Accelerating Deep Neural Networks on Low Power Heterogeneous Architectures

Manolis Loukadakis, José Cano, Michael O'Boyle

Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh, UK

Abstract. Deep learning applications are able to recognise images and speech with great accuracy, and their use is now everywhere in our daily lives. However, developing deep learning architectures such as deep neural networks in embedded systems is a challenging task because of the demanding computational resources and power consumption. Hence, sophisticated algorithms and methods that exploit the hardware of the embedded systems need to be investigated. This paper is our first step towards examining methods and optimisations for deep neural networks that can leverage the hardware architecture of low power embedded devices. In particular, in this work we accelerate the *inference time* of the VGG-16 neural network on the ODROID-XU4 board. More specifically, a serial version of VGG-16 is parallelised for both the CPU and GPU present on the board using OpenMP and OpenCL. We also investigate several optimisation techniques that exploit the specific hardware architecture of the ODROID board and can accelerate the inference further. One of these optimisations uses the CLBlast library specifically tuned for the ARM Mali-T628 GPU present on the board. Overall, we improve the inference time of the initial serial version of the code by 2.8X using OpenMP, and by 9.4X using the most optimised version of OpenCL.

1 Introduction

Deep Neural Networks (DNNs) are thriving the last few years because of the evolution of Artificial Intelligence (AI), which is in turn closely related to the increasing capacity of computing systems and architectures. These networks achieve great accuracy in numerous AI applications like speech [1,2,3] and image recognition [4,5,6], self-driving cars [7] and playing complicated games such as the Google's AlphaGo [8,9]. The performance of DNNs comes from their complex architecture. From a mathematical point of view, DNNs are basically a series of transformations and functions with learnable parameters. These parameters are learned during a training phase for a specific dataset. The training phase is a combination of forward-propagation, in which the error of the prediction is calculated, and a back-propagation procedure, in which the calculated error is minimised. Then, an inference phase uses the trained parameters (weights) in order to predict the class of specific unseen inputs with minimum error.

There are several types of neural networks, such as Feed Forward Neural Networks which achieve good performance in numerical and linguistic data, Recurrent Neural Networks, which have a wide application in machine translation and natural language processing, and Convolutional Neural Networks which achieve exceptional accuracy in image recognition. In this work we focus on the latter.

An important factor that has helped the widespread of DNNs is the rapid development of Graphics Processing Units (GPUs). Training and inference procedures are much faster when running on GPUs rather than on CPUs. In addition, the combination of the development of GPUs and the evolution of smartphones has brought neural networks to our daily lives. As a consequence, more and more neural networks are developed in embedded systems, exploiting the specific characteristics of embedded GPUs. However, the training procedure is an extremely heavy operation and in most cases is avoided in embedded systems. The systems research community has focused its interest mainly on accelerating the inference phase based on a pre-trained model of the neural network, which is trained in a high performance computing system. This is the approach followed in this work.

In addition, even the inference procedure can be affected by the limited capacity of embedded devices, both in performance and power consumption. Therefore, it is paramount to explore new methods and optimisations that can exploit specific embedded heterogeneous architectures in the best possible way.

This paper is actually our first step towards this objective. In particular, we focus on accelerating the inference phase of the VGG-16 neural network on the ODROID-XU4 board, which includes the ARM Cortex-A big.LITTLE processor and the Mali-T628 GPU. We propose two parallel implementations of the serial version of the VGG-16 neural network, the first one using OpenMP (traditionally used for homogeneous multi-core programming) and the second using OpenCL (the *de facto* standard for heterogeneous programming). The OpenMP version is parallelised on the CPU of the board, but the OpenCL version leverages the hardware architecture of the Mali GPU, where several optimisations techniques are examined and developed to boost the performance.

The contributions of this paper are as follows:

- We develop an OpenMP version of the VGG-16 neural network that runs only on the CPU of the board. This implementation improves the inference time of the initial serial C version of the code by 2.8X.
- We implement a baseline model in OpenCL without any hardware optimisation which provides a 0.8X speed up over the serial version.
- We use OpenCL work groups to speed up the performance of the baseline OpenCL model by 2.3X (1.9X over the serial version).
- The use of vector data types and SIMD instructions allows to parallelise the inference procedure further achieving an improvement of 11.6X over the baseline OpenCL (9.4X over the serial version).
- Finally, we use the CLBlast library optimised for the Mali-T628 GPU to reduce the inference time of the baseline OpenCL by 7.8X.

