Accelerating Deep Learning inference on mobile systems

Darian Frajberg, Carlo Bernaschina, Christian Marone, and Piero Fraternali

Politecnico di Milano,
Dipartimento di Elettronica, Informazione e Bioingegneria,
Piazza Leonardo da Vinci, 32, Milan, Italy
{first.last}@polimi.it

Abstract. Artificial Intelligence on the edge is a matter of great importance towards the enhancement of smart devices that rely on operations with real-time constraints. We present PolimiDL, a framework for the acceleration of Deep Learning on mobile and embedded systems with limited resources and heterogeneous architectures. Experimental results show competitive results with respect to TensorFlow Lite for the execution of small models.

Keywords: Deep Learning · Mobile Sensing · Acceleration · Mobile Devices · Embedded Systems · Continuous Vision

1 Introduction

The recent success of Deep Learning (DL) has boosted its application to many areas, with remarkable results that are influencing people's lives [20]. Typical implementations of DL models focus on the maximization of accuracy for a given task, and architectures to achieve such an objective have become significantly deeper and more complex over time [25][10]. Powerful workstations with Graphics Processing Units (GPUs) were fundamental for the success of DL, making their computationally expensive training possible. On the other hand, even though resources of embedded systems, such as smartphones, tablets, wearable devices, drones and Field Programmable Gate Arrays (FPGAs), are rapidly improving, they are still not completely suitable for the deployment of big and complex models. Furthermore, the use of remote cloud services for the execution of models has its own drawbacks due to the use of the network, such as cost, availability, latency, and privacy issues [5]. These limitations promote the interest in compact architectures for accelerating execution, by optimizing computation, storage, memory occupation, and energy consumption. The efficient execution of DL on the edge can benefit areas such as robotics, autonomous vehicles, augmented reality, health monitoring and digital assistance, which rely on smart devices with real-time constraints.

In this work, we present PolimiDL, a framework for accelerated DL inference on mobile and embedded systems. PolimiDL speeds-up the execution time of ready-to-use models, by applying multiple optimization methods, and increases

efficiency of operations without impacting accuracy. Its implementation is very generic, with neither hardware nor platform specific components, and supports devices with very heterogeneous architectures. The development of PolimiDL was started with the goal of deploying DL models on mobile devices, when no other stable solutions were available. It is currently deployed in PeakLens[1] [6], a real world AI-enhanced augmented reality (AR) mobile app for the identification of mountain peaks, with +370k installs. Experimental results demonstrate the effectiveness of PolimiDL to accelerate inference time, achieving competitive results with respect to TensorFlow Lite on a set of DL models and across multiple mobile devices.

The contributions of the paper can be summarized as follows:

- We introduce the problem of DL acceleration for devices with limited resources and discuss the design requirements of solutions capable of supporting a wide spectrum of device architectures.
- We propose a framework (PolimiDL) for DL acceleration that improves performance on mobile devices and embedded systems without accuracy loss.
- We release a public implementation of the framework in an open repository[2].

The rest of the paper is structured as follows: in Section 2, we discuss the related work; in Section 3, we introduce the requirements; in Section 4, we present PolimiDL; and in Section 5 we describe the evaluation of the framework; finally Section 6 concludes and gives an outlook on the future work.

2 Related Work

Compression techniques. Compression techniques target large scale architectures and aim at reducing the number of parameters and floating point operations (FLOPs), possibly tolerating small accuracy drops in favor of execution acceleration and optimization of computational resources, storage, memory occupation and energy consumption. Quantization [27] reduces numerical precision of CNNs to accelerate run-time performance and reduce storage and memory overhead, with minor accuracy loss. Pruning [8] removes redundant connections, thus the number of weights, and proved to efficiently compress state of art models by one order of magnitude. Alternative options include knowledge-distillation [11] to compress and transfer knowledge from complex models to simpler ones, and tensor decomposition methods [17] followed by low-rank approximation, for the reduction and compression of weights. The effectiveness of compression depends on the size and redundancy of the original model and most compression techniques are applied either after or at training-time. Post-training compression is easy to apply, but may induce a sensible accuracy loss, especially when no fine-tuning is performed on the models afterwards. On the other hand, training-aware compression tends to achieve better results, but requires more time and

[1] https://www.peaklens.com
[2] https://github.com/darianfrajberg/polimidl

it is more complex to apply.

Optimized model architectures. Lightweight architectures with compact layers pursue the design of an optimized network topology, yielding small, fast and accurate models, suitable for resource-constrained devices. SqueezeNet [15] is a first-generation optimized CNN architecture, with modules composed by small Convolutional kernels; it achieves the same accuracy as AlexNet [18] with 50 times less parameters and can be effectively compressed on disk. MobileNet [12] is a family of efficient models for mobile vision applications, which perform different trade-offs of accuracy, computation and number of parameters. Such models, released by Google, are based on Depthwise Separable Convolutions [4] and outperformed most of previous state-of-the-art models. MobileNet v2 [22] further improves MobileNet, by incorporating inverted residual connections. Recently, reinforcement learning has been also exploited for the discovery of efficient building blocks, to support the design phase. Tan et al. [26] proposed MnasNet, an automated neural architecture search that exploits a multi-objective reward to address both accuracy and latency measured in real-world mobile devices.

