Analytical Characterization and Design Space Exploration for Optimization of CNNs Extended Abstract

Rui Li[1], Yufan Xu[1], Aravind Sukumaran-Rajam[2], Atanas Rountev[3], and P. Sadayappan[1]
[1]University of Utah, [2]Washington State University, [3]Ohio State University

1. Motivation

Convolutional Neural Networks (CNNs) have had transformative impact. Since they are computationally very demanding, there have been numerous efforts towards optimized implementation of CNNs [8, 11, 7, 2]. The cost of data movement dominates the cost of floating-point arithmetic computations on all current hardware platforms. Hence loop tiling is a crucial transformation for optimizing CNNs. The CNN computation can be expressed as a 7-dimensional loop nest. Multi-level tiling is required for limiting data movement in a memory hierarchy with multiple levels of caches. Tiling creates one additional tile-loop per level of tiling, for each tiled iterator. The total design space corresponds to all possible permutations of tile-loops at each level, for all possible combinations of tile sizes. This space is explosively large and therefore all prior efforts at optimizing CNNs have only been able to explore a limited subset of the full design space.

2. Limitations of the State of the Art

The current state of the art in optimizing CNNs include (1) vendor libraries, such as Nvidia's cuDNN [3] for GPUs and Intel's oneDNN [9] for multicore CPUs, and (2) optimizing code generation frameworks such as TVM [2]. Libraries for CNN implement a small number of code variants and use models/heuristics to select tile sizes and code variants at run time. TVM uses empirical auto-tuning to measure the performance of candidate code versions and also employs a machine learning model to guide the auto-tuning process. With current CNN libraries, only a limited number of tiled code variants are implemented, using insights from developers to determine the tile loop structure. With auto-tuning-based code generators, many more code versions are evaluated during the tuning process, but the space must still be structured by experts with good insights because the full search space is much too large to search effectively in a reasonable amount of time.

3. Key Insights

In this paper, we solve the design-space exploration problem for optimizing multi-level tiled CNN code in a principled and comprehensive way. To achieve this, we develop the first approach that analytically models the data movement for any CNN stage in a multi-level memory hierarchy. Using this model, we show how to explore *the entire search space*, looking for the configuration that minimizes the bandwidth-scaled data movement in the limiting level of the memory hierarchy. The key insight of the presented approach, which differentiates it from previous CNN optimization efforts, is that analytical modeling and reasoning enables dramatic pruning of the extremely large space of loop permutations and tile sizes, reducing it to a small number of constrained non-linear optimization problems that can be solved by off-the shelf solvers.

	Auto tuning	**Micro Kernel**	**Design Space Exploration**
oneDNN	×	Highly optimized	Minimal
TVM	✓	NA	Limited
MOpt	×	Not highly optimized	Comprehensive

Table 1: Strengths/limitations of oneDNN, TVM, and MOpt

Figure 1 shows the components of the *MOpt* system (*M*odeling-based *Opt*imizer) for generating optimized CNN code for multicore processors, based on a new comprehensive design-space exploration approach for tile-loop optimization. The leftmost component represents a conceptual methodology for pruning the space of possible permutations of tile-loops for single-level tiling, using analytical modeling of data movement volume to identify a very small subset—containing only 8 elements—of the full space of tile-loop permutations, guaranteed to contain an optimal configuration that minimizes data volume for tiled execution. The right portion of the figure shows the tool components for code generation for a specific CNN. For the set of pruned tile-loop permutations, constrained non-linear optimization problems are automatically generated and solved using an off-the-shelf solver (AMPL [4] with Ipopt [12]) to produce optimal tile sizes $T_{i,j}$ and data movement costs C_i (here j ranges over the levels of the memory hierarchy). The best solution gives the tile sizes and tile-loop permutation to be used to generate customized C code for the CNN stage, with tile loops surrounding a CNN microkernel that implements register-tiling using vector intrinsics.

Table 1 contrasts the strengths and limitations of CNN libraries like Intel's oneDNN, state-of-the-art auto-tuning code-generator TVM, and MOpt. oneDNN is a vendor library that includes highly optimized microkernels developed and optimized by Intel engineers over many years. However, it dynamically chooses among a small number of pre-determined tiled code structures based on the CNN array sizes provided at invocation, i.e., it performs minimal design-space exploration. TVM performs a search through a limited design space, as specified by the tuning script.

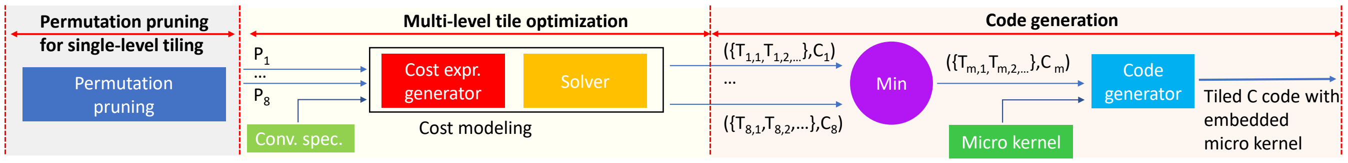
Permutation pruning for single-level tiling
Permutation pruning
Multi-level tile optimization
P_1
...
P_8
Cost expr. generator
Solver
Cost modeling
Conv. spec.
Code generation
$(\{T_{1,1},T_{1,2},\ldots\},C_1)$
...
$(\{T_{8,1},T_{8,2},\ldots\},C_8)$
Min
$(\{T_{m,1},T_{m,2},\ldots\},C_m)$
Code generator
Micro kernel
Tiled C code with embedded micro kernel