In summary, we improve the inference time of the initial serial code by 2.8X using OpenMP, and by 9.4X using the best implementation of OpenCL.

2 Background

This section introduces some fundamental concepts of convolutional neural networks that are required to understand the architecture of VGG-16, which is the specific neural network that we target in this work.

2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are composed of a stack of layers that transform a 3D input to a 3D output. Typically, three main types of layers are used to build a CNN:

- **Convolutional**: This layer is the most demanding computational part of the CNN. It consists of multiple 3-dimensional filters which are called kernels. These filters are learnable parameters. A typical example of a filter is 10x10x3 (i.e. ten pixels height, ten pixels width, and three channels depth). The kernels are sliding on the input images during the forward propagation. In each slide *dot product* is computed (convolution operation). Each sliding of kernel produces an *activation map*. The activation maps from the multiple kernels are stacked and form the output. Moreover, three hyperparameters determine the size of the output which are *depth*, *stride*, and *padding*. The depth refers to the number of kernels that are used. The stride is the step of sliding of the filter. Finally, sometimes is convenient to preserve the output size by padding with zeros the input image.
- **Max Pooling**: This layer is placed between convolutional layers and reduces the spatial size of the output. A max pooling operation is performed in each depth independently, where every MAX operation compares four elements of a 2x2 region of the output and selects the maximum value. Therefore, the depth dimension remains the same but the width and height are halved.
- **Fully Connected**: This layer sums a weighting of the previous layer of features, where all the elements of all the features of the previous layer are used in the calculation. Actually, there can be more than one layer which are placed after the convolutional and max pooling layers.

In addition, between the convolutional layers the *relu* activation function [10] is the most commonly used (it identifies likely features on the previous layer). Other layers like *Batch Normalization* [11] and *Dropout* [12] can also be used to regularise the training procedure and speed up the convergence of the network.

2.2 VGG-16 Neural Network

The VGG-16 neural network (Figure 1) was developed by Simonyan and Zisserman [13] in the context of the ILSVRC 2014 competition[1]. It achieves an accuracy of 70.5% and is the most computationally expensive neural network of

[1] `http://www.image-net.org/challenges/LSVRC`

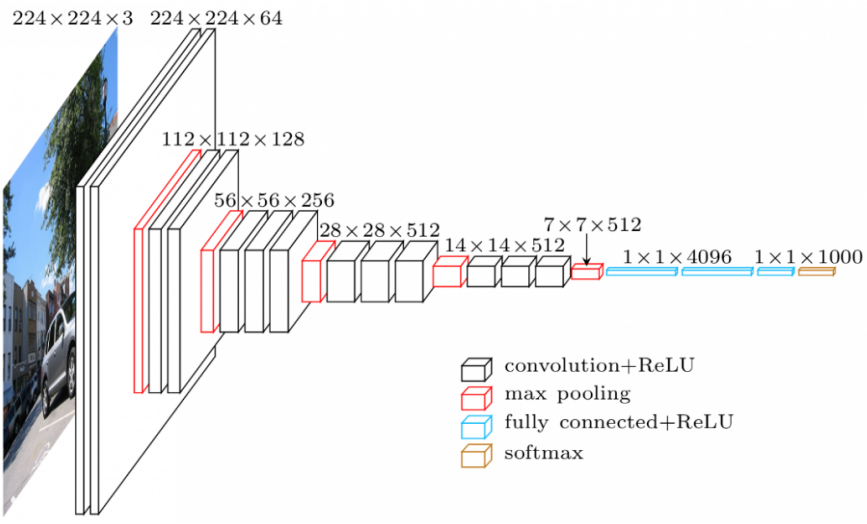

Fig. 1: VGG-16 neural network architecture.

the ILSVRC contest because of the large number of parameters and convolutional layers that contains. The task in the competition is to classify 1,000 RGB images from the ImageNet dataset [14]. The VGG-16 network was the runner up of this competition and GoogLeNet [8] was the winner with an accuracy of 89%. GoogLeNet is deeper than the VGG-16 network, but the number of parameters is reduced because of the use of inception modules. Thus, parallelising the VGG-16 network in an embedded system is more challenging.

The VGG-16 network consists of 16 layers, where convolutional layers (13) with 3x3 filters and 2x2 max pooling layers are stacked. Between these layers, the *relu* activation function is applied. Then, there are three fully connected layers which contain most of the parameters of the network. Finally, a *softmax* function is used to produce the probabilities for each class. Figure 1 depicts the exact architecture. Note that an important downside of VGG-16 is the demanding memory usage because of its enormous number of parameters (140M), which leads to a deceleration of the performance and wasteful power consumption.