Hardware acceleration (HA). HA is the use of dedicated hardware to complement general-purpose CPUs and perform computationally intensive work more efficiently, e.g. by favoring specific operations and data-parallel computation. Digital Signal Processors (DSPs), GPUs and, more recently, Neural Processing Units (NPUs) are examples of it. Prominent mobile system on chip (SoC) vendors have incorporated specialized hardware for accelerated AI inference, focusing on vector and matrix-based instructions. Nonetheless, such instructions and the access to them depend on the proprietary Software Development Kits (SDKs) of each vendor, which are incompatible and impair the porting of acceleration solutions. Given the need of standardization, Google has recently published the Android Neural Networks API[3] (NNAPI), which defines a layer of abstraction that provides unified access to DL run-time acceleration. Its support for current devices is still limited due to its availability from Android 8.1 and requires specialized vendor drivers, otherwise computation falls back to the CPU. Similarly, recent versions of OpenGL[4] and Vulkan[5] were introduced for GPU-based efficient computations, but their support is reduced for older devices and depend on vendors' implementation. From iOS 8, Apple devices feature the Metal API[6], designed to maximize performance and let developers access HA. Apple has the advantage of targeting a limited and relatively homogeneous set of devices, while having full control over the production, which simplifies integration and support.

Heterogeneous computing scheduling. While HA relies on dedicated physical components designed to speed-up specific operations, heterogeneous com-

[3] https://developer.android.com/ndk/guides/neuralnetworks
[4] https://www.opengl.org
[5] https://www.khronos.org/vulkan
[6] https://developer.apple.com/metal/

puting scheduling comprises the design of strategies to efficiently coordinate and distribute the workload among processors of different types [1]. Previous research works [19][14] have proposed DL scheduling techniques for embedded systems. Results show a good level of optimization, with accuracy loss up to 5%. However, for maximum efficiency, these methods require specific drivers (e.g., to support recent versions of OpenCL) or custom implementations for different architectures with direct access to hardware primitives.

Mobile DL Frameworks. Frameworks for the execution of DL models on mobile and embedded systems pursue optimized deployment on devices with limited resources, by managing memory allocation efficiently and exploiting the available hardware resources at best. We built PolimiDL, our own optimized framework for DL acceleration on mobile devices and embedded systems, when no efficient off-the-shelf solutions were available; recently, some new tools were released, such as TensorFlow Lite[7], Caffe2[8], and Core ML[9]. Training is performed off-board, with mainstream tools such as TensorFlow, Caffe or MXNet, and the resulting models is converted into the format of the mobile framework for deployment. Open Neural Network Exchange Format[10] (ONNX) proposes the standardization of models definition, to simplify the porting of models trained with different tools. Furthermore, CoreML already exploits Metal HA on iOS devices, while NNAPI support for Android frameworks and devices is still not totally stable nor fully integrated.

Benchmarking. Performance benchmarking measures indicators to compare run-time architectures. For mobile DL, relevant metrics include accuracy, execution time, memory overhead, and energy consumption. Shi et al. [24] assessed the performance of various open-source DL frameworks, by executing different models over a set of workstations with heterogeneous CPU and GPU hardware. The work [23], defines guidelines to asses DL models on Android and iOS devices, and [9] studies the latency-throughput trade-offs with CNNs for edge Computer Vision. Finally, Ignatov et al. [16] present a publicly available Android mobile app to benchmark performance on a set of DL Computer Vision tasks. Scores are calculated by averaging performance results over all the devices and the corresponding SoCs evaluated.

3 Requirements

Before introducing the architecture and use of PolimiDL, we pinpoint the requirements for its development. When dealing with specific hardware architectures and vendors, maximum performance can be reached by developing ad-hoc optimised solutions. Nonetheless, such approach may comprise scalability and

[7] http://www.tensorflow.org/mobile/tflite
[8] http://caffe2.ai/docs/mobile-integration.html
[9] http://developer.apple.com/documentation/coreml
[10] https://onnx.ai

maintenance, when targeting many heterogeneous architectures and devices, as in the case of the Android market nowadays. Moreover, as highlighted in Section 2, current acceleration approaches still have limitations: 1. HA primitives are still not completely standardized and stable, but are tightly dependent on SoC vendors; 2. cloud-offloading can imply cost, availability, latency and privacy issues; 3. retraining or modifying the architecture of ready-to-use models can be extremely time-consuming; 4. post-training compression of already small models can detriment accuracy. Under the above mentioned drivers, the requirements at the base of PolimiDL can be summarized as follows:

