflect the actual memory traffic; hence we model the data movement in terms of the number of memory transactions. In our design, the output tensor has a very high degree of reuse. Hence the memory traffic from the output tensor is low and we ignore it in our analysis. The transferred data volume from global memory to shared memory for the input tensor is $ceil((W^x + F_x - 1) \times (W^y + F_y - 1)/32)$; for the kernel it is calculated as $ceil(W^k \times F_x \times F_y/32)$. The traffic from shared-memory to registers for the input tensor is calculated as $((R^y + F_y - 1) \times (R^x + F_x - 1)) \times numWarp \times numTB \times C$ and the kernel is calculated as $(F_x \times F_y \times R^k) \times numWarp \times numTB \times C$, where $numTB = (B_x \times B_y \times B_k)$, $numWarp = (W_x \times W_y \times W_k)$. Tile-sizes at different memory levels were chosen as factors of the problem size which did not violate capacity constraints. We rank-ordered all possible configuration according to the estimated data movement, and the $Top - n$ configurations are evaluated to select the best version.

ML Modeling: In order to develop the machine learning model, we generated a training dataset for the same set of problem sizes as described earlier for the hybrid model. The training dataset was thus identical to the dataset used in CNNOpt hybrid modeling except that analytic features were excluded. We ensured that the training dataset did not contain any problem size used in the final evaluation. We compared multiple ML models such as LinearRegression, DecisionTreeRegressor, RandomForestRegressor, AdaBoostRegressor, and GradientBoostingRegressor from scikit-learn[12] and we selected the RandomForestRegressor to build a black-box ML model since it achieved the best overall accuracy in the validation set. The trained ML model was then used for the test benchmarks from real CNN networks 2 by enumerating all possible tile sizes that did not violate capacity constraints to predict execution time. The top 30 candidates were selected for actual execution on the target GPU from these predictions, and the best one was chosen.

Experimental Results: Table 9 compares the performance of the three models where the $Top-n$ ($n = 30$) configurations are executed to find the best configuration. **1)** The LoP of the Analytical Model (defined in equation 5) is the highest among all methods in most test cases and shows the ineffectiveness of a pure Analytical Model, as dynamic run-time factors such as occupancy, thread idling, tail effects, and synchronizations are ignored. The complex interplay between these factors is difficult to capture using an analytical model. **2)** The performance of black-box machine learning method excels over pure Analytical Modeling and shows that the ML modeling can capture more complicated relationships between various factors and their impact on performance. However, in a few cases, the ML models have the worst LoP, reflecting a lack of robustness across the benchmarks. **3)** CNNOpt achieves the best prediction results and the LoP of $Top-30$ is less than 1% on average. Note that our approach is highly scalable and can be employed on very deep neural networks. The optimization of each layer is independent and thus does not depend on the depth of the network. In addition, many modern networks have multiple layers with the same problem sizes; hence the solution to a single layer can be reused across many layers.

5 RELATED WORK

TVM utilizes a scheduling language and includes a template-guided search framework AutoTVM[2]. FlexTensor[19] claims it optimizes tensor computation programs without human interference, allowing programmers to only work on high-level programming abstraction without considering the hardware platform details. But, FlexTensor limits itself to explore a very small schedule space. Ansor[18] is based on auto-scheduling, which avoids using predefined templates as AutoTVM, which also achieves better performance than AutoTVM and FlexTensor. CNN optimization on CPU by Li et al[8] achieves better performance than auto-tuning and library. No previous performance-model-driven approach to GPU CNN optimization has been demonstrated to achieve competitive performance to either manual library development or auto-tuning.

The polyhedral compilation model formulates the optimization of schedules for affine computations as an integer linear programming (ILP) problem. It finds an affine loop transformation that minimizes the data reuse distance between dependent statements. Tiramisu[1] and Tensor-Comprehensions[15] are two polyhedral compilers that also target the deep learning domain. Tiramisu provides a scheduling language, but relies on manual scheduling. Tensor Comprehensions can search for GPU code automatically, but it

Table 8: Number of trials to achieve 95% and 100% of best performance, for Ansor and CNNOpt, on 2080Ti and V100.

