FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System

Size Zheng
zhengsz@pku.edu.cn
CECA, Department of CS
Peking University

Yun Liang*
ericlyun@pku.edu.cn
CECA, Department of CS
Peking University

Shuo Wang
shvowang@pku.edu.cn
CECA, Department of CS
Peking University

Renze Chen
crz@pku.edu.cn
CECA, Department of CS
Peking University

Kaiwen Sheng
sheng_kaiwen@pku.edu.cn
CECA, Department of CS
Peking University

Abstract

Tensor computation plays a paramount role in a broad range of domains, including machine learning, data analytics, and scientific computing. The wide adoption of tensor computation and its huge computation cost has led to high demand for flexible, portable, and high-performance library implementation on heterogeneous hardware accelerators such as GPUs and FPGAs. However, the current tensor library implementation mainly requires programmers to manually design low-level implementation and optimize from the algorithm, architecture, and compilation perspectives. Such a manual development process often takes months or even years, which falls far behind the rapid evolution of the application algorithms.

In this paper, we introduce FlexTensor, which is a schedule exploration and optimization framework for tensor computation on heterogeneous systems. FlexTensor can optimize tensor computation programs without human interference, allowing programmers to only work on high-level programming abstraction without considering the hardware platform details. FlexTensor systematically explores the optimization design spaces that are composed of many different schedules for different hardware. Then, FlexTensor combines different exploration techniques, including heuristic method and machine learning method to find the optimized schedule configuration. Finally, based on the results of exploration, customized schedules are automatically generated for different hardware. In the experiments, we test 12 different kinds of tensor computations with totally hundreds of test cases and FlexTensor achieves average 1.83x performance speedup on NVIDIA V100 GPU compared to cuDNN; 1.72x performance speedup on Intel Xeon CPU compared to MKL-DNN for 2D convolution; 1.5x performance speedup on Xilinx VU9P FPGA compared to OpenCL baselines; 2.21x speedup on NVIDIA V100 GPU compared to the state-of-the-art.

• **Software and its engineering → Source code generation**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies** → *Machine learning*.

code generation, compiler optimization, heterogeneous systems, machine learning

ACM Reference Format:
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 (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378508

1 Introduction

Tensor computations are arithmetic operations among multi-dimensional arrays. A tensor is a natural way to represent multi-factor or multi-relational data and has found numerous applications for machine learning [18, 25, 45], data analysis, and data mining [42], social networks [39], and so on. The tensor computations are often computing-intensive kernels

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378508

composed of deeply-nested loops, which are perfect candidates to be accelerated on high-performance and low-power hardware accelerators such as GPUs and FPGAs. Therefore, a central problem with tensor-oriented data analytics is how to design a high-performance library for various tensor algorithms on heterogeneous systems.

One of the most widely used approaches is to write hand-optimized low-level kernels, where the programmers manually design high-performance libraries by manipulating the optimization options, including compute, memory, interconnect, and data bit-width, etc. This low-level implementation approach entangles algorithm with hardware and compilation options for a specific platform. This approach has several drawbacks. First, the hand-tuned libraries such as cuDNN [11] for GPUs, MKL [22] for Intel CPU, FBLAS [35] for FPGAs take months or even years to develop, which is much slower than the rapid evolution of the algorithms. For example, the emerging shift convolution [59] operator used in the neural networks for embedded devices currently lacks high-performance library support. Consequently, fewer researchers will continue to explore these new tensor operators, which may bury operators of high potential values. Second, for a specific hardware architecture, there are many optimization schedules and parameters to decide at application, architectural, and compilation levels, forming a large space with non-trivial performance trade-offs to explore. Manual designs without systematic exploration can easily lead to a sub-optimal implementation. Third, different hardware systems vary hugely in terms of computation, memory, and many other details, rendering manual implementation on one platform hardly portable to other platforms.

In recent years, there has been a growing interest in leveraging high-level abstraction to describe tensor algorithms and employing tensor compilers to generate the low-level implementations automatically [8, 44, 53]. This is a promising solution as it requires less manual effort and can potentially be applied to new tensor operators and platforms where the library support is not ready. For example, Halide [44] separates compute from schedule and uses high-level abstractions to generate codes for image processing pipelines. TVM [8] is a whole compilation stack for deep learning. It provides high-level domain-specific languages (DSLs) for programmers to develop new operators and performs code generation according to compute and schedule specifications given by programmers. However, TVM still requires the programmers to design a schedule template for each tensor operator manually [9]. The schedule template is composed of different schedule primitives such as split, reorder, and fuse. Designing a high-performance schedule template is as difficult as writing a low-level implementation. The reason is that though the high-level abstraction improves programming productivity, the programmers should still carefully decide how to manipulate loops using the primitives and reason about the performance trade-offs of different schedules.

In this paper, we introduce FlexTensor, which is a schedule exploration and optimization framework for tensor computations on heterogeneous systems. Given a high-level description of the tensor algorithms (only describe mathematical calculations), FlexTensor will automatically decide how to map the tensor algorithms onto low-level implementations for different hardware platforms including CPUs, GPUs, and FPGAs, providing high programming productivity, good platform portability, and high performance. There is no need for users to write any schedule or template manually. FlexTensor is composed of two parts: the front-end and the back-end. The front-end of FlexTensor takes tensor computations written in Python as inputs. It employs static analysis to analyze the computation pattern and generates a hardware-specific schedule space. The back-end of FlexTensor leverages a heuristic and machine learning combined method to find the optimized schedule configuration. Our heuristic method is based on simulated annealing and our machine learning algorithm is based on Q-learning. For schedule implementation, FlexTensor applies different optimizations, including multi-level loop tiling, loop reordering, loop parallelization, and memory customization based on the schedule configurations for different hardware.

The contributions of this work are as follows,

- We develop a framework called FlexTensor to automatically optimize tensor computations for CPU, GPU, and FPGA without human interference.
- We propose static analysis in the front-end of FlexTensor to analyze different tensor computation patterns and form the design space for different hardware.
- We propose a heuristic and machine learning combined approach in the back-end of FlexTensor to explore the design space and automatically implement the optimized schedules for different hardware.

We evaluate FlexTensor using different tensor operators on CPU, GPU, and FPGA. FlexTensor achieves average 1.83x performance speedup on NVIDIA V100 GPU compared to cuDNN; 1.72x performance speedup on Intel Xeon CPU compared to MKL-DNN for 2D convolution; 1.5x performance speedup on Xilinx VU9P FPGA compared to OpenCL baselines; 2.21x performance speedup on NVIDIA V100 GPU compared to the state-of-the-art.

2 Background & Motivation

In this section, we first introduce tensor computation, schedule, and heterogeneous hardware. Then, we present the motivation of automated tensor optimization for the heterogeneous systems.

Table 1. Definitions of different tensor computations

Operator	Definition	Description
GEMV	$O_i = A_{i,k} \circ B_k$	Matrix-vector multiply
GEMM	$O_{i,j} = A_{i,k} \circ B_{k,j}$	Matrix-matrix multiply
Bilinear	$O_{i,j} = A_{i,k} \circ B_{j,k,l} \circ C_{i,l}$	Bilinear transformation
1D convolution	$O_{b,k,i} = I_{b,rc,i+rx} \circ W_{k,rc,rx}$	1D sliding window convolution
Transposed 1D convolution	$O_{b,k,i} = I_{b,rc,i+rx} \circ W_{rc,k,L-rx-1}$	Transposed convolution for 1D array
2D convolution	$O_{b,k,i,j} = I_{b,rc,i+rx,j+ry} \circ W_{k,rc,rx,ry}$	2D sliding window convolution
Transposed 2D convolution	$O_{b,k,i,j} = I_{b,rc,i+rx,j+ry} \circ W_{rc,k,X-rx-1,Y-ry-1}$	Transposed convolution for 2D matrix
3D convolution	$O_{b,k,d,i,j} = I_{b,rc,d+rd,i+rx,j+ry} \circ W_{k,rc,rd,rx,ry}$	3D sliding window convolution
Transposed 3D convolution	$O_{b,k,d,i,j} = I_{b,rc,d+rd,i+rx,j+ry} \circ W_{rc,k,D-rd-1,X-rx-1,Y-ry-1}$	Transposed convolution for 3D cube
Group convolution	$O^g_{b,k,i,j} = I^g_{b,rc,i+rx,j+ry} \circ W^g_{k,rc,rx,ry}$	2D convolution separated into groups
Depthwise convolution	$O_{b,k,i,j} = I_{b,c,i+rx,j+ry} \circ W^c_{k,rx,ry}$	2D convolution separated by channels
Dilated convolution	$O_{b,k,i,j} = I_{b,rc,i+rx\times dx,j+ry\times dy} \circ W_{k,rc,rx,ry}$	2D convolution with kernel dilation