1. **Focus on execution**. It should be possible to train a model using tools already known to the developer. The framework should focus just on execution concerns, without the need of re-training.
2. **Minimum dependencies**. It should be possible to execute an optimized model independently of the Operating System, hardware platform or model storage format.
3. **Easy embedding**. It should be possible to embed the framework and optimized models into existing applications easily, without the need of ad-hoc integration procedures.
4. **End-to-end optimization**. Optimization should be applied as early as possible and span the model life-cycle (generation, compilation, initialization, configuration, execution).
5. **Offline support**. Computation should occur only on-board the embedded system, without the need of a network connection for work off-loading.
6. **No accuracy loss**. The acceleration for constrained devices should not reduce accuracy w.r.t. to the execution on a high performance infrastructure.

4 The PolimiDL Framework

PolimiDL aims at speeding-up the execution time of ready-to-use models by applying multiple optimizations that increase the efficiency of operations without modifying the model's output. Its implementation is highly generic, with neither hardware nor platform specific components; this enables performance gains on heterogeneous devices and simplifies maintenance, eliminating the need of targeting different platforms by means of different tools. It is written in C++ and can be compiled for all major platforms, requiring only a very simple interface layer to interact with the platform-specific code. PolimiDL exploits multi-threaded execution, based on the STL Concurrency Extensions[11], and SIMD instructions, based on the well-known Eigen Library[12]. Figure 1 illustrates the general workflow of the proposed framework, with its main stages (in red) and data/artifacts (in green), showing the stage in which each optimization takes place. The pipeline starts by training a model via some external DL framework, such as TensorFlow or Caffe2, on a workstation or cloud accelerated learning infrastructure, such as

[11] https://isocpp.org/wiki/faq/cpp11-library-concurrency
[12] https://eigen.tuxfamily.org

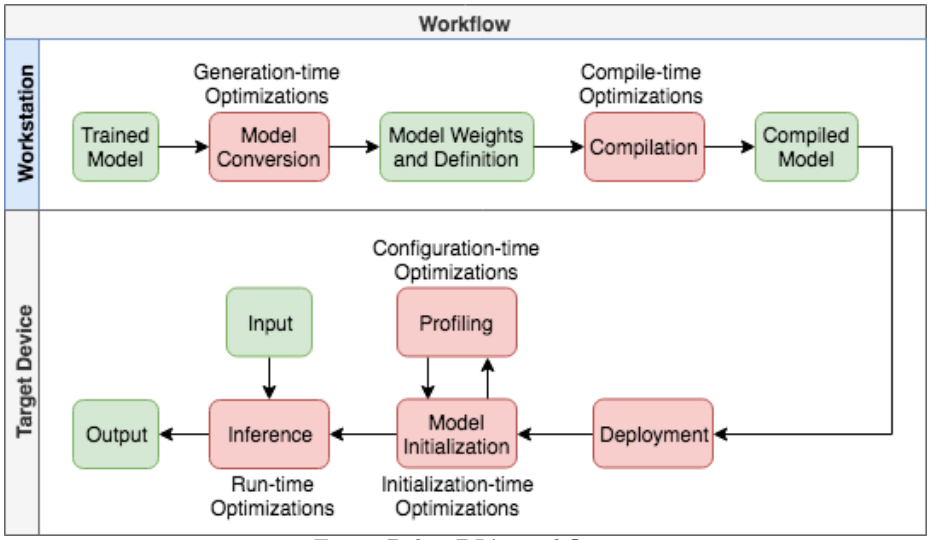

Fig. 1: PolimiDL's workflow.

Google Cloud[13]. The trained model is converted into a PolimiDL compatible format, while applying generation-time optimizations. Next, the model is compiled for the target architectures and compile-time optimizations are applied, enabling SIMD instructions where supported. Once the model is deployed on the target device, an initialization stage applies initialization-time optimizations to determine the best memory layout. The first time a model is deployed, the initialization step can include the profiling of the model, which enables configuration-time optimizations to determine the best scheduling approach. Finally, the model is ready to process inputs by applying run-time optimizations, which involve dynamic workload scheduling to speed-up inference.

4.1 Generation-time optimizations

Layers fusion. Consecutive in-place layers with identical filter size can be fused into one single layer, thus reducing the number of iterations over the cells of an input matrix. Such technique has been applied to fuse multiple combinations of layers, such as Batch Normalization/ReLu6 and Bias/ReLu. Potentially, Batch Normalization/ReLu6 fusion can be further extended by incorporating a Pointwise Convolution beforehand, taking into account that such combination of layers is frequently used for Depthwise Separable Convolutions.

