AsyMo: Scalable and Efficient Deep-Learning Inference on Asymmetric Mobile CPUs

Manni Wang*
Xi'an Jiao Tong University
Microsoft Research
manny.w.m.n@stu.xjtu.edu.cn

Shaohua Ding*
National Key Laboratory for Novel Software Technology, Nanjing University
Microsoft Research
dingshaohua@megvii.com

Ting Cao
Microsoft Research
ting.cao@microsoft.com

Yunxin Liu
Microsoft Research
Yunxin.Liu@microsoft.com

Fengyuan Xu
National Key Laboratory for Novel Software Technology, Nanjing University
fengyuan.xu@nju.edu.cn

Abstract

On-device deep learning (DL) inference has attracted vast interest. Mobile CPUs are the most common hardware for on-device inference and many inference frameworks have been developed for them. Yet, due to the hardware complexity, DL inference on mobile CPUs suffers from two common issues: the poor performance scalability on the asymmetric multiprocessor, and energy inefficiency.

We identify the root causes are improper task partitioning and unbalanced task distribution for the poor scalability, and unawareness of model behaviour for energy inefficiency. Based on that, we propose a novel technique called *AsyMo* for the thread pool implementation of DL frameworks to solve the two issues. The key design principle is to leverage the execution determinism of DL inference, and build an optimal execution plan offline by jointly considering model structures and hardware characteristics. For performance scalability, AsyMo implements *cost-model-directed partitioning* and *asymmetry-aware task scheduling* to properly divide and fairly schedule tasks on asymmetric CPUs. For energy saving, AsyMo determines the least-energy cost frequency based on data reuse rate of a model.

AsyMo is evaluated on different models and DL frameworks. All gain substantial improvement. For example, AsyMo shows up to 46% performance and 37% energy-efficiency improvement for convolution-dominant models, and up to 97% performance and 1.22× energy-efficiency improvement for fully-connect-dominant models, compared to an optimized TensorFlow on off-the-shelf mobile CPUs.

*Both authors contribute equally to this paper. Work is done during their internship at Microsoft Research.

ACM MobiCom '21, October 25–29, 2021, New Orleans, LA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8342-4/21/10...$15.00
https://doi.org/10.1145/3447993.3448625

CCS Concepts

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Computing methodologies** → **Parallel computing methodologies**.

Keywords

Mobile CPU, Deep Neural Networks, Asymmetric multiprocessor, Cost model, Energy efficiency

ACM Reference Format:
Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: Scalable and Efficient Deep-Learning Inference on Asymmetric Mobile CPUs. In *The 27th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '21), October 25–29, 2021, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3447993.3448625

1 Introduction

DL technology is used extensively in mobile and edge applications [59], such as image editing, face detection, and speech recognition. On-device DL inference is gaining momentum, due to the advantages in privacy protection, internet resilience, and quick response compared to on-cloud inference. Therefore, many DL frameworks and libraries have provided dedicated support for on-device inference.

Nearly all on-device inferences run on mobile CPUs, according to a recent study from Facebook [58]. Though various AI accelerators have been developed, mobile CPUs are still the most used due to their general availability, mature programming environment, robust support for diverse models, and increasingly better performance. Mobile GPUs are also widely available, but they provide only as much performance as mobile CPUs on majority Android devices, and many DL models are not supported on mobile GPUs [58]. Thus, we focus on mobile CPUs in this paper.

Issues However, we find that current on-device inference on mobile CPUs suffers from two common inefficiency issues. The first issue is *poor performance scalability on asymmetric multiprocessor (AMP).* Mobile CPUs feature an asymmetric design such as ARM big.LITTLE technology [21]. There is a big-core processor

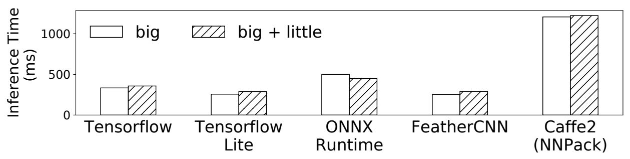


Figure 1: DL inference barely gains speedup by using both processors compared to just using the big processor for ResNet-50 in different frameworks on Kirin 970 with Android.

with higher CPU performance, as well as a little-core processor with lower performance and energy cost. Unfortunately, as shown in Fig. 1, DL inference barely gains speedup by using both processors compared to just the big-core processor, although the little one provides additional compute capability (*e.g.,* 58% more on Kirin 970). This is undesirable since inference latency is crucial for end users. DL frameworks should be capable to utilize all the available compute capability.

The other issue is *energy inefficiency because of improper CPU frequency setting*. OS cannot identify the most energy-efficient frequency because of the unawareness of DL behaviour characteristics. It tends to set the highest CPU frequency for inference, which is good for performance rather than energy. For example, ResNet-101 [22] consumes 81% more energy while saves only 32% inference time on Snapdragon 845 at the highest frequency compared to the least-energy cost frequency. Besides, the OS frequency scaling is not responsive enough particularly for short-run inferences. For example, the frequency only starts to gradually increase after 20% of total inference time for MobileNet V1 [26]. PredJoule [5] identifies efficient frequency through extensive energy measurements for every layer of a DL model on mobile CPUs. This is infeasible for the large numbers of models.

Root causes We deeply analyze the reasons for the poor scalability issue. The first reason we find is the *unbalanced task*[1] *distribution* on AMP cores. DL models are composed of tensor operations, particularly matrix multiplication (MM). For parallel execution, the thread pool (*e.g.,* Eigen [15] and OpenMP [46]) of DL frameworks partitions MMs into sub-MM tasks, and then assigns the tasks to each thread of the pool in a round-robin way. The OS then schedules these threads to run on the AMP cores. Unfortunately, current scheduling result is not ideal. We find that (1) the task distribution between the big and little processors is not proportional to their compute capability and the little one executes much fewer tasks; (2) the task distribution between the cores within each processor is also unbalanced due to the interference-prone mobile environment. Therefore, the performance gain by adding the little processor is far below expectation. It is necessary to implement asymmetry-aware task assignment in DL frameworks rather than just relying on OS scheduling.

Another implicit reason for the poor scalability is *inferior task partitioning*. We find that without proper partitioning, performance gain from adding the little processor is still under expectation (*e.g.,* 20% vs 58% on Kirin 970) even with asymmetry-aware task assignment.

[1]A task in this paper means a serial computation unit which can be assigned to run on a core.

MM partitioning (also called blocking or tiling) is a classical question and used to be actively studied on symmetric CPUs [7, 34, 54–56, 64]. It is not that active in recent years because the large cache on server CPUs can hold the data for most MMs, and thus performance is not sensitive to block size. However, block size still has big impact on mobile CPUs (*e.g.,* 30% on Kirin 970). The de-facto partitioning method used by current mobile DL frameworks is based on ATLAS [56]. It partitions matrices according to two considerations: the smaller matrix is always the inner matrix in the loop, and the blocks of a sub-MM task can be held in cache.

Challenges There are four major challenges on mobile AMP CPUs that current partitioning methods cannot solve. (1) Hardware asymmetry. Current partitioning uses unified block size on AMP cores, which harms performance and misleads the fair task assignment based on the number of tasks. (2) Separated cache between the big and little processors on most mobile CPUs. Accessing remote cache can cause 5× latency than local cache. Current partitioning and scheduling neglects this and may cause remote cache accesses. (3) High competition for small cache (*e.g.,* 2 M cache shared by four cores on the big processor of Kirin 970). ATLAS carefully determines innermost matrix for cache reuse. However, the high competition from multi-cores for the small cache can possibly cause cache thrashing. (4) Interference-prone environment. Overlooking this, current partitioning always results in lagging threads.

Our approach This paper proposes the AsyMo system with novel techniques to solve both the performance scalability and energy efficiency issues on mobile AMP CPUs. The primary design principle is based on the fact that DL inference is *deterministic*. That is, given a DL model, its execution is entirely determined by the model itself. Therefore, by jointly considering the model structure and AMP CPU characteristics, the optimal model execution plan *e.g.,* task partition and frequency setting, can be built offline. Guided by this principle, AsyMo integrates the following components.

For performance scalability, AsyMo coordinates a *cost-model-directed block partitioning* and an *asymmetry-aware task scheduling* method, which comprehensively consider the challenges of mobile AMP CPUs. The partitioning is first conducted on a processor level, and then on a core level to find the task size predicted to have the minimum MM latency by the cost model. The scheduling can balance tasks on each core and avoid unnecessary data movement between processors. The cost model is formulated by considering the task-size impact on every aspect that contributes to the latency, such as memory accesses, task scheduling cost, and degree of parallelism. The parameters of the cost model can be trained by common MMs for DL inference on each new CPU rather than each DL model, or be set by empirical values.

AsyMo determines the least-energy frequency based on the finding that DL models are typical computation- or memory-intensive workloads on mobile CPUs, which can be determined by the data reuse rate (*i.e.,* operational intensity) of a DL model. Therefore, AsyMo offline profiles energy curves over frequency for computation and memory-access benchmarks on target CPUs. Based on the data reuse rate of a DL model, the least-energy frequency can be found on the corresponding energy curve.

As far as we know, AsyMo is the first thread pool implementation that can achieve performance scalabiliy for DL inference on mobile AMP CPUs, and also gain substantial energy saving.