3 Deep Learning Optimisation Techniques in Embedded Systems

We now discuss some related work on different deep learning optimisations techniques in low power heterogeneous embedded systems that include GPUs. Two types of optimisation techniques are analysed. One that jointly leverages mathematical and machine learning concepts in order to reduce the memory requirements and speed up the computations, and another that exploits the hardware architecture of the embedded devices.

It is known that CNNs require a lot of memory because of the large amount of parameters they generate. However, many of these parameters are redundant. Hence, removing them will not affect the accuracy of the prediction. Various compression techniques such as convolutional tensor using matrix approximations, low rank tensors approximations, and monochromatic convolution approximations have been proposed in [15]. In [16] it is presented a novel technique of dimensionality reduction of the parameters that combines pruning, vector quantization, and huffman coding and achieves 35X to 45X reduction of the memory requirements without affecting the inference accuracy. Finally, other algorithms that present compression techniques of the weights are proposed in [17,18]. However, all these methods require re-training of the neural network (typically in very powerful GPUs like NVIDIA Titan) in order to obtain the final compressed weights, and we are targeting pre-trained networks.

It is also known that the memory bandwidth is a bottleneck to the inference speed. Several methods that try to reduce the memory bandwidth are proposed in the literature. In [19], Maxout networks [20] are trained using different float precision for the weights, proving that using low precision floats is sufficient for the inference and even for the training phase. However, simply reducing the float precision in more complex networks would degrade the accuracy. This issue is studied in [21] for LeNet [22], CovNet [23] and GoogLeNet [8]. The work is inspired by the idea that variations of the accuracy do not only depend on reducing the floating precision across the whole network but of every layer. Finally, in [24] the use of fixed-point and floating-point representations in data is proposed to reduce the memory bandwidth. This technique leverages the fact that multiplication is more efficient when is operated on floating-point data and the addition is more optimal when is operated in fixed-point data. Note that in this work we focus on leveraging the target architecture. We leave the memory bandwidth problem for future work.

The second set of techniques are based on exploiting the hardware architecture and parallelising the inference procedure on the GPU to boost the performance. We focus on embedded systems that contain low power GPUs. The *DeepX* framework [25] accelerates the execution time of the inference by dynamically decomposing the neural network architecture into segments. Then each segment is executed in a different computing device (e.g. CPU, GPU, etc). In addition, *Runtime Layer Compression* is applied on-the-fly only to individual layers, thus reducing drastically the computing resources required. *DeepSense* [26] is another framework developed for low power GPUs that leverages the architecture of GPU devices like Mali and Adreno, and achieves high performance on the inference for various deep CNNs such as AlexNet, VGG-16, VGG-M, and VGG-F. It includes several optimisations techniques such as *branch divergence elimination* and *vectorisation*, which take advantage of the SIMD instructions supported by Mali and Adreno GPUs. Finally, *CNNdroid* [27] is another framework optimised for android phones that contains all types of layers. Similar to [26], CNNdroid takes advantage of the SIMD instructions of the Mali GPU. In this work we consider similar optimisations to [26,27] and also others.

4 Accelerating the VGG-16 Neural Network

We now describe the proposed implementations of VGG-16 in OpenMP and OpenCL, along with the optimisation techniques used to accelerate the inference further on the ARM Mali GPU. Since we observed that the convolutional layer requires roughly 90% of the inference time, we focus on examining parallelisation techniques for this layer. Note that in order to check the inference accuracy of each parallel version, we compared the predicted class of each image with the one provided by the serial version of the code.

4.1 Experimental Setup

Hardware Platform. The ODROID-XU4 board includes the ARM Cortex-A15 and Cortex-A7 big.LITTLE CPU, the low power ARM Mali-T628 GPU, and 2Gbyte of shared LPDDR3 RAM. The GPU supports 64-bit data types (scalar/vector, integer/float) which makes it suitable for accelerating applications that require significant computations like deep neural networks.

Imagenet Dataset. This dataset contains 15 million RGB images classified in 22,000 classes [14]. However the dataset used in this work is actually a subset of ImageNet with 1.2 million training images, 50,000 validation images and 150,000 testing images classified into 1,000 categories. Since the images are not fixed size, they are cropped to 224x224 pixels which is the input size that VGG-16 accepts.

