arXiv:2002.12418v1 [cs.CV] 27 Feb 2020

MNN: A Universal and Efficient Inference Engine

Xiaotang Jiang [1] **Huan Wang** [2] **Yiliu Chen** [1] **Ziqi Wu** [1] **Lichuan Wang** [1] **Bin Zou** [1] **Yafeng Yang** [1] **Zongyang Cui** [1] **Yu Cai** [1] **Tianhang Yu** [1] **Chengfei Lv** [1] **Zhihua Wu** [1]

Abstract

Deploying deep learning models on mobile devices draws more and more attention recently. However, designing an efficient inference engine on devices is under the great challenges of model compatibility, device diversity, and resource limitation. To deal with these challenges, we propose Mobile Neural Network (MNN), a universal and efficient inference engine tailored to mobile applications. In this paper, the contributions of MNN include: (1) presenting a mechanism called pre-inference that manages to conduct runtime optimization; (2) delivering thorough kernel optimization on operators to achieve optimal computation performance; (3) introducing backend abstraction module which enables hybrid scheduling and keeps the engine lightweight. Extensive benchmark experiments demonstrate that MNN performs favorably against other popular lightweight deep learning frameworks. MNN is available to public at: https://github.com/alibaba/MNN.

1 Introduction

Deep learning has become the de facto method for artificial intelligence in various tasks including computer vision, user intention recognition (Guo et al., 2019), and autopilot (LeCun et al., 2015). As edge devices (*e.g.*, smartphones, IoT devices, wearable devices) are ubiquitous now, deep learning on edge devices, especially mobile devices, attracts growing attention (Shi et al., 2016). There are many advantages for deep learning on mobiles, for example, low latency, privacy protection, and personalized service. To make full use of on-device deep learning technology, inference engines tailored to mobile devices have been developed and extensively used in mobile applications, for example, TF-Lite (Google, 2017a)(Google, 2017a), NCNN (Tencent, 2017), CoreML (Apple, 2017), *etc*.

The major challenges for mobile inference engines can be categorized into three aspects: model compatibility, device diversity, and resource limitation.

(1) Model compatibility. Most deep learning models deployed on mobile devices are trained from well-known deep learning frameworks such as TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), Caffe (Jia et al., 2014), CNTK (Yu et al., 2014), MXNet (Chen et al., 2015). As a result, it is a basic requirement that an inference engine should have the model compatibility for different formats and different operators. More importantly, the engine should also allow for proper scalability to support new operators emerging in the future.

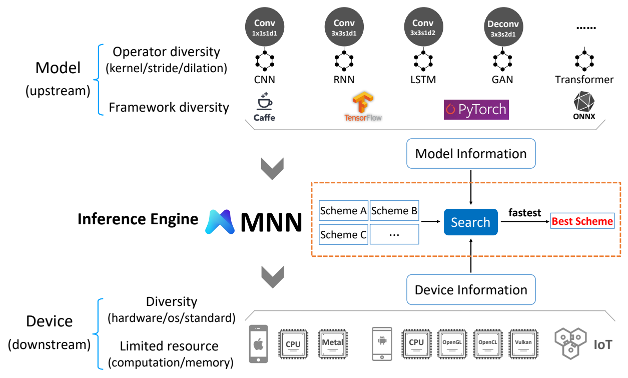

Figure 1. Overview of the proposed Mobile Neural Network.

(2) Device diversity. Almost every well-known mobile applications are used extensively on various devices, ranging from low-end equipment with single-core CPU to high-end equipment with co-processors such as Apple Neural Engine (ANE). To achieve great performance on various devices, mobile inference engines have to take hardware architectures or even device vendors (like ARM Mali GPU or Qualcomm Adreno GPU) into consideration. In addition, the engine is also expected to take care of the software diversity problem well, such as different operating systems (Android/iOS/embedded OS) and different solution standards (OpenCL (Khronos, 2009)/OpenGL (Khronos, 1992)/Vulkan (Khronos, 2015) for Android GPU).

[1]Alibaba Group, Hangzhou, China. [2]Department of Electrical and Computer Engineering, Notheastern University, Boston, USA. Correspondence to: Ziqi Wu <mingyi.wzq@alibaba-inc.com>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020. Copyright 2020 by the author(s).

(3) Resource limitation. Despite the rapid hardware development, memory and computation power are still constrained on mobile devices and are orders of magnitude lower than their desktop and server counterparts.

Concluded from the challenges above, a good mobile inference engine should have the following two properties: (1) *Universality* to address both model compatibility and device diversity; (2) *Efficiency* to inference models on devices with great performance and to use as little memory and energy consumption as possible.

To satisfy above properties, we introduce a new mobile inference engine named *Mobile Neural Network (MNN)*. Our contributions can be summarized as follows:

- We present a mechanism called pre-inference which manages to perform runtime optimization through online cost evaluation and optimal scheme selection.
- We deliver in-depth kernel optimization by utilizing improved algorithm and data layout to further boost the performance of some widely-used operations.
- We propose backend abstraction module to enable hybrid scheduling and keep engine itself as lightweight as possible. Integrating MNN into applications only increases the binary size by $400 \sim 600$KB.

It should be noted that MNN has been extensively adopted in many mobile applications. Meanwhile, we open-source the entire project to enrich the community and hope to involve more people to make it better.

2 Related Work

With the present rise of demand for on-device deep learning, much attention has been spent on mobile inference solutions, especially from several major companies.

CoreML is a framework of Apple to integrate machine learning models into iOS software applications, which can leverage multiple hardware like CPU, GPU, and ANE. For Android smartphones, Google also provides their own solution for on-device inference, *i.e.*, ML-kit and Neural Networks API (NNAPI) (Google, 2016). However, the main drawback of these solutions lies in their limited universality. For example, CoreML requires iOS 11+ and NNAPI requires Android 8.1+, which exclude many existing mobile phones and the embedded devices.

In late 2017, Google released TensorFlow Lite (TF-Lite) (Google, 2017a), an efficient mobile deep learning framework. TF-Lite is optimized for less powerful devices such as mobile phones and embedded devices. Almost at the same time, Facebook released Caffe2 (Paszke et al., 2017) to help developers deploy deep learning models on mobile devices. Both of them support a wide range of devices and many applications have already been developed based on them. However, on-device inference with TF-Lite or Caffe2 may sometimes go against the goal to be lightweight. For example, TF-Lite utilizes Accelate, Eigen, and OpenBLAS libraries for floating-point acceleration while Caffe2 depends on Eigen to accelerate matrix operations. Integrating mobile deep learning frameworks with these dependencies will bloat the binary size of mobile applications and bring in unnecessary overhead.

Many efforts have been devoted to solving the problem above. NCNN (Tencent, 2017), MACE (Xiaomi, 2018), and Anakin (Baidu, 2018) are the representatives. They follow a paradigm of what we call *manually search* or *non-automated search*. In this paradigm, operators are optimized *case by case* through carefully-designed assembly instructions and do not rely on any external libraries. For example, a unique program function is implemented for a 3×3 convolution of stride 1; another function has to be implemented separately for a 3×3 convolution of stride 2. This kind of implementation allows the mobile inference engine to be lightweight and efficient. However, the case-by-case optimization is time-consuming and can hardly cover all new emerging operators.

