Compiler-Level Matrix Multiplication Optimization for Deep Learning

Anonymous Authors[1]

Abstract

As an important linear algebra routine, GEneral Matrix Multiplication (GEMM) is considered a fundamental operator in deep learning. Compilers need to translate these routines into low-level code optimized for specific hardware. Compiler-level optimization of GEMM has significant performance impact on training and executing deep learning models. However, current deep learning frameworks rely on hardware-specific operator libraries in which GEMM optimization has been mostly achieved by manual tuning, restricting the performance of deep learning frameworks on different target hardware. In this paper, we propose two novel algorithms for GEMM optimization based on the TVM framework. The Greedy Best First Search (G-BFS) guided method is a lightweight method based on search. The Neighborhood Actor Advantage Critic (N-A2C) method is based on reinforcement learning. Experimental results show significant performance improvement of both approaches over state-of-the-art optimization methods, in terms of computation time and the fraction of space explored. The proposed approaches have potential to be applied for other operator-level optimization.

1. Introduction

In recent years, deep learning has been attracting increasing attention from both academia and industry (LeCun et al., 2015). With its own advances in algorithmic and architectural design, significant improvement in computational hardware (e.g. GPU and TPU), and availability of enormous amount of labeled data, deep learning has shown success in many domains, such as computer vision, natural language processing, speech recognition, health care, finance, etc. There is no doubt that deep learning solutions will be adopted in even more areas in the upcoming years.

However, computing deep learning models (both training and inference) efficiently on various hardware is a difficult task, which involves end-to-end compiler optimization at several levels from computational graphs to operators, and down to executable code on target hardware (Chen et al., 2018a). A computational graph is a global view of operators and data flow among them. Within a graph, operators specify various computation that is required for individual operations on tensors. Current optimization techniques at the computational graph level are hardware agnostic and independent from the implementation of operators within a graph. For example, a standard procedure, operator fusion, combines multiple operators into a single kernel, avoiding latency introduced by write-back and loading of intermediate results into memory. At the operator level, existing optimization is mostly limited to specific implementation for a framework (e.g. TensorFlow XLA) or proprietary library for a particular target device (e.g. Nvidia cuDNN). Such libraries have been built mostly based on expert knowledge and manual tuning to perform efficiently and effectively.

Deep learning models are expected to run on diverse hardware platforms (CPU, GPU, FPGA, ASIC, SoC, etc.) with very different characteristics that can be leveraged to optimize various deep learning operations. To efficiently map deep learning operation/workload to broader range of hardware, TVM has been proposed (Chen et al., 2018a) as a general compiler framework for automated tensor operator optimization. In this framework, a configuration space can be defined for each operator. Optimization of an operator is to find a configuration that can optimize a performance metric (e.g., the lowest running time). Configuration is the detailed specification of how an operator is computed and executed on target hardware. For example, in matrix multiplication, tiling is required to make computation more efficient, but various tiling strategies may generate configurations that have significantly different performance. Configuration space for individual operators may have different structures and properties.

In this paper, we aim at more efficient operator optimization. We focus on the GEneral Matrix Multiplication (GEMM) operator which is the fundamental operator in deep learning. GEMM computes the product (multiplication) of two ma-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

trices and the operator can be translated into nested loops controlled by a fixed set of few parameters. Its optimization can be reduced to finding the optimal combination of parameter values in this set. We analyze structure of its configuration space and design improved tuning approaches for GEMM optimization. The contributions of the paper are three-fold:

- We consider the relation between different configurations and define the neighboring relation between two configurations. We employ a Markov Decision Process (MDP) for exploration over the configuration space.
- Based on the neighboring relations within the configuration space, we propose a Greedy Best-First-Search (G-BFS) guided method and a Neighborhood Actor Advantage Critic (N-A2C) method to search for an optimal configuration given two matrices to be multiplied.
- We evaluate the performance of our proposed methods in TVM framework for Nvidia Titan XP GPU. We compared them with state-of-the-art GEMM tuners using XGboost (Chen & Guestrin, 2016) and RNN controller (Chen et al., 2018a). We demonstrate that both our methods can discover high-performance configurations efficiently with smaller fraction of explored configuration space and shorter time to complete. By exploring only 0.1% of the configuration space, our methods discover configurations of **24%** less computing cost than what the XGBoost method can find and configurations of **40%** less computing cost than what the RNN method can find, for multiplying two 1024×1024 matrices.

1.1. Related Work

As there are multiple AI frameworks and a wide range of hardware involved in deep learning applications nowadays, it is important but challenging for compiler-level optimization to efficiently and flexibly harmonize AI algorithms with the underlying hardware and optimize the performance of various deep learning applications. Much work has explored this space and achieve good performance improvement. CuDNN (Chetlur et al., 2014) provides highly efficient implementations of various deep neural network layers and it is considered the standard for accelerating deep learning on Nvidia GPUs. NNPACK (Dukhan, 2016) and PCL-DNN (Das et al., 2016) similarly provide accelerations in x86 processors. Latte (Truong et al., 2016) provides a natural abstraction for specifying new layers and applies domain-specific and general computation graph optimization. XLA (Accelerated Linear Algebra) (Leary & Wang, 2017) optimizes TensorFlow computations in terms of speed, memory usage, and portability via just-in-time (JIT) compilation or ahead-of-time (AOT) compilation.