2.1 Tensor Computation & Schedules

A tensor is a multi-dimensional array of data. It can be dense or sparse. In this paper, we only consider dense tensors. Operations among tensors such as addition and multiplication are examples of tensor computation. We list the mathematical definitions of tensor operators considered in this paper in Table 1, which include matrix multiplication and convolution. To save space, we avoid presenting long formulas and use the Einstein summation convention to express each operator. We use $\circ$ to express multiply and sum operation, and the subscripts that appear on the right side but don't appear on the left side represent for reduction. For example, the 2D convolution definition $O_{b,k,i,j} = I_{b,rc,i+rx,j+ry} \circ W_{k,rc,rx,ry}$ can be interpreted as $O_{b,k,i,j} = \sum_{rc}\sum_{rx}\sum_{ry} I_{b,rc,i+rx,j+ry} \times W_{k,rc,rx,ry}$. Some operators have special parameters. For instance, in group convolution P^g, I^g, W^g are tensors of group g; for depthwise convolution, W^c is the filter tensor for channel c; dilated convolution uses dx, dy as dilation factors.

Traditional implementation of these tensor computations entangles the mathematical calculations with hardware-specific optimizations, which is very difficult to program and takes a long time to develop. To ease the process, [8, 44] leverages the idea of writing *compute* and *schedule* separately to implement algorithms. Compute is the description of tensor computations, while schedule is a list of optimization primitives for compute. Different schedule primitives considered in this paper are listed in Table 2. In practice, writing a schedule is non-trivial because programmers have to choose the right set of schedule primitives and proper parameters from millions of possible combinations.

2.2 Heterogeneous Systems

Recently, it is prevalent to use heterogeneous hardware to accelerate tensor computation. Different hardware varies in architecture, programming model, and optimization.

On CPUs, two critical optimizations that impact the performance are parallelism and locality. To exploit parallelism, we can split the workload among CPU cores using a parallel programming model. For example, OpenMP [14] provides pragmas to enable parallel execution of loops (#pragma omp parallel) and barrier among parallel threads (#pragma omp barrier). We use tiling to exploit data locality.

GPUs employ massive threading for high throughput. NVIDIA CUDA [40] programming model provides an abstraction of grid, block, and thread hierarchy to expose parallelism for programming. Parallel programming on GPUs is challenging because the final performance depends on both thread-level parallelism and single-thread performance. Besides, GPUs allow programmers to explicitly configure on-chip memory (shared memory) within thread blocks, providing opportunities for better locality optimization but at the same time adding extra difficulty to programming.

FPGAs are reconfigurable hardware, which allow programmers to tailor the hardware for tensor computations by customizing pipeline, parallelism, and data bit-width. However, FPGAs are very difficult to program and optimize. Traditional FPGA programming requires RTL (register transfer level) implementation, but writing RTL is tedious and time-consuming. Though HLS (high level synthesis) [13] lowers the programming barriers for FPGAs by using high-level programming models such as C and C++, it still requires high expertise and significant effort to optimize.

2.3 Motivation

Here we use two examples to illustrate the challenges in writing schedules on heterogeneous systems. First, different combinations of schedule primitives lead to different performance. In Figure 1a, we use three different schedules to optimize 2D convolutions on NVIDIA V100 GPU. We choose three different input shapes (C2, C8, and C13 listed in Table 4) for illustration and the batch size is 8. *Schedule-a* splits the batch dimension for tiling, while *schedule-b* binds batch dimension to different thread blocks, and *schedule-c* simply fuses all loops together. The difference in these schedules is

Table 2. Different schedule primitives for different target platforms and their parameters.

Target	Name	Description	Parameter
All	split	divide a loop into several sub-loops	loop to split and split factors
	fuse	merge several loops into a hyper-loop	adjacent loop to fuse
	reorder	change execution orders of loops	loops to reorder and new order
	unroll	unroll a loop by given depth	which loop to unroll and unroll depth
	vectorize	apply vector operation to a loop	which loop to vectorize
	inline	inline a function	which node to inline
	compute at	put producer in the body of consumer	which node and how deep to compute at
CPU	parallel	use multithreading	which loop to parallel
GPU	cache	use shared memory to store inputs/results	which tensor to cache and how much data to cache
	bind	assign a loop to parallel blocks/threads	which loop to bind to block/thread
FPGA	buffer	how much input to buffer at a time	rows and columns of inputs to buffer
	pipeline	pipeline of data read/write and computation.	number of stages in pipeline and number of pipelines
	partition	memeory partition to increase available bandwidth	partion number

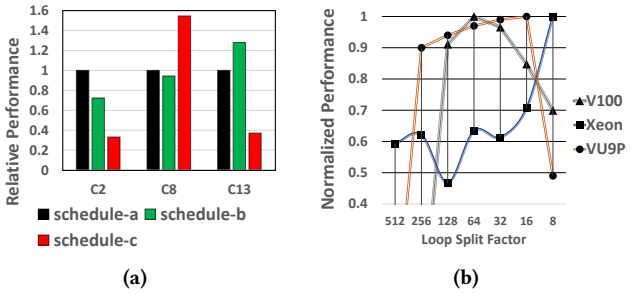


Figure 1. (a) Relative performance of three different schedules for 2D convolutions on the same platform. (b) Normalized performance of different split factors for 2D convolution on V100, Xeon E5 and VU9P.

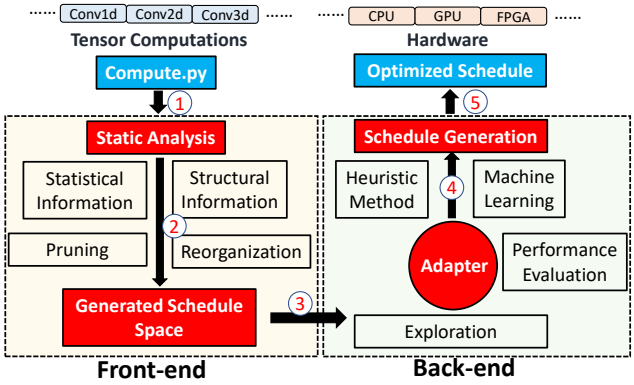


Figure 2. Overview of FlexTensor.

very small (only different in the order of the primitives or simply omit some primitives), but impacts the performance noticeably. We also find that for different input shapes, different schedules are favorable. For C2, *schedule-a* is the best; for C8, the best one is *schedule-c*; for C13, *schedule-b* performs the best. Second, the hardware heterogeneity further adds complexity. In Figure 1b, we compare three different platforms: NVIDIA V100 GPU, Intel Xeon E5 CPU, and Xilinx VU9P FPGA. We vary the split factor (8 to 512) of the inner loop for 2D convolution on the three platforms. As shown, the performance trend and the optimal factor are different on the three platforms.

From the above examples, it's clear that writing an optimized schedule on heterogeneous systems presents huge challenges for programmers. Recent approaches [8, 9] tend to rely on schedule templates to automatically optimize tensor computations. They can only auto-tune the parameters for schedule templates. But the schedule templates are still written manually by the programmers and new templates should be developed for new operators. In this paper, we propose to use automatic schedule space generation along with an efficient search to generate schedules for tensor computations, which is a template-free, fully-automatic framework without human interference.

3 Overview of FlexTensor

Figure 2 presents the overview of FlexTensor. To work with FlexTensor, users describe the tensor computation in mathematical form using Python and register the optimization task into FlexTensor. The flow of FlexTensor can be divided into two parts: the front-end and the back-end.

The front-end analysis employs a static analyzer to extract useful statistical information as well as structural information of operators. Statistical information includes number of loops, trip counts of loops, and structural information includes number of nodes in the graph and number of tensors of each node. FlexTensor relies on these information to generate a schedule space. In the front-end analysis, FlexTensor will also prune the schedule space by eliminating the

suboptimal designs and rearrange the schedules based on their structural similarities.