Weights fusion. Layers applying functions with constant terms comprising multiple weights can be pre-computed and encoded as unique constant weights, thus reducing the operations at run-time and potentially avoiding temporary

[13] https://cloud.google.com/products/ai/

memory allocation for such layers. Weigh fusion applies, e.g., to the Batch Normalization (BN) layer, in which a subset of the vector weights involved in the normalization, scale and shift steps (γ,σ^2,ϵ) can be factored into a constant vector weight (ω) as follows:

$$BN(x_i) = \gamma * \left(\frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta \tag{1}$$

$$\omega = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{2}$$

$$BN(x_i) = \omega * (x_i - \mu) + \beta \tag{3}$$

where:

- x_i is the input of the layer
- γ, μ, σ^2, β are constant vector weights
- ϵ is a constant value

Weights rearrangement. Layers' weights, which are multidimensional matrices, are generally stored as linear arrays ordered by a default schema (e.g., output channel, row, column and input channel). PolimiDL stores the weights in different ways based on the layer type. Weights associated to predefined Convolutional layer types are stored in an order such that Eigen's GEMM matrix operations do not require any memory reshaping at run-time. These optimizations are executed automatically and transparently to the developer, who need not know their details.

4.2 Compile-time optimizations

Fixed network architecture. The architecture of a model is fixed at compile-time, which enables the compiler to perform per-layer optimizations. As an example, Convolutional layers can exploit loop-unrolling [13], because the number of channels and the kernel size are known at compile-time, potentially generating different machine code for each configuration of layer parameters. This approach can be seen as a limiting factor, because the model architecture cannot be updated at run-time or by simply changing a configuration file. However, it is important to notice that changing the model architecture is not expected to occur after the model has been deployed. Besides, in PolimiDL a model can be compiled as a set of Shared Objects (.so) files for the corresponding target architectures (armeabi-v7a, x86 and arm64-v8a for Android), enabling model updates by a simple file replacement. Given a fixed model architecture, PolimiDL supports the update of layer weights at run-time. When the run-time update of weights is not required, then the weights can be stored together with the network architecture, by embedding them in the .so files; this avoids the overhead of loading them from secondary memory, as opposed to TensorFlow Lite, where architecture and weights are stored as an external file loaded from disk.

Shared memory allocation & "tick-tock" piping. Efficient memory allocation and management is critical in embedded systems, where the amount of memory is limited and access time is slower than in workstations. Exploiting spatial locality [2] to reduce cache misses can decrease inference time and energy consumption significantly. For this purpose, layers in PolimiDL do not own the memory they read inputs from, write outputs to, or use to store intermediate results: memory is injected as a dependency from the layer scheduler. Given this organization, the memory required by a model can be reduced to just 3 areas: 1. Layer Input 2. Layer Output 3. Temporary data. These areas are properly sized at run-time, to contain the largest layer in the model. A disadvantage of this approach is the need to copy the output of a layer back into the input area to feed it to the next layer. PolimiDL alleviates this inconvenience by inverting the input and output buffers of subsequent layers. With this schema, data goes back and forth between the two buffers in a "tick-tock" fashion. Tick-tock buffer swapping is skipped for "in-place" layers, i.e., layers that can use the same buffer area for both input and output: they do not trigger an input/output buffer flip. ReLu layer is a clear example, because it performs value-wide operations enabling in-place modifications. Furthermore, given the fixed model architecture, layer piping can be computed at compile-time via the template meta-programming capabilities of C++, without incurring in any run-time costs.

4.3 Initialization-time optimizations

Memory pre-allocation. Pre-allocating memory buffers to contain the layers of a complete model without memory reuse may be feasible for server computation, but is certainly not the best option for embedded systems with hardware constraints. We have shown how the proposed framework reduces this memory requirements via shared buffers and the "tick-tock" piping. PolimiDL further reduces memory requirements by fusing the 3 buffers (input, output and temporary) into a single one. During initialization, each layer is queried about its memory requirements: input size, output size and temporary data, which can differ based on hardware capabilities, e.g., number of threads, or input size, in the case of Fully Convolutional Networks. A single buffer is allocated and sized to contain data of the most demanding layer. The upper and lower end of the buffer are used as input/output areas respectively, following the "tick-tock" strategy, while the area in between is used for the temporary data. This approach further reduces memory requirements as a single memory cell can store input, output or temporary data in different layers.

Small tasks for low memory consumption. While some layers require little or no temporary memory to work efficiently, others have a space-time trade-off. As an example, Convolutional layers can exploit SIMD instructions if their 3D input is unrolled into 2D matrices, where each row is the linearized input to the kernel. While unrolling the entire input and exploiting Eigen's SIMD and cache optimization capabilities may reduce the computation time significantly, it also

increases the memory requirements of the layer by increasing the size of the temporary buffer. In these cases, PolimiDL does not perform a full input unroll, but divides the operation into smaller tasks, which can be executed independently. In this way, the temporary memory required by the tasks has a fixed size.