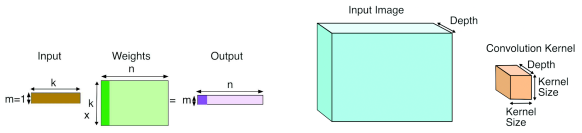
However, the aforementioned approaches perform either graph-level or operator-level optimization during compilation and they are not generic enough to accommodate all AI frameworks and hardware. Based on the structure of Halide (Ragan-Kelley et al., 2013), (Chen et al., 2018a) proposes a general end-to-end compilation optimization framework combining Neural Network Virtual Machine (NNVM) (NNVM, 2017) for computation graphs and Tensor Virtual Machine (TVM) (Chen et al., 2018a) for tensor operators. Currently in TVM, a tuning method based on XGBoost (Chen & Guestrin, 2016) is considered the state-of-the-art method for GEMM configuration optimization and has shown superior performance over other methods (Chen et al., 2018a).

For configuration optimization, the intuitive method is grid search, where all possible configuration candidates are tested sequentially so as to find the configuration with best performance. It guarantees to find the global optimal configurations, but the number of tested configuration candidates grows exponentially with the dimensions of configuration space (Bellman, 1961), so its usage is limited in problems with small search space or in combination with manual search (Hinton, 2012; LeCun et al., 2012; Larochelle et al., 2007). Random search is proposed where the configurations are randomly selected to be tested, and is shown empirically and theoretically to be more efficient than grid search for configuration tuning (Bergstra & Bengio, 2012).

As an instance of Bayesian optimization, sequential model-based optimization (SMBO) shows its strength in configuration tuning by iteratively updating the underlying expected improvement, exploring new data through an acquisition function, and training a regression model (Hutter et al., 2011; Bergstra et al., 2011; Hoffman & Shahriari, 2014). The general method has been widely adopted and implemented (Kandasamy et al., 2018; Snoek et al., 2012).

From another perspective, a series of evolutionary approaches have been explored, including the broad class of genetic algorithms (GA) (Holland, 1975; Goldberg, 1989), differential evolution (Storn & Price, 1997), estimation of distribution algorithms (Larrañaga & Lozano, 2001; Bosman et al., 2007) and particle swarm optimization (Kennedy et al., 2001). Evolutionary strategies (ES) (Rechenberg & Eigen, 1973; Schwefel, 1977) have shown to perform efficiently in configuration tuning. Based on the concept, Covariance Matrix Adaptation Evolution Strategy (CMA-ES), first proposed by (Hansen & Ostermeier, 2001), has shown excellent performance by smartly update the mean, step size and covariance matrix for each evolution (Loshchilov & Hutter, 2016) . Natural Evolution Strategies (Wierstra et al., 2014) applies natural gradients to update configuration search policy so as to achieve high expected fitness and discover the high-performance configuration.

Recently, researchers at Google apply deep reinforcement



(a) Fully-Connected Layer (b) Convolutional Layer

Figure 1. Illustration of deep neural network layers

learning with a RNN controller to optimize the configurations of neural network architectures and its components (Bello et al., 2017; Mirhoseini et al., 2017; Ramachandran et al., 2018; Zoph & Le, 2016; Pham et al., 2018). The excellent tuning performance in wide range of applications shows the potential of deep reinforcement learning in the configuration tuning area.

2. Problem Statement

2.1. Matrix Multiplication

Matrix multiplication is a critical operation in many machine learning algorithms, particularly in the domain of deep learning. Training parameters (weights) of a deep neural network in a vectorized fashion essentially involves multiplication of matrices with various sizes.

Fully-Connected (FC) layers (Fig. 1a) and convolutional (Conv) layers (Fig. 1b) are building blocks of feed-forward and convolutional neural networks (Warden, 2015). It is straightforward to identify matrix multiplication in computing output value of a FC layer: each input has k elements, and FC layer has n neurons each with k weights. An FC layer is the multiplication of a $m \times k$ matrix (m is sample size) and a $k \times n$ matrix. A Conv layer appears to be a specialized operation, but it can be computed with matrix multiplication after rearranging data in a matrix format: each depth-wise (channel) slice of input can be added into an input matrix as a row; similarly each kernel can be added into a kernel matrix as a column. Convolution operation becomes multiplication of those two matrices. When using AlexNet on image classification with ImageNet dataset, vast majority of computation time on forward pass (94.7% on GPU, and 88.7% on CPU) is consumed by Conv and FC layers (Jia, 2014).