In stark contrast to the manual search, there is another philosophy on the opposite, which we call *automated search*, pioneered by TVM (DMLC, 2016). TVM is an open deep learning compiler stack that can compile various deep learning models into libraries in an end-to-end manner. Not only does it resolve the redundant dependency problem, but also provides graph-level and operator-level optimization customized for the model and backend. As a result, the performance of TVM is pretty encouraging and scalable in terms of both model and device diversity. However, these advantages come at a price. The runtime library generated by TVM is *model-specific*, which means if we want to update the model (which can be very common and frequent for many AI-driven software applications), we need to re-generate the code and release a new version. This mechanism bears a burden of cost and sometimes it is impracticable for mobile applications. In this paper, we are motivated to develop a semi-automated search architecture featured by enhanced universality and better performance in mobile deployment.

It is worthy of mention that there are some other works related to on-device deep learning, *e.g.*, computational graph DSL (Domain-Specific Language) (Abadi et al., 2016; Bastien et al., 2012) to perform optimization in the graph level or operator fusion and replacement (Google, 2017b; Wei et al., 2017). These works are orthogonal to our contributions in this paper and are partially referenced by MNN.

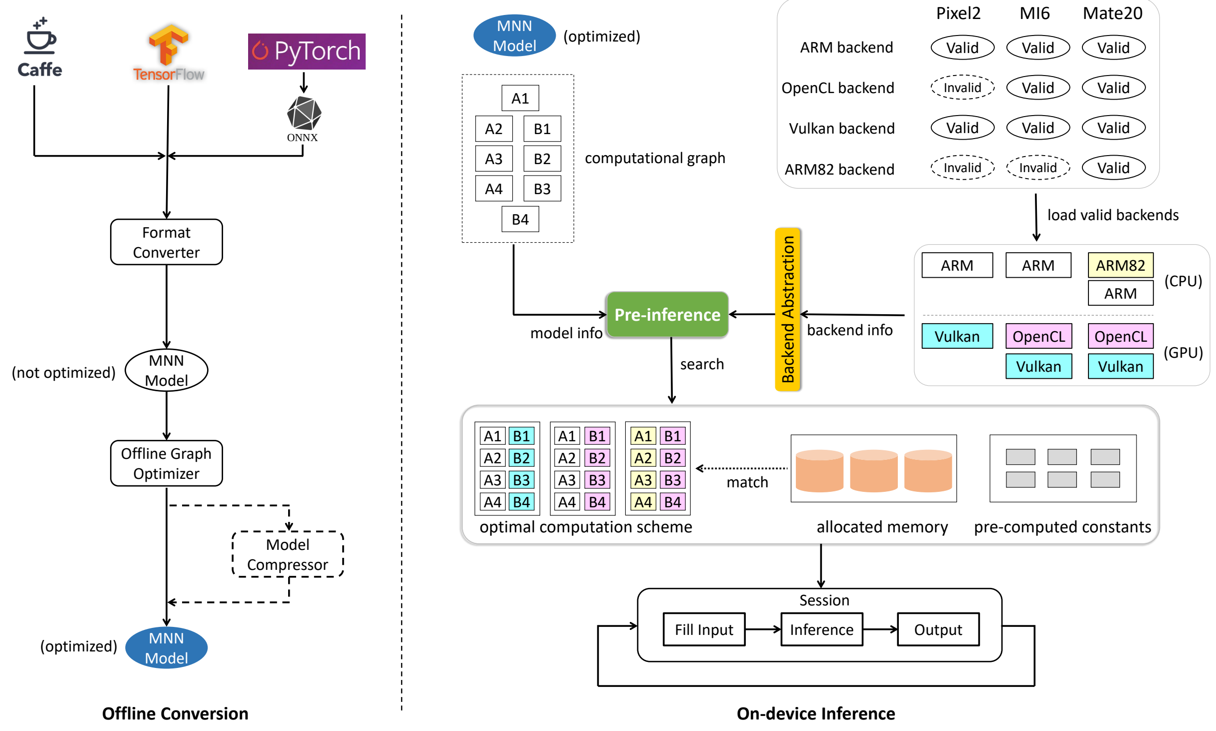

Figure 2. Architecture detail of the proposed mobile inference engine Mobile Neural Network (MNN).

3 Mobile Neural Network (MNN)

3.1 Overview of MNN

The basic workflow of MNN is composed of two parts, offline conversion and on-device inference. Figure 2 summarizes the entire architecture of MNN and this section briefly shows how MNN optimizes the workflow by walking through its components.

For the part of offline conversion, the converter first takes models as input from different deep learning frameworks and transforms them to our own model format (`.mnn`). Meanwhile, some basic graph optimizations are performed, such as operator fusion (Ashari et al., 2015), replacement, and model quantization (Rastegari et al., 2016).

For on-device inference, three modules are involved: pre-inference, operator-level optimization, and backend abstraction. For each operator, the pre-inference module offers a *cost evaluation mechanism*, which combines the information (*e.g.*, input size, kernel shape) with backend properties (*e.g.*, number of cores, hardware availability) to dynamically determine the optimal computation solution from a solution pool. Then the operator-level optimization module utilizes advanced algorithms together with techniques like SIMD (Single Instruction Multiple Data), pipelining to further boost the performance.

Moreover, MNN supports various hardware architectures as backend. Since no single standard fits all hardware specifications, MNN supports different software solutions such as OpenCL, OpenGL, Vulkan, and Metal. All backends are implemented as independent components and a set of uniform interface is provided with backend abstraction module to hide raw details (*e.g.*, memory management on heterogeneous backends).

With the proposed architecture, not only do we achieve high performance for on-device inference, but also make it easy to extend MNN to more ongoing backends (such as TPU, FPGA, *etc.*). In the rest of this section, we present more details of the architecture of MNN.

3.2 Pre-inference

Pre-inference is the fundamental part of the proposed semi-automated search architecture. It takes advantage of a common phenomenon that the input size is typically fixed (or can be pre-processed into a target size) in many deep learning applications. Based on this, memory usage and computational cost can be determined *ahead of* formal inferences. Thus, some optimization techniques like memory pre-allocation and reuse can be conducted to further improve the performance. Details of pre-inference can be specified into two parts: computation scheme selection and preparation-execution decoupling.

Computation scheme selection. We propose a *cost evaluation mechanism* to select the optimal scheme from the scheme pool, which takes into consideration both the algorithm implementation and the backend characteristics,

$$C_{\text{total}} = C_{\text{algorithm}} + C_{\text{backend}}, \tag{1}$$

where C stands for the cost.