The back-end analysis adopts an exploration based method to find the optimized schedule. In the huge schedule space, FlexTensor explores with a heuristic and machine learning combined approach. Different points in the space are evaluated by either measuring the performance on target device or querying an analytical model to estimate performance. Then FlexTensor generates customized schedules for different hardware based on hardware-specific characteristics. At last, FlexTensor automatically generates low-level code for different hardware including NVIDIA GPUs, Intel Xeon CPUs, and Xilinx FPGAs based on the final schedule generated by the back-end. We leverage existing tensor compiler [8] for code generation on GPUs and CPUs and extend it to FPGAs.

4 Front-end Analysis and Schedule Space

This section explains the front-end of FlexTensor. The front-end mainly comprises two parts: static analysis of tensor computations and generation of schedule space. In the static analysis phase, FlexTensor collects useful statistical information and structural information, and the subsequent phase uses the information to generate schedule space.

4.1 Static Analysis

A tensor computation can be represented as a mini-graph where nodes represent nested-loops, and edges represent data organized in tensor format. Each node in the graph has some input tensors and output tensors, for simplicity, we just consider one output tensor. If node P's output tensor is used as input by node Q, then node Q is called the consumer of node P. We denote input tensors as $\mathrm{I}_1, \mathrm{I}_2, \ldots, \mathrm{I}_\mathrm{N}$, and output tensor as O. Then the computation can be expressed as follows,

$$\mathrm{O}[i_1, i_2, \ldots, i_M] = \mathcal{F}(\mathrm{I}_1, \mathrm{I}_2, \ldots, \mathrm{I}_\mathrm{N})$$

where M is the dimension of output tensor, and N is the number of input tensors. The function $\mathcal{F}$ is used to compute each point in the output tensor. In general, there are two types of loops in each node (nested-loops): *spatial loops* and *reduce loops*. Spatial loops are the loops without data dependency, so they are the best candidates for parallelization, while reduce loops have data dependency and usually run in serial.

To characterize a tensor computation, we need to know how the mini-graph is constructed and how the nested-loops are organized. The information we need falls into two categories: statistical information and structural information, which correspond to the characteristics of graph nodes and edges, respectively. In details, statistical information includes number of spatial loops (noted as *#sl*), number of reduce loops (noted as *#rl*), trip counts of spatial loops (noted as *stc*), trip counts of reduce loops (noted as *rtc*) and loop orders (noted as *order*). Structural information includes number of nodes in mini-graph (noted as *#node*), number of input tensors of each node (noted as *#in*), number of output tensors of each node (noted as *#out*) and number of consumer nodes of each node (noted as *#cs*).

Figure 3 presents an example using GEMM. Figure 3 (a) shows the mini-graph of GEMM, the nodes *op A* and *op B* represent for operations that produce tensor A and B, respectively; tensor A and tensor B are used by GEMM node to produce tensor C. Inside the GEMM node, the nested-loops are shown in Figure 3 (b), *loop i* and *loop j* have no data dependency, so they are spatial loops, and *loop k* has data dependency, so it's a reduce loop. Figure 3 (c) shows the statistical and structural information of GEMM example.

4.2 Schedule Space Generation

The schedule space is generated by enumerating different combinations of schedule primitives and corresponding parameters using statistical and structural information. The enumeration process follows a specific order in the schedule space. In particular, we try the split, reorder, and fuse primitives in Table 2 first, and then other primitives. This ensures that the configuration points with the same number of parameters are put together.

Each point in the schedule space is encoded using a vector, and each value in the vector represents a specific choice of primitive or parameter. Example in Figure 3 (d) is a schedule for GEMM. The schedule splits the three loops of GEMM into twelve sub-loops, then reorders them and generates a larger outer-most loop by fusion. The outer-most loop is parallelized, and the inner-most loop is vectorized. Figure 3 (e) shows how to encode the schedule in (d) as a point in schedule space. We refer to a loop by its nested depth in the loop-nests (1 for the outer-most loop). For split, we record the split factors. For reorder, we record the new order of loops. For fuse, the loops not recorded are meant to be fused with their neighboring outer loops. For unroll, each loop corresponds to a value 0 (not to unroll) or 1 (unroll). Parallel and vectorize are not encoded as we always parallelize the outer-most loop and vectorize the inner-most loop.

To explore this large space efficiently, we first propose to prune the design space by deleting the points that are unlikely to lead to good performance and then rearrange the space by exploiting structural similarity. We prune the space in three ways: 1) limit the depth of primitives combination; 2) prune the parameter space, 3) pre-determine certain decisions for different hardware. First, some of the primitive combinations can be used recursively, which will lead to an infinite number of configurations (e.g., a single loop can use split and fuse recursively). To avoid this, we set a threshold on the depth of primitive combinations (e.g., at most use split and fuse for four times). Second, for parameter pruning, we mainly consider pruning split factors because

split factors account for most of the parameter space. We find that divisible split is efficient in most cases, while other split choices have inferior results. Hence, we limit the split factors to divisible split choices for each loop. At last, we fix certain decisions for different hardware based on the previous optimization studies [8, 37, 66]. For example, on CPU, we only parallelize the outer-most loop (after fusion) and vectorize the inner-most loop; on GPU, we bind outer loops to thread blocks and inner loops to threads; on FPGA, we use three-stage pipeline design, etc.

Besides pruning, we also consider rearranging the schedule space for the subsequent exploration. The rearranging process starts after pruning so that the pruned points will never appear in the exploration process. The schedule space can be represented using a 1D list. However, systematic exploration needs to exploit a local region in the search space (ideally knowing which direction to search along) instead of random sampling. The 1D list has poor locality because there are only two directions to search along. So we propose to change the original 1D list to a high-dimensional space. In the high-dimensional space, the local region has more points. For example, considering splitting a loop of size L to N parts, different choices are generated by N-factorization of integer L, and they can be represented as $[f_1, f_2, ..., f_N]$ where $f_1 \times f_2 ... \times f_N = L$. We rearrange the 1D list of choices into a $\frac{N\times(N-1)}{2}$-dimensional space by adding directions to space. For any point $p = [f_1, f_2, ...f_N]$, the neighboring point at direction (i, j) is $[g_1, g_2, ...g_N]$ where $g_i > f_i, g_j < f_j$ and $\forall k \neq i, k \neq j, g_k = f_k$. In this rearrangement, the neighboring points will have a similar structure. Thus there is a high possibility that these points can lead to similar performance.

5 Back-end Exploration and Optimization

The back-end of FlexTensor explores the schedule space and generates an optimized schedule. Our exploration techniques combine heuristic and machine learning methods, and our schedule implementation is customized for different hardware.

5.1 Exploration with Heuristics and Machine learning

The schedule space generated by the front-end is enormous (e.g., often larger than 10^{11}), which renders exhaustive search infeasible. For an efficient search, we combine heuristics methods with machine learning methods. To explain our method clearly, we denote the searching space as G. Each point in G is represented as a vector, as illustrated in section 4.2. Recall that the front-end reorganizes the original schedule space into a space of higher dimension, and each point in the schedule space has multiple neighboring points. For each point p in G, its adjacent points are different from p at only one position of p (for example, modifying the split factor only or changing a value from 0 to 1 to enable/disable one schedule primitive).