Implemented at thread pool level, the techniques of AsyMo is complementary with the optimizations at framework and kernel levels. We implement AsyMo based on the thread pool of Eigen [15], and then apply it to TensorFlow [18] (TF) and TensorFlow Lite [19] (TFLite) which are the most widely used DL frameworks (39% in total) on mobile devices according to the DL app analysis [59]. Besides, we also apply AsyMo to FeatherCNN [35] and ONNX Runtime [42] (ORT) to show its portability. FeatherCNN shows better performance than other libraries such NNPACK and OpenBLAS [35]. We evaluate MM computation, as well as end-to-end DL inference for a range of CNN (Convolution Neural Network) and RNN (Recurrent Neural Network) models on different mobile CPUs (Snapdragon 845 [47] and Kirin 970 [23]) and OSes. For MM executions, the speedup can reach up to 49%. For inference, AsyMo can achieve 46% performance and 37% energy efficiency improvement for convolution-dominant models, and up-to 97% and 1.22× improvement respectively for fully-connected-dominant models compared to an optimized-version TF v2.0 on Kirin 970 with Android.

To sum up, the key contributions of the paper are:

- Analyze performance and energy issues for DL inference on mobile CPUs, and identify the root causes (Section 3).
- Propose asymmetry-aware partitioning and scheduling methods to improve performance scalability of DL inference on mobile AMP CPUs (Section 4.2 and Section 4.3).
- Propose to find the least-energy frequency based on the data reuse rate of a DL model as well as the energy-frequency curves of the CPU (Section 4.4).
- Implement AsyMo and achieve significant performance and energy improvement on different frameworks, CPUs and OSes for both RNN and CNN models (Section 5 and Section 6).

2 Background

To understand the performance scalability and energy issues, this section introduces current task scheduling and partitioning for DL inference, as well as OS frequency scaling for mobile AMP CPUs.

Parallelism in DL inference A DL model is generally represented as a dataflow graph. A major advantage is for parallel processing. A node of the graph is a compute operation (abbreviated as op) and the connected edges are the tensors consumed or produced by an op. Two levels of parallelism exist in the graph: *the inter- and intra-op parallelism*. They can be implemented by two thread pools for better performance.

Fig. 2 shows a simple dataflow graph as an example. The inter-op thread pool can parallelly process ops without data dependency, such as op B and C. After being pre-processed, B and C are partitioned into tasks (notated as b and c) and sent to the shared intra-op thread pool to execute. Each inter-op thread waits until all the tasks of its op are finished to resume post processing. Compared to the compute tasks in the intra-op pool, the time cost of the inter-op pool is minor. Thus, intra-op thread pool is the optimization target of AsyMo.

Current thread pool implementations *e.g.,* Eigen and OpenMP, evenly distribute op's tasks to the task queue of each thread without consideration for AMP CPUs. The number of threads in the intra-op pool is normally set to the number of CPU cores (*i.e.,* hardware threads). The threads in the pool will then be scheduled to run on CPU cores by the underlying OS thread scheduler.

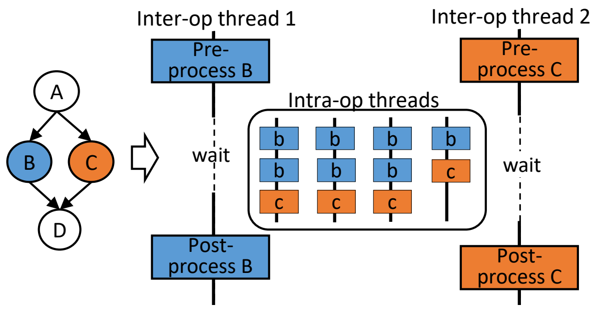


Figure 2: Inter- and intra-op parallel processing for op B and C in the example dataflow graph. The notation b and c show the partitioned tasks for B and C.

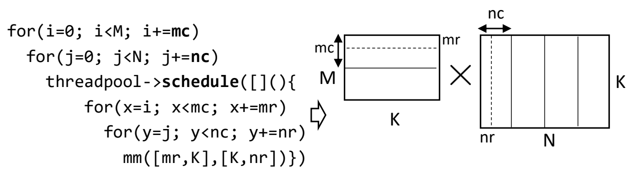

```
for(i=0; i<M; i+=mc)
  for(j=0; j<N; j+=nc)
    threadpool->schedule([](){
        for(x=i; x<mc; x+=mr)
          for(y=j; y<nc; y+=nr)
            mm([mr,K],[K,nr])})
```

Figure 3: $(M,K) \times (K,N)$ is partitioned into $(mc,K) \times (K,nc)$ tasks for parallel execution. $(mr,K) \times (K,nr)$ is the basic computing block.

Table 1: The execution time % of MM in DL models

MobileNetsV1	SqueezeNet	ResNet-18	SSD-MobileNetV1	Char-RNN
65.76%	72.48%	83.84%	68.00%	97.45%
ResNet-50	ResNet-101	VGG-16	RNN-classifier	AlexNet
84.81%	88.27%	96.00%	99.11%	94.82%

MM partitioning MM is the major cost of DL inference. Table 1 shows the MM time cost in representative CNN and RNN models on Kirin 970, ranging from 66% to 99% of total time (experimental settings in Section 6). The reason is that the convolution op (Conv) for CNN is normally implemented as MM for better performance [9]. The fully-connect op (FC) for RNN is a matrix-vector multiplication (MV) when the batch size is one for inference, which is a special case of MM. Thus, proper partitioning for MM is critical for inference performance.

Fig. 3 illustrates an example MM $(M,K) \times (K,N)$ which is partitioned into $(mc,K) \times (K,nc)$ tasks and then assigned to threads in the pool. To fully utilize SIMD registers, MM kernel normally has an elementary block size that cannot be partitioned *i.e., mr* and *nr* in the figure. Albeit as the accumulation dimension, *K* can be partitioned too if necessary.

The selection of *mc* and *nc* in current DL frameworks is basically based on the heuristics of ATLAS and hardware thread number, to make sure that the computation is equally divided for the threads and the task data can be held in cache. As aforementioned, these partitioning methods cannot solve the performance issue on mobile AMP CPUs.

Mobile AMP and OS DVFS Single-ISA asymmetric multicore architecture [33] is proposed to achieve both the performance and

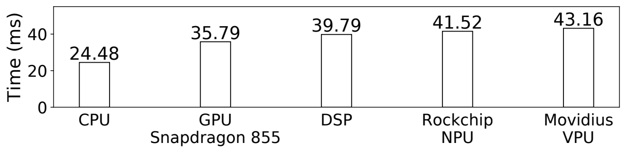


Figure 4: MobileNetV3 runs the fastest on a Snapdragon CPU than mobile AI accelerators. Frameworks used are TensorFlow Lite for the CPU; Qualcomm SNPE [48] for the GPU and DSP; RKNN for the Rockchip NPU [49]; and OpenVINO [29] for Intel Movidius VPU.

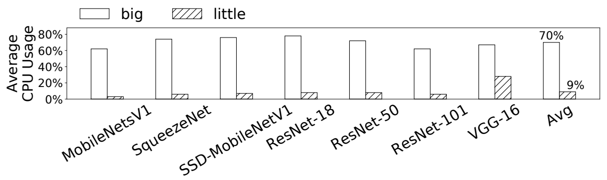


Figure 5: Average CPU usage of big and little processors for CNN inference in TF on Kirin 970 with Android. The little processor is seriously underutilized.

power requirements. It is widely adopted by mobile productions, represented by ARM big.LITTLE technology. Compared to the big-core processor, the little-core processor has lower CPU frequency, smaller cache and memory bandwidth, in-order pipeline, and lower power cost. Before DynamicIQ Shared Unit (DSU) technique [2], ARM big and little processors have separate caches. DSU enables optional shared L3 cache. (Section 6.1 has the detailed specs for the platforms used in this paper.)

For CPU frequency setting, the big and little processors normally have isolated power domains, and thus can set frequency separately. Each processor has a range of frequencies that OS DVFS (Dynamic Voltage and Frequency Scaling) can set based on the current workload for energy efficiency. Per-core frequency setting is not supported in ARM CPUs yet.

For timely frequency response according to workload change, the *Schedutil* governor [8] of OS DVFS is integrated into the state-of-the-art OS thread scheduler for mobile AMP CPUs called Energy-aware scheduling (EAS) [3] (note the difference between thread scheduling in OS and task scheduling in intra-op thread pool). Thus, *Schedutil* can configure frequency immediately when the scheduler changes thread. The frequency response is much faster than the workload-sampling-based *Ondemand* DVFS governor [8].

However, as we will show in Section 3, EAS *Schedutil* still has mismatch with short-run DL inference, which motivates the direct frequency setting of AsyMo.

On-device DNN accelerators Though various AI accelerators have been developed, CPUs are the dominant hardware for on-device inference [58] for the following reasons. Firstly, CPUs are always available on every mobile/edge device, but AI accelerators are not. For example, Edge TPU is currently only available on Google Pixel 4. Secondly, the ecosystem of AI accelerators is closed and immature. Most of the accelerators and their inference frameworks are black box [29, 48, 49], raising the barrier from wide usage. Thirdly, specialized accelerators lack the flexibility to adapt novel DL algorithms, and thus may not always perform well. For example, we evaluate MobileNetV3 [25], the latest version of MobileNet designed to run on mobile devices, on a range of AI chips including Google Edge TPU [20] and Intel Movidius VPU [44]. Fig. 4 shows the results of chips that can successfully run the model. The mobile CPU runs the fastest albeit with much lower theoretical peak performance. Moreover, mobile CPUs have similar micro architecture, and the optimization techniques can be generally applied. However, AI accelerators have quite different architecture. The optimizations need to be customized for each one. Based on these reasons, mobile CPUs should still play an important role for on-device inference in the coming future.

3 Performance evaluation and motivation

We have introduced the current design problems for DL inference on mobile AMP CPUs. This section analyzes how the problems harm performance scalability and energy efficiency.

3.1 Poor performance scalability on AMP

As shown in Fig. 1, current DL inference barely gains speedup by using both processors. To understand the reason, we take TF as an example, and record the processor usage for a range of CNN models in Fig. 5. Clearly, the little-core processor is seriously underutilized, only 9% usage on average. The usage for the big processor is not ideal either, only 70% on average.