(1) Take convolution scheme pool as an example, in which there are generally two fast implementation algorithms to choose: sliding window and Winograd (Lavin & Gray, 2016). Our general idea is to dynamically choose the algorithm that minimizes the computational cost based on different convolution settings. Therefore, the optimal computation scheme to minimize the $C_{\text{algorithm}}$ term can be determined as follows:

1. If kernel size $k = 1$, it is just a matrix multiplication. Strassen algorithm (Strassen, 1969) will be applied.
2. If kernel size $k > 1$, we employ Winograd to transform convolution into matrix multiplication. Winograd convolution has many sub-options for different output tile size n, with regard to which, the arithmetic cost can be formulated as follows. Based on the cost, we choose the optimal output tile size $\hat{n}$ that minimizes the cost C,

$$\begin{aligned} C(n) &= 2i_c(n+k-1)^3 \\ &\quad + i_c o_c (n+k-1)^2 \\ &\quad + n(n+k-1)(2n+k-1), \\ \Rightarrow \hat{n} &= \arg\min_n C(n). \end{aligned} \tag{2}$$

3. Then, if $\hat{n}$ equals to 1, sliding window is picked out; otherwise, we adopt Winograd convolution.

This cost evaluation mechanism for convolution is summarized as

$$\text{Scheme} = \begin{cases} \text{sliding window}, & \text{if } k > 1 \text{ and } \hat{n} = 1, \\ F(\hat{n} \times \hat{n}, k \times k), & \text{if } k > 1 \text{ and } \hat{n} > 1. \end{cases} \tag{3}$$

(2) Then next question is how we determine the second term C_{backend} in Equation 1. Generally, the idea is to sum up the time cost of all the operators on different backends, then choose the backend with the minimal cost,

$$C_{\text{backend}} = \sum_{\text{op}} C_{\text{op}}, \tag{4}$$

where C denotes the cost as before and op represents the operator. The key of backend cost evaluation is apparently to obtain the C_{op} term, which relies on the different backend cases. If an operator is not supported on a GPU backend of interest (*e.g.*, OpenCL/OpenGL/Vulkan), it will be scheduled to run on CPU. The cost of an operator on CPU or GPU can be formulated as

$$C_{\text{op}} = \begin{cases} \frac{\texttt{MUL}}{\texttt{FLOPS}} \times 1000, & \text{if on CPU}, \\ \frac{\texttt{MUL}}{\texttt{FLOPS}} \times 1000 + t_{\text{schedule}}, & \text{if on GPU}, \end{cases} \tag{5}$$

where `MUL` stands for the number of multiplication, indicating the computational complexity of an operator. `FLOPS` (FLoating-Operations Per Second) is a common index to quantify the computation power of CPU or GPU. As we see, the main difference of GPU from CPU as the backend is that it has an additional term t_{schedule}, which accounts for the cost to prepare the command buffer and command description for GPU. Note that, `FLOPS` and t_{schedule} are known constants for a specific backend (please refer to the Appendix for how to determine them in detail).

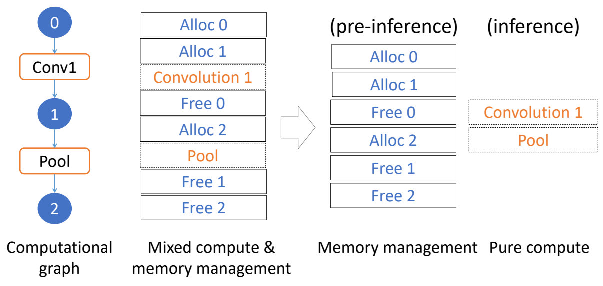

Figure 3. Memory optimization of MNN: decouple memory allocation from computation.

Table 1. Inference time (ms) comparison of different computation schemes. "Sliding" represents "sliding window" and "WinoMin/-Max" means "Winograd convolution with minimal/maximal block size". The four numbers in the convolution argument setting successively mean kernel size, input channel, output channel, and the spatial size of input, respectively. The best results are **in bold** and the second best are <u>underlined</u>.

Scheme	Convolution argument setting		
	$(2, 3, 16, 224)$	$(2, 512, 512, 16)$	$(3, 64, 64, 112)$
Sliding	**32.1**	895.1	895.1
WinoMin	42.2	<u>287.7</u>	389.8
WinoMax	57.3	539.3	<u>237.4</u>
Ours	<u>32.7</u>	**286.0**	**236.4**

Table 1 shows the performance comparison between fixed scheme and our scheme selection under different convolution settings. We can see that each scheme of sliding window or Winograd can be especially fit in certain case, but not universally good. Our method gets the best (or comparable to the best) in different cases, partly showing the claimed universal efficiency.

Preparation-execution decoupling. During the program execution, the computation is normally interlaced with the

Table 2. Inference time (ms) comparison without (w/o) and with (w/) the proposed preparation-execution decoupling. Hardware setting: (1) MI6 – CPU: Kryo 280, GPU: Adreno 540; (2) P10 – CPU: Cortex A73, GPU: Mali G71.

Scheme	CPU (4 threads)	GPU (Vulkan)
MI6 (w/o)	30.9	63.6
MI6 (w/)	**28.9** ($\downarrow$ 6.5%)	**15.8** ($\downarrow$ 75.2%)
P10 (w/o)	29.0	41.0
P10 (w/)	**26.8** ($\downarrow$ 7.6%)	**20.7** ($\downarrow$ 49.5%)

memory allocation and freeing. Speaking of mobile applications, however, time spent on memory management cannot be ignored. Since input size is determined or can be pre-processed into a target size, MNN can infer the exact required memory for the entire graph by virtually walking through all operations and summing up all allocation and freeing. In this sense, MNN pre-allocates required memory as a memory pool during the pre-inference stage and reuses it in the following inference sessions. The whole process is illustrated in Figure 3.

It should also be noted that by decoupling preparation from execution, MNN achieves better performance when the selected scheme is GPU-related. This is because setting up command buffer and its related command descriptions is, to some extent, time-consuming and thus have negative impact on the inference performance.

This simple idea can be quite effective in practice. The inference time using this technique can drop by about $7\% \sim 8\%$ on CPU and $50\% \sim 75\%$ on GPU. More details can be found in Table 2.

3.3 Kernel Optimization

Kernel is the detailed implementation of an operator, whose optimization can be specified into two ways, *i.e.*, the algorithm and the schedule (Jiang et al., 2018). In other words, we need to choose the optimal algorithm with the lowest arithmetic complexity and make the most of the available hardware resources for fastest execution.

3.3.1 Winograd optimization

Winograd algorithm is a well-known minimal filtering algorithm proposed by Shmuel Winograd (Winograd, 1980) and has been used to accelerate convolution in DNNs (Lavin & Gray, 2016).

Given an input feature map of size $[i_w, i_h, i_c]$, its output feature map $[o_w, o_h, o_c]$, Winograd convolution can be formulated as

$$Y = A^T \left[\sum_{\text{channel}} (GWG^T) \odot (B^T XB) \right] A, \tag{6}$$

where G, B, A are the transformation matrices for kernel W (spatial size $[k, k]$), input X (spatial size $[n+k-1, n+k-1]$), and output Y (spatial size $[n, n]$), respectively. These three matrices only depend on the shape of W and X.