Libraries and Methods. We selected a serial version of the pre-trained VGG-16 neural network implemented in C [2]. OpenMP 4.0 was used for the CPU parallel version. Finally, for the OpenCL version we used OpenCL 1.1 and the OpenCL C++ wrapper API 1.1 which simplifies the host code. Furthermore, the CLBlast library [28] was used and optimised for the Mali-T628 GPU.

4.2 OpenMP Implementation

OpenMP is an API for shared memory parallelisation (i.e. all processors use the same address space). Since OpenMP does not support yet ARM Mali GPUs, the VGG-16 network is parallelised only on the CPU of the board using up to 8 threads (cores) of the Cortex-A processor. Specifically, the outer *for* loop of the convolutional layer is parallelised using dynamic scheduling of threads (because of the different amount of data required to process in each loop). Besides, the execution of the threads among the layers is synchronised because each output is the input of the next layer, so we have to wait until all the operations from the previous layer finish. OpenMP suffers from some overheads caused by threads initialisation, loops scheduling, etc. The results of the average execution time per layer, compared with the serial C version, are shown in Figure 3. As we see, the OpenMP implementation is much more efficient than the serial code, decreasing the overall inference time of one image from 79 to 28 seconds (2.8X speed up).

[2] `https://github.com/ZFTurbo/VGG16-Pretrained-C`

4.3 OpenCL Implementation

The main program which contains the specifications of the architecture and the work flow of the VGG-16 network (host code) is executed on the CPU sequentially. Since the different layers of the network are parallelised and can potentially run on both CPU and GPU devices, the OpenCL code needs to be carefully designed to avoid data transfer overheads that may degrade the overall performance of the inference.

OpenCL kernels communicate with the host through buffers, which can be accessed directly from memory pointers. So, the arrays in the GPU are handled as they are represented in memory, i.e. as 1-dimensional arrays. As a result, all the matrices are transformed to 1-dimensional arrays and passed through the buffers to the kernels. This transformation is performed in the host code at the start of the program. Then, all layers handle 1-dimensional arrays and the final output is reformed back to a multidimensional matrix. However, the transformation of the matrices is not a simple procedure. A naive option is depicted in Figure 2, where the matrices are simply flattened row by row.

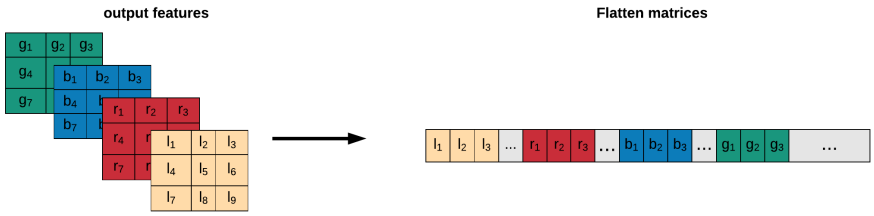

Fig. 2: 3D input to 1D array row by row transformation.

Convolutional Layer. We now discuss several methods to optimise the performance of the convolutional layer by exploiting the architecture of the GPU.

– **Baseline implementation**: The *Index Space* (NDRange) determines the total number of work-items and the number of work-items per work-group, thus being an important aspect of OpenCL that can dramatically affect the performance of the code. A naive implementation can lead to poor results, so a careful exploration of the design space needs to be done. For example, a naive design space for the convolutional layer is that each work-item computes one pixel of the output, and we leave the OpenCL driver to determine the size of the work-groups. This design could accelerate the performance of the VGG-16 neural network but surely is not the optimal one. Figure 3 also shows the execution times for the baseline OpenCL version. As can be seen, the times are actually worse than the serial code for all the layers excepting for the last three (we assume that for these layers the OpenCL driver finds a good value for the work-group size), providing an overall speed up of