2.2. GEMM and Matrix Tiling

GEMM is a general procedure ubiquitously used in linear algebra, machine learning, statistics, and many other areas and is implemented in the BLAS (Basic Linear Algebra Subprograms) library (BLA, 2002). It multiplies two input matrices to produce an output matrix. The key difference between GEMM in deep learning and regular matrix multiplication is that the input matrices handled in deep learning are normally much larger. For example, a single layer in a typical convolution neural network may require multiplication of a 256×1024 matrix by a 1024×128 matrix to produce a 256×128 matrix. Regular three-for-loop (Fig. 2) computation requires 34 million ($256 \times 1024 \times 128$) floating point operations (FLOPs). Modern deep neural networks may have hundreds of convolutional layers (e.g. ResNet152 (He et al., 2015)), and such networks may need several billions of FLOPs to finish operations in all layers for an input image.

```
for i in range(m):
    for j in range(n):
        C[i][j] = 0
        for l in range(k):
            C[i][j] += A[i][l] × B[l][j]
```

Figure 2. Computing matrix multiplication

High cache hit rate of memory access is critical for complex numerical computation, such as GEMM. The large sizes of matrices usually forbid the entire matrices being loaded into memory or cache, however, GEMM can optimize memory access by iteratively splitting computation into smaller tiles, often referred to as the *tiling process*. A resulted matrix is initialized with zeros. GEMM uses outer products to compute part of a tile of the result and accumulates it on top of what has been stored in that tile. A tile is loaded from memory into cache and accumulates a new result on top of that. Fig. 3 (Matthes et al., 2017) illustrates a tiling strategy of GEMM.

Original memory access patterns need to be transformed to adapt to the cache policy of a particular hardware. It is not straightforward to decide an optimal tiling strategy because it requires accurate estimate of accessed array regions in loops to match with cache size of target hardware and meet other constraints. Optimal tiling chooses a tile size for each loop to collectively achieve lowest running time on target hardware.

2.3. Problem Formulation

We use TVM (Chen et al., 2018a) to investigate the performance of matrix tiling for GEMM. TVM facilitates tiling optimization by generating Intermediate Representation (IR) of a particular configuration. Fig. 4 is a simple example IR of GEMM tiling configuration with a blocking factor of 32 on x86 CPU for GEMM with $(m = 1024, k = 1024, n = 1024)$ (short as $(1024, 1024, 1024)$).

Definition: Generally, a GEMM tiling configuration can be defined as

$$\overrightarrow{\xi} = \overrightarrow{\xi_m} \times \overrightarrow{\xi_k} \times \overrightarrow{\xi_n}, \quad (1)$$

where

$$\overrightarrow{\xi_m} = \{[m_0, \ldots, m_i, \ldots m_{d_m-1}] \,|\Pi_{i=0}^{d_m-1} m_i = m\}, \quad (2)$$

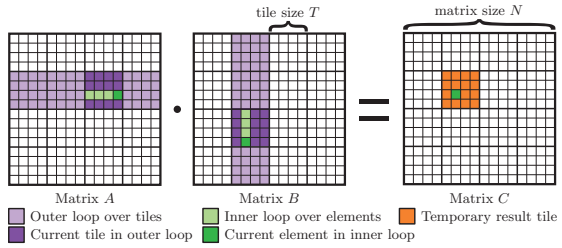


Figure 3. An Example of Tiling Strategy

```
produce C {
 for (x.outer, 0, 32) {
   for (y.outer, 0, 32) {
     for (x.inner.init, 0, 32) {
       for (y.inner.init, 0, 32) {
         C[((((x.outer*1024) + y.outer) + (x.inner.init*32))*32)
           + y.inner.init)] = 0.000000f }}
     for (k.outer, 0, 256) {
       for (k.inner, 0, 4) {
         for (x.inner, 0, 32) {
           for (y.inner, 0, 32) {
             C[((((x.outer*1024) + y.outer) + (x.inner*32))*32) + y.inner)]
             += A[((((x.outer*8192) + k.outer)*4) + k.inner) + (x.inner*1024))]
              * B[((((y.outer + (k.outer*128)) + (k.inner*32))*32) + y.inner)] }}}}}}}
```

Figure 4. IR of GEMM with a blocking factor of 32

$$\overrightarrow{\xi_k} = \{[k_0, \ldots, k_l, \ldots k_{d_k-1}] \,|\Pi_{l=0}^{d_k-1} k_l = k\}, \quad (3)$$

$$\overrightarrow{\xi_n} = \{[n_0, \ldots, n_j, \ldots n_{d_n-1}] \,|\Pi_{j=0}^{d_n-1} n_j = n\}. \quad (4)$$

Multiplication of two matrices $A(m \times k)$ and $B(k \times n)$ produces matrix $C(m \times n)$. d_m, d_k and d_n are the number of nested loops for each dimension m, k and n, respectively. m_i, k_l, n_j, $\forall i \in [0, d_m)$ $\forall l \in [0, d_k)$ $\forall j \in [0, d_n)$, are the number of iterations of a respective loop. The configuration in Fig. 4 is $m_0 = m_1 = 32$, $k_0 = 256, k_1 = 4$, $n_0 = n_1 = 32$, and $d_m = d_k = d_n = 2$.