	95%				100%			
	Ansor		CNNOpt		Ansor		CNNOpt	
	v100	2080	v100	2080	v100	2080	v100	2080
R1*	130	1686	1	1	1025	1793	6	9
R2	833	513	22	1	2001	1922	115	10
R4*	385	199	8	1	1665	454	11	37
R6	897	1410	7	1	1153	1922	29	10
R7	706	130	5	1	1475	139	20	1
R9	513	203	1	1	897	1857	4	2
R10	150	1164	1	4	2004	1412	1	4
R13	321	1606	5	1	1345	1611	7	5
Y0	836	312	28	12	1217	1555	29	19
Y2	451	420	2	1	1409	1820	2	64
Y4	449	259	1	48	1985	1541	3	183
Y8	321	833	4	1	1601	1601	11	1
Y12	1089	641	4	1	1601	1994	4	1
Y18	202	653	3	3	1859	2002	8	3
D1	961	705	1	2	1474	2003	1	3
D2	1025	577	15	2	1025	1409	584	104
D3	1537	470	3	9	1601	1224	3	9
D4	328	1537	3	1	2002	1537	332	28
D5	641	330	1	6	641	1025	1	9
D6	328	281	1	4	1730	2005	2	9
avg	606	697	6	6	1486	1542	59	26

Table 9: LOP comparison: Black-box ML, Analytical, and Hybrid Modeling, on 2080Ti and V100

	ML		Analytical		CNNOpt	
	2080Ti	v100	2080Ti	v100	2080Ti	v100
R1*	0.1%	11.6%	0%	2.4%	0%	0%
R2	0.5%	10.5%	73.6%	81.5%	0%	7.8%
R4*	2.8%	10.3%	10.1%	27.5%	2.1%	0%
R6	0%	2.5%	105.5%	243.6%	0%	5.1%
R7	0%	5.3%	1.1%	1.8%	0%	0%
R9	0%	0%	94.3%	244.4%	0%	0%
R10	147.7%	2.8%	15.2%	47.2%	0%	0%
R12	0%	0%	2.1%	5.4%	0%	0%
Y0	3.5%	0%	18.2%	9.5%	0.2%	0%
Y2	0%	0%	19.4%	18.3%	0%	0%
Y4	0%	0%	13.0%	5.8%	3.7%	0%
Y8	0%	0%	0.5%	7.9%	0%	0%
Y12	0%	0%	0%	1.6%	0%	0%
Y18	0%	0%	0%	0%	0%	0%
D1	0%	0%	0%	0%	0%	0%
D2	1.7%	5.3%	18.5%	15.8%	3.2%	2.6%
D3	9.1%	3.9%	30.5%	31.9%	0%	0%
D4	7.3%	7.8%	51.1%	55.1%	0.6%	2.6%
D5	0%	0%	27.6%	50.0%	0%	0%
D6	0%	0%	18.8%	6.3%	0%	0%

cannot outperform AutoTVM/Ansor. This is because of the lack of certain optimizations and the inaccurate implicit cost model in the polyhedral formulation.

CUTLASS[7] is a collection of CUDA C++ template abstractions for implementing high-performance GEMM and related computations at all levels and scales within CUDA. CUTLASS does not include any tile-size optimization mechanism or a modeling approach. Our modeling approach can be extended for tensor cores as planned future work.

6 CONCLUSION

This paper has developed an effective domain-specific code optimization approach for synthesizing high-performance GPU implementations for the important CNN kernels that dominate execution time of deep learning pipelines for image analysis. The developed approach is applicable both to the off-line context where code synthesis time is not a concern, as well as scenarios where code optimization time is a constraint. Improved performance over both state-of-the-art cuDNN vendor library and the state-of-the-art TVM/Ansor auto-tuning framework were demonstrated. Significant reduction of code optimization time over TVM/Ansor was also demonstrated.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback that helped us improve the paper. The support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged. We are also very thankful for the use of the RI2 compute cluster in the CSE Department at the Ohio State University. Work at the University of Utah was supported in part by the National Science Foundation through awards 2018016 and 2112606.

REFERENCES

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[3] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. *Advances in Neural Information Processing Systems* 31 (2018), 3389–3400.