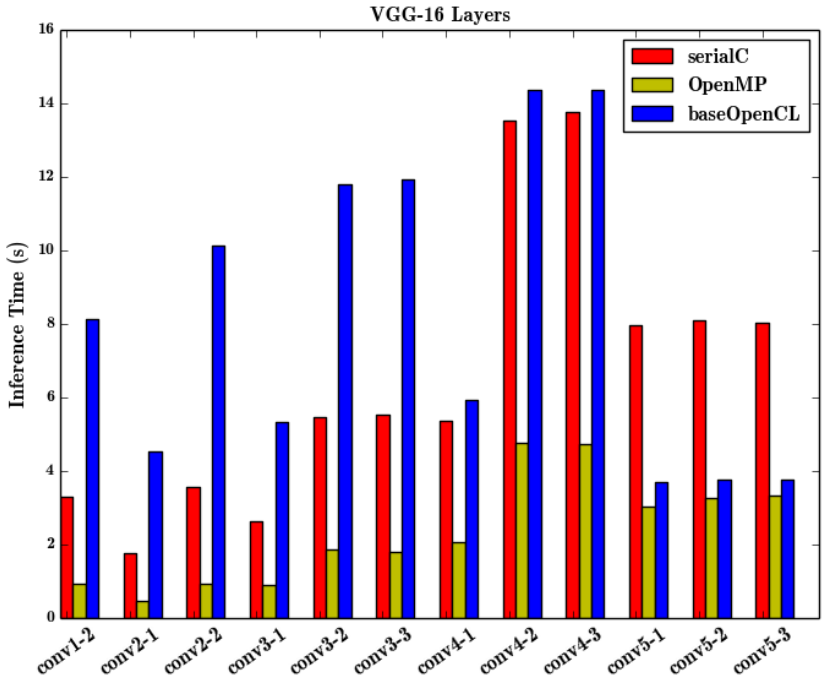

Fig. 3: Time for Serial C, OpenMP, and baseline OpenCL implementations.

0.8X over the serial code. Note that this implementation uses the row-by-row matrix transformation discussed previously (Figure 2).

- **Work-groups optimisation**: Another design option is to explicitly indicate the size of the work-groups we want to use. Note that Mali GPUs do not have local memory, so using work groups' local memory will not increase the throughput of the implementation. However, the 16KB L1 cache memory (which has 64 bytes cache line size) can be used effectively with work-groups, thus speeding up the performance. The most suitable work group size for most Mali GPUs is a multiple of two, as there are 2 arithmetic pipelines per core. We tested work-group sizes of 1x1, 2x2, 4x4, 8x8, and 16x16 work-items and found that the best option is size 4x4. As the work-group size increases from 1x1 to 4x4 the performance is enhanced (up to 2.3X over the baseline). However, when the work group size is increased even more, then more L1 cache misses occur and the performance is degraded. Figure 7 shows the average speed up per layer using the optimal (4x4) work-group size.
- **Kernel Vectorisation**: The kernel code can be optimised further by exploiting the architecture of the Mali GPU, which supports SIMD (Single Instruction, Multiple Data) instructions in each shader core, whose operations are referred to as vector operations. OpenCL provides built-in *vector data types* where the I/O operations can be reduced. Read and write in

memory can be performed efficiently using the OpenCL's functions *vload[n], vstore[n]* where n indicates the number of elements that are loaded/stored in parallel in one processor cycle (permitted values are 2, 4, 8 and 16). Vector data types give us the possibility of parallelising the convolutional layer further. However, these functions read and write bytes in consecutive memory addresses. So, in order to be able to fully exploit the benefits of the vector data types we have to transform the matrices in a different way. That is, instead of flattening the matrices by rows (Figure 2) we flatten them by depth (Figure 4). Thus, pixels with same index and different depth are stored consecutively in memory. In this way it is possible to read mutiple data in one processor cycle and reduce the I/O overhead. For example, in the second convolutional layer the load operations are reduced from 64×9 to 4×9.

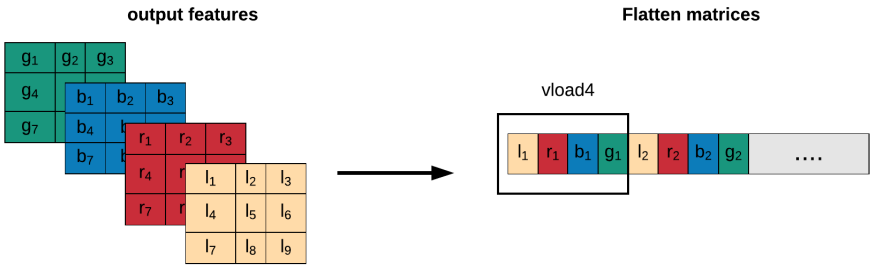

Fig. 4: Flatten matrices by depth.

Figure 7 also shows the average speed up per layer after applying the two optimisations described so far to the baseline OpenCL implementation. This vectorised version reduces the overall inference time of the best work-groups version from 42 to 8.5 seconds on average (5X speed up). We see how the vectorised kernels improve the average performance of every layer. However, the fluctuations in the inference time per layer are due to the max pooling layers that interfere among the convolutional layers and reduce the input matrices dimensions.