Constrained by the definition of tiling configuration, we can formulate the following optimization problem:

$$\min_s T_{cost}(s; m, k, n, d_m, d_k, d_n).$$

The objective function of this problem is to find an optimal tiling configuration that has minimal running time on target hardware. T_{cost} denotes the running time for the configuration s, given the dimension of matrices as m, k, n and the number of the nested loops on each dimension as d_m, d_k, d_n.

3. Methodology

3.1. The XGBoost Tuner

In TVM framework, (Chen et al., 2018b) proposed a configuration tuning method for operator-level optimization in which the search is guided by a performance prediction model trained with eXtreme Gradient Boosting (XGBoost). This XGBoost guided tuner (or XGBoost tuner) follows an iterative search process. In each iteration, a large number of configuration candidates are derived from configuration space by random walk. According to the predicted performance from a trained XGBoost model, the best candidates are selected and tested on hardware. The performance feedback from the hardware is collected and applied to further train the XGBoost model so as to improve its prediction accuracy.

The XGBoost tuner outperforms the other classic tuners including genetic algorithm search, random search, etc., for GEMM. Nevertheless, training the XGBoost model for a large configuration space would incur a high cost. In order to further improve the operator-level configuration tuning performance, we propose a new configuration search model which allows exploitation of relations between similar configurations, followed by two efficient tuning methods.

3.2. Configuration Search Modeling

For better analysis, we model the configuration tuning problem as a Markov Decision Process (MDP), where each configuration can be regarded as a unique state. We define the state as follows.

$$s = [s_m, s_k, s_n, J], \quad (5)$$

where $s_m = [m_0, m_1, \ldots, m_{d_m-1}] \in \xi_m$, $s_k = [k_0, k_1, \ldots, k_{d_k-1}] \in \xi_k$, $s_n = [n_0, n_1, \ldots, n_{d_n-1}] \in \xi_n$, and J is the binary number indicating whether the state is legitimate (e.g. the product of m_i's must be m, the numbers must be integers, etc.).

As in the GEMM application, with similar configuration settings, i.e., the configuration parameters for each dimension of two states are equal or close, the performance of this two states may not exists large difference. Taking advantage of the relations among similar configurations, and considering the constraints of the matrices size in each configuration. We define the concept of action space as follows.

$$\mathcal{A} = \left[\begin{array}{l} s_x[i] = 2s_x[i] \text{ and } s_x[j] = s_x[j]/2, \\ \\ \text{where } \forall x \in \{m, k, n\}, \forall i, j \in [0, d_x) \text{ and } i \neq j \end{array} \right]. \quad (6)$$

Accordingly, we define a step function $step$, i.e.,

$$s' = step(s, a). \quad (7)$$

With the input of any action $a \in \mathcal{A}$, the current state s is transferred to state s'. We define s and s' are neighbor states. Based on the setting of action space, we guarantee the Euclidean distance between the neighbor state s and s' is the minimum value compared with the distance between the state s and all the other states.

Moreover, in order to better evaluate the performance of each action based on different states, if the agent takes action a and goes from the state s to the state s', we define the rewards as follows

$$r(s,a) = \frac{1}{T_{cost}(s'; m, k, n, d_m, d_k, d_n)}. \quad (8)$$

Following the above modeling, the agent is expected to determine its policy $\boldsymbol{\pi}$ so as to efficiently approach and discover the state s^* with the lowest running time in hardware system. In the following subsections, we will analyze two different configuration tuning approaches guided by G-BFS and N-A2C reinforcement learning, followed by a discussion for their strengths with different scenarios.

3.3. G-BFS Method

The G-BFS method is guided by Greedy Best-First-Search and follows the flowchart in Fig. 5(a). We initialize an empty priority queue $\mathcal{Q}$ (ordered by increasing cost), an empty list S_v to record all visited configuration candidates, and a random or hand-crafted starting state s_0. We first test and enque the starting state s_0 and record its running time $T_{cost}(s_0)$ into the priority queue Q. Based on the configuration search model, for each iteration, we deque the top configuration candidate s from the priority queue Q. We iterate through all actions $a \in \mathcal{A}$ and collect all corresponding neighbor states as

$$g(s) = [s' = step(s,a) \ \ \forall a \in \mathcal{A}]. \quad (9)$$

We randomly select ρ ($\rho \in \{1, 2, \ldots, len(g(s))\}$) configuration candidates from $g(s)$, and test them in hardware. For each state s' sampled from $g(s)$, if state s' is legitimate and has not been visited before, we enque state s' and its tested running time $T_{cost}(s')$ into Q and add state s' in the visited list S_v. If its tested running time $T_{cost}(s')$ is smaller than the current minimum running time, we set state s' as the optimal state visited and record its running time as t^{min}_{cost}. The iteration continues until the priority queue is empty or the computation time reaches the maximum time specified by the user. The current stored optimal state $s*$ and its running time t^{min}_{cost} are returned as tuning results. The summary of the algorithm is shown in Algorithm 1.