We optimize Winograd convolution based on a proposed Winograd generator with popular parallel computing techniques such as pipelining and SIMD.

(1) Block division and pipelining. In Winograd convolution (Lavin & Gray, 2016), X is a small tile instead of the whole input feature map, thus leaving us the first issue: block division, *i.e.*, how to determine n.

To resolve it, we divide the block from the *output* perspective. For the output of size $[o_w, o_h, o_c]$, let T be the multiplier for parallel computing (namely, we compute T output blocks per time). Then there should be

$$T = \left\lfloor \frac{o_w o_h}{\hat{n}^2} \right\rfloor, \tag{7}$$

where $\hat{n}$ is the aforementioned optimal output tile size decided at the pre-inference stage (Equation 2).

When blocks are computed together, we must try our best to avoid pipeline stalls to hide latency. The trick is well-known as avoiding data dependence in pipeline (Kennedy & Allen, 2001). This is achieved by careful *assembly instruction rearrangement* in MNN.

(2) Hadamard product optimization. Hadamard product is an essential step in Winograd convolution (see Equation 6). However, it has a problem that memory access takes much time, draging down the whole acceleration.

From Equation 6, it can be found that the summation plus the Hadamard product can be transformed into a dot product. Combining many dot product together gives us matrix multiplication, which is a good indicator for parallelism and amortizing memory access overhead. In this spirit, we propose to transform the Hadamard product to matrix multiplication building upon *data layout re-ordering*. The consequent new data layout is called NC4HW4 (DMLC, 2016; Apple, 2014). Briefly, NC4HW4 is a data layout re-ordering method that splits out V ($V = 4$ in this paper) data elements as a unit to make a new dimension for a tensor. The V elements are placed contiguously in memory so as to leverage the vector registers in CPUs to compute V data in a single instruction (*i.e.*, SIMD). After this re-ordering, the Winograd convolution is illustrated as Figure 4.

(3) Winograd generator. Most existing inference frameworks using Winograd (Google, 2017a; Tencent, 2017; Xiaomi, 2018) hardcode the A, B, G matrices for common kernel and input sizes in their source codes (Google; Xiaomi), which has relatively poor scalability in face of new cases. In contrast, MNN keeps a universal interface through a proposed *Winograd generator*, allowing for Winograd

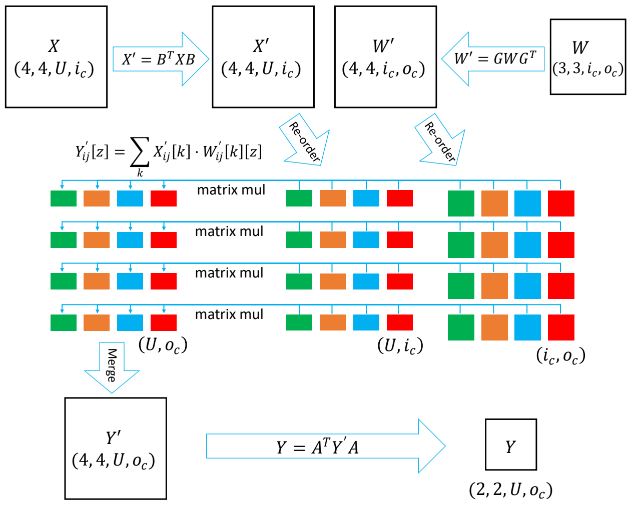

Figure 4. Illustration of the optimized Winograd algorithm in MNN (*best viewed in color*).

convolution of *any* kernel and input sizes. We adopt the following formula to generate the A, B matrices,

$$x \cdot (x-f)(x+f) \cdot (x-2f)(x+2f) \cdots \\ (x - \frac{(n+k-1)f}{2})(x + \frac{(n+k-1)f}{2}), \tag{8}$$

where f is a scalar used to minimize the numerical errors. We set $f = 0.5$ in this paper.

3.3.2 *Large matrix multiplication optimization*

As aforementioned (Section 3.2), convolution operations with kernel size 1 are converted into large matrix multiplication in MNN, where comes the Strassen algorithm (Strassen, 1969) for acceleration. To the best of our knowledge, MNN is the *first* mobile inference engine to adopt Strassen algorithm to accelerate large matrix multiplication.

Strassen is a fast algorithm that trades expensive multiplications with cheap additions, whose acceleration effect is maximized when it is applied *recursively* (Blahut, 2010). In practice, we need to decide when the recursion should stop. Conditioned on a fact that on modern processors, multiplication is nearly the same costly as addition, so we can just compare their cost via their numbers. In MNN, for a matrix multiplication of size $[n,k] \times [k,m] \Rightarrow [n,m]$, the number of direct multiplication is mnk, while it only needs $7 \cdot \frac{m}{2}\frac{n}{2}\frac{k}{2}$ multiplications using Strassen. The extra cost of using Strassen is 4 matrix additions of size $[\frac{m}{2}, \frac{k}{2}]$, 4 matrix additions of size $[\frac{n}{2}, \frac{k}{2}]$, and 7 matrix additions of size $[\frac{m}{2}, \frac{n}{2}]$. Therefore, the recursion goes on only when the benefit is over cost, formulated as

$$mnk - 7 \cdot \frac{m}{2}\frac{n}{2}\frac{k}{2} > 4 \cdot \frac{m}{2}\frac{k}{2} + 4 \cdot \frac{n}{2}\frac{k}{2} + 7 \cdot \frac{m}{2}\frac{n}{2}. \tag{9}$$

Table 3. Time cost (ms) of matrix multiplication comparison on P10 without (w/o) and with (w/) the optimized Strassen algorithm. Matrix size of (a, b, c) means the matrix multiplication of size [a, b] times [b, c].

Matrix size	w/o Strassen	w/ Strassen
$(256, 256, 256)$	23	23
$(512, 512, 512)$	191	**176** ($\downarrow$ 7.9%)
$(512, 512, 1024)$	388	**359** ($\downarrow$ 7.5%)
$(1024, 1024, 1024)$	1501	**1299** ($\downarrow$ 13.5%)

```
class XPUBackend final : public Backend {
    XPUBackend(MNNForwardType type, MemoryMode mode);
    virtual ~XPUBackend();
    virtual Execution* onCreate(const vector<Tensor*>& inputs,
                                const vector<Tensor*>& outputs, const MNN::Op* op);
    virtual void onExecuteBegin() const;
    virtual void onExecuteEnd() const;
    virtual bool onAcquireBuffer(const Tensor* tensor, StorageType storageType);
    virtual bool onReleaseBuffer(const Tensor* tensor, StorageType storageType);
    virtual bool onClearBuffer();
    virtual void onCopyBuffer(const Tensor* srcTensor, const Tensor* dstTensor) const;
}
```

Figure 5. Backend class in MNN (*best viewed in color*).

Once this inequation cannot hold, the recursion of Strassen should stop.

Table 3 demonstrates the advantage of Strassen compared with direct matrix multiplication with different matrix sizes. We can see that the Strassen method outperforms the direct one by $7.5\% \sim 13.5\%$ improvement.