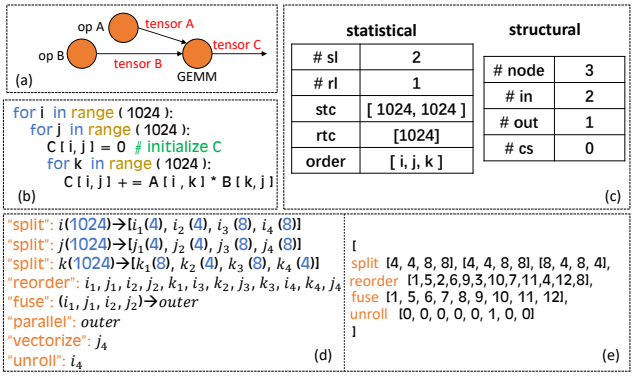


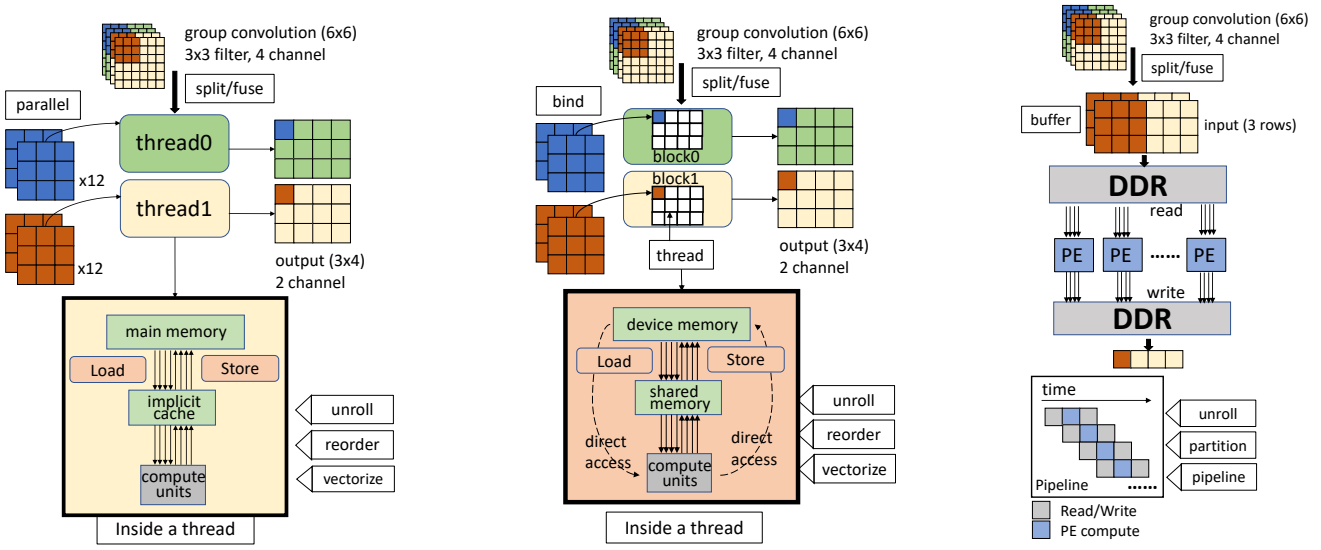
Figure 3. GEMM Example. (a) GEMM mini-graph structure. (b) GEMM code example. (c) Statistical and structural information of GEMM example. (d) A schedule example. (e) Encode schedule in (d) as a point in schedule space.

In the back-end, FlexTensor explores the schedule space to find good schedule configurations. It maintains a set H, each point in H has already been evaluated, and we associate a performance value E with each point. To obtain the performance value, we can either run the program on the target device to collect the real performance or use an analytic model to obtain an estimated performance (illustrated in Section 5.2). During exploration, two critical decisions need to be made: first, which point in H is selected as the starting point for the next step; second, given the starting point p, which direction d to move along to get a new point in G. For the first decision, we use a heuristic method, and for the second decision, we use a machine learning method.

Heuristic Method. Our heuristic method is based on simulated annealing [26]. For a point p in H, its performance value is denoted as E_p. During the exploration, FlexTensor tracks the best point (with the highest E value) in H, and we denote its performance value as E^*. FlexTensor chooses a point p in H as the starting point for next step with probability $\exp^{-\gamma\frac{(E^*-E_p)}{E^*}}$ where γ is a hyperparameter. The closer E_p is to E^*, the more likely p is chosen as the starting point. We can also choose more than one starting point at a time.

Machine Learning Method. Once a point p is chosen as the starting point, there are many possible directions to move along. If we try out all the directions, the exploration process will take extremely long. Instead, we want to try the "best" direction d in G. The "best" direction should ultimately lead the the best performance.

To do this, we design a method based on reinforcement learning to predict the "best" direction. The problem of finding the best direction for a certain point is similar to finding the best action for a certain state, and can be solved by reinforcement learning algorithm [50]. In reinforcement learning



(a) Schedule for CPU. **(b)** Schedule for GPU **(c)** Schedule for FPGA

Figure 4. FlexTensor's schedule generation for CPU, GPU and FPGA. Use group convolution as example for illustration.

theory, there is a state set S, an action set A, and a reward function r. For a state $s \in S$, taking action $a \in A$ results reward value $r(s, a)$. Every time an action is performed, the current state s_t changes to a new state $s_{t+1} = f(s_t, a_t)$, where t is current time step and f is transformation function. The aim of reinforcement learning is to maximize $\sum_{t \leq T} r(s_t, a_t)$ by always finding good actions a_t, where T is the maximum attempted step. Accordingly, we treat point p as a state, directions ds in G are actions, and a new point e is obtained if we move from p along direction d. Our purpose is to always find a series of good direction ds to finally reach the optimal point. So the reward value can be designed as $r(p, d) = E_e - E_p$ where E_e is the performance value of point e and E_p is the performance value of point p, thus $\sum_{t \leq T} r(s_t, a_t) = E_{s_T}$. Maximizing E_{s_T} is equivalent to finding the optimal point. In practice, we normalize reward value for stability, so $r(p, d) = \frac{E_e - E_p}{E_p}$.

In this paper, we use a Q-learning algorithm to maximize the objective function. Q-learning is a special reinforcement learning algorithm proposed in [57]. The idea of Q-learning is to assign a value to each action (the direction in our problem), which is called Q-value. The bigger the Q-value is, the better the action is. The crucial part of Q-learning is to calculate Q-values. To do this, we design a neural network to predict the Q-values. More clearly, we build a network composed of four fully connected layers and use ReLU [3] activation. Our algorithm can start with multiple starting points. In fact, the number of starting points can be set by the user. For each starting point p, we query the neural network (use p as input feature) to obtain the Q-values of all the directions that can be reached from p. We choose the direction with the biggest Q-value and move from p to its neighbor point e along this direction (i.e., single step). The searching process can involve multiple steps, which can be set by the user. During the search process, we record the visited points to avoid repeated searching, and there is no back-tracing. We evaluate the new point e and record the results as a tuple (p, e, $\frac{E_e - E_p}{E_p}$).

Q-learning is an online learning algorithm, so the neural network training is conducted during exploration. The training process is similar to that of [36]. To train our four-layer network (denoted as X), we create another network (denoted as Y) with the same structure. In the beginning, both networks are initialized with the same parameters, and they are not accurate because the parameters are not trained. The training process happens periodically (every five trials), and we use the collected data to train the network X. For each data tuple (p, e, $\frac{E_e - E_p}{E_p}$), we first calculate target value (i.e. label) using the formula: $target = \alpha \times \max Y(e) + \frac{E_e - E_p}{E_p}$ where α is a hyperparameter; then we use mean squared error $(X(e) - target)^2$ as loss value and run AdaDelta algorithm [64], which is an optimizing algorithm used to perform backward process and update network X; at last, the parameters of X are copied to network Y as a backup. The network Y is used to make the training procedure stable [36].

5.2 Performance Comparison

There are two ways to obtain the performance value for the points to evaluate during exploration. One is to measure the real performance by executing the program on the target device, while another is to use analytical models to estimate the performance. Real measurement returns the

precise performance and thus gives accurate starting points and directions to search. Also, it is easy to implement and portable to different devices. However, this approach could take a very long time to complete if the compilation and execution overhead is not small. On the other hand, using an analytical model is very fast, which is very useful when the target platform is not available, or the real measurement takes a long time to complete. However, it is non-trivial to build an analytical model as it depends on many parameters in application, architecture, and compilation level, which vary a lot for different hardware platforms.

In this work, we choose to use a measurement approach on CPUs and GPUs as the compilation and measurement overhead is relative small on these platforms ($\leq$ 1s). However, on FPGAs, it may take hours to synthesize a high-level schedule to a netlist [49]. This lengthy synthesis process prevents us from real measurement on FPGAs. Instead, we use the performance models from [32, 49] to predict the performance on FPGAs. The performance on FPGAs can be estimated by multiplying the number of rounds of parallel processing with the execution time of one round. The execution time of one round is bounded by the maximum time of computation, data read, and data write. We only present the simplified model and omit the details:

$$Execution\_time = \frac{workload}{\#PE} \times \max(R, C, W)$$

where R is the data read time, W is the data write time, C is the computation time, $\#PE$ is the number of parallel processing elements (PEs). This model is derived from our three-stage pipeline for FPGA PE design, as illustrated in Figure 4c. The final time cost is determined by the workload size, the longest stage in the pipeline ($\max(R, C, W)$), and the number of PEs ($\#PE$).