Convolution as Matrix Multiplication. We now optimise the VGG-16 neural network by transforming the convolution operation to a general matrix-matrix multiplication. It is well-known that matrix multiplication is one the of the most optimised operations in GPUs. We accelerate the inference time by using the *CLBlast* library [28], which is an open source OpenCL BLAS library that provides optimised routines for a wide variety of devices including the Mali-T628 GPU (it is similar to the cuBLAS library that supports only NVIDIA GPUs). However, using the CLBlast library for the convolution operation is not trivial, as it requires changes in the input matrices.

The CLBlast library provides accelerated low level routines for dense algebra operations, among others the GEMV (Generalised Matrix-Vector Multiplication)

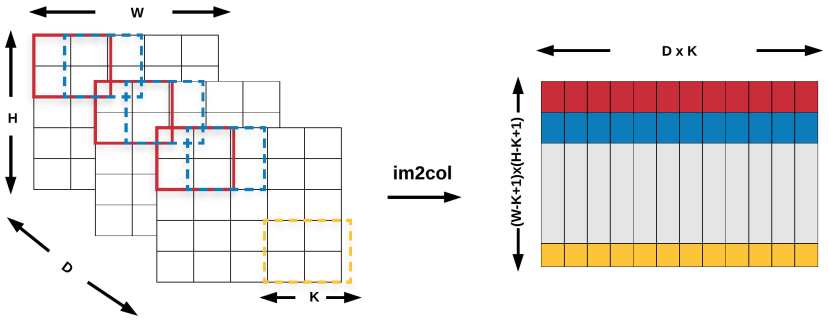

Fig. 5: Im2Col operation in input matrices.

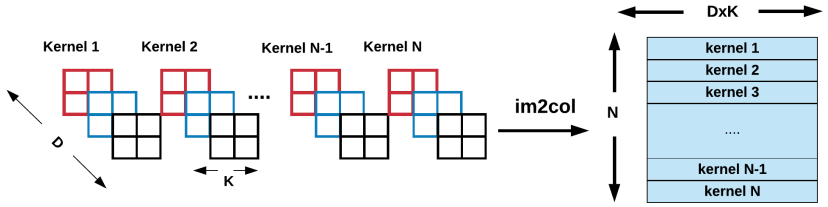

Fig. 6: Im2Col operation in kernels.

and GEMM (Generalised Matrix - Matrix Multiplication) routines. CLBlast is implemented in OpenCL and can be *tuned* using CLTune [29] for several GPUs. In the proposed implementation the *GEMM* routine is used, which contains 14 parameters that define for example: *work group size*; *register tiling configuration*; *vector widths* of both matrices; *loop unroll factors*; whether to use local memory or not; etc. It is worth to remember again that memory in the Mali-T628 GPU is unified so using local memory would not increase the throughput of our program.

How GEMM works for Convolutions? In order to apply the *GEMM* routine to convolution operations we need to change the structure of the matrices in a way that the matrix multiplication would give the same results as the convolution operation (Figures 5 and 6). The feature matrices have dimensions $D \times H \times W$ where D is the depth, H is the height, and W is the width of each feature map. The kernel matrices have dimensions $D \times N \times K \times K$ where N stands for the number of kernels and is the output depth of the feature maps, D is the input depth of channels, and K is kernel's height and width (note that kernels are always square matrices).

Therefore, it is required to transform the 3-D matrices to 2-D ones. This transformation can be done by using the *im2col* operation, which rearranges image blocks to columns. As a result, the input matrices after the *im2col* operation are transformed to matrices with $(H - K + 1) \times (W - K + 1)$ rows and $D \times K$ columns (Figures 5). On the other hand, the *im2col* operation for kernels is simpler (Figure 6). Each kernel is simply flattened from 3 dimensions