As explained in last section, the intra-op thread pool evenly distributes tasks to each thread, and then OS assigns threads to CPU cores. We record the actual number of tasks executed on each core for every op. Results expose two-level unbalanced task distribution which causes the low CPU utilization.

The first level is **unbalance between big and little processors**. The number of tasks executed on the little processor is much less than its capability. Take MM in MobileNets V1 as an example, the average number of sub-MM tasks executed on a little core is 0.68, while the number on a big core is 3.82, so the ratio is 5.63. However, their capability ratio (*i.e.,* performance ratio) for running MM is only 1.73.

OS EAS is designed for mobile AMP. It schedules threads based on comprehensive considerations on asymmetric core capability, CPU utilization, and predicted energy cost. The result here shows that the intra-op threads are improperly assigned to the big processor much more often than the little one by EAS.

The second level is **unbalance within a processor**. For example, the task distribution of an MM in VGG-16 is 90, 77, 94, 95 for the big-core processor, and 40, 36, 25, 33 for the little-core processor (each processor has four cores in Kirin 970). This is possibly due to the interference from some Android background services or improper decisions of EAS. This unbalance degrades the average CPU usage. The lagging core that executes the most tasks becomes the performance bottleneck.

Therefore, the OS EAS for intra-op thread scheduling is far from ideal, due to the lack of workload understanding on each thread. For example, EAS does not know that there are equally-sized tasks on each thread, which should be distributed proportionally according to core capability. By comparison, with the information of tasks, fair task assignments for AMP should be implemented in the intra-op thread pool. This motivates the asymmetry-aware task scheduling of AsyMo.

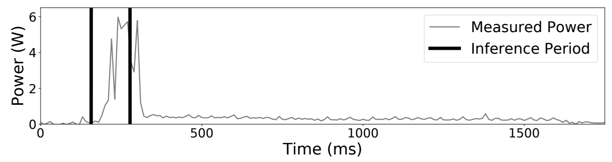


Figure 6: A mismatch between power curve and inference period for MobileNets V1 by OS EAS *Schedutil* on Kirin 970 with Android (idle power is subtracted).

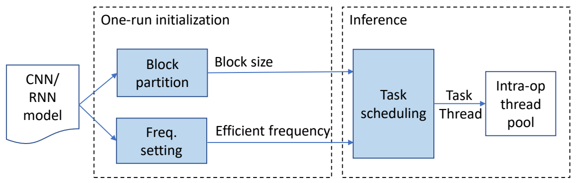


Figure 7: AsyMo (blue blocks) workflow.

3.2 DVFS mismatch

Through integration with OS thread scheduler, EAS *Schedutil* is expected to set CPU frequency in a timely fashion. We evaluate it using the short-run MobileNets V1 as an example, and measure the CPU power curve during inference in Fig. 6 (measurement methodology in Section 6.1).

Surprisingly, there is **a big mismatch between OS CPU frequency scaling and DL inference**. The CPU power, an indicator of CPU frequency, only starts to gradually increase about 34 ms (20% of total inference time) after the inference starts. When the inference is done, the power starts to descend about 25 ms later, and drags a long tail to finally reach the idle power about 1,283 ms later, causing a big waste of energy. We also evaluate the traditional sampling-based *Ondemand* governor. Much worse than *Schedutil*, the power starts to increase about 775 ms after the inference starts. This makes the inference time 6× more than the *Schedutil* result.

Users can avoid OS frequency scaling and set CPU frequency directly for their workload by the *Userspace* governor [8], with <1 ms frequency transition latency. By setting the highest CPU frequency for the inference period of MobileNets V1, we reduce the energy cost by 57% compared to EAS *Schedutil*, which shows a big potential in energy saving by using a proper CPU frequency.

As such, AsyMo sets CPU frequency directly for DL inference to eliminate the extra energy cost and performance slowdown due to DVFS mismatch.

4 AsyMo system design

4.1 System overview

The design of AsyMo is based on the unique characteristics of both DL inference and mobile AMP CPUs.

The DL inference characteristics considered are: *(i)* **deterministic execution**. Given a DL model graph, the entire execution is determined, such as ops and tensor size. The task partition and efficient frequency can be configured during the one-run framework initialization after a model is loaded; *(ii)* **embarrassingly parallel tensor operation**. The AsyMo just needs to balance the number of tasks on each core according to core capability rather than the need to identify critical path or data dependencies for general applications [30]; *(iii)* **typical hardware behaviour characteristics**. Conv is a typical computation-intensive workload while FC is memory-intensive (explanation in Section 4.4), which facilitates the selection of least-energy frequency for models dominant by these ops.

The mobile AMP CPU characteristics considered are: *(i)* **asymmetric core capability.** Task partitioning and scheduling are customized for big and little processor respectively; *(ii)* **small caches.** The partitioning carefully selects block size to reduce total memory accesses; *(iii)* **separated caches.** The partitioning and scheduling avoid costly data transfer between processors. *(iv)* **interference-prone environment.** The execution on mobile CPUs is easily to be interfered by system services or other workloads. The partitioning and scheduling need to avoid some core become a bottleneck.

Considering these unique characteristics, the workflow of AsyMo is shown in Fig. 7. After a model is loaded in the one-run framework initialization, the *cost-model-directed block partitioning* of AsyMo calculates the proper task size for every MM of the model on both big and little processors. The least-energy frequency for the model is found by AsyMo based on its data reuse rate (*i.e.,* memory- or computation-intensive) as well as the processor's energy-frequency curves.

After all the preparation in initialization, during the inference run, the *asymmetry-aware scheduler* of AsyMo binds each intra-op thread to one CPU core, and schedules tasks fairly to each thread.

To meet diverse requirements of apps, AsyMo supports two configurable modes: 1) *latency-first* mode where the highest CPU frequency is set for best performance; 2) *energy-first* mode where the most energy-efficient CPU frequency obtained by AsyMo is used to achieve minimum energy cost. Note that both the two modes have better performance and lower energy cost than the default DL inference without AsyMo (except for VGG-16 shown in Section 6).

Next, we introduce each technique of AsyMo in detail.

4.2 Cost-model-directed block partitioning

Current partitioning method based on ATLAS cannot solve the challenges on mobile AMP CPUs, which results in inferior performance. AsyMo partitioning comprehensively considers all these challenges. This section will first explain the ideas behind AsyMo, and then introduce the proposed cost model.

Design guidelines Due to the interference-prone environment, we find that there are always threads lagging, even though balanced task assignment is implemented and also task stealing from a busy thread to a free thread is enabled. Based on this, if a task is too big, just task stealing cannot help reduce the task on lagging threads. Therefore, we construct a partitioning guideline on mobile CPUs: **for better task balance, task size should be minimized.**

However, we find that reducing task size may increase memory accesses. The reason is that all the tasks are parallelly executed without particular order. Unless the cache can hold all the data of an MM, as is frequently the case on servers but rare on mobile CPUs, cache thrashing is possible *i.e.,* every task has to load data from memory. Thus, another partitioning guideline for small-cache mobile CPUs is that **task size should be maximized but remain within cache limitations, so that memory accesses are minimal.**

The two competing guidelines result in a trade-off task size which has the least MM latency. Empirical search of the configuration space

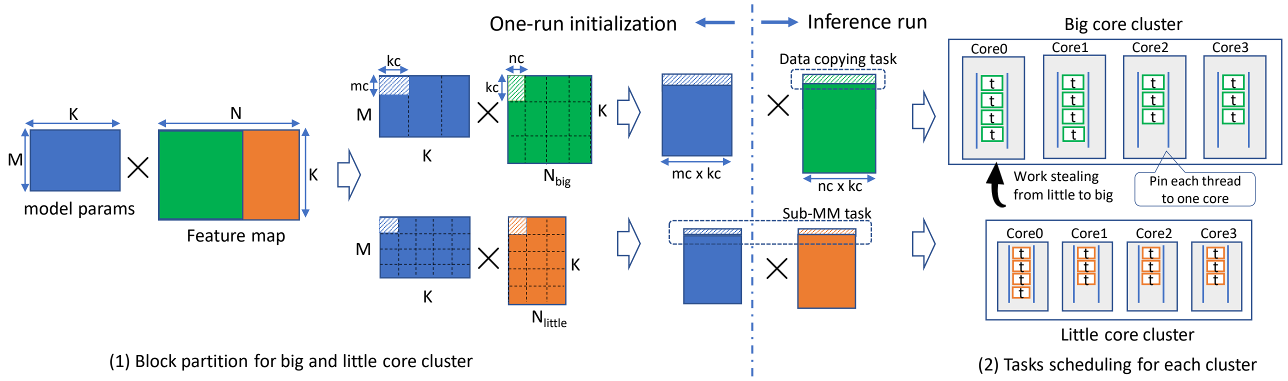


Figure 8: an MM execution process of AsyMo. During initialization, (1) AsyMo calculates the block partition strategy for big and little processor; (2) during inference, schedules the feature map copying and sub-MM tasks (notated by t in the figure) to each core within a processor.

on mobile CPUs is too large (in the order of millions) to be practical. By considering task-size impact on every aspect that contributes to the latency, including memory accesses, degree of parallelism, and scheduling cost, we formulate an MM cost model. It can predict the cost of each task size and find the size with minimum MM latency. Plus, the input, filter, and feature map size of DL models are normally within a specific set, so the model is easy to achieve good accuracy. This model only needs to be offline trained once for each CPU, and then can be applied for various DL inference.

MM cost model Table 2 shows the cost model. The input is block size on each dimension and output is predicted latency. The cost calculation process first calculates the cost of a sequential unit in Step (1) and (2). Since K is the accumulation dimension, the tasks along K normally run sequentially. Step (3) calculates the parallel MM execution cost based on the cost of one sequential unit, the number of sequential units ($\frac{M}{mc} \cdot \frac{N}{nc}$), and the number of threads.