5.3 Optimized Schedule Implementation

After we find a good schedule point, we use the primitives listed in Table 2 to implement the schedule on the target device. We can choose different orders to generate the schedule depending on the priority given to the graph and the nodes. Alternative orders are *bottom-up order*, *top-down order* and *recurrent order*. The bottom-up order generates a schedule for each node in a mini-graph first and then generates schedules for the whole graph. The top-down order is on the opposite. And the recurrent order is to repeat bottom-up order or top-down order for several times. Here we adopt bottom-up order because it's straight forward and efficient in most cases. Algorithm 1 shows the outline of our algorithm. The mini-graph structure of the tensor computation is obtained by *get_graph* function in line 1. And the graph is traversed in post-order to obtain each node in line 2. We implement schedule for each node by invoking function *Schedule_for_node* in line 5, then function *Schedule_for_graph* is used to implement schedule for the mini-graph in line 8.

```
1:  graph ← get_graph(operator)
2:  node_lst ← post_order_traverse(graph);
3:  op_config_lst ← [];
4:  for every node in node_lst do
5:      op_config ← Schedule_for_node(node);
6:      op_config_lst.append(op_config)
7:  end for
8:  graph_config ← Schedule_for_graph(graph);
9:  config ← Config(op_config_lst, graph_config);
10: return config
```

Algorithm 1: FlexTensor schedule outline

Schedule for CPU. Figure 4a illustrates CPU schedule generation for a group convolution example. For CPU, register blocking and vectorization are of critical importance. Register blocking can be enabled through multi-level tiling [24]. Multi-level tiling uses split primitive and reorder primitive recursively to produce a series of small tiles. We split spatial loops and reduce loops according to the split factors from exploration results. After splitting, the loops become smaller and can be potentially held by the cache to exploit data locality. To exploit parallelism, we dynamically fuse several outer loops into one outer-most loop and parallelize it. Vectorization is applied to the inner-most loop. The trip counts of the inner-most loop are determined by the split factors and tuned in the exploration phase. FlexTensor can dynamically decide the vectorization length to adapt to different instruction sets such as AVX2 and AVX512. In Figure 4a, splitting along channel dimension produces two sub-convolutions, and each thread takes one sub-convolution workload. Loop unroll, loop reordering and loop vectorization are performed after that.

Schedule for GPU. Figure 4b is a GPU schedule generation example. For GPU, block/thread decomposition and configuration of shared memory are important. Block and thread decomposition is reflected in split factors and implemented by multi-level tiling. The tiling process can produce many sub-loops. Outer loops are bound to thread blocks, and some inner loops are bound to threads so that each thread block can handle only a small part of outputs. The exploration phase tunes different split factors and order of loops, so different decomposition strategies are tried. For shared memory configuration, each block always loads data to shared memory before computation. The size of shared memory used in each block is determined by the trip counts of inner loops, which are also tuned during exploration. To further improve performance, we always use a tile of registers to store the intermediate results of computation and only write back the results after computation is done. For the group convolution example, we use two blocks and twelve threads

Table 3. Benchmark specifications.

Tensor Computations		Analysis Results		Library Support		FLOPs	Precision	Test Cases
Operator	Abbr.	#sl/rl	#node	CPU	GPU			
GEMV	GMV	1/1	1	MKL	cuBlas	16K-1M	float32	6
GEMM	GMM	2/1	1	MKL	cuBlas	32K-8.6G	float32	7
Bilinear	BIL	2/2	1	MKL	cuBlas	1G	float32	5
1D convolution	C1D	6/2	2	MKL-DNN	cuDNN	50M-200M	float32	7
Transposed 1D convolution	T1D	9/2	3	PyTorch	cuDNN	50M-200M	float32	7
2D convolution	C2D	8/3	2	MKL-DNN	cuDNN	77M-3.7G	float32	15
Transposed 2D convolution	T2D	12/3	3	PyTorch	cuDNN	77M-3.7G	float32	15
3D convolution	C3D	10/4	2	PyTorch	cuDNN	77M-6.6G	float32	8
Transposed 3D convolution	T3D	15/4	3	PyTorch	cuDNN	77M-6.6G	float32	8
Group convolution	GRP	4/3	2	MKL-DNN	cuDNN	20M-900M	float32	14
Depthwise convolution	DEP	4/3	2	MKL-DNN	cuDNN	250K-3.6M	float32	7
Dilated convolution	DIL	4/3	2	MKL-DNN	cuDNN	100M-1.2G	float32	11

per block. For inner loops, unroll and reorder primitives are used to fine-tune performance.

Schedule for FPGA. Figure 4c shows schedule generation for FPGA. We leverage a widely used three-stage coarse-grained pipeline architecture to build tensor computation accelerators on FPGAs. The three-stage pipeline is composed of three parts which are data read, compute, and write stages. On one hand, the schedule for data read and write stages are built and configured based on the DDR bandwidth, on-chip memory buffer size, and the total size of transferred data. On the other hand, the schedule for the compute pipeline stage is determined based on the available DSP resources for parallel data processing and the BRAM memories for local on-chip memory buffer.

6 Experiments

6.1 Experiments Setup

We implement FlexTensor in Python and use TVM [8] tools (version 0.6.dev) for code generation. We evaluate FlexTensor for a variety of tensor computations on different hardware. The details of the tensor computations are shown in Table 3. Their mathematical definitions have been introduced in Table 1. Operators examined in the experiments find applications in various domains. For example, GMV, C2D, T2D are used in image processing [30, 45]; GMM, C1D, T1D are used in natural language processing [25]; C3D, T3D are used in video processing [18, 51]; DEP [12] is used for mobile and embedded devices [47].

Different operators have different numbers of spatial loops and reduce loops, and their mini-graphs contain different numbers of nodes. For example, T1D requires expansion and padding before convolution, so its mini-graph has three nodes. The column **Analysis Results** in Table 3 gives details of the structural and statistical information of each operator. We evaluate each operator using multiple test cases, which vary in the input size and computation workload. The columns **Test Cases** and **FLOPS** show the number of test cases and computation workload, respectively.

We compare FlexTensor with the hand-tuned libraries in PyTorch [43] (version 1.0). PyTorch integrates multiple libraries such as MKL, MKL-DNN, cuBlas, and cuDNN. For the operators with no or poor library support, PyTorch [43] uses its native implementation (called native library). On GPU, cuDNN backend for PyTorch can be enabled by setting *torch.backends.cudnn.enabled = True*, and we use cuDNN v7 for PyTorch. When cuDNN is disabled, PyTorch uses the native library [21] on GPU. For each benchmark, we use float32 precision with batch size 1 for inference. For GPU experiments, we use NVIDIA V100 (16GB device memory), P100 (16 GB device memory), and Titan X (Pascal). For CPU experiments, we use Intel Xeon E5-2699 v4 CPU. For FPGA experiments, we use Xilinx VU9P FPGA.

6.2 Overall Speedups on GPUs

We test all the benchmarks on GPUs with different test cases. The geometric mean speedups (normalized performance) of different test cases for each benchmark are shown in Figure 5. We compare FlexTensor with native PyTorch (without cuDNN) and cuDNN. CuDNN has no support for GMV, GMM, and BIL, so we only compare to cuBlas. FlexTensor outperforms both native PyTorch and cuDNN for most operators, and the average speedup of all benchmarks to cuDNN is 1.83x on V100, 1.68x on P100 and 1.71x on Titan X. FlexTensor achieves good speedups thanks to the exploration of huge schedule space (the size ranges from 3.9×10^9 to 2.4×10^{12}) for choosing proper schedule primitives and parameters as well as the target-specific optimization process for implementing high-performance schedules on GPU.