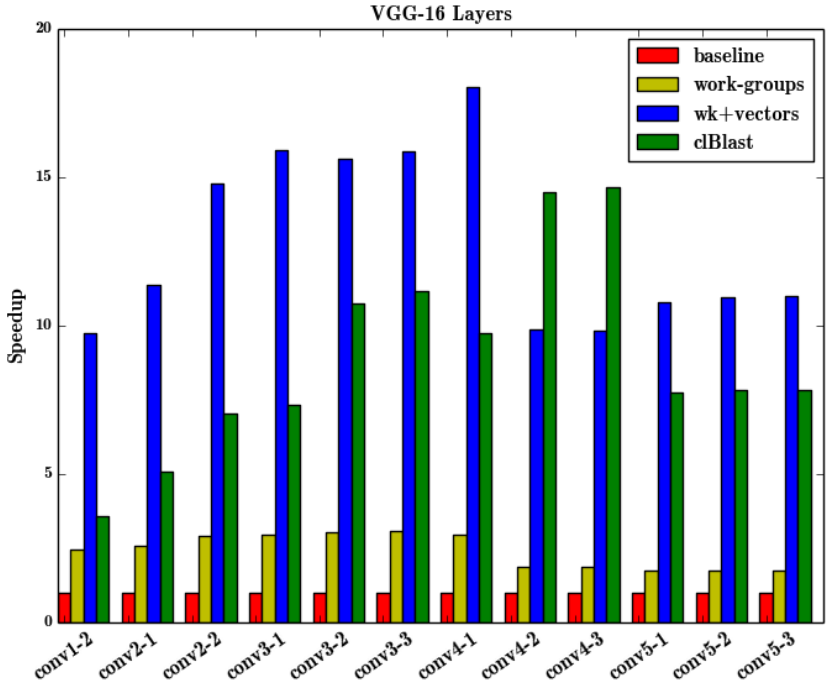

Fig. 7: Inference time speedup for work-groups, work-groups+vectorised, and CLBlast OpenCL implementations (baseline OpenCL = 1).

to 1. Consequently, the kernels are transformed to a matrix with N rows and $D \times K$ columns. Note that all these matrix transformations are performed on the board's CPU.

Figure 7 also shows the speed up for the OpenCL version using the CLBlast library. As can be seen, the average improvement per layer is less than the vectorised version excepting for layers conv4-2 and conv4-3 (probably because CLBlast finds very good parameters for them). Also, the speed up increases for successive layers according to the amount of work each kernel needs to perform to compute an output pixel. Therefore, smaller matrices may have less improvement. The overall inference time is reduced from the 98 seconds of the baseline OpenCL implementation to 13 using the CLBlast version (7.8X speed up).

CLBlast is a useful library for deep learning applications. However, according to [28] the *GEMM* kernel is optimised for specific values of parameters. The parameters can be overrided and tuned according to the parameters of our program by using the function *OverrideParameters*. Finally, note that this implementation has an important drawback. The size of the matrices that have to be processed becomes almost doubled. This means that the power consumption will increase because of the large data transfer between the CPU and the GPU. However, the improvement in performance outweighs the cost of the sizes.

5 Conclusions and Future Work

We have presented multiple parallel versions of the VGG-16 neural network for the CPU and GPU of the ODROID-XU4 board using both OpenMP and OpenCL programming frameworks. The optimisations investigated for OpenCL take into account the hardware specifications of the ARM Mali-T628 GPU present on the board. The OpenMP implementation, that runs exclusively on the CPU, reduced the inference time of the serial version from 79 to 28 seconds. The baseline OpenCL implementation, in which no optimisations were applied to the kernels or the index space, achieved $98s$ of average inference time which is worse than OpenMP and the serial version of the code. Then, the utilisation of work groups boosted the performance achieving an inference time of $42s$. Additionally, the vectorisation of the kernels, which takes advantage of the SIMD instructions of the Mali GPU by using OpenCL *vector data types*, further enhanced the inference time to $8.5s$. Finally, the CLBlast library was also used to boost the performance of the inference to achieve an overall inference time of $13s$. Note that CLBlast does not contain a routine to perform the convolution operation. However, by changing the structure of the matrices the convolution operation can be transformed to a general matrix multiplication, which is a built-in routine (*GEMM*) in CLBlast that can be optimised for the Mali-T628 GPU.

A possible extension of this work would be to investigate other techniques such as transforming the convolution operation to a FFT operation. This method would resolve the usage of extra memory space required by the matrix multiplication. Moreover, the Mali-T628 GPU contains 2 GPU devices which have 4 and 2 compute units respectively. In this work we have only used the 4 units of the first device. Thus, another possible extension would be to parallelise the VGG-16 network using both devices by redesigning the index space and break up the matrices appropriately in order to be able to run in different queues. Furthermore, we haven't taken into account the power consumption of the ODROID-XU4 board during the inference, which is a very important factor in embedded systems that must be considered. Specifically, the power consumption can be decreased by reducing the amount of data transferred between the CPU and the GPU devices. This can be achieved by lowering the float precision of the data, although it is not so trivial because the accuracy of the network may degrade. Hence, new approaches to save power without significant loses of accuracy in the predictions must be explored. Finally, we also want to explore optimisations for other deep neural networks and low power heterogeneous embedded devices.