Step (4) calculates other cost, including framework cost and task scheduling cost proportional to number of tasks. The unbalance cost is the time that some threads wait for others to finish all their tasks. Thus, it is a multiple of sequential cost. Including this cost is particularly important for the interference-prone environment. Finally, the whole MM latency cost for a task size can be obtained in Step (5).

Table 2: Latency cost model for a $(M,K) \times (K,N)$ MM.

Variables	mc, nc, kc: block size on each dimension
Constants	mr, nr: elementary block size (ref. Fig. 3) $Thread\#$: thread number = core number
Training parameters	t_{flop}, t_{data}: cost of a FLOP; a data access. t_{sched}, t_{fram}: cost of a task scheduling; framework. p: unbalance coefficient.
Cost calculation process	(1) FLOPs and data size of a task: $FLOP_{task} = mc \cdot nc \cdot kc$ $Data_{task} = \frac{DataTypeSize}{CachelineSize}(mc \cdot kc + kc \cdot nc + 2 \cdot mc \cdot nc)$ (2) Cost of a sequential unit: $Cost_{seq} = \frac{K}{kc} \cdot (t_{comp} \cdot FLOPs_{task} + t_{data} \cdot Data_{task})$ (3) Cost of parallel execution: $Cost_{par} = \frac{1}{Thread\#} \cdot \frac{M}{mc} \cdot \frac{N}{nc} \cdot Cost_{seq}$ (4) Cost of unbalance + scheduling + framework: $Cost_{other} = p \cdot Cost_{seq} + t_{sched} \cdot \frac{M}{mc} \cdot \frac{N}{nc} \cdot \frac{K}{kc} + t_{fram}$ (5) Total MM cost: $Cost = Cost_{par} + Cost_{other}$
Constraints	(1) Cache size: $mr \cdot kc + nr \cdot kc + mr \cdot nr \leq L1Size$ $(mc \cdot kc + nc \cdot kc + mc \cdot nc) \times Thread\# \leq L2Size$ (2) Least memory accesses: $mc = nc = \sqrt{\frac{MN}{i \cdot Thread\#}}, i \in \mathbb{N}$

To reduce memory accesses, the cost model sets two constraints for the block size. Constraint (1) is to make sure that the data of one task can be held in cache. The elementary compute block (introduced in Section 2) should reside in L1 cache. This constraint is for the two-level cache on most mobile CPUs and L2 is public to all cores within a processor.

Constraint (2) is to reduce total memory accesses of MM. It is calculated by multiplying the number of tasks with the memory accesses of a task, *i.e.,* $\frac{M}{mc} \cdot \frac{N}{nc} \cdot \frac{K}{kc} \cdot Data_{task}$, which equals to Eq. 1. Since $\frac{M}{mc} \cdot \frac{N}{nc} = i \times Thread\#, i \in \mathbb{N}$, *i.e.,* the number of sequential units is a multiple of the number of threads, for a fixed i, the minimum value of Eq. (1) is achieved when $mc = nc$. That is how Constraint (2) is calculated.

From Constraint (2), we can get a candidate mc and nc for each integer i. Through the cost model, we can find the least-cost mc and nc as the final partitioning result.

$$Data_{total} = \frac{N}{nc} \cdot M \cdot K + \frac{M}{mc} \cdot N \cdot K + 2 \cdot \frac{K}{kc} \cdot M \cdot N \qquad (1)$$

A special case is when $M \ll N$ (or $N \ll M$) such as MV, the calculated block size may be larger than M (or N). If so, AsyMo will not partition M (or N), but only partition the other dimensions.

Partitioning for big and little processors As shown in Fig. 8 (1), AsyMo first conducts processor-level partitioning by dividing the bigger matrix into two sub-matrices. Then, it conducts core-level partitioning within each processor. Reasons for the processor-level division are: (1) the proper block size is different on each processor; (2) the separation can avoid costly data transfer between the two processors. The division ratio is based on the offline measured processor capability (*i.e.,* performance difference running MM) as well as the current processor usage.

To sum up, besides of the design for AMP, the specific advantage of AsyMo partitioning is that it considers the trade-off between task balance and memory access reduction. The result block size is normally much smaller than other partitioning methods. Even with interference, collaborated with task stealing, tasks are easier to be balanced among threads. Thus, results show much improved performance scalability (results in Section 6).

4.3 Asymmetry-aware scheduling

The guidelines for AsyMo scheduling are: 1) to balance the number of tasks on each core and processor; 2) *to avoid unnecessary data movement between processors*.

AsyMo sets the intra-op thread pool equal to the core number, and uses OS thread affinity to bind each thread to a core as shown in Fig. 8(2). This can guarantee that the tasks actually executed on each core is consistent with AsyMo scheduling.

Based on the partitioning result for big and little processor, AsyMo schedules each task to the shortest thread queue in the corresponding processor. If a thread's task queue is empty, it can steal tasks from other threads (*i.e.,* work stealing). This is preferentially done from the longest thread queue within a processor. If a thread on a big core fails to steal a task within the processor, it is allowed to steal a task from a thread on a little core. However, the other way around is forbidden, because the block size of a big core can be too large for the cache of a little core.

There are two kinds of tasks scheduled in the thread pool as shown in Fig. 8: the data-copying tasks and sub-MM tasks. Data copying is an optimization method to fully utilize cache space and locality. It is used in general MM implementation after being proposed by ATLAS and achieves better performance. After blocks are partitioned, it copies the data of a block into a continuous memory space (*i.e.,* block-major format) before computation, and this is the data-copying task.

AsyMo schedules the data copying task and the corresponding compute task within the same processor, which can avoid the costly accesses to the remote cache. Other thread pool implementations, however, randomly schedule data copying and sub-MM tasks to cores. It is possible that the data for a sub-MM task resides in the remote cache, which causes data transfer between processors.

4.4 Frequency setting for energy efficiency

AsyMo selects the least-energy frequency based on the guideline that the least-energy frequency for a DL model is determined by its data reuse rate (*i.e.,* the operational intensity in a roofline model [57]). The reason is as follows.

Design considerations Energy cost of a workload is calculated as the sum of static and dynamic energy *i.e.,* $Energy = Time \times Power_{static} + Time \times Power_{dynamic}$. $Power_{static}$ normally keeps constant with CPU frequency, while $Time \propto \frac{1}{freq.}$ and $Power_{dynamic} \propto voltage^2 \times freq$ [12]. As frequency increases, the static energy cost reduces, while the dynamic energy increases. Therefore, there will be a trade-off frequency with the least total energy cost.

This least-energy frequency depends on the data reuse rate of the workload, because it impacts the response of *Time* to CPU frequency. For example, for high computation-intensive workloads, *Time* reduces accordingly as frequency increases, and so does the static energy. By comparison, for high memory-intensive workloads, *Time* does not reduce much with CPU frequency, since it has less impact on memory access latency. This is why for energy efficiency, memory-intensive workloads normally prefer a lower CPU frequency.

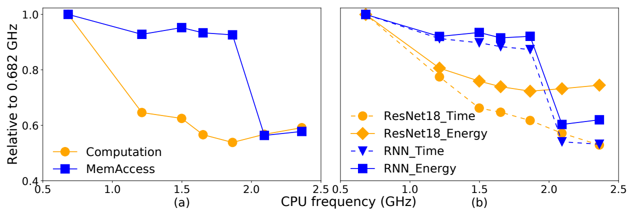


Figure 9: The least-energy frequencies are consistent on (a) the offline measured energy curves for memory-access and computation workloads, and (b) the ResNet-18 and Char-RNN models (measurement method in Section 5).

For DL inference, MM (for Conv) and MV (for FC) are typical computation- and memory-intensive (without parameter copying in initialization) workload respectively. This can be calculated by a simplified data-reuse formula for MM (without considering block partition). Assuming that input matrices are loaded from memory once and the result matrix is written back to memory. Multiply and accumulation operations are counted separately. The data reuse for an MM $(M,K) \times (K,N)$ is calculated by Eq. (2).

$$Data\_reuse = \frac{2MNK}{MK + NK + 2MN} = \frac{2}{\frac{1}{N} + \frac{1}{M} + \frac{2}{K}} \tag{2}$$

For MV, data reuse is bounded by 2, since N or M is 1. For MM, the smallest dimension is much larger than 1 ranging from 20 to 540 in our benchmark DL models. Thus, **the maximum data reuse of MM is bounded by the smallest dimension of the input matrices.** The greater the data reuse, the fewer memory accesses that are needed. Therefore, MM is computation intensive and MV is memory intensive.

Based on all the considerations above, *the idea of AsyMo is to offline profile energy curves over frequency for computation and memory-access benchmarks on target CPUs. Then, AsyMo can find the least-energy frequency for MM and MV ops of a model by the corresponding curve.*

Experimental verification To verify the idea, we profile the energy curves over frequency for self-written memory-access and computation benchmarks offline, and also the real energy curves for a Conv-dominant model ResNet-18, and an FC-dominant model Char-RNN [43] shown in Fig. 9. The lowest frequencies and curve shapes between the two figures are consistent. AsyMo can find the efficient frequencies for DL inference based on the offline profiled curves.

The figures show unexpected results for the energy curves of memory accesses. As Fig. 9 (b) shows, before 1.86 GHz, the Char-RNN result is basically as expected. The time and energy do not reduce much with CPU frequency. However, there is a big time and energy drop after 1.86 GHz.