3.4 Backend Abstraction

Backend abstraction module is introduced to make all the hardware platforms (*e.g.*, GPU, CPU, TPU) and software solutions (*e.g.*, OpenCL, OpenGL, Vulkan) encapsulated into a uniform `Backend` class. Through `Backend` class, resource management, memory allocation, and scheduling are disentangled with the concrete operator implementations.

The `Backend` class consists of several abstract functions, as shown in Figure 5. As for memory management, `onAcquireBuffer` is responsible for allocating new memory for tensors and `onReleaseBuffer` for releasing them. For operator implementation, `onCreate` is designed to create execution instance for each operator.

Advantages of this specific module are three-folds.

(1) Reduce complexity. A large number of operators and innumerable devices make the operator optimization a non-trivial task. The major challenge lies in the fact that heterogeneous backends typically have different ways to manage

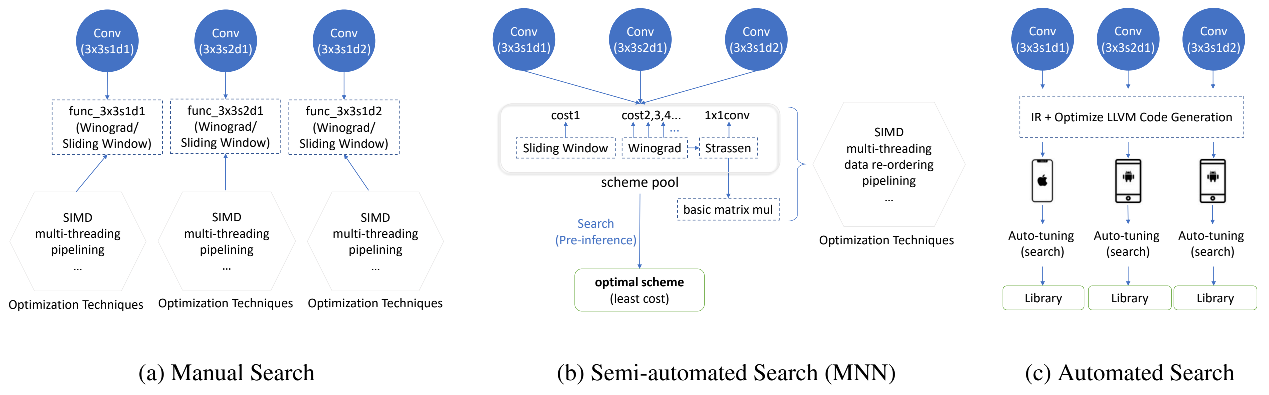
Conv (3x3s1d1)
Conv (3x3s2d1)
Conv (3x3s1d2)
func_3x3s1d1 (Winograd/ Sliding Window)
func_3x3s2d1 (Winograd/ Sliding Window)
func_3x3s1d2 (Winograd/ Sliding Window)
SIMD multi-threading pipelining ...
Optimization Techniques
(a) Manual Search
cost1
cost2,3,4...
1x1conv
Sliding Window
Winograd
Strassen
scheme pool
basic matrix mul
Search (Pre-inference)
optimal scheme (least cost)
SIMD multi-threading data re-ordering pipelining ...
Optimization Techniques
(b) Semi-automated Search (MNN)
IR + Optimize LLVM Code Generation
Auto-tuning (search)
Library
(c) Automated Search

Figure 6. Design paradigm comparison between MNN and the other two kinds of popular mobile inference engines.

resources, *e.g.*, allocating/freeing memory and scheduling data. Dealing with these issues makes implementation error-prone and cumbersome. The `Backend` class uniformly manages resource loading such as GPU shader and completes the optimal memory allocation according to the declarations of operators. With backend abstraction, MNN manages to divide the task into two independent parts. The "front-end operator" developers can focus on the efficient implementation for fast operator execution and all unrelated backend details are hidden. The "backend" developers can be devoted to exploiting different backend specifications and offering more convenient APIs. This separation of tasks are rather meaningful in practice since the lowering contribution barriers is highly appreciated in open-source projects.

(2) Enable hybrid scheduling. Heterogeneous computing mainly involves the backend selection and data transmission among different backends. When creating an inference session in MNN, one can configure targeted backends. If there are multiple backends available, MNN can decide the optimal backends for operators according to aforementioned backend evaluation (Section 3.2). As a result, MNN supports the flexible combination of operator execution on different backends even in a single inference. For example, convolution may run on CPU and the following ReLU activation may run on GPU. With the help of `Backend` class, developers do not need to worry about the scheduling and transmission, which automatically proceed under the hood.

(3) More lightweight. Implementation of each backend can work as an independent component while maintaining the uniform interface in MNN. Whenever some backends are unavailable on certain device, the corresponding concrete implementation can be easily peeled off from the whole framework. For example, Metal (Apple, 2014) is not supported on Android platforms and thus we can just take away the Metal module without touching the specific operator implementations. Through the decoupling of operator and backend, MNN can be competitively lightweight. This is of critical importance since mobile applications have a rigorous restriction on the binary size.

Through these uniform backend interfaces, to the best of our knowledge, MNN is the inference engine that supports the *most comprehensive* backends (see Table 4). Moreover, it is scalable enough for users to integrate new backends such as NPU, FPGA, *etc*.

3.5 Method Summary

As compared with some other typical design paradigms shown in Figure 6, we illustrate the design philosophy behind MNN and highlight several MNN-specific advantages in this part.

For an inference engine, high performance is the vital factor to decide whether it will be adopted by developers or not. Out of this motivation, different inference engines (*e.g.*, TF-Lite, NCNN, TVM) put continuous efforts into optimization to achieve better performance in difference ways.

For MNN, we make many improvements to meet the fundamental requirement of high performance. Taking the overwhelming operator diversity into consideration, we are not satisfied with the case-by-case optimization solution. Although this solution can be rather simple and effective, it often encounters the problem that some operators are left out of optimization and become the performance bottleneck (as shown by an example in Section 4.2). Instead, MNN first spots the compute-intensive unit *of a smaller granularity* (*i.e.*, the basic matrix multiplication), which is highly optimized by means of fast algorithms and paralleling techniques. Consequently, operators built upon this basic unit can naturally benefit from acceleration without being specially optimized.

Besides, maintainability, scalability, and the deployment cost all have a great impact on the long-term growth of an inference engine. Compared with auto-tuning in TVM, MNN is able to select the optimal computation scheme with less time and realize the runtime optimization through the

Table 4. Backend comparison of different mobile inference engines. "–" means that the mobile engine does not support that kind of backend (at the time this paper is submitted); "/" stands for "not applicable". Metal (Apple, 2014) is Apple's exclusive GPU standard on iOS, while OpenCL (Khronos, 2009), OpenGL (Khronos, 1992), and Vulkan (Khronos, 2015) are the concurrent standards on Android.

