Rovder, S., Cano, J. and O'Boyle, M. (2019) Optimising Convolutional Neural Networks Inference on Low-Powered GPUs. Twelfth International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2019), Valencia, Spain, 21 Jan 2019.

There may be differences between this version and the published version. You are advised to consult the publisher's version if you wish to cite from it.

http://eprints.gla.ac.uk/183820/

Deposited on: 11 April 2019

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

Optimising Convolutional Neural Networks Inference on Low-Powered GPUs