To explain this result, we profile the memory access latency on two mobile CPUs shown in Fig. 10. Surprisingly, on Kirin 970, the random access latency (solid line) drops between 1.86 and 2.09 GHz, matching the RNN energy and time curve. Similar latency drop happens on Snapdragon 845 too. Thus, **for ARM CPUs, the random memory access latency drops at certain CPU frequencies.** This

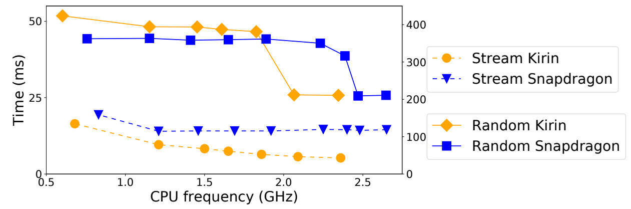


Figure 10: The latency of random (right axis) and stream (left axis) memory access on a single big core at different frequency step of Kirin 970 and Snapdragon 845. Snapdragon 845 has bigger frequency range than Kirin 970.

is why the memory-intensive RNN has much lower energy cost at a higher frequency.

Frequency setting for other ops We have also considered to extend AsyMo to set different frequency for different ops of a model based on its data reuse rate. However, current DL models are either dominant by Conv *e.g.,* CNN models or by FC *e.g.,* RNN models. Other ops like ReLU and Softmax take little time to run or fuse with the Conv layers. Thus, to avoid extra frequency transition cost, current AsyMo only sets frequency for Conv and FC ops.

5 Cost model training and energy profiling

We implement AsyMo in Eigen [15] due to its popular usage. DL frameworks can call the APIs of AsyMo to utilize its thread pool and frequency setting for DL inference. The asymmetry-aware thread pool of AsyMo is implemented in Eigen's `NonBlockingThreadPool`. The block partitioning is added in `TensorContractionThreadPool` which conducts block partitioning, and then enqueues the data-copying and sub-MM tasks.

Cost model training The cost model only needs to be trained once for each CPU rather than each model. The input, filter, and feature map size of DL models are normally within a specific set. The training data set can be much smaller than general MM, but still effective for DL models.

We select 15 MM sizes and 20 MV sizes in the data set. The MM sizes are chosen from the Conv ops of two representative CNN models: VGG-16, an example of complex models; and MobileNets V1, an example of light-weight models. For an $(M,K) \times (K,1)$ MV, M and K are set as the power-of-two values within range $(256,2048)$ and $(124,2048)$ respectively. Each MM or MV also includes a range of block sizes. In total, there are 270 settings in the data set. AsyMo trains the cost model through linear regression and K-folder ($K = 10$) cross validation using scikit-learn package.

The R^2 value (an index to show how close the data to the regression line) of the trained model is 0.97 and 0.98 respectively for the big and little processor. Applying the best partitioning found by AsyMo to our benchmark DL models, the performance difference is only 3% to 5% compared to the best empirically searched result. The time cost of profiling and training is about one hour on Kirin 970.

Profiling of energy-frequency curves To find the least-energy frequency, AsyMo offline profiles energy curves over frequency for computation and memory-access benchmarks on target CPUs. This profiling only needs to be done once for each CPU. The time cost to profile energy is insignificant. Particularly, the energy curves are purely hardware related, and ideally they should be provided by hardware vendors.

We use Android APIs to read the voltage and current of the battery and USB power supply on the mobile phone, and power monitor on the Hikey970 development board. We thus can get the real power of the computation and memory access at each frequency, *i.e.,* P_{freq}^{mem} and P_{freq}^{comp}, as well as the average memory access latency t_{freq}^{mem} (Section 6.1 for detailed measurement methodology). Then, the energy curve for computation and memory access is calculated by $\frac{1}{freq} \times P_{freq}^{comp}$ and $t_{freq}^{mem} \times P_{freq}^{mem}$ respectively.

Both the big and little core processors have a range of frequencies to be set. However, for the little processor, lowering frequency is not very helpful for power reduction. For example, on Kirin 970, only 7% power difference between the lowest (0.51 GHz) and highest (1.84 GHz) frequency on the little processor (Cortex A53), while 26% power difference between the lowest (0.68 GHz) and highest (2.36 GHz) frequency on the big processor (Cortex A73). Thus, AsyMo fixes the little core processor at the highest frequency, and only scales the frequency of the big processor.

6 Evaluation

6.1 Experimental methodology

Hardware and OS We use a Hikey970 development board with HiSilicon Kirin 970 SoC (Kirin 970 for short), running Android 9 Pie OS (Android for short), as the main experimental platform. Compared to a phone, it is easier to conduct power monitoring and temperature control on a development board. To show the portable performance, we also evaluate AsyMo on Kirin 970 running Debian Linux 4.9.78 aarch64 (Debian for short), and Google Pixel 3 XL with Snapdragon 845 SoC (Snapdragon 845 for short) running Android 9 Pie. We will state particularly when the results are from these two settings. The default DVFS governor for both OS is *Schedutil*. Note that although both Kirin 970 and Snapdragon 845 run Android 9 Pie, since Android has customized codes for different hardware, the behaviour may still be different.

Table 3 shows the CPU specs for Kirin 970 and Snapdragon 845. Peak performance is measured by self-implemented MM. Memory bandwidth and latency are profiled by LMbench [41]. The latency and bandwidth vary with the CPU frequency. We list the min latency and max bandwidth, respectively.

We measure the real power cost by Monsoon high voltage power monitor [28] on the Hikey970 board. On the Pixel 3 XL phone, the power of battery and USB is read from the Android APIs. The sampling rate is set to 5 kHz. Energy is the integral of power over time. We calculate it by multiplying the average of measured power during inference and the inference time. Note that the power measured is for the whole development board or phone rather than just the CPU, so the static power can be higher. However, it won't affect the fairness of the energy comparison.

The core utilization is sampled from /proc/stat every 200 ms. We keep the inference running while sampling the time, and make sure >20 samples for each model. The profiling thread runs on a different machine rather than the measured CPU to avoid increasing core utilization. The reported core utilization is the average of all the samples for a core.

Table 3: Experimental platform specs

Kirin 970	big core cluster	little core cluster
CPU	ARM Cortex A73, 4 cores	ARM Cortex A53, 4 cores
Pipeline	Out-of-order	In-order
CPU Frequency	0.68∼2.36 GHz	0.51∼1.84 GHz
Peak Performance	8.8 GFLOPs	5.1 GFLOPs
L1 D/I	64 KB private	32 KB private
L2	2 MB shared	1 MB shared
Cacheline size	64 B	64 B
Memory read bandwidth	4.6 GBs (max)	0.86 GBs (max)
Random memory access latency	200 ns (min)	200 ns (min)

Snapdragon 845	big core cluster	little core cluster
CPU	ARM Cortex A75, 4 cores	ARM Cortex A55, 4 cores
Pipeline	Out-of-order	In-order
CPU Frequency	0.83∼2.80 GHz	0.30∼1.77 GHz
Peak Performance	13.0 GFLOPs	5.4 GFLOPs
L1 D/I	64 KB private	32 KB private
L2	256 KB private	128 KB private
Shared L3	2 MB	
Cacheline size	64 B	64 B
Memory read bandwidth	10 GBs (max)	3.2 GBs (max)
Random memory access latency	200 ns (min)	210 ns (min)

Table 4: Time and energy cost of DL models in TensorFlow⋆ and AsyMo at max CPU frequency on Kirin 970 with Android (foot size is the min and max of the measured time)

	Flops	Params	Time (s)		Energy (J)	
Model	(10^9)	(10^6)	TF⋆	AsyMo	TF⋆	AsyMo
MobileNets V1	1.14	4.25	0.09 (0.08, 0.11)	0.06 (0.06, 0.07)	0.87	0.75
SqueezeNet	1.67	1.25	0.11 (0.09, 0.13)	0.07 (0.07, 0.08)	1.12	0.88
SSD-MobileNetV1	2.47	6.82	0.17 (0.15, 0.19)	0.12 (0.11, 0.13)	1.78	1.31
ResNet-18	3.47	16.02	0.18 (0.16, 0.19)	0.12 (0.11, 0.13)	1.78	1.29
ResNet-50	6.69	25.61	0.34 (0.30, 0.37)	0.22 (0.21, 0.22)	3.24	2.44
ResNet-101	14.39	44.68	0.63 (0.58, 0.69)	0.43 (0.42, 0.45)	5.57	5.29
VGG-16	30.80	68.15	1.24 (1.21, 1.27)	0.62 (0.61, 0.63)	11.69	7.71
Char-RNN	0.13	3.28	0.38 (0.30, 0.46)	0.03 (0.02, 0.07)	2.38	0.43
AlexNet	1.44	60.97	0.39 (0.38, 0.41)	0.06 (0.05, 0.08)	2.69	0.61
RNN-classifier	0.76	12.64	1.03 (0.84, 1.06)	0.11 (0.11, 0.11)	7.52	1.46

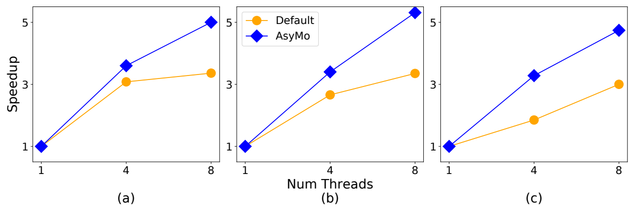


Figure 11: Performance scalability of MM over the number of cores (1-4 are big and 5-8 are little cores) by applying AsyMo to (a) TF; (b) FeatherCNN; (c) ORT at max CPU frequency on Kirin 970 with Android.

Framework and model configurations AsyMo is applied to TF, TFLite, FeatherCNN and ORT. The version of TensorFlow and TFLite is the recently released v2.0 with Eigen 3.3.7. They are compiled for Android by the C++ compiler of android-ndk-r18b with the option –cpu=arm64-v8a.