4.4 Configuration time optimizations

Scheduling optimization. PolimiDL features a task scheduler, explained in detail in Section 4.5, which enables layers to divide the workload into tasks executed by different threads. The optimal size for a scheduled task may vary depending on the specific layer, the underlying architecture, or even on the input size for Fully Convolutional Neural Networks. Task sizes can be considered as parameters, which can be: 1. set to a default value, which may not be optimal 2. inferred by executing a profiling routine during initialization, which may increase the initialization time 3. inferred once for all on the specific device, stored and loaded at subsequent initialization steps. The profiling for each layer is performed by assessing the execution time of different task sizes. A full exploration of the task size space is not possible, given the high time and computation requirements. The sizes used during the assessment are generated by a power law heuristics. Task sizes may be bounded to a maximum value, dictated by the available temporary memory. It is important to notice that the available temporary memory may be more than the one requested at initialization time. This is because the buffer is initialized to contain the largest layer and, as a consequence, layers with smaller footprint can exploit the extra temporary memory.

4.5 Run-time optimizations

Dynamic workload scheduling. Static and even distribution of workload among available cores does not represent the most suitable solution, due to the unpredictable nature of mobile resources availability, more evident in asymmetric architectures such as ARM big.LITTLE [3]. A static scheduling strategy can under-utilize resources, wasting processing power. Conversely, dynamic multi-threaded scheduling of tasks can adapt well to different contexts and allows cores to be better exploited. Tasks are forwarded to a fixed size thread-pool (by default the number of workers is set to $max(1, \#cores - 1)$). In PolimiDL, during the development of a layer, it is possible to opt-out from dynamic scheduling or to enable it just when profiling shows a significant improvement. Dynamic scheduling should not be applied blindly, as computational intensive layers, such as Convolutions, perform better when dynamically scheduled, while others, such as ReLu, may perform worse due to memory bottlenecks. Therefore, dynamic scheduling is disallowed by default for layers that would be harmed by it.

4.6 Layers coverage

Table 1 summarizes the layers supported by PolimiDL and their features[14].

[14] Given the open source release of PolimiDL, the supported layers may be subject to modifications and further extensions.

Table 1: Layers supported by PolimiDL.

Layer name	In place	Temporary memory	Schedulable
Convolution	No	Yes	Yes
Depthwise Convolution	No	Yes	Yes
Pointwise Convolution (out channels $\leq$ in channels)	Yes	Yes	Yes
Pointwise Convolution (out channels $>$ in channels)	No	No	Yes
Max Pooling	No	Yes	No
Average Pooling	No	Yes	Yes
Batch Normalization	Yes	No	Yes
Bias	Yes	No	No
ReLu	Yes	No	No
ReLu6	Yes	No	No
Softmax	Yes	No	No

Fully Connected layers can be supported by introducing a standard Convolution in which the corresponding kernel size is equal to the expected layer input size. Given an expected input size of 1x1xN, such operation can be managed efficiently by using a 1x1xNxM Pointwise Convolution, where N represents the input channels and M the output classes.

4.7 Limits to generalization

The applicability of PolimiDL to a specific model is subject to the support of the required layers and to the availability of a converter from the source DL framework format. PolimiDL currently supports conversion from the TensorFlow format. Furthermore, features such as batch inference, model quantization and the inclusion of additional layers may require adaptations of the architecture design; for example, residual skip connections would require more complex buffers piping. Shared object libraries with self-contained weights declared as variables can be used for small models, common in embedded systems; but they may suffer from compilation constraints when big models, such as VGG-16 [25], are involved. Finally, PolimiDL currently runs on CPU only and does not support GPU processing, due to the still limited and non-standard access to it, which would require multiple implementations.

5 Evaluation

5.1 Experimental setup

The evaluation benchmarks inference execution time of DL models on heterogeneous embedded systems, comparing PolimiDL with the state-of-the-art solution for edge inference: TensorFlow Lite[15]. Measurements are collected by means of an Android benchmark application, implemented by extending TensorFlow Lite's

[15] The latest stable version at the time of writing is tensorflow-lite:1.13.1

sample application[16] to support multiple experiments. The use of multiple devices and models is critical for performance evaluation, given the non-linear correlation between hardware features and tasks characteristics [16].

The evaluation process is conducted as follows:

- Initialization and pre-processing times are not considered in the overall processing time.
- One warm up inference run is executed before the actual measurements.
- 50 consecutive inference iterations are executed and averaged to reduce variance.
- Three complete evaluation sessions with all models and devices are averaged, to further reduce variance.
- Models are run on mobile devices having above 80% of battery charge and pausing for 5 minutes between executions.

Models. Evaluation exploits hardware with limited resources and models with a small-size architecture achieving a good trade-off between accuracy and latency. Three models with diverse characteristics, listed in Table 2, are evaluated.

Table 2: Models used for evaluation.