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *European conference on computer vision*. Springer, 630–645.

[7] Andrew Kerr. 2020. Nvidia CUTLASS CUDA templates for Linear Algebra. https://github.com/NVIDIA/cutlass.

[8] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. 2021. Analytical characterization and design space exploration for optimization of CNNs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 928–942.

[9] Nvidia. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[10] Nvidia. 2017. NVIDIA TURING GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[11] Nvidia. 2022. NVIDIA Nsight CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[13] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[14] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7263–7271.

[15] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[16] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[17] Zichen Zhang, Sami Khanal, Amy Raudenbush, Kelley Tilmon, and Christopher Stewart. 2022. Assessing the efficacy of machine learning techniques to characterize soybean defoliation from unmanned aerial vehicles. *Computers and Electronics in Agriculture* 193 (2022), 106682.

[18] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.

[19] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the scripts and data required to reproduce the experimental results in the PACT 2022 paper titled "Effective Performance Modeling and Domain-Specific Compiler Optimization of CNNs for GPUs". The git repository contains:

- The CNNOpt, cuDNN, and Ansor generated source code;
- The scripts to depict the performance chart in Fig. 3;
- The scripts to depict the correlation in Fig. 4;
- The scripts to illustrate efficiency and rapidity of analytical metrics in ML performance modeling in Table 8, 9.

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

- **Program:** cuda source code of CNNOpt and corresponding code generator.
- **Compilation:** Detailed instructions to compile different frameworks and scripts to run each framework is provided below. A copy of these instructions can also be found at the github repository.
- **Run-time environment:** GCC >= 8.5, CUDA 11.3.0, cuDNN v8.2.0, Conda, Linux platform such as Ubuntu or CentOS.
- **Hardware:** Nvidia 2080Ti or Nvidia V100.
- **Execution:** All scripts are explained in the READEME file and follow the workflow in section A.5.
- **Output:** The script reports performance chart in Fig. 3, the execution time and shared memory correlation in Fig. 4, and the LoP experiment in Table 8, 9
- **How much disk space required (approximately)?:** > 50 GB.
- **How much time is needed to prepare workflow (approximately)?:** Creating conda virtual environment and install dependency should be less than 5 mins.
- **How much time is needed to complete experiments (approximately)?:** Should be less than 1 hour.
- **Publicly available?:** Yes

A.3 Description

A.3.1 How Delivered . Our artifact is available on a public git repository:
https://github.com/HPCRL/AE-PACT

A.3.2 Hardware Dependencies . Nvidia 2080Ti or Nvidia V100.

A.3.3 Software Dependencies . Conda, CUDA 11.3.0, cuDNN v8.2.0, scikit-learn, numpy, pandas, six, matplotlib, Linux

A.4 Installation

Clone the repository (recursively):
https://github.com/HPCRL/AE-PACT
See the below file for instructions:
https://github.com/HPCRL/AE-PACT/blob/master/README

A.5 Experiment Workflow

Prepare conda environment:
```
$ conda create –name pactae python=3.8
$ conda activate pactae
$ conda install numpy pandas six matplotlib
$ pip3 install -U scikit-learn
```
For Performance Chart:
```
$ bash gen_profile_duration_log.sh
$ bash bash all_fig.sh
```
For Execution time and Shared Memory correlation :
```
$ bash sm-cor-2080.sh > sm-2080.log
$ bash sm-cor-v100.sh > sm-v100.log
$ python SM_correlation.py
```
For LoP experiments:
```
$ cd lop-v100/
$ bash run.sh
$ cd lop-2080/
$ bash run.sh
```

A.6 Evaluation and Expected Result

We expect the performance results to be close to those reported in the paper (Fig. 3). The results compare the achieved speedup of the code generated by CNNOpt and cuDNN over the optimized code generated by TVM/Ansor on Nvidia 2080Ti and V100 machine. After running scrips in Execution time and Shared Memory correlation, all result figures should be similar with Fig. 4. The LoP experiment takes around 20-30 min to produce LoP table and dominant time cost is to generate hybrid model and comparison pure ML model. The printout table shows LoP between Analytical Modeling, ML Modeling, and CNNOpt Hybrid modeling.