The version of FeatherCNN is v0.1-beta. It is compiled for Android using android-ndk-r19c and Android API version 21. ONNX Runtime is v1.1.0 with Eigen 3.3.90, compiled by android-ndk-r18b with Android API version 24.

AsyMo is evaluated with a range of typical DL models with various model or computation complexity and memory usage [6]. The CNN models included are MobileNets V1 [26], SqueezeNet [27], SSD-MobileNetV1 [38], ResNet-18/50/101 [22], VGG-16 [50], and AlexNet [32]. The RNN models are Char-RNN [43] and RNN classifier. CNN models are commonly dominated by Conv ops, except for AlexNet whose FC ops take ∼ 78% total time. Thus, **we categorize the models into either Conv-dominant or FC-dominant groups.** The types of the model parameters are all Float32. All the CNN models use default input size and NN structure. Except for VGG-16, a FC-4096 layer is removed because of an out-of-memory error. The configurations for the two RNN models are: Char-RNN: hidden_size 512, layers 2, batch_size 1, input_size 65, length 40, cell LSTM; RNN classifier: hidden_size 1024, layers 3, batch_size 1, input_size 512, length 20, cell GRU.

The reported inference time and energy are the arithmetic mean of 20 runs with 1 s delay between each run (except for the continuous inference experiment). **The first inference time is excluded.** The reason is that current DL frameworks normally use lazy initialization. The first inference also includes some one-time framework cost and runs several times slower than the following ones.

Optimized TF baseline We made two modifications to default TF as our baseline. One is to pre-copy model parameters into continuous memory during initialization to eliminate copies during inference. The other is the parallel implementation for MV (default TF only supports sequential MV). Block number of this parallel MV is equal to thread number. These two modifications are straightforward and have already been addressed in some DL frameworks/libraries [14, 34, 40]. The optimized baseline (denoted as TF⋆) is 32% faster than default TF for Conv-dominant models on average, and achieves up to 4.30× improvement in performance for FC-dominated models on Kirin 970 with Android.

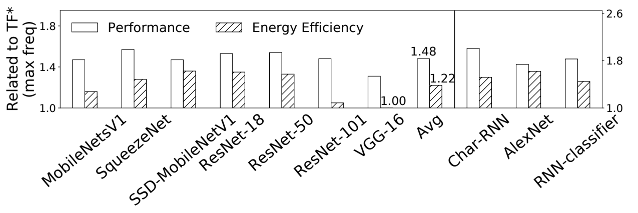
6.2 Results

This section will first show performance scalability improvement for MM by utilizing AsyMo in TF, ORT and FeatherCNN. Then, the whole DL model inference results are shown by utilizing all the techniques of AsyMo compared to TF v2.0⋆. The model evaluation is mainly on TF because of its sound support for various models.

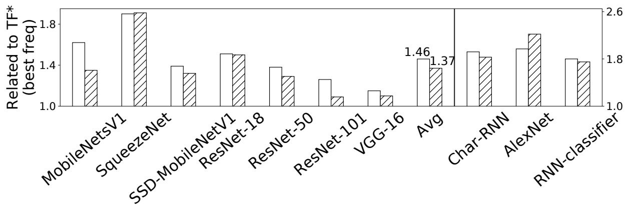
MM results. Fig. 11 shows the performance scalability improvement of MM (an average over a range of MM size used by DL) by utilizing AsyMo to different frameworks. Default TF and ORT use Eigen of different versions for thread pool implementation, while FeatherCNN uses OpenMP. Every of them has different serial MM kernel implementation. AsyMo can gain portable speedup on all of them. By the better partition strategy and fair scheduling, AsyMo can improve the CPU utilization for both big and little processors, and achieve performance scalability on AMP. For example, in FeatherCNN (Fig. 11 (b)), the 4-big-core speedup is 2.65× while AsyMo is 3.4×. On both big and little processors, FeatherCNN is only 3.35× while AsyMo is 5.31×.

Model inference results. Now we show results of real DL model inference on both big and little processors. AsyMo can greatly reduce latency and energy compared to TensorFlow⋆ in both latency-first or energy-first mode.

For the latency-first mode, CPU frequency is set at max (*i.e.,* 2.36 GHz) for both AsyMo and TensorFlow⋆ for fair comparison. Fig. 12a shows the performance and energy efficiency improvement of AsyMo ($\frac{Time_{TF}}{Time_{AsyMo}}$ and $\frac{Energy_{TF}}{Energy_{AsyMo}}$) for DL models. Table 4 lists the actual measured time (average, max, min), energy, and model



(a)



(b)

Figure 12: The relative performance and energy efficiency improvement of AsyMo on Kirin 970 with Android for Conv- (left axis) and FC-dominant (right axis) groups at (a) max CPU frequency; (b) most efficient frequency for AsyMo and *Schedutil* for TensorFlow⋆.

size. The average performance for Conv-dominant improves 48% by AsyMo, while energy efficiency increases by 22%. The energy improvement is smaller than performance because the improved core utilization by AsyMo also increases power consumption by around 20%. For the three FC-dominated models (right axis), the performance improvement is 1.01×, 74% and 83%, and the energy efficiency improvement is 52%, 62% and 45% respectively. The FC-dominant improvement is much better than Conv-dominant. This is because as Section 6.1 said, Tensorflow⋆ simply partitions MV into thread number (8 in this case) blocks, which is much fewer than desired. RNN-classifier's improvement is lower than Char-RNN because its inter-op parallelism is explored by TensorFlow, and the ops can run parallelly. Char-RNN can only benefit from the paralellism provided by Tensorflow⋆. AlexNet is lower than the other two because besides FC, its Conv op takes about 16% total running time in TensorFlow.

For energy-first, AsyMo can find the efficient frequency through its energy-frequency curves. Since AsyMo changes FC-dominant from a memory-intensive workload to computation intensive, the efficient frequency for all the benchmark models is set to 1.86 GHz in AsyMo on Hikey970. Fig. 12b shows the improvement of AsyMo at this frequency compared to default TensorFlow with EAS *Schedutil* which mostly sets CPU to the highest frequency. AsyMo improves energy efficiency by 37% for Conv-dominant on average compared to *Schedutil* governor. The efficiency improvement for three FC-dominated models is 83%, 1.22× and 75% respectively. The performance improvement is similar as Fig. 12a.

As discussed in Section 3.2, the energy cost reduction comes from 1) the efficient frequency selected by AsyMo using less power than max frequency; 2) eliminating the additional energy cost from the long power tail of *Schedutil* after the inference is done; 3) reduced running time by removing the mismatched frequency period at the beginning of inference using *Schedutil*. The third reason can also explain why for some models *e.g.,*SqueezeNet, the speedup of efficient frequency over *Schedutil* is even higher than Fig. 12b.

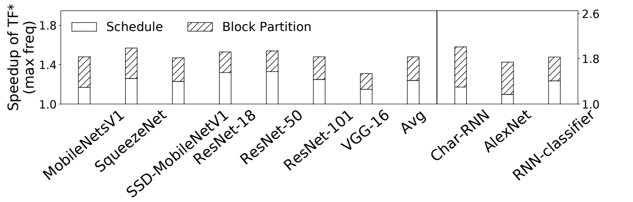


Figure 13: The performance improvement breakdown for prearranged parameters, asymmetry-aware scheduling, and cost-model-based block partition compared to default TensorFlow for Conv- (left axis) and FC-dominant (right axis) groups at max CPU frequency on Kirin 970 with Android.

Performance improvement breakdown To show the performance improvement of each technique of AsyMo, Fig. 13 breaks down the speedup at max frequency in Fig. 12a for asymmetry-aware scheduling and cost-model-based block partition.

AsyMo scheduling improves performance by 24% for Conv-dominant models on average. This improvement comes from the fair task scheduling and better cache locality. The improvement for the long-running VGG-16 is relatively lower than the other models. It is because as explained in Section 3.1, the default workload balance for VGG-16 is better than others and the default little core usage is 28% already. The small increase for the three FC-dominant models is because the eight blocks are too few to be scheduled fairly.

Block partition improves another 24% on average. Compared to default Eigen block partition, the partition cost model of AsyMo considers reducing both data accesses and sequential waiting time, and generates about 6 times more tasks for each op on average. More tasks *i.e.,* smaller blocks facilitate the workload balance and improve the parallelism degree. This is particularly important for ops having a small number of tasks by default Eigen and thus more sensitive to imbalance. For this reason, the improvement for SqueezeNet and MobileNets V1 is a bit higher than others.

Results on other platforms The results above are from Kirin 970 with Android. To show the portability of AsyMo, we also evaluate it on different hardware–Snapdragon 845, and different OS–Debian Linux 4.9.78. Fig. 14 shows the performance improvement results.

The average performance improvement on Snapdragon 845 is 52% for Conv-dominant models, about 29% higher than Kirin 970. The major reason is the default CPU usage of big cores for Snapdragon 845 is lower than Kirin 970. Thus, there is more space for AsyMo to improve. The performance improvement for FC-dominant models is 63%, 66% and 37% respectively, smaller than the result on Kirin 970. It is because the little core capability is about a third of the big core on Snapdragon 845, and a half on Kirin 970. Assume a big core's capability is 1, then from a sequential MV running on one big core to perfect parallelism on four big and four little cores on Snapdragon 845 can speedup $4+\frac{4}{3}=5.33$, while on Kirin 970 can speedup $4+\frac{4}{2}=6$.