Acknowledgment

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 732204 (Bonseyes). This work is supported by the Swiss State Secretariat for Education Research and Innovation (SERI) under contract number 16.0159. The opinions expressed and arguments employed herein do not necessarily reflect the official views of these funding bodies.

References

1. Y. Zhang, M. Pezeshki, P. Brakel, S. Zhang, C. Laurent, Y. Bengio, and A. C. Courville, "Towards end-to-end speech recognition with deep convolutional neural networks," *CoRR*, vol. abs/1701.02720, 2017. [Online]. Available: http://arxiv.org/abs/1701.02720
2. Y. Zhang, W. Chan, and N. Jaitly, "Very deep convolutional networks for end-to-end speech recognition," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 4845–4849.
3. M. Ravanelli, P. Brakel, M. Omologo, and Y. Bengio, "A network of deep neural networks for distant speech recognition," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 4880–4884.
4. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
5. X. Wu, "High performance binarized neural networks trained on the imagenet classification task," *CoRR*, vol. abs/1604.03058, 2016. [Online]. Available: http://arxiv.org/abs/1604.03058
6. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.
7. M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316
8. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 1–9.
9. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, 2013.
10. R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, "Understanding deep neural networks with rectified linear units," *CoRR*, vol. abs/1611.01491, 2016. [Online]. Available: http://arxiv.org/abs/1611.01491
11. S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML)*, 2015, pp. 448–456.
12. G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580
13. K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *International Conference on Learning Representations (ICLR)*, 2015.
14. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.
15. E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," in *Proceedings*

of the 27th International Conference on Neural Information Processing Systems - Volume 1, ser. NIPS'14, 2014, pp. 1269–1277.
16. S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," in *International Conference on Learning Representations (ICLR)*, 2016.
17. G. Soulié, V. Gripon, and M. Robert, "Compression of deep neural networks on the fly," in *Artificial Neural Networks and Machine Learning – ICANN 2016: 25th International Conference on Artificial Neural Networks*. Springer International Publishing, 2016, pp. 153–160.
18. W. Chen, J. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing convolutional neural networks in the frequency domain," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, 2016, pp. 1475–1484.
19. M. Courbariaux, Y. Bengio, and J. David, "Low precision arithmetic for deep learning," *CoRR*, vol. abs/1412.7024, 2014. [Online]. Available: http://arxiv.org/abs/1412.7024
20. I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013, pp. III–1319–III–1327.
21. P. Judd, J. Albericio, T. H. Hetherington, T. M. Aamodt, N. D. E. Jerger, R. Urtasun, and A. Moshovos, "Reduced-precision strategies for bounded memory in deep neural nets," *CoRR*, vol. abs/1511.05236, 2015. [Online]. Available: http://arxiv.org/abs/1511.05236
22. C.-C. J. Kuo, "Understanding convolutional neural networks with a mathematical model," *Journal of Visual Communication and Image Representation*, vol. 41, pp. 406 – 413, 2016.
23. Y. Li, B. Sun, T. Wu, and Y. Wang, "Face detection with end-to-end integration of a convnet and a 3d model," in *Computer Vision – ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III*. Springer International Publishing, 2016, pp. 420–436.
24. L. Lai, N. Suda, and V. Chandra, "Deep convolutional neural network inference with floating-point weights and fixed-point activations," *CoRR*, vol. abs/1703.03073, 2017. [Online]. Available: http://arxiv.org/abs/1703.03073
25. N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "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 (IPSN)*, 2016, pp. 23:1–23:12.
26. L. N. Huynh, R. K. Balan, and Y. Lee, "DeepSense: A GPU-based Deep Convolutional Neural Network Framework on Commodity Mobile Devices," in *Proceedings of the 2016 Workshop on Wearable Systems and Applications (WearSys)*, 2016, pp. 25–30.
27. S. S. L. Oskouei, H. Golestani, M. Kachuee, M. Hashemi, H. Mohammadzade, and S. Ghiasi, "GPU-based Acceleration of Deep Convolutional Neural Networks on Mobile Platforms," *CoRR*, vol. abs/1511.07376, 2015. [Online]. Available: http://arxiv.org/abs/1511.07376
28. C. Nugteren, "Clblast: A tuned opencl BLAS library," *CoRR*, vol. abs/1705.05249, 2017. [Online]. Available: http://arxiv.org/abs/1705.05249
29. C. Nugteren and V. Codreanu, "CLTune: A Generic Auto-Tuner for OpenCL Kernels," *CoRR*, vol. abs/1703.06503, 2017. [Online]. Available: http://arxiv.org/abs/1703.06503