Model	Task	Mult-Adds	Parameters	Input size
PeakLens original	Semantic segmentation	2G	429K	320x240x3
PeakLens optimized	Semantic segmentation	198M	21K	320x240x3
MobileNet	Object classification	569M	4.24M	224x224x3

PeakLens original [7] is a Fully Convolutional Neural Network model [21] for the extraction of mountain skylines, which exhibits a good balance between accuracy, memory consumption, and latency; it is exploited in the implementation of PeakLens, a real-world AR application for mountain peak recognition on mobile phones. The model was trained with image patches for binary classification, by adapting the LeNet architecture as shown in Table 3, and can be applied to pixel-wise classification of full images.

Table 3: PeakLens original model.

Layer Type	Input Shape	Filter Shape	Stride
Conv	29 x 29 x 3	6 x 6 x 3 x 20	1
Pool (max)	24 x 24 x 20	2 x 2	2
Conv	12 x 12 x 20	5 x 5 x 20 x 50	1
Pool (max)	8 x 8 x 50	2 x 2	2
Conv	4 x 4 x 50	4 x 4 x 50 x 500	1
ReLU	1 x 1 x 500	-	1
Conv	1 x 1 x 500	1 x 1 x 500 x 2	1

PeakLens optimized is a modified version of the PeakLens model replacing standard Convolutions with Depthwise Separable Convolutions, inspired by MobileNet [12]. The optimized version improves accuracy and performance and

[16] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/examples/android/app

reduces the number of parameters by one order of magnitude. The architecture is shown in Table 4: each Depthwise Separable Convolution block consists of a sequence of Depthwise Convolution, Relu, Pointwise Convolution, and Relu.

Table 4: PeakLens optimized model.

Layer Type	Input Shape	Filter Shape	Stride
Conv	29 x 29 x 3	3 x 3 x 3 x 32	1
ReLU	27 x 27 x 32	-	1
Depthwise Separable Conv	27 x 27 x 32	3 x 3 x 32 x 32	2
Depthwise Separable Conv	13 x 13 x 32	3 x 3 x 32 x 64	1
Depthwise Separable Conv	11 x 11 x 64	3 x 3 x 64 x 64	2
Depthwise Separable Conv	5 x 5 x 64	3 x 3 x 64 x 128	1
Conv	3 x 3 x 128	3 x 3 x 128 x 2	1

MobileNet [12] is a well-known state-of-the-art CNN architecture for efficient inference on mobile devices, developed by Google for diverse tasks, such as image classification and object detection. Multiple versions of MobileNet trained on ImageNet are publicly available[17], among which the biggest version has been chosen for evaluation (MobileNet_v1_1.0_224).

Devices. Six distinct Android devices with heterogeneous architectures are used, Table 5 lists the devices and their characteristics.

Table 5: Devices used for evaluation.

Device	Android Version	Chipset	CPU	RAM (GB)
Asus ZenFone 2 ZE500CL (Z00D)	5.0	Z2560 Intel Atom	2-cores 1.6 GHz (4 threads)	2
Google Pixel	9.0	MSM8996 Qualcomm Snapdragon 821	2-cores 2.15 Ghz Kryo + 2-cores 1.6 Ghz Kryo (4 threads)	4
LG G5 SE	7.0	MSM8976 Qualcomm Snapdragon 652	4-cores 1.8 GHz Cortex-A72 + 4-cores 1.2 GHz Cortex-A53 (8 threads)	3
LG Nexus 5X	8.1	MSM8992 Qualcomm Snapdragon 808	4-cores 1.44 GHz Cortex-A53 + 2-cores 1.82 GHz Cortex-A57 (6 threads)	2
Motorola Nexus 6	7.0	Qualcomm Snapdragon 805	4-cores 2.7 GHz Krait (4 threads)	3
One Plus 6T	9.0	SDM845 Qualcomm Snapdragon 845	4x 2.8 GHz Kryo 385 + 4x 1.8 GHz Kryo 385 (8 threads)	6

[17] https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md

Configurations. Multiple configurations are tested to analyze the impact of the scheduler thread pool size. *#threads* is the number of usable threads, which depends on the device (see Table 5). The evaluated configurations comprise:

- **min(4,#threads):** the thread-pool has a maximum of 4 workers, which is TensorFlow Lite's default configuration.
- **max(1,#threads-1):** the thread-pool employs all available threads but one.
- **#threads:** the thread-pool comprises all threads, for maximum parallelism.

5.2 Experimental results.

We report the experimental results obtained with TensorFlow Lite and PolimiDL for each combination of model, device, and configuration.

Table 6 reports the results for *PeakLens original* model. PolimiDL outperforms TensorFlow Lite in almost all cases (highlighted in green), with reductions of up to 57.32% (Motorola Nexus 6); TensorFlow Lite performs better (highlighted in red) just in one device (LG Nexus 5X) with a single configuration. Overall, PolimiDL consistently reduces average execution time by above 30%.

Table 6: Experimental results of PeakLens original model.