We notice FlexTensor falls short for operators T2D and T3D. The reason is that our implementation is based on the direct spatial convolution, while cuDNN uses implicit

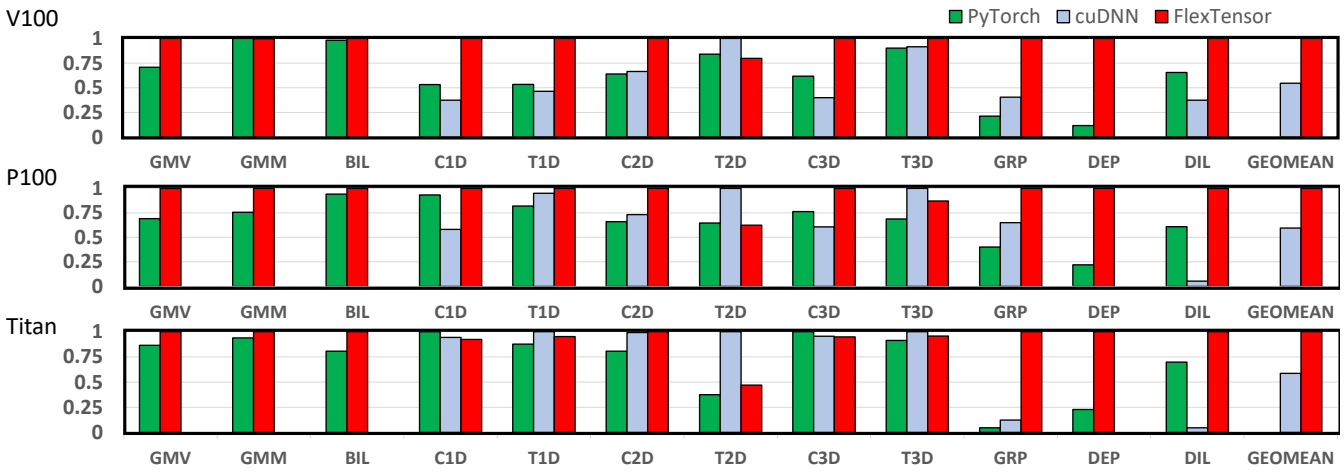
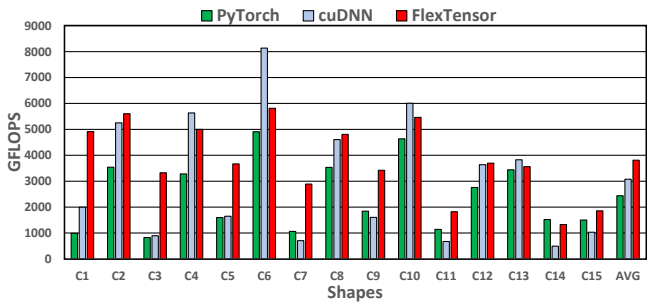
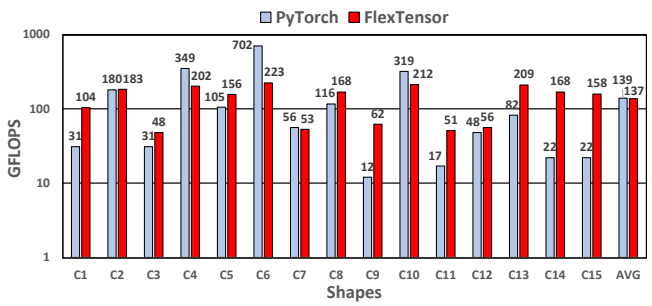


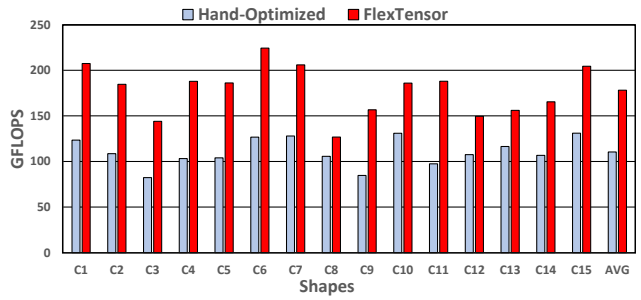
Figure 5. Normalized performance of native PyTorch, cuDNN and FlexTensor on different GPUs.



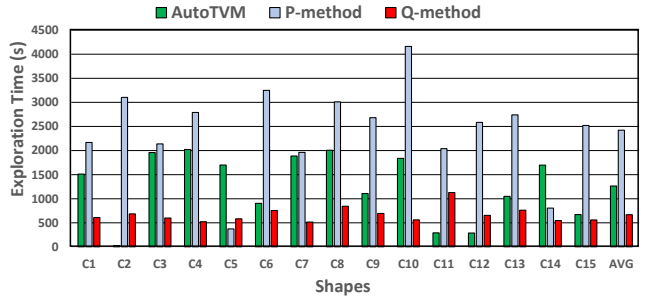
(a) Results of FlexTensor compared to PyTorch (without cuDNN) and cuDNN library on NVIDIA V100 GPU for 2D convolutions.



(b) Results of FlexTensor compared to PyTorch on Intel Xeon E5-2699 v4 for 2D convolutions.



(c) Results of FlexTensor comapred to hand-optimized OpenCL baselines on Xilinx VU9P FPGA for 2D convolutions.



(d) Exploration time comparison of AutoTVM, P-method, and Q-method.

Figure 6. Performance for 2D convolution on heterogeneous hardware.

GEMM algorithm and fast algorithms [52]. This needs algorithm level transformations, which are not supported by our schedule primitives.

Currently, some operators are not well supported by the libraries, which reflects the fact that it is very challenging to design a high-performance hand-tuned library. For example, GRP, DEP, and DIL have library support in PyTorch and cuDNN, but the performance is poor. Actually, in cuDNN, GRP and DIL reuse the kernels of C2D. For DEP operator, the implementation in cuDNN is even slower than that of PyTorch, and PyTorch doesn't use cuDNN for this operator internally. So we only compare to native PyTorch for DEP.

As shown in Figure 5, FlexTensor outperforms cuDNN for GRP and DIL, ranging from 1.54x to 21.35x speedup on GPUs. For DEP, the speedup of FlexTensor to PyTorch ranges from 4.39x to 8.53x.

6.3 Detailed Performance on Heterogeneous Hardware

We use C2D as a case study to evaluate FlexTensor on different hardware. The C2D test cases are taken from all the 15 different convolution layers in YOLO-v1 [45] as shown in Table 4, where C is input channel, K is output channel, H and W are input height and width, k is kernel size, and st is stride.

On GPU, FlexTensor exceeds native PyTorch and cuDNN for almost all the layers. The geometric mean speedup to PyTorch and cuDNN is 1.56x and 1.5x, respectively. The absolute performances are shown in Figure 6a. FlexTensor can achieve an average throughput of 3519.71 GFLOPS. FlexTensor can achieve high performance because it strikes a balance between inter-thread and intra-thread workload decomposition by exploring numerous schedules. FlexTensor can dynamically adapt to different input shapes and search for proper schedules, so for most layers, it is better than hand-optimized results. We also notice that FlexTensor fails to speed up some layers such as C4 and C6 because FlexTensor only uses a direct convolution algorithm for them, while cuDNN uses the Winograd algorithm, which greatly reduces the computation complexity [29].

Figure 6b presents the results on CPU in absolute performance. On CPU, FlexTensor also exceeds PyTorch (which is using MKL-DNN backend) for most layers. The geometric speedup to the library is 1.72x. FlexTensor uses NCHWc layout [17] for C2D to exploit the vectorization feature on CPU. FlexTensor can achieve high-performance thanks to efficient tiling and vectorization. We allow FlexTensor to automatically decide vectorization length to adapt to different platforms, and find that the schedules generated by FlexTensor all use a vectorization length of eight in our experiments because Xeon E5-2699 v4 uses AVX2 instructions and allows vectorization of at most eight floating-point operations.

On FPGA, the baseline uses schedules in [65] for performance optimization. The absolute results are shown in Figure 6c. The geometric speedup is 1.5x. The reasons behind the better performance of FlexTensor are two folds. On one hand, FlexTensor enables a larger design space exploration by solving an optimization problem under certain FPGA resource constraints. On the other hand, FlexTensor provides a better schedule strategy to reduce the off-chip memory access overhead by overlapping data communication and computation operations.

6.4 Performance for New Operators

We also evaluate FlexTensor using two new tensor operators, which lack of good library support. They are block circulant matrix [10, 56] (abbreviated as BCM) and shift operation [59] (abbreviated as SHO). BCM is used for embedded devices to prune parameters, and SHO is a parameter-free operator used in Shift-Net [63]. We compare FlexTensor with our hand-tuned implementation for these two operators. In our hand-tuned implementation, we use 4-level tiling with hand-optimized split factors and unroll loops to a maximum depth of 200. For BCM, FlexTensor achieves an average 2.11x speedup compared to the GPU baseline on V100. For SHO, FlexTensor achieves 1.53x speedup compared to the GPU baseline on Titan X. Therefore, users of these new operators can rely on FlexTensor to automatically generate a high-performance implementation.