Energy consumption of Snapdragon 845 is also evaluated. For the latency-first mode, CPU frequency is set at max (2.65 GHz). The energy efficiency is improved by 23% on average for Conv-dominant

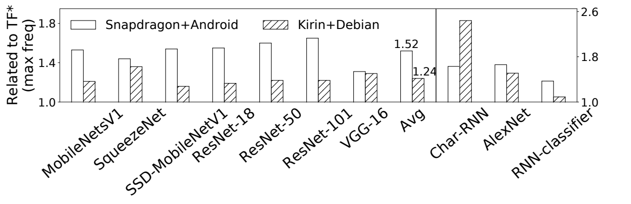


Figure 14: The performance improvement of AsyMo compared to TensorFlow⋆ on Snapdragon 845 with Android and Kirin 970 with Debian at max CPU frequency.

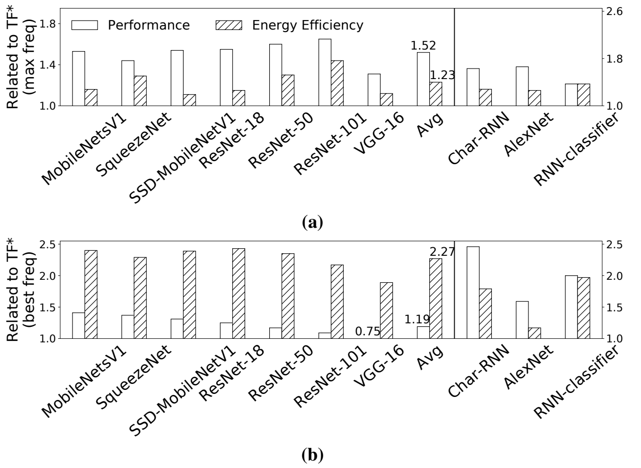


Figure 15: The relative performance and energy efficiency improvement of AsyMo on Snapdragon 845 with Android for Conv- (left axis) and FC-dominant (right axis) groups at (a) max CPU frequency; (b) most efficient frequency for AsyMo and *Schedutil* for TensorFlow⋆.

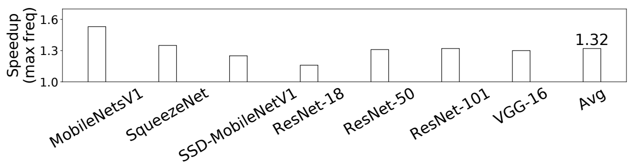


Figure 16: The performance improvement of AsyMo compared to TFLite⋆ on Kirin 970 with Android.

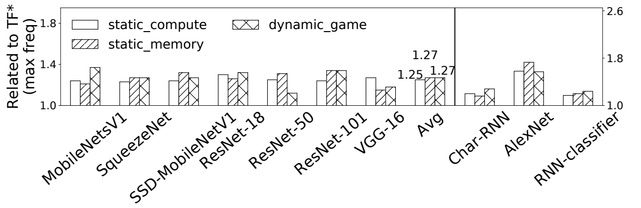


Figure 17: The relative performance improvement of AsyMo on TensorFlow⋆ with background load interference for Conv- (left axis) and FC-dominant (right axis) groups at max CPU frequency on Snapdragon 845 with Android.

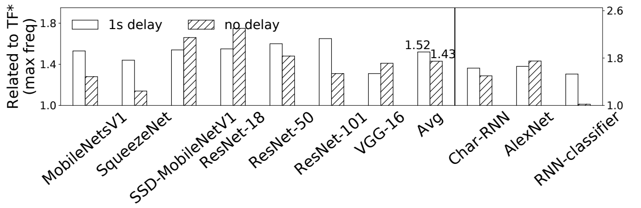


Figure 18: The relative performance improvement of AsyMo on TensorFlow⋆ w/o delay between inference runs for Conv- (left axis) and FC-dominant (right axis) groups on Snapdragon 845 with Android.

models, as shown in Fig. 15a. Due to the smaller static power, the increase in CPU utilization would result in higher increase in power, compared to Kirin 970, which is why the energy efficiency improvement is similar even with more improvement in performance. For the energy-first mode, CPU frequency is set to 1.21 GHz according to the energy curve of computation workload. The average performance improvement for Conv-dominant models is 19%, and the energy efficiency improves by 1.27×, as shown in Fig. 15b. Running at this low frequency, models gain acceleration mostly from correcting the mismatch discussed in Section 3.2, which decreases as inference time increases. This explains why VGG16 is 25% slower.

The speedup for Kirin 970 with Debian is about 16% smaller on average for Conv-dominant models compared to Kirin 970 with Android. This is because counter-intuitively, the baseline CPU utilization of Conv-dominant on Debian is better than Android, especially for MobileNets V1, although this Debian version doesn't have EAS scheduling for big and little cores. It is possibly because Android has more background services which disturb the inference running.

TFLite is designed for DL inference on mobile devices. Eigen is also its fallback choice for the thread pool implementation and the Float32 Conv ops. Thus, we also evaluate the performance of AsyMo compared to TFLite⋆ with Eigen library for Conv-dominant models in Fig. 16. TFLite has its own sequential implementation for MV and doesn't use Eigen's, so we didn't evaluate FC-dominant for TFLite. The average performance improvement is 32%. The result shows that although TFLite is particularly designed for mobile devices, AsyMo can still gain great improvement.

Background load interference The robustness of AsyMo under background load interference is evaluated with both static and dynamic background load. For static load, we use a controllable load generator *stress* [1] and run a single-thread sqrt() (which represents computation load) or malloc() (which represents memory load). For dynamic load, we use the replay tool RERAN [17] to record the user input of playing game Cut the Rope [63] level 10 to level 12, and replay it with each test. The result in Fig. 17 shows that Conv-dominant models can still gain 25% speedup with static computation load, 27% speedup with static memory load, and 27% speedup under dynamic game load. FC-dominant models can gain up to 58% speedup with static computation load, 73% speedup with static memory load, and 57% speedup under dynamic game load. The smaller block size and work stealing mechanism can help balance the loads among cores when there is background interference.

Continuous inference In previous experiments, a 1 s delay is inserted between inferences to maintain a neat and stable experimental environment for generating reproducible results. We also evaluate the performance for long continuous inference runs (use

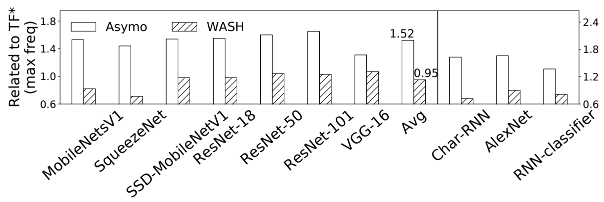


Figure 19: The relative performance improvement of AsyMo and WASH [31] on TensorFlow⋆ for Conv- (left axis) and FC-dominant (right axis) groups at max CPU frequency on Snapdragon 845 with Android.

1000 times here) to show the potential thermal throttling impact on AsyMo. As shown in Fig. 18, Conv-dominant models can gain 43% speedup on average. The reason for performance improvement drop is that AsyMo increases CPU utilization and therefore would generate more heat under continuous inference runs. The thermal throttling may be trigger earlier. A performance increase is also observed for SSD-MobileNetV1, ResNet-18, VGG-16 and AlexNet. This is possibly because continuous running reduces the time to awake sleeping threads.

Comparison with other AMP scheduler There are some general thread scheduling algorithms for AMP machines. For a quantitative comparison, we implement WASH [31], one of the state-of-the-art AMP thread scheduler, on Tensorflow⋆. Since it is a general-purpose thread scheduler, WASH is unaware of MM block partition, and just schedules threads according to the core capability (no critical threads for MM). Therefore, for WASH, we use the default block partition strategy of Tensorflow, and schedule threads according to core capability (*i.e.,* three-times threads on a big core compared to a little core on Snapdragon 845). Experimental result in Fig. 19 shows that the baseline performance is reduced by 5% on average for Conv-dominant models with WASH. This illustrates that without proper block partition of AsyMo, just scheduling threads according to core difference cannot benefit performance.

7 Related work

Block partition tuning There are many works on auto-tuning GPU work group size by empirical searching [36, 45, 53] or model-based searching [13, 16, 24, 51]. Empirical searching runs different configurations on real hardware to find the best one and thus has high searching cost. Model-based searching either manually derives a performance model or automatically trains a machine learning model to predict the running cost of different configurations. However, CPU and GPU are different in architecture and thus, the performance models for GPUs cannot be applied directly to CPUs.

The MM block partition strategies for CPUs normally use heuristics to empirically search for the optimal block size [7, 54]. [34] recommends using the largest block size possible that does not incur self interference within an array. ATLAS [54] tries to put the matrix that can be held in cache as the innermost matrix since it will be invoked many times, or it will put the two operand sub-matrices and the result sub-matrix into cache. PHiPAC [7] searches the best block size according to the register and cache size. TVM [10] and NeoCPU [39] are designed to generate low-level optimized code for DL models, by searching a code space and chooses a better operator according to the predicted or measured cost during compilation. SOL [52] is a middleware that transparently support heterogenous hardware and determines optimal memory layout in the compiling session. DeepCPU [64] accelerates RNN inference on x86 server CPUs. Its block partition strategy also considers reducing the slow-memory accesses. All those methods are mainly designed for symmetric server CPUs with large cache, and the search-based mechanisms imposes high search overhead. To compare, AsyMo formalizes an analytical cost-prediction model to quickly find the best partition for each DL model. AsyMo is designed for asymmetric mobile CPUs with much small cache and also considers parallelism, heterogeneous cache size, scheduling overhead, and framework overhead for DL.

AMP thread scheduling There are thread scheduling algorithms designed for AMP machines [3, 31, 62]. They schedule threads to big or little cores by monitoring their hardware behaviours or criticality. WASH [31] proportionally schedules threads to big and little cores according to the core capability. COLAB [62] makes coordinated core assignment and thread selection decisions based on the performance estimation of each thread on different cores, as well as identified communication patterns and bottleneck threads. Gomatheeshwari et al. [4] utilize a lightweight-deep neural network (LW-DNN) to predict the optimal cores for each workload. Compared with AsyMo, these works are for general thread scheduling at the OS or language virtual machine level. They are unknown of the code logic running on each thread, such as the matrix multiplication in this paper. Therefore, these schedulers cannot conduct block partition for matrix multiplication and then schedules them. By comparison, AsyMo partitions blocks first and then schedules the sub-block tasks according to the core capability.