	TensorFlow Lite (ms)			**PolimiDL (ms)**		
Device	**Min (4,Threads)**	**Max (1,Threads-1)**	**All Threads**	**Min (4,Threads)**	**Max (1,Threads-1)**	**All Threads**
Asus ZenFone 2	1352.67	1672.67	1353.00	936.00 (-30.80%)	1138.00 (-31.96%)	936.67 (-30.77%)
Google Pixel	207.67	255.33	210.33	145.00 (-30.18%)	171.00 (-33.03%)	145.00 (-31.06%)
LG G5 SE	418.67	290.00	272.67	273.00 (-34.79%)	209.00 (-27.93%)	200.33 (-26.53%)
LG Nexus 5X	423.67	370.33	336.33	432.33 (+2.05%)	342.33 (-7.56%)	282.33 (-16.06%)
Motorola Nexus 6	336.67	505.33	337.67	169.00 (-49.80%)	215.67 (-57.32%)	168.33 (-50.15%)
One Plus 6T	176.00	144.33	145.33	104.00 (-40.91%)	91.00 (-36.95%)	89.00 (-38.76%)
			Average	(-30.74%)	(-32.46%)	(-32.22%)

Table 7 reports the results for *PeakLens optimized*. This model is smaller, yet more accurate than the original one. PolimiDL outperforms TensorFlow Lite and reduces inference time significantly. This is due to the design of memory management, which exploits well spatial locality and reduces cache misses. The performance gain is highly consistent: execution times are reduced on average of more than 62% in all the configurations. Improvement is particularly sensible for low-end devices, such as ZenFone 2, where the reduction is greater than 77%.

Finally, Table 8 reports the results for *MobileNet*. Performance of the two frameworks are quite comparable, but PolimiDL reduces overall execution time. The most significant gains are achieved on the ZenFone 2 and Nexus 6 devices.

Table 7: Experimental results of PeakLens Optimized model.

	TensorFlow Lite (ms)			PolimiDL (ms)		
Device	**Min (4,Threads)**	**Max (1,Threads-1)**	**All Threads**	**Min (4,Threads)**	**Max (1,Threads-1)**	**All Threads**
Asus ZenFone 2	740.67	807.67	743.33	166.00 (-77.59%)	179.33 (-77.80%)	167.67 (-77.44%)
Google Pixel	82.00	95.00	82.67	30.00 (-63.41%)	35.33 (-62.81%)	31.00 (-62.50%)
LG G5 SE	185.67	138.33	138.00	94.33 (-49.19%)	68.00 (-50.84%)	70.67 (-48.79%)
LG Nexus 5X	204.33	193.00	181.00	84.67 (-58.56%)	80.33 (-58.38%)	77.00 (-57.46%)
Motorola Nexus 6	140.33	225.67	135.67	52.33 (-62.71%)	66.00 (-70.75%)	49.00 (-63.88%)
One Plus 6T	66.67	68.67	66.33	22.00 (-67.00%)	22.67 (-66.99%)	22.33 (-66.33%)
			Average	(-63.08%)	(-64.59%)	(-62.73%)

TensorFlow Lite performs slightly better (not over 5%) on certain settings involving devices with big.LITTLE architecture (LG G5 SE and LG Nexus 5X). Despite the fact that PolimiDL features dynamic scheduling, it is the Operating System the ultimate responsible of the allocation of tasks to workers and low frequency cores seem to be prioritized for this model and devices. Nonetheless, the average execution time, when using all threads but one, is reduced by ≈16%.

Table 8: Experimental results of MobileNet model.

	TensorFlow Lite (ms)			PolimiDL (ms)		
Device	**Min (4,Threads)**	**Max (1,Threads-1)**	**All Threads**	**Min (4,Threads)**	**Max (1,Threads-1)**	**All Threads**
Asus ZenFone 2	734.00	775.33	733.33	371.00 (-49.46%)	377.33 (-51.33%)	374.33 (-48.95%)
Google Pixel	75.67	82.33	77.00	74.00 (-2.20%)	82.67 (+0.40%)	73.67 (-4.33%)
LG G5 SE	263.67	274.67	275.67	276.67 (+4.93%)	259.00 (-5.70%)	256.33 (-7.01%)
LG Nexus 5X	217.33	225.00	223.33	222.33 (+2.30%)	234.33 (+4.15%)	226.00 (+1.19%)
Motorola Nexus 6	224.33	298.33	227.67	203.67 (-9.21%)	176.00 (-41.01%)	163.33 (-28.26%)
One Plus 6T	56.67	56.67	57.67	49.67 (-12.35%)	51.67 (-8.82%)	53.00 (-8.09%)
			Average	(-11.00%)	(-17.05%)	(-15.91%)

The activation of NNAPI has been assessed in TensorFlow Lite for the supported devices, but results are not reported due to unstable performance. NNAPI reduces execution time on the Google Pixel, but doubles it on the LG Nexus 5X.