In Fig. 5(b), we show the exploration situation in the middle of the tuning algorithm when $\rho = len(g(s))$, where the red nodes denote the state currently stored in the priority queue and the grey nodes are all the visited states. In future iterations, the method will explore from the current most promising red nodes expands its visited state areas. In Fig. 5(c), we take an example of 2-dimensional configuration search on the randomly generated rewards function. We discover that the proposed G-BFS method is able to correct itself from exploring wrong directions and efficiently expand its neighborhood to the optimal states. Moreover, when the value of $\rho = len(g(s))$, given unlimited tuning time, the algorithm is guaranteed to visit all the configuration states.

Algorithm 1 G-BFS Method

```
Initialization: Q=PriorityQueue(), S_v, s_0
Q.push((T_cost(s_0), s_0));
Add s_0 in S_v;
while Q ≠ ∅ and t < T^max do
    (T_cost(s), s) = Q.pop();
    B_test = Sample ρ randomly from g(s);
    for s' in sampled configuration candidates do
        if s'[−1] = True and s' ∉ S_v then
            Q.push((T_cost(s'), s'));
            Add s' in S_v;
            if t^min_cost > T_cost(s') then
                t^min_cost = T_cost(s');
                s* = s';
            end if
        end if
    end for
end while
Return: The optimal configuration s* with cost t^min_cost.
```

3.4. N-A2C Method

As the G-BFS method explores only one step from the considered state for each iteration, its performance may be affected when the running time from similar states exhibits large random noise. In the N-A2C method, as shown in Fig. 6(a), for each episode, we let the exploration being generated in a ς-step neighborhood, and the direction of exploration is guided with A2C reinforcement learning method (Bhatnagar et al., 2009). The center of the exploration neighborhood is periodically updated with the optimal states ever visited.

We summarize the tuning method in Algorithm 2. Initially, we set a random or experience-estimated starting state s_0, a fixed-size memory buffer $\mathcal{M}$ to record the latest searching information and an empty hashtable H_v recording all the visited states with their running time. For the A2C reinforcement learning model, both actor and critic firstly establish their neural networks with random weights, respectively. Following the configuration search model, for each episode, while the collected configuration candidates for testing is less than the predefined tested batch size, iteratively, from the same starting point, the agent explore $\mathcal{T}$ continuous steps. For each step, with probability of τ, the agent take action a guided by the policy $\pi(s)$ from the actor's nerual network; With probability of $1 - \tau$, the agent choose a random action a from current state. Based on the current state s and action a, we get the next state s' from (7). If the next state s' has not been visited before, we add the state s' in collected

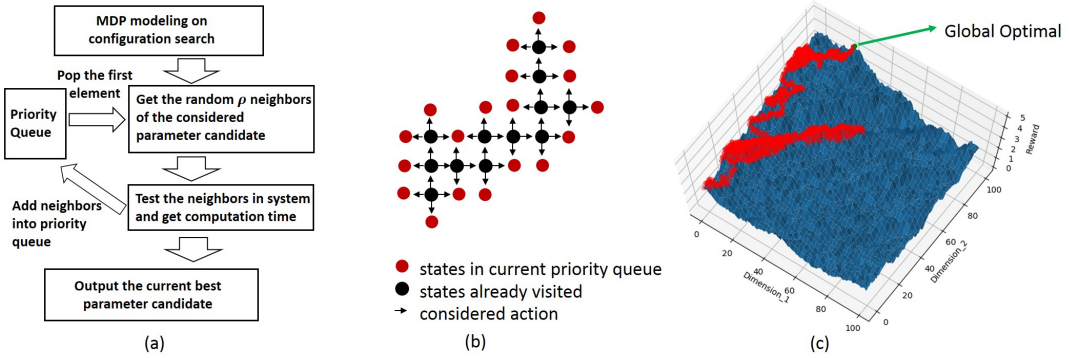


Figure 5. G-BFS Method

candidate batch $\mathcal{B}_{collect}$.

When the number of collected candidates $\mathcal{B}_{collect}$ achieve the predefined test size, we let the system run the collected configuration candidates. Based on the corresponding running time, the stored hashmap H_v and memory buffer $\mathcal{M}$ is updated. Based on updated $\mathcal{M}$, by randomly selecting history exploration data, the neural networks of A2C reinforcement learning is trained.

Generally, the proposed N-A2C method is able to efficiently search the optimal GEMM configuration with fixed exploration step in each episode. Nevertheless, in order to find the most important exploration neighborhood, the exploration step $\mathcal{T}$ can have a soft start, which means starting with a large value and gradually reduce to a small number. Moreover, in order to fully explore the configuration space, after a long time exploration within the same starting points, the exploration step $\mathcal{T}$ will gradually increase to further explore new configuration candidates.