Energy efficiency for mobile inference PredJoule [5] empirically measures the energy cost of each layer of a DL model under different CPU/GPU DVFS settings to find the least energy settings under latency requirements. However, the setting space is large and real measurement can be slow. Besides, as shown in the paper, the Conv and FC layers dominate the total cost. It is not rewarding to measure the energy of layers like ReLU and Softmax. They are normally fused with the conv layers anyway. An energy estimation tool [60] for DL models was developed and used for energy-aware model pruning [61] and compression [37], but it is specialized for the Eyeriss [11] DL accelerator only, and thus cannot be used for commercial hardware. AsyMo derives energy model to find the efficient CPU frequency for real ARM CPUs.

8 Conclusion

This paper reveals the performance scalability issue due to imbalanced task distribution, and energy inefficiency due to DVFS mismatch for mobile DL inference. To solve these, AsyMo is proposed to properly partition the MM blocks, schedule tasks among threads for fairness, and set the most energy efficient frequency. Both performance and energy efficiency get improved greatly by AsyMo in different DL frameworks on various platforms.

9 Acknowledgments

We thank our anonymous reviewers and shepherd for the helpful comment and feedback. We thank Dr. John Zigman for the valuable discussions and suggestions. Shaohua Ding and Fengyuan Xu were partially supported by the National Natural Science Foundation of China under Grant NSFC-61872180.

References

[1] 2019. *stress-android*. https://github.com/m-ric/stress-android/

[2] ARM. 2017. *ARM documentation set for DynamIQ Shared Unit*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexa.dsunit/index.html

[3] ARM. 2019. *Energy Aware Scheduling (EAS)*. https://developer.arm.com/tools-and-software/open-source-software/linux-kernel/energy-aware-scheduling

[4] Gomatheeshwari B and J. Selvakumar. 2020. Appropriate allocation of workloads on performance asymmetric multicore architectures via deep learning algorithms. *Microprocessors and Microsystems* 73 (2020), 102996. https://doi.org/10.1016/j.micpro.2020.102996

[5] Soroush Bateni, Husheng Zhou, Yuankun Zhu, and Cong Liu. 2018. PredJoule: A Timing-Predictable Energy Optimization Framework for Deep Neural Networks. In *RTSS*. IEEE Computer Society, 107–118.

[6] Simone Bianco, Rémi Cadène, Luigi Celona, and Paolo Napoletano. 2018. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access* 6 (2018), 64270–64277.

[7] Jeff A. Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. 1997. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*. ACM, 340–347.

[8] Dominik Brodowski. 2020. *CPUFreq Governors*. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt

[9] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie 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 18)*. USENIX Association, Carlsbad, CA, 578–594.

[11] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*. IEEE Computer Society, 367–379.

[12] Intel Corporation. 2004. Enhanced Intel Speed Step Technology for the Intel Pentium M Processor (White Paper).

[13] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. 2015. Autotuning OpenCL Workgroup Size for Stencil Patterns. *CoRR* abs/1511.02490 (2015).

[14] Marat Dukhan. 2018. *NNPack, acceleration package for neural networks on multi-core CPUs*. https://github.com/Maratyszcza/NNPACK

[15] Eigen. 2020. *Eigen*. https://eigen.tuxfamily.org/

[16] Thomas L. Falch and Anne C. Elster. 2015. Machine Learning Based Auto-Tuning for Enhanced OpenCL Performance Portability. In *IPDPS Workshops*. IEEE Computer Society, 1231–1240.

[17] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *2013 35th International Conference on Software Engineering (ICSE)*. 72–81. https://doi.org/10.1109/ICSE.2013.6606553

[18] Google. 2019. *TensorFlow: An end-to-end open source machine learning platform*. https://www.tensorflow.org/

[19] Google. 2019. *TensorFlow Lite: Deploy machine learning models on mobile and IoT devices*. https://www.tensorflow.org/lite

[20] Google. 2020. *Edge TPU*. https://cloud.google.com/edge-tpu/

[21] Peter Greenhalgh. 2011. *Big.LITTLE Processing with ARM Cortex^{TM}-A15 & Cortex-A7*. https://www.cl.cam.ac.uk/~rdm34/big.LITTLE.pdf

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. [n.d.]. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[23] HiSilicon. 2019. *Kirin*. http://www.hisilicon.com/en/Products/ProductList/Kirin

[24] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. GRNN: Low-Latency and Scalable RNN Inference on GPUs. In *EuroSys*. ACM, 41:1–41:16.

[25] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. arXiv preprint, arXiv:1905.02244.

[26] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. [n.d.]. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* ([n. d.]). http://arxiv.org/abs/1704.04861

[27] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). http://arxiv.org/abs/1602.07360

[28] Monsoon Solutions Inc. 2019. *Monsoon*. https://www.msoon.com/online-store

[29] Intel. 2020. *OpenVINO Deploy high-performance, deep learning inference*. https://software.intel.com/en-us/openvino-toolkit

[30] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Portable performance on asymmetric multicore processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 24–35. https://doi.org/10.1145/2854038.2854047

[31] I. Jibaja, T. Cao, S. M. Blackburn, and K. S. McKinley. 2016. Portable performance on Asymmetric Multicore Processors. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 24–35.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[33] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA*. IEEE Computer Society, 64–75.

[34] Monica D Lam, Edward E Rothberg, and Michael E Wolf. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review* 25, Special Issue (1991), 63–74.

[35] Haidong Lan, Jintao Meng, Christian Hundt, Bertil Schmidt, Minwen Deng, Xiaoning Wang, Weiguo Liu, and Yu Qiao. 2019. FeatherCNN: Fast Inference Computation with TensorGEMM on ARM Architectures. *IEEE Transactions on Parallel and Distributed Systems* PP (09 2019), 1–1. https://doi.org/10.1109/TPDS.2019.2939785

[36] Yinan Li, Jack J. Dongarra, and Stanimire Tomov. 2009. A Note on Auto-tuning GEMM for GPUs. In *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 5544)*, Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, G. Dick van Albada, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 884–892. https://doi.org/10.1007/978-3-642-01970-8_89

[37] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework. In *MobiSys*. ACM, 389–400.

[38] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. http://arxiv.org/abs/1512.02325 To appear.

[39] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1025–1040. https://www.usenix.org/conference/atc19/presentation/liu-yizhi

[40] Hao Lu Marat Dukhan, Yiming Wu and Bert Maher. 2018. *Quantized Neural Network PACKage*. https://github.com/pytorch/QNNPACK

[41] Larry McVoy and Carl Staelin. 1996. Lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference (USENIX ATC'96)*.

[42] Microsoft. 2019. *ONNX Runtime*. https://github.com/microsoft/onnxruntime

[43] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. [n.d.]. Recurrent neural network based language model.. In *INTERSPEECH 2010*.

[44] Intel Movidius. 2020. *Ultimate Performance at Ultra-Low PowerIntel Movidius Myriad X VPU*. https://www.movidius.com/myriadx

[45] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In *MCSoC*. IEEE Computer Society, 195–202.

[46] OpenMP. 2020. *The OpenMP API specification for parallel programming*. https://www.openmp.org/

[47] Qualcomm. 2019. *Snapdragon 845 Mobile Platform*. https://www.qualcomm.com/products/snapdragon-845-mobile-platform

[48] Qualcomm. 2020. *Snapdragon Neural Processing Engine SDK*. https://developer.qualcomm.com/docs/snpe/overview.html

[49] Rockchip. 2020. *High performance AI development platform*. http://t.rock-chips.com/en/

[50] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015*. http://arxiv.org/abs/1409.1556

[51] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. 2017. Towards Pervasive and User Satisfactory CNN across GPU Microarchitectures. In *HPCA*. IEEE Computer Society, 1–12.

[52] Nicolas Weber and Felipe Huici. 2020. SOL: Effortless Device Support for AI Frameworks without Source Code Changes. arXiv:2003.10688 [cs.DC]

[53] Benvan Werkhoven, Jason Maassen, Henri E.Bal, and Frank J.Seinstra. 2014. Optimizing convolution operations on GPUs using adaptive tiling. In *Future Generation Computer System*. 14–26.

[54] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[55] R. Clinton Whaley and Jack J. Dongarra. 1999. Automatically Tuned Linear Algebra Software. In *PPSC*. SIAM.

[56] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated Empirical Optimization of Software and the ATLAS Project. *PARALLEL COMPUTING* 27 (2000), 2001.

[57] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. https://doi.org/10.1145/1498765.1498785

[58] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim M. Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 331–344. https://doi.org/10.1109/HPCA.2019.00048

[59] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 2125–2136. https://doi.org/10.1145/3308558.3313591

[60] Tien-Ju Yang, Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2017. A method to estimate the energy consumption of deep neural networks. In *ACSSC*. IEEE, 1916–1920.

[61] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *CVPR*. IEEE Computer Society, 6071–6079.

[62] T. Yu, R. Zhong, V. Janjic, P. Petoumenos, J. Zhai, H. Leather, and J. Thomson. 2021. Collaborative Heterogeneity-Aware OS Scheduler for Asymmetric Multicore Processors. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2021), 1224–1237. https://doi.org/10.1109/TPDS.2020.3045279

[63] ZeptoLab. 2020. *Cut the Rope*. https://cuttherope.net/#ctr

[64] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 951–965. https://www.usenix.org/conference/atc18/presentation/zhang-minjia