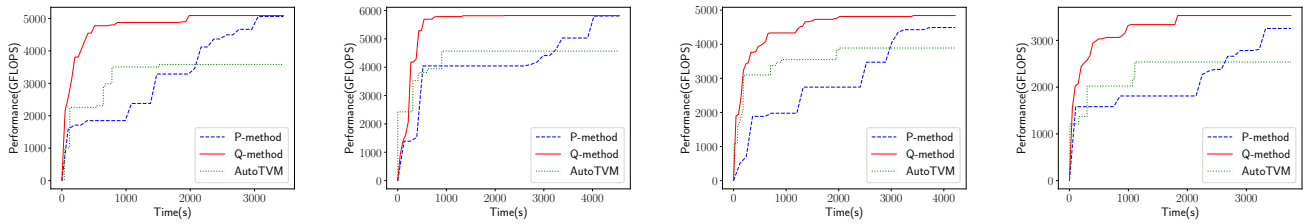
6.5 Comparison to State-of-the-Art

We compare with AutoTVM [9] to show the performance and exploration efficiency. AutoTVM requires users to write schedule-templates to optimize operators manually. AutoTVM generates schedule space according to the template and explores the space for optimized schedules. It uses XGBoost [6] to build a cost model to guide the searching process. To build such a model, AutoTVM tries thousands of implementations previously, and the data recorded from these trials are used to train the cost model. We compare FlexTensor with AutoTVM using C1D, T1D, C2D, T2D, C3D, T3D, and GRP. AutoTVM has no template support for C1D, T1D, C3D, T3D currently, so we design optimized templates for these operators manually. FlexTensor exceeds AutoTVM for all the operators except for T2D (0.95x), and the average speedup of these benchmarks is 2.21x. FlexTensor achieves better performance because it systematically explores the space of schedule primitives while AutoTVM relies on static templates. By comparing the schedule space size of AutoTVM and FlexTensor for the C2D operator, we find that FlexTensor explores a space 2027x larger than AutoTVM on average.

We name our Q-learning based method as *Q-method*. We compare *Q-method* with another method called *P-method*, which doesn't use Q-learning to guide search and tries all possible directions at each trial. For each trial, *P-method* explores all the directions for each starting point, while *Q-method* queries the Q-learning algorithm to get one direction for each starting point and only explores that direction. For AutoTVM, it randomly gets a new point to explore for each trial. We use C2D on V100 as a case study. Figure 6d shows the exploration time taken by *P-method*, *Q-method*, and AutoTVM when a similar performance is achieved. More clearly, we first run AutoTVM and let it converge to a stable performance, then run *P-method* and *Q-method* to reach a similar performance and record the time cost. On average, *Q-method* takes 27.6% and 52.9% of the time of *P-method* and AutoTVM to achieve a similar performance, respectively. We also collect the stable performance of *P-method* and *Q-method*. The final performance of *P-method* is 1.41x better than AutoTVM and *Q-method* is 1.54x better than AutoTVM.

Table 4. Configurations of 15 distinctive convolution layers in YOLO v1.

Name	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
C	3	64	192	128	256	256	512	256	512	512	1024	512	1024	1024	1024
K	64	192	128	256	256	512	256	512	512	1024	512	1024	1024	1024	1024
H/W	448	112	56	56	56	56	28	28	28	28	14	14	14	14	7
k, st	7,2	3,1	1,1	3,1	1,1	3,1	1,1	3,1	1,1	3,1	1,1	3,1	3,1	3,2	3,1



(a) C1 **(b)** C6 **(c)** C8 **(d)** C9

Figure 7. Performance vs. Exploration time

Figure 7 shows how the performance improves as the exploration time increases for test cases C1, C6, C8, and C9. As shown, *Q-method* always converges to a good performance in a short time, while *P-method* and AutoTVM take longer.

6.6 Case Study of DNNs

So far, we have evaluated FlexTensor using stand-alone operators only. Here, we test FlexTensor using two real-world DNNs: YOLO-v1 [45] (24 conv-layers; totally 30 layers) and OverFeat [48] (5 conv-layers; totally 8 layers) with batch size 1. FlexTensor can be used for full DNNs with multiple operators by partitioning the DNNs into sub-graphs and applying fusion techniques to fuse sub-graphs into operators [46]. The fused operators are fed to FlexTensor to generate schedules. We test the two networks on NVIDIA V100 GPU, and the end-to-end speedups of FlexTensor are 1.07x for YOLO-v1 and 1.39x for OverFeat compared to AutoTVM.

7 Related Works

Accelerating tensor computations on heterogeneous hardware is of critical importance and has attracted a lot of attention from both academia and industry. The best practice currently is still optimizing algorithms manually and developing libraries for different hardware. Most deep learning frameworks [1, 7, 23, 43] rely on these libraries to achieve high performance. On CPUs, high-performance library MKL [22] is designed to accelerate linear algebra applications, and MKL-DNN [20] is designed for deep learning applications. MKL-DNN integrates optimization techniques from [17], which uses JIT techniques to optimize CNN on Intel Xeon CPUs. Besides, SWIRL [54] can generate high-quality fused, vectorized, and parallelized code for DNN on CPUs. On GPUs, there are optimized libraries such as cuBlas [41] and cuDNN [11]. CuBlas [41] can accelerate linear algebra kernels to extreme high-performance, while cuDNN [11] accelerates deep learning applications by assembling a set of efficient algorithms such as Winograd [29] and FFT [34]. Optimizing algorithms on GPU is challenging as the final performance depends on both single thread performance and the thread level parallelism [31, 61, 62]. On FPGAs, prior works [33, 60] implement high-performance convolutions by manually designing hardware architecture and dataflow. All these works require manual design of high-performance implementation, which requires years of experience and expertise in both algorithms and hardware.

However, as deep-learning algorithms rapidly evolve, the long developing time and heavy labor cost of libraries become a bottleneck. To solve this problem, prior works design frameworks to lower the programming barrier using high-level abstractions and compilation techniques. ATLAS [58], BTO [4] FFTW [16], and Leo [15] use auto-tuning to achieve high-performance. Halide [44] can generate high-performance codes for image processing pipelines. TACO [27] focuses on sparse operators and generates codes from high-level expressions. TVM [8] generates codes for deep learning networks on many different kinds of hardware. HeteroCL [28] provides a code generation backend for FPGA. However, these frameworks still require the programmers to write the schedules manually, which is non-trivial.

Recently, researchers have tried to integrate automation into the code generation process. Halide auto-scheduler [2] provides fully automatic approaches by using tree searching and random programs, which can scale to whole graph level, but its main focus is still code-generation for image processing pipelines. PlaidML [19] leverages an analytical model to achieve high performance for tensor operators

automatically. Tensor Comprehensions [53] uses polyhedral model [5, 38, 55] to automatically optimize algorithms. PlaidML and Tensor Comprehensions are fully automatic, but they are limited to a narrow range of hardware and only achieve limited performance speedup. AutoTVM [9] can auto-tune operators on heterogeneous hardware with good portability, and achieve high performance for a wide range of operators. But AutoTVM still requires programmers to write schedule templates manually. Compared to AutoTVM, FlexTensor achieves better performance without human interference. FlexTensor can be used to generate arbitrary dense tensor operators.

8 Conclusion

Accelerating tensor computations on heterogeneous systems is urgently demanded in numerous applications. Designing high-performance libraries manually for different hardware is time-consuming and hardware-specific. In this paper, we propose an automatic optimization framework called FlexTensor, which uses pruning techniques and machine learning techniques to generate high-performance schedules for tensor computations on CPU, GPU, and FPGA. Experiments on CPU, GPU, and FPGA show that FlexTensor is able to achieve competitive or better performance than hand-tuned libraries. Compared to cuDNN on NVIDIA V100 GPU, the average speedup is 1.83x; compared to MKL-DNN on Intel Xeon CPU, the average speedup for 2D convolution is 1.72x; compared to OpenCL baselines on VU9P FPGA, the average speedup is 1.5x; compared to the state-of-the-art on NVIDIA V100 GPU, the average speedup is 2.21x.