Figure 1: MOpt Overview

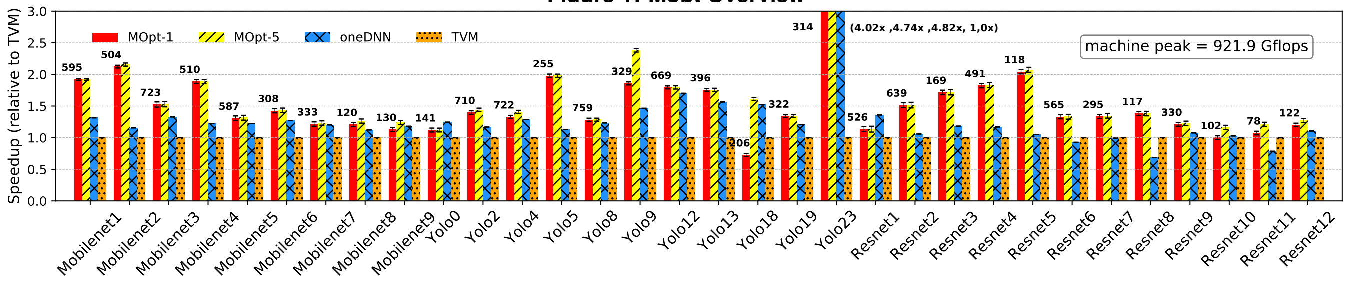

Figure 2: Performance (relative to TVM) and variance for Mobilenet, Yolo-9000, and Resnet-18 on i7-9700K

MOpt's strength is comprehensive design-space exploration to seek tile-loop structures and tile sizes that minimize the data volume at the bottleneck resource in the multi-level cache hierarchy. It does not use any empirical auto-tuning in its search and uses a micro-kernel that is not as highly optimized as oneDNN's. Nevertheless, the achieved performance of MOpt's code on the CNN stages of three DNN pipelines is almost always better and often much better than TVM's code, and comparable and sometimes much better than oneDNN.

4. Experimental Results

We compare the performance of code generated by MOpt with two state-of-the-art frameworks: (i) Intel oneDNN (v1.5) library, and (ii) TVM (v0.6). All MOpt codes and oneDNN were compiled using the Intel ICC 2019 compiler with flags "-O3 -march=native -qopenmp". TVM recommends using the LLVM framework; hence we used LLVM-8. TVM tuning was based on the built-in template: "generic.schedule_conv2d_nchw" [1]. We used XGBTuner as the ML tuning model, and we set "LLVM -mcpu=core-avx2 to vectorize the code. For each CNN benchmark, we ran TVM's auto-tuner with its internal ML model to find the best configuration in 1000 trials. The experiments were carried out on an 8-core Intel Core i7-9700K CoffeeLake processor, with 32KB L1 cache per core, 256KB L2 cache per core, and a shared 12MB L3 cache.

We evaluated the performance of MOpt generated code on all CNN benchmarks used by TVM in their extensive comparative evaluation [2] against various other CNN optimization frameworks. The benchmarks used by TVM include all 12 conv2d operators from Resnet-18 [5] and the 9 depthwise conv2d operators from MobileNet [6]. In addition we used all 11 conv2d operators from Yolo-9000 [10]. The bar charts and the left vertical axes in Figure 2 show performance normalized to performance of TVM-optimized code. We show performance of two MOpt code versions (i) MOpt-1: The code version generated with the configuration with minimum modeled cost, and (ii) MOpt-5: The best among the top five code versions based on model prediction. The inclusion of MOpt-5 highlights the potential for performance improvement by use of limited empirical auto-tuning with MOpt. Since the modeling in MOpt is based on an idealized fully associative cache, occasionally we find (e.g., Yolo9 and Yolo18) that conflict misses cause a significant drop in performance; but when the top five configurations are considered, the best among the top-5 always performed very well. Geometric means of speed-up over oneDNN are 1.16× on Yolo, 1.37× on ResNet, and 1.24× on MobileNet. Geometric means of speed-up over TVM are 1.73× on Yolo, 1.40× on ResNet, and 1.52× on MobileNet.

5. Key Contributions

The main contributions of this work are as follows.

1) It presents the first comprehensive analytical modeling for data movement volume for multi-level tiled CNN execution on a system with a multi-level memory hierarchy, covering the full space of permutations and tile sizes.

2) It is the first analysis that exploits algebraic properties of the non-linear analytical expressions for data-movement volume to dramatically prune the number of distinct cases from thousands to only eight in order to find the global optimum in the entire space of tile-loop permutations for a single-level tiled CNN. The factor of reduction in the search space that is enabled by this algebraic analysis is exponentially higher for multi-level tile-size optimization.

3) The utility of the proposed new analytical modeling and optimization is demonstrated via a custom code generator for high-performance multicode CPU code for CNNs. The evaluation considers all CNN stages from MobileNet, ResNet-18, and Yolo9000. The achieved performance is comparable to or better than both the state-of-the-art oneDNN library from Intel, and the state-of-the-art TVM framework for auto-tuned code generation.

References

[1] TVM CNN tuning script. `https://github.com/apache/incubator-tvm/blob/v0.6/topi/python/topi/x86/conv2d.py`.

[2] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[3] 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.

[4] R. Fourer, D. M. Gay, and B. W. Kernighan. AMPL. A Modeling Language for Mathematical Programming. 2003.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

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

[7] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *SC'16: Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 633–644, 2016.

[8] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

[9] oneDNN: oneAPI Deep Neural Network Library. `https://01.org/onednn`.

[10] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017.

[11] Y. M. Tsai, P. Luszczek, J. Kurzak, and J. Dongarra. Performance-portable autotuning of OpenCL kernels for convolutional layers of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 9–18. IEEE, 2016.

[12] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.