Mobile Inference Engine	#Operator (CPU)	#Operator (GPU)				Supported OS
		Metal	OpenGL	OpenCL	Vulkan	
CoreML (Apple, 2017)	110[1]	110[1]	/	/	/	iOS
TF-Lite (Google, 2017a)	93	17	19	–	–	iOS+Android
MACE (Xiaomi, 2018)	61	–	–	29	–	Android
NCNN (Tencent, 2017)	65	–	–	–	32	iOS+Android
MNN (Ours)	94	55	15	33	35	iOS+Android

[1] Since CoreML is not open-sourced, we cannot tell which operator belongs to CPU or GPU. Thus the number here is the total number of operators.

proposed pre-inference mechanism. Note that, by transferring the search stage from offline compilation to online pre-inference, we also avoid the restriction on the binary validation (*e.g.*, the iOS code signature). Meanwhile, by providing a set of uniform interface to hide raw backend details, MNN is naturally equipped with the property of modularity. Not only does this property make MNN lightweight, but also it is more convenient for contributors to extend MNN to more backends, as partly shown by number of operators supported on different backends in Table 4.

4 Benchmark Experiments

In this section, we comprehensively evaluate the performance of MNN. We first explain our experiment settings, then present the experimental results on different hardware platforms and networks, compared with other mobile inference engines. Finally, we share an online case to show the production application of MNN.

4.1 Experiment Settings

- Inference engines. We compare the performance with state-of-the-art mobile inference engines, including CoreML (Apple, 2017), TF-Lite (Google, 2017a), NCNN (Tencent, 2017), and MACE (Xiaomi, 2018).
- Devices. For iOS, we adopt iPhone8 and iPhoneX (processor: Apple A11 Bionic), as they are popularly adopted in benchmarks[1]. For Android, MI6 (processor: Snapdragon 835) and Mate20 (processor: Kirin 980) are adopted.
- CPU and GPU. (1) For CPU, thread $\{2, 4\}$ are evaluated considering that modern devices usually have two or four processors and multi-threading is a common acceleration technique. CPU affinity is set to use *all* the available cores in line with NCNN benchmark (Tencent, 2017). (2) For GPU, we evaluate the Metal Performance Shaders on iPhones. On Android devices, three standard backends (*i.e.*, OpenCL, OpenGL, and Vulkan) are evaluated since MNN has supported them all (see Table 4).
- Networks. MobileNet-v1 (Howard et al., 2017), SqueezeNet-v1.1 (Iandola et al., 2016), and ResNet-18 (He et al., 2016) are chosen as benchmark networks since they have been extensively used in mobile applications.
- Run settings. We report the inference time of one 224×224 RGB image (*i.e.*, batch size is 1), averaged by 10 runs. Before benchmark, one warm-up inference is conducted for fair comparison with other works (Tencent, 2017; Google, 2017a).

[1]https://www.tensorflow.org/lite/performance/benchmarks

4.2 Experimental Results

Performance on different smartphones and networks. Figure 7 shows the performance of MNN compared with other four inference engines. We have the following observations.

(1) Generally, MNN outperforms other inference engines under *almost all* settings by about $20\% \sim 40\%$, regardless of the smartphones, backends, and networks.

(2) For CPU, on average, 4-thread inference with MNN is about 30% faster than others on iOS platforms, and about 34% faster on Android platforms (*e.g.*, Mate20).

(3) For Metal GPU backend on iPhones, MNN is much faster than TF-Lite, a little slower than CoreML but still comparable, which is reasonable since CoreML is Apple's exclusive GPU solution tailored to iOS while MNN is meant to support backends of different operating systems. For Android GPU backends, other engines usually have their performance blind spots. For example, NCNN with Vulkan backend is not very fast on MI6; TF-Lite with OpenGL still has much room for improvement on ResNet-18 network. In contrast, MNN obtains favorable results on *all* different

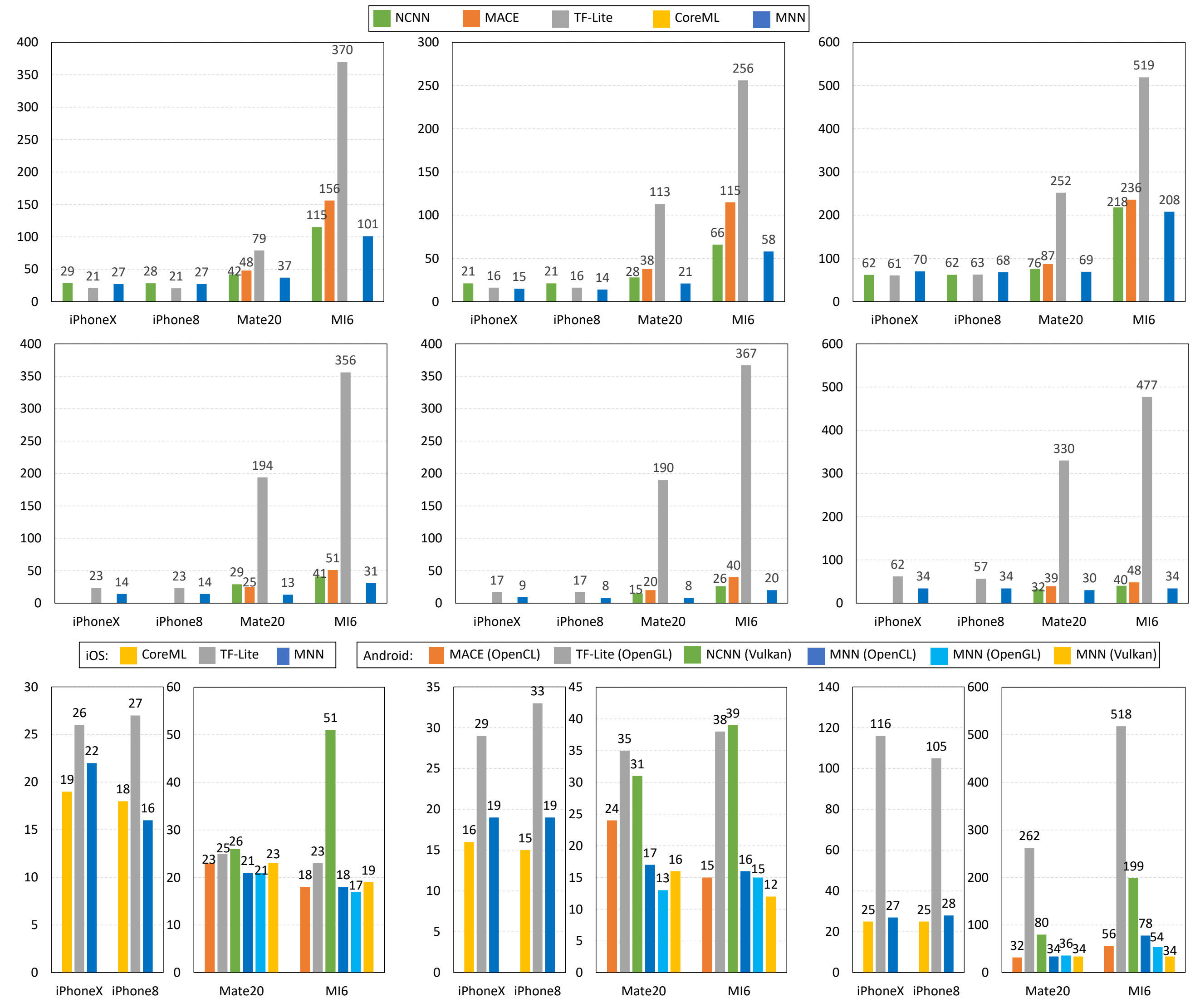

Figure 7. Inference time (ms) comparison on MobileNet-v1 (left), SqueezeNet-v1.1 (middle), and ResNet-18 (right). Row 1: CPU with 2 threads. Row 2: CPU with 4 threads. Row 3: GPU. (*Best viewed in color*)

hardware platforms and networks. Note that we achieve this comprehensive performance advantage by means of the proposed semi-automated search architecture rather than case-by-case heavy optimization.