In Fig. 6(b), we show a simple exploration map with the proposed N-A2C method. With more explorations, the exploration neighborhood changes with the update of optimal states ever visited. In Fig. 6(c), we take an example of 2-dimensional configuration on the randomly generated rewards function. Due to the large randomness in the example, we set the exploration step $\mathcal{T}$ as 100 and the global optimal state can be efficiently discovered with the assistance of the A2C reinforcement learning algorithm.

3.5. Discussion

We compare the G-BFS guided tuning method, N-A2C reinforcement learning method and the XGBoost guided tuning method in Table 1.

Since our proposed configuration search model is based on neighborhood information which is the ground for both our tuners, exploration by G-BFS and N-A2C methods are more confined in neighborhoods of current candidates and

Table 1. Comparison of Configuration Search Methods

Methods	XGBoost	G-BFS	N-A2C
Adapt to large space	No	Yes	Yes
Adapt to fluctuations	Weak	Weak	Strong
Lightweight	No	Yes	Yes
Hyper-parameters	Yes	only 1	Yes
Trapped in Local Optima	No	No	No

increasing the size of the configuration space does not have a large impact as it does on the XGBoost method, as the XGBoost method considers the overall configuration space. Furthermore, when the performance feedback of neighboring configurations has large fluctuations, XGBoost and G-BFS may not be able to efficiently discover the global optimal solution, while the N-A2C method can explore multiple steps from the neighborhood, so as to locate the optimal configuration more efficiently. Moreover, random exploration exists within both XGBoost and N-A2C tuning methods, but in G-BFS, when the value of $\rho = len(g(s))$, there will be no randomness during the configuration search. Compared with other configuration optimization methods, G-BFS is more lightweight and only requires the hyper-parameter setting of ρ. Finally, all three methods are able to jump out of local optima to get to global optimal solutions.

4. Experimental Results

Performance of the proposed GEMM configuration tuning approaches is evaluated in the TVM framework on Nvidia Titan Xp GPU. Following similar settings in TVM for GPUs, we set the number of nested loops for each dimension as $d_m = 4, d_k = 2, d_n = 4$. We set the random selection parameter $\rho = 5$ for the G-BFS method, and the maximum search step $\mathcal{T} = 3$ for the N-A2C method. For generality, we set the initial state for the proposed methods as $s_0 = [[m, 1, 1, 1], [k, 1], [n, 1, 1, 1]]$, referring to the situation without multi-level matrix tiling, and the performance of our proposed methods can be further improved by setting

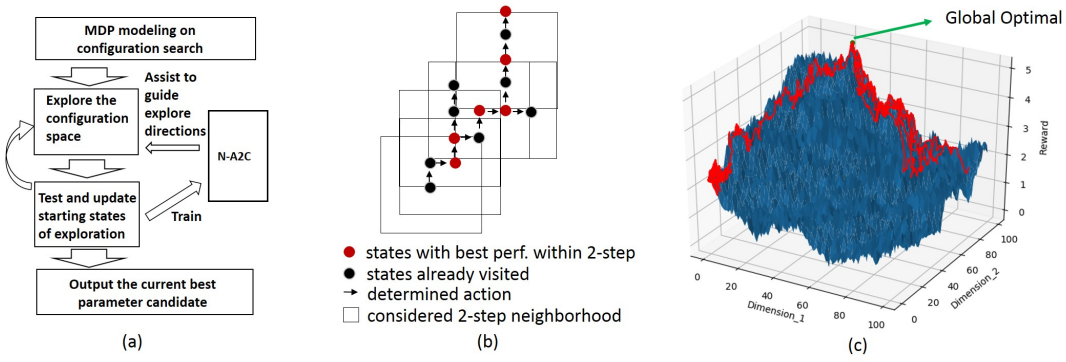
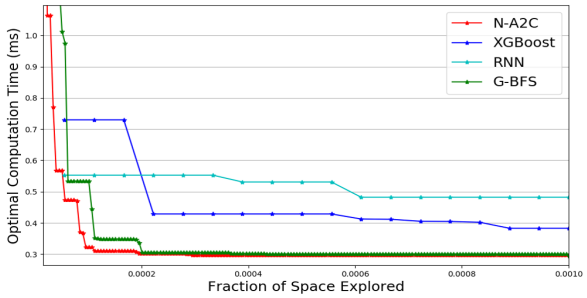
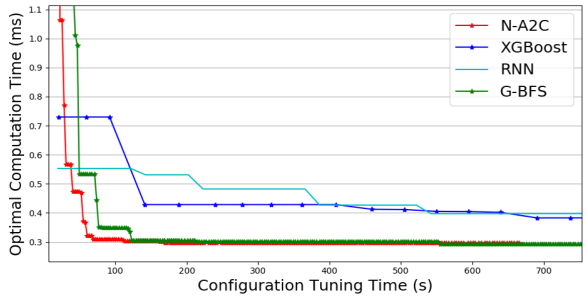


Figure 6. N-A2C Method



(a) Optimal computation time versus ratio of visited data



(b) Optimal computation time versus searching time

Figure 7. GEMM configuration tuning in (1024, 1024, 1024)

the initial state to a more meaningful configuration. In order to investigate the performance of our proposed methods, we compare them with the state-of-the-art algorithms including the XGBoost guided configuration tuning methods (or "the XGBoost method") in TVM framework and the general configuration optimization method using a RNN controller by Google researchers. Without specific explanations, the computation time for each configuration is the arithmetic mean for 10 repeated trials on the tested GPU hardware.