Acknowledgments

We would first like to thank the anonymous reviewers for their suggestions. We would also like to thank Jonathan Ragan-Kelley for his thoughtful suggestions during the revision process. This work was partially supported by Beijing Natural Science Foundation (No. JQ19014), National Natural Science Foundation China (No. 61572048), and PKU-SenseTime Joint Lab.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12. https://doi.org/10.1145/3306346.3322967

[3] Abien Fred Agarap. 2018. Deep Learning using Rectified Linear Units (ReLU). *CoRR* abs/1803.08375 (2018). arXiv:1803.08375 http://arxiv.org/abs/1803.08375

[4] Geoffrey Belter, Elizabeth R. Jessup, Ian Karlin, and Jeremy G. Siek. 2009. Automating the generation of composed linear algebra kernels. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*. https://doi.org/10.1145/1654059.1654119

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 101–113. https://doi.org/10.1145/1375581.1375595

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. 785–794. https://doi.org/10.1145/2939672.2939785

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[9] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 3393–3404. http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs

[10] Yu Cheng, Felix X. Yu, Rogério Schmidt Feris, Sanjiv Kumar, Alok N. Choudhary, and Shih-Fu Chang. 2015. An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 2857–2865. https://doi.org/10.1109/ICCV.2015.327

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[12] François Chollet. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. *CoRR* abs/1610.02357 (2016). arXiv:1610.02357 http://arxiv.org/abs/1610.02357

[13] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees A. Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[14] Leonardo Dagum and Ramesh Menon. 1988. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering* 5 Issue 1 (1988).

[15] Naila Farooqui, Christopher J. Rossbach, Yuan Yu, and Karsten Schwan. 2014. Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications. In *2014 Conference on Timely Results in Operating Systems, TRIOS '14, Broomfield, CO, USA, October 5,*

2014. https://www.usenix.org/conference/trios14/technical-sessions/presentation/farooqui

[16] Matteo Frigo and Steven G. Johnson. 1998. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, Seattle, Washington, USA, May 12-15, 1998*. 1381–1384. https://doi.org/10.1109/ICASSP.1998.681704

[17] Evangelos Georganas, Sasikanth Avancha, Kunal Banerjee, Dhiraj D. Kalamkar, Greg Henry, Hans Pabst, and Alexander Heinecke. 2018. Anatomy of high-performance deep learning convolutions on SIMD architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. 66:1–66:12. http://dl.acm.org/citation.cfm?id=3291744

[18] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. 2017. Learning Spatio-Temporal Features with 3D Residual Networks for Action Recognition. In *2017 IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2017, Venice, Italy, October 22-29, 2017*. 3154–3160. https://doi.org/10.1109/ICCVW.2017.373

[19] Intel(R) PlaidML https://ai.intel.com/plaidml. [n.d.]. https://ai.intel.com/plaidml

[20] Inter(R) MKL-DNN https://github.com/intel/mkl dnn. [n.d.]. https://github.com/intel/mkl-dnn

[21] Torch/CUNN https://github.com/torch/cunn. [n.d.]. https://github.com/torch/cunn

[22] Intel(R) MKL https://software.intel.com/en us/mkl. [n.d.]. https://software.intel.com/en-us/mkl

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[24] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay V. Rajopadhye, and Michelle Mills Strout. 2007. Multi-level tiling: M for the price of one. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*. 51. https://doi.org/10.1145/1362622.1362691

[25] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. *CoRR* abs/1408.5882 (2014). arXiv:1408.5882 http://arxiv.org/abs/1408.5882

[26] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. 1983. Optimization by Simulated Annealing. In *Science, 220(4598):671-680*.

[27] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *PACMPL* 1, OOPSLA (2017), 77:1–77:29. https://doi.org/10.1145/3133901

[28] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*. 242–251. https://doi.org/10.1145/3289602.3293910

[29] Andrew Lavin. 2015. Fast Algorithms for Convolutional Neural Networks. *CoRR* abs/1509.09308 (2015). arXiv:1509.09308 http://arxiv.org/abs/1509.09308

[30] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

[31] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A coordinated tiling and batching framework for efficient GEMM on GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*. 229–241. https://doi.org/10.1145/3293883.3295734

[32] Yun Liang, Shuo Wang, and Wei Zhang. 2018. FlexCL: A Model of Performance and Power for OpenCL Workloads on FPGAs. *IEEE Trans. Computers* 67, 12 (2018), 1750–1764. https://doi.org/10.1109/TC.2018.2840686

[33] Liqiang Lu and Yun Liang. 2018. SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. 135:1–135:6. https://doi.org/10.1145/3195970.3196120

[34] Michaël Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast Training of Convolutional Networks through FFTs. *CoRR* abs/1312.5851 (2013). arXiv:1312.5851 http://arxiv.org/abs/1312.5851

[35] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2019. FBLAS: Streaming Linear Algebra on FPGA. *CoRR* abs/1907.07929 (2019). arXiv:1907.07929 http://arxiv.org/abs/1907.07929

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236

[37] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4 (2016), 83:1–83:11. https://doi.org/10.1145/2897824.2925952

[38] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 429–443. https://doi.org/10.1145/2694344.2694364

[39] Makoto Nakatsuji, Qingpeng Zhang, Xiaohui Lu, Bassem Makni, and James A. Hendler. 2017. Semantic Social Network Analysis by Cross-Domain Tensor Factorization. *IEEE Trans. Comput. Social Systems* 4, 4 (2017), 207–217. https://doi.org/10.1109/TCSS.2017.2732685

[40] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. *ACM Queue* 6, 2 (2008), 40–53. https://doi.org/10.1145/1365490.1365500

[41] NVIDIA(R). [n.d.]. CUBLAS Library https://www.nvidia.com/. https://developer.download.nvidia.cn/compute/DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf

[42] Evangelos E. Papalexakis, Christos Faloutsos, and Nicholas D. Sidiropoulos. 2017. Tensors for Data Mining and Data Fusion: Models, Applications, and Scalable Algorithms. *ACM TIST* 8, 2 (2017), 16:1–16:44. https://doi.org/10.1145/2915921

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 519–530. https://doi.org/10.1145/2491956.2462176

[45] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 779–788. https://doi.org/10.1109/CVPR.2016.91

[46] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level IR for Deep Learning. *CoRR* abs/1904.08368 (2019). arXiv:1904.08368 http://arxiv.org/abs/1904.08368

[47] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[48] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. 2014. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. http://arxiv.org/abs/1312.6229

[49] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David M. Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. 97–108. https://doi.org/10.1109/ISCA.2014.6853196

[50] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning - an introduction*. MIT Press. http://www.worldcat.org/oclc/37293240

[51] Du Tran, Lubomir D. Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2015. Learning Spatiotemporal Features with 3D Convolutional Networks. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 4489–4497. https://doi.org/10.1109/ICCV.2015.510

[52] Nicolas Vasilache, Jeff Johnson, Michaël Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. 2015. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.7580

[53] 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. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[54] Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. 2019. SWIRL: High-performance many-core CPU code generation for deep neural networks. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1275–1289. https://doi.org/10.1177/1094342019866247 arXiv:https://doi.org/10.1177/1094342019866247

[55] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *TACO* 9, 4 (2013), 54:1–54:23. https://doi.org/10.1145/2400682.2400713

[56] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. 2018. C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*. 11–20. https://doi.org/10.1145/3174243.3174253

[57] Christopher J. C. H. Watkins and Peter Dayan. 1992. Technical Note Q-Learning. *Machine Learning* 8 (1992), 279–292. https://doi.org/10.1007/BF00992698

[58] R. Clint Whaley. 2011. ATLAS (Automatically Tuned Linear Algebra Software). In *Encyclopedia of Parallel Computing*. 95–101. https://doi.org/10.1007/978-0-387-09766-4_85

[59] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter H. Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2018. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 9127–9135. https://doi.org/10.1109/CVPR.2018.00951

[60] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. 62:1–62:6. https://doi.org/10.1145/3061639.3062244

[61] Xiaolong Xie, Yun Liang, Xiuhong Li, and Wei Tan. 2019. CuLDA: Solving Large-scale LDA Problems on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019*. 195–205. https://doi.org/10.1145/3307681.3325407

[62] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*. 395–406. https://doi.org/10.1145/2830772.2830813

[63] Zhaoyi Yan, Xiaoming Li, Mu Li, Wangmeng Zuo, and Shiguang Shan. 2018. Shift-Net: Image Inpainting via Deep Feature Rearrangement. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIV*. 3–19. https://doi.org/10.1007/978-3-030-01264-9_1

[64] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. *CoRR* abs/1212.5701 (2012). arXiv:1212.5701 http://arxiv.org/abs/1212.5701

[65] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks *(FPGA '15)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[66] Jialiang Zhang and Jing Li. 2017. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*. 25–34. http://dl.acm.org/citation.cfm?id=3021698