(4) The multi-thread CPU inference using MNN on high-end devices (*e.g.*, iPhone8 and iPhoneX) is highly competitive compared with that using GPU backends, which demonstrates the effectiveness of the proposed in-depth kernel optimization of MNN.

Bottleneck of case-by-case optimization. Figure 8 shows an under-performance example of the case-by-case optimization on Inception-v3 (Szegedy et al., 2015). One can clearly see that NCNN requires abnormally more inference time than the others. This is because some special operators in this network (*e.g.*, 1×7 and 7×1 convolution) are not optimized by NCNN (for now). Consequently, they become bottlenecks during execution and drag down the overall performance severely. This specific case shows the limited scalability of case-by-case optimization. MNN is free from this problem because our computation solution is a general one which is applicable for various convolution cases.

Comparison with TVM. We compare the performance of MNN with TVM on various networks, as shown in Figure 9. Although MNN does not apply model-specific tuning and optimization, MNN still has encouraging performance and is even slightly faster than TVM. In addition, generating model-specific code using auto-tuning and compiling in TVM usually brings some deployment overhead. In Table 5[2]

[2]One may feel curious why the compiling results are faster than auto-tuning. This is because, TVM provides developers with

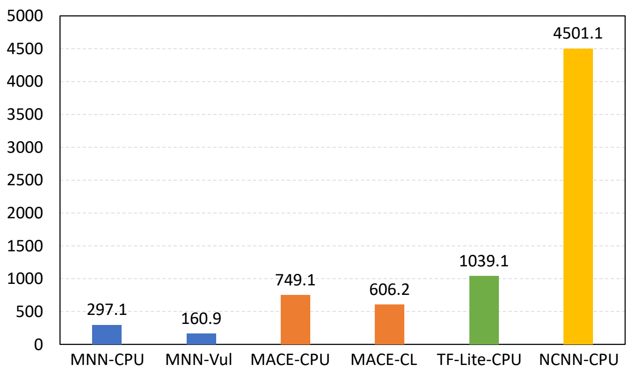

Figure 8. Bottleneck of case-by-case optimization on Inception-v3, evaluated on Huawei P20 (Kirin 970). "MNN-Vul" means"MNN-Vulkan" and "MACE-CL" stands for "MACE-OpenCL".

Table 5. Time cost (s) of auto-tuning and compiling for ResNet-18 with TVM on Samsung Galaxy S8 (GPU: Adreno 540).

#Trial	Auto-tuning	Compiling
1	355	40
10	1477	41
30	4583	41

we show the time cost of auto-tuning and compiling for ResNet-18 with TVM. Even with a small number of trials for tuning on a single device, TVM still takes much time to generate code. Since most mobile applications cover lots of different device types, the process of code generation will take much longer time and demand more resources (*e.g.*, servers), which is hardly affordable for many developers. MNN is free from these issues because all optimizations are performed at runtime without performance loss.

4.3 Online Case Study

Searching commodities is an indispensable stage for shopping on E-commerce platforms. However, due to the complexity of commodity categories, the traditional way of text search cannot meet the expectation of users these days. Therefore, searching commodity from images become a must-have feature for E-commerce platforms. This part shows a real application case of MNN in this scenario.

In the E-commerce application, MNN is employed to run deep learning models on mobile devices for main object detection and detected results are then used in commodity search. This service covers more than 500 kinds of mobile devices and has over 10 million daily users. Table 6 shows the top-5 popular devices used in this service and the average inference time, where MNN achieves stable and smooth search experience with average inference time 90.2 (ms) across all different devices, regardless of their broad diversity. This shows the encouraging universality of MNN.

the mechanism (*i.e.*, tensorize) to replace the auto-tuned code with hand-optimized implementation. Thus, it is reasonable for our optimized direct compilation to outperform auto-tuning.

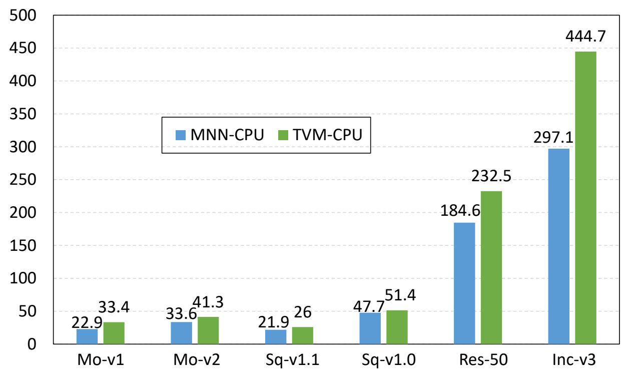

Figure 9. CPU inference time (ms) comparison between MNN and TVM (DMLC, 2016) on Huawei P20 Pro (SoC: HiSilicon Kirin 970). "Mo/Sq/Res/Inc" is short for MobileNet/SqueezeNet/ResNet/Inception, respectively. TVM data is adopted from their released benchmark: https://github.com/dmlc/tvm/wiki/Benchmark.

Table 6. Top-5 popular devices and their average inference time (AIT, ms) using MNN in a very large-scale real production case.

Device	CPU type	GPU type	AIT
EML-AL00	Kirin 970	Mali-G72 MP12	87.9
PBEM00	SDM670	Adreno 615	84.5
PACM00	Cortex-A73	Mali-G72 MP3	92.0
COL-AL10	Cortex-A73	Mali-G72 MP12	95.1
OPPO R11	Kryo 260	Adreno 512	91.4

5 Conclusion and Future Work

Mobile inference engine is critical to deep learning model deployment on mobile applications. To deal with the challenge of model compatibility and device diversity, we introduce Mobile Neural Network (MNN), which proposes a novel mobile engine design paradigm (semi-automated search) to get the best of both universality and efficiency. Pre-inference mechanism with a thorough kernel optimization and backend abstraction endows MNN favorable universality and state-of-the-art on-device inference performance.

MNN is still fast evolving, which is being improved in many aspects, for example, (1) applying auto-tuning during backend evaluation, (2) integrating model compression tools (*e.g.*, pruning) to slim the model on the fly, (3) providing more tools for user convenience, (4) providing more language support including JavaScript and Python.