In simulations, we evaluate and compare the configuration tuning efficiency in a perceptron network, which is the fundamental unit for state-of-the-art neural network architectures and mainly includes GEMM for computation. During training of the network, we denote the setting of neural network in the format of $(number_of_inputs, batch_size, number_of_outputs)$, which corresponds to (m, k, n) in matrices $A(m \times k)$ and $B(k \times n)$ for GEMM.

In Fig. 7, we set $number_of_inputs = 1024$, $batch_size = 1024$ and $number_of_outputs = 1024$, and show the performance of our proposed configuration tuners. We first analyze the optimal computation time discovered with respect to the fraction of visited configurations in Fig. 7a. Based on the sizes of matrices and the number of nested loops, there are totally 899756 configuration candidates. With the value of visited fraction increasing, the optimal computation time discovered generally decreases. Compared with the XGBoost and RNN methods, the proposed N-A2C and G-BFS methods are able to discover the configuration candidate with lower fraction of visited configuration candidates. Fig. 7b plots the optimal cost (hardware computation time) discovered by the four methods as tuning progresses over time. The proposed N-A2C and G-BFS methods generally use less tuning time to find the configuration with lower cost, compared with the XGBoost and RNN methods.

In Fig. 8, we evaluate and compare the configuration tuning efficiency in the perceptron network. In Fig. 8a, we compare the discovered optimal computation time when the fraction of visited configuration candidates reaches 0.1%. The total numbers of configuration candidates for $(512, 512, 512)$, $(1024, 1024, 1024)$, $(2048, 2048, 2048)$ matrix tiling are 484000, 899756 and 1589952, respectively. As the sizes of matrices increase, longer computation time is required, and G-BFS and N-A2C can search configuration more efficiently than the XGBoost and RNN methods. Specifically, with 0.1% exploration of $(1024, 1024, 1024)$'s configuration space, the proposed G-BFS and N-A2C methods are able to discover configurations of 24% lower computation time than what the XGBoost method can find and configurations of 40% lower computation time than what the RNN method can find. When the sizes of matrices increase, the

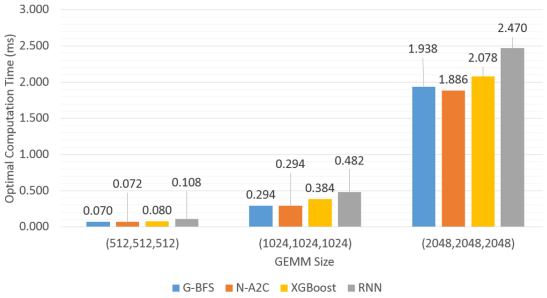
Algorithm 2 N-A2C Method

Initialization: s_0, $\mathcal{M}$, H_v
for each episode **do**
  while $len(\mathcal{B}_{collect}) < len(\mathcal{B}_{test})$ **do**
    $s = s_0$;
    for each step until T steps **do**
      if $rand() < \tau$ **then**
        a follows $\pi(s)$;
      else
        a is random selected from $\mathcal{A}$;
      end if
      $s' = step(s, a)$;
      if s' not in H_v **then**
        Add s' in $\mathcal{B}_{collect}$;
      end if
      $s = s'$;
    end for
  end while
  for s' in $\mathcal{B}_{collect}$ **do**
    if $t_{cost}^{min} > T_{cost}(s')$ **then**
      $t_{cost}^{min} = T_{cost}(s')$;
      $s^* = s'$;
      $s_0 = s^*$;
    end if
    $H_v[s'] = T_{cost}(s')$;
    Store $(s, a, r(s, a), s')$ to $\mathcal{M}$, where $\forall s$, $\forall a$ satisfying $step(s, a) = s'$;
    Train actor's and critic's neural networks with $\mathcal{M}$;
  end for
end for
Return: The optimal configuration s^* with cost t_{cost}^{min}.

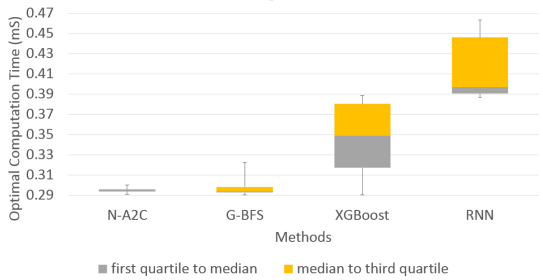
N-A2C method outperform G-BFS method, as the N-A2C method is able to go multiple steps from the current state. In Fig. 8b, we compare the optimal discovered computation time of a configuration when the tuning time is limited to 750 seconds. In order to show the variance of performance incurred by random exploration for each method, we draw a box plot with the minimum, first quartile, median, mean, third quartile, and maximum values during the 10 trials on the $(1024, 1024, 1024)$ tiling. We can see that our methods exhibit more stable performances than the other two methods.

5. Conclusion

In this paper, we propose a Greedy Best First Search guided (G-BFS) method and a Neighborhood Actor Advantage Critic (N-A2C) method for compiler-level GEMM optimization, taking advantage of performance of neighborhood configurations. The G-BFS method, though being lightweight, have shown strength of stable performance; and the N-A2C method performs better for large matrices. Empirical results show that both methods achieve significant performance improvement over state-of-the-art tuning methods such as those using XGBoost and RNN controller. Both methods are general in the sense that they are applicable to other compiler-level tuning tasks and can be used for optimization of other tensor operators with large configuration space.



(a) When the fraction of visited configurations reaches 0.1%



(b) When the tuning time reaches 750 seconds

Figure 8. Comparisons of configuration tuning efficiency

References

An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002. ISSN 0098-3500. doi: 10.1145/567806.567807. URL http://doi.acm.org/10.1145/567806.567807.

Bellman, R. E. *Adaptive control processes: a guided tour*. Princeton university press, 1961.

Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. Neural optimizer search with reinforcement learning. *arXiv preprint arXiv:1709.07417*, 2017.

Bergstra, J. and Bengio, Y. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb): 281–305, 2012.

Bergstra, J. S., Bardenet, R., Bengio, Y., and Kégl, B. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554, 2011.

Bhatnagar, S., Sutton, R. S., Ghavamzadeh, M., and Lee, M. Natural actor–critic algorithms. *Automatica*, 45(11):2471–2482, 2009.

Bosman, P. A. N., Grahl, J., and Thierens, D. Adapted maximum-likelihood gaussian models for numerical optimization with continuous edas. *Software Engineering [SEN]*, (E0704), 2007.

Chen, T. and Guestrin, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pp. 785–794. ACM, 2016.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: end-to-end optimization stack for deep learning. 2018a. URL http://arxiv.org/abs/1802.04799.

Chen, T., Zheng, L., Yan, E. Q., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. 2018b. URL http://arxiv.org/abs/1805.08166.

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

Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B., and Dubey, P. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

Dukhan, M. Nnpack, 2016.

Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., 1989.

Hansen, N. and Ostermeier, A. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9 (2):159–195, 2001.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *arXiv prepring arXiv:1506.01497*, 2015.

Hinton, G. E. A practical guide to training restricted boltzmann machines. In *Neural networks: Tricks of the trade*, pp. 599–619. Springer, 2012.

Hoffman, M. W. and Shahriari, B. Modular mechanisms for bayesian optimization. In *NIPS Workshop on Bayesian Optimization*, pp. 1–5. Citeseer, 2014.

Holland, J. H. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Jia, Y. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, Univ. of California at Berkeley, Berkely, Calif., 2014.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191*, 2018.

Kennedy, J., Eberhart, R. C., and Shi, Y. Swarm intelligence. 2001. *Kaufmann, San Francisco*, 1:700–720, 2001.

Larochelle, H., Erhan, D., Courville, A., Bergstra, J., and Bengio, Y. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pp. 473–480. ACM, 2007.

Larrañaga, P. and Lozano, J. A. *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media, 2001.

Leary, C. and Wang, T. Xla: Tensorflow, compiled. *TensorFlow Dev Summit*, 2017.

LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature*, 521 (7553):436, 2015.

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. Efficient backprop. In *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 2012.

Loshchilov, I. and Hutter, F. CMA-ES for hyperparameter optimization of deep neural networks. *CoRR*, abs/1604.07269, 2016. URL http://arxiv.org/abs/1604.07269.

Matthes, A., Widera, R., Zenker, E., Worpitz, B., Huebl, A., and Bussmann, M. Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library. Technical report, https://arxiv.org/abs/1706.10086, 2017.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

NNVM. Nnvm compiler: Open compiler for ai frameworks. 2017. URL http://tvmlang.org/2017/10/06/nnvm-compiler-announcement.html.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

Ramachandran, P., Zoph, B., and Le, Q. V. Searching for activation functions. 2018.

Rechenberg, I. and Eigen, M. Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. *Frommann-Holzboog Verlag, Stuttgart*, 1973.

Schwefel, H.-P. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: mit einer vergleichenden Einführung in die Hill-Climbing-und Zufallsstrategie*. Birkhäuser, 1977.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Storn, R. and Price, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, Dec 1997. ISSN 1573-2916. doi: 10.1023/A:1008202821328. URL https://doi.org/10.1023/A:1008202821328.

Truong, L., Barik, R., Totoni, E., Liu, H., Markley, C., Fox, A., and Shpeisman, T. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks. *ACM SIGPLAN Notices*, 51(6):209–223, 2016.

Warden, P. Why gemm is at the heart of deep learning? https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/, 2015.

Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, 2014.

Zoph, B. and Le, Q. V. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.