In conclusion, experimental results demonstrate the potential of PolimiDL by showing competitive results with respect to the well-known TensorFlow Lite

platform. Results are particularly improved when dealing with small models and low-power devices; this finding corroborates the potential of the proposed framework for supporting the implementation of augmented reality applications for mass market mobile phones, which is the use exemplified by PeakLens app.

6 Conclusion and future work

In this paper we presented PolimiDL, an open source framework for accelerating DL inference on mobile and embedded systems, which has proved competitive with respect to TensorFlow Lite on small models. Implementation is generic and aims at supporting devices with limited power and heterogeneous architectures. Future work will concentrate on adding support for more layers, quantization, and conversion from more DL frameworks. Moreover, experimentation will be extended by evaluating additional models, configurations, metrics (e.g. energy consumption and memory accesses) and devices (e.g. Raspberries and drones).

Acknowledgements. This research has been partially supported by the PENNY project of the European Commission Horizon 2020 program (grant n. 723791).

References

1. AlEbrahim, S., Ahmad, I.: Task scheduling for heterogeneous computing systems. The Journal of Supercomputing **73**(6), 2313–2338 (2017)
2. Anderson, A., Vasudevan, A., Keane, C., Gregg, D.: Low-memory gemm-based convolution algorithms for deep neural networks. arXiv preprint arXiv:1709.03395 (2017)
3. Cho, H.D., Engineer, P.D.P., Chung, K., Kim, T.: Benefits of the big. little architecture. EETimes, Feb (2012)
4. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1251–1258 (2017)
5. Dinh, H.T., Lee, C., Niyato, D., Wang, P.: A survey of mobile cloud computing: architecture, applications, and approaches. Wireless communications and mobile computing **13**(18), 1587–1611 (2013)
6. Fedorov, R., Frajberg, D., Fraternali, P.: A framework for outdoor mobile augmented reality and its application to mountain peak detection. In: International Conference on Augmented Reality, Virtual Reality and Computer Graphics. pp. 281–301. Springer (2016)
7. Frajberg, D., Fraternali, P., Torres, R.N.: Convolutional neural network for pixel-wise skyline detection. In: International Conference on Artificial Neural Networks. pp. 12–20. Springer (2017)
8. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems. pp. 1135–1143 (2015)
9. Hanhirova, J., Kämäräinen, T., Seppälä, S., Siekkinen, M., Hirvisalo, V., Ylä-Jääski, A.: Latency and throughput characterization of convolutional neural networks for mobile computer vision. In: Proceedings of the 9th ACM Multimedia Systems Conference. pp. 204–215. ACM (2018)

10. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
11. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. arXiv preprint arXiv:1503.02531 (2015)
12. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
13. Huang, J.C., Leng, T.: Generalized loop-unrolling: a method for program speedup. In: Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No. PR00122). pp. 244–248. IEEE (1999)
14. Huynh, L.N., Lee, Y., Balan, R.K.: Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In: 15th Annual International Conference on Mobile Systems, Applications, and Services. pp. 82–95. ACM (2017)
15. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. arXiv preprint arXiv:1602.07360 (2016)
16. Ignatov, A., Timofte, R., Chou, W., Wang, K., Wu, M., Hartley, T., Van Gool, L.: Ai benchmark: Running deep neural networks on android smartphones. In: European Conference on Computer Vision. pp. 288–314. Springer (2018)
17. Kim, Y.D., Park, E., Yoo, S., Choi, T., Yang, L., Shin, D.: Compression of deep convolutional neural networks for fast and low power mobile applications. arXiv preprint arXiv:1511.06530 (2015)
18. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)
19. Lane, N.D., Bhattacharya, S., Georgiev, P., Forlivesi, C., Jiao, L., Qendro, L., Kawsar, F.: Deepx: A software accelerator for low-power deep learning inference on mobile devices. In: Proceedings of the 15th International Conference on Information Processing in Sensor Networks. p. 23. IEEE Press (2016)
20. Lane, N.D., Georgiev, P.: Can deep learning revolutionize mobile sensing? In: Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications. pp. 117–122. ACM (2015)
21. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 3431–3440 (2015)
22. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4510–4520 (2018)
23. Sehgal, A., Kehtarnavaz, N.: Guidelines and benchmarks for deployment of deep learning models on smartphones as real-time apps. arXiv preprint arXiv:1901.02144 (2019)
24. Shi, S., Wang, Q., Xu, P., Chu, X.: Benchmarking state-of-the-art deep learning software tools. In: 7th International Conference on Cloud Computing and Big Data (CCBD). pp. 99–104. IEEE (2016)
25. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
26. Tan, M., Chen, B., Pang, R., Vasudevan, V., Le, Q.V.: Mnasnet: Platform-aware neural architecture search for mobile. arXiv preprint arXiv:1807.11626 (2018)
27. Wu, J., Leng, C., Wang, Y., Hu, Q., Cheng, J.: Quantized convolutional neural networks for mobile devices. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4820–4828 (2016)