Acknowledgements

We thank Chaoyue Niu for helpful discussions and the anonymous reviewers for their valuable comments to improve our work.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A system for large-scale machine learning. In *IEEE Symposium on Operating Systems Design and Implementation*, 2016.

Apple. Metal: Accelerating graphics and much more. https://developer.apple.com/metal/, 2014. Accessed: 2019-09-01.

Apple. CoreML. https://developer.apple.com/documentation/coreml, 2017. Accessed: 2019-09-01.

Ashari, A., Tatikonda, S., Boehm, M., Reinwald, B., Campbell, K., Keenleyside, J., and Sadayappan, P. On optimizing machine learning workloads via kernel fusion. In *ACM SIGPLAN Notices*, 2015.

Baidu. Anakin. https://github.com/PaddlePaddle/Anakin, 2018. Accessed: 2019-09-01.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., and Bengio, Y. Theano: new features and speed improvements. In *NeurIPS Workshop*, 2012.

Blahut, R. E. *Fast algorithms for signal processing*. Cambridge University Press, 2010.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

DMLC. TVM: Tensor virtual machine, open deep learning compiler stack. https://github.com/dmlc/tvm, 2016. Accessed: 2019-09-01.

Google. TensorFlow Winograd. https://github.com/tensorflow/tensorflow/blob/9590c4c32dd4346ea5c35673336f5912c6072bf2/tensorflow/core/kernels/winograd_transform.h. Accessed: 2019-09-01.

Google. Neural Networks API. https://developer.android.google.cn/ndk/guides/neuralnetworks, 2016. Accessed: 2019-09-01.

Google. TensorFlow Lite. https://tensorflow.google.cn/lite, 2017a. Accessed: 2019-09-01.

Google. XLA: Accelerated linear algebra. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, 2017b. Accessed: 2019-09-01.

Guo, L., Hua, L., Jia, R., Zhao, B., Wang, X., and Cui, B. Buying or browsing?: Predicting real-time purchasing intent using attention-based deep network with multiple behavior. In *SIGKDD*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

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

Iandola, F., Moskewicz, M., and Ashraf, K. SqueezeNet: Alexnet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv preprint arXiv:1602.07360*, 2016.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *ACM MM*, 2014.

Jiang, Z., Chen, T., and Li, M. Efficient deep learning inference on edge devices. In *MLSys*, 2018.

Kennedy, K. and Allen, J. R. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2001.

Khronos, G. OpenGL: Open Graphics Library. https://opengl.org/, 1992. Accessed: 2019-09-01.

Khronos, G. OpenCL: Open Computing Language. https://www.khronos.org/opencl/, 2009. Accessed: 2019-09-01.

Khronos, G. Vulkan. https://www.khronos.org/vulkan, 2015. Accessed: 2019-09-01.

Lavin, A. and Gray, S. Fast algorithms for convolutional neural networks. In *CVPR*, 2016.

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

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NeurIPS Workshop*, 2017.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

Reddi, V. J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.-J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al. Mlperf inference benchmark. *arXiv preprint arXiv:1911.02549*, 2019.

Shi, W., Cao, J., Zhang, Q., Li, Y., and Xu, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

Strassen, V. Gaussian elimination is not optimal. *Numerical Mathematics*, 13(4):354–356, 1969.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *CVPR*, 2015.

Tencent. NCNN. `https://github.com/Tencent/ncnn`, 2017. Accessed: 2019-09-01.

Wei, R., Schwartz, L., and Adve, V. DLVM: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

Winograd, S. *Arithmetic complexity of computations*, volume 33. SIAM, 1980.

Xiaomi. MACE Winograd. `https://github.com/XiaoMi/mace/blob/9b0b03c99cf73cd019050c6b9ee80a4753265da0/mace/ops/arm/fp32/conv_2d_3x3_winograd.cc`. Accessed: 2019-09-01.

Xiaomi. MACE: Mobile ai compute engine. `https://github.com/XiaoMi/mace`, 2018. Accessed: 2019-09-01.

Yu, D., Eversole, A., Seltzer, M., Yao, K., Huang, Z., Guenter, B., Kuchaiev, O., Zhang, Y., Seide, F., Wang, H., et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.

A MLPerf Evaluation

We also conduct MobileNet-v2 benchmark with MNN using the benchmark tool MLPerf (Reddi et al., 2019) on 4 CPU threads of Pixel 3. Results are shown in Table 7.

B More comparison on Pixel phones

To further compare with TF-Lite, we also conduct evaluations of Inception-v3 float model on the CPU of Pixel 2 and 3. Results are shown in Table 8. As we see, MNN is consistently faster than TF-Lite with either single thread or multi-threads, in line with the results in the main paper.

Table 7. MLPerf benchmark results.

Item of evaluation	Value
min_query_count	1024
max_query_count	5000
QPS w/ loadgen overhead	64.22
QPS w/o loadgen overhead	64.27
Min latency (ns)	13212918
Max latency (ns)	36022504
Mean latency (ns)	15560004
50.00 percentile latency (ns)	15600783
90.00 percentile latency (ns)	16407241

Table 8. CPU inference time (ms) comparison on Pixel phones.

Phone type	#Threads	TF-Lite	MNN
Pixel 2	1	974	**664**
Pixel 2	4	310	**214**
Pixel 3	1	873	**593**
Pixel 3	4	239	**160**

C Backend cost evaluation

Both CPU and GPU use `FLOPS` to measure the capability of the processors. Only GPU has the t_{schedule} term. Their values are determined as follows.

- `FLOPS`. For CPU, if the OS is Linux or Android, we can access the maximal frequency of each CPU core. Then choose the largest k frequencies and add them together as the `FLOPS` term, where k is the pre-specified number of threads (such as two threads or four threads). For the other CPU systems, set `FLOPS` to 2×10^9. For GPU, we estimate the `FLOPS` through practical running. Specifically, we run the MobileNet-v1 network for 100 times and obtain the `FLOPS` values for a bunch of common mobile GPUs. The results are shown in the list below. For those GPUs not in this list, we set the `FLOPS` as 4×10^9, namely, faster than CPU, as is the normal case.

  The list of GPU `FLOPS` (10^9): Mali-T860: 6.83; Mali-T880: 6.83; Mali-G51: 6.83; Mali-G52: 6.83; Mali-G71: 31.61; Mali-G72: 31.61; Mali-G76: 31.61; Adreno (TM) 505: 3.19; Adreno (TM) 506: 4.74; Adreno (TM) 512: 14.23; Adreno (TM) 530: 25.40; Adreno (TM) 540: 42.74; Adreno (TM) 615: 16.77; Adreno (TM) 616: 18.77; Adreno (TM) 618: 18.77; Adreno (TM) 630: 42.74; Adreno (TM) 640: 42.74.

- t_{schedule}. This term depends on the adopted graphical API. For OpenCL and OpenGL, it is empirically set to 0.05 (ms), which is the normal average time of calling API like `clEnqueueNDRKernel`. For Vulkan, since it only needs to summit `commandBuffer`, which is less time-consuming, thus t_{schedule} can be estimated as 0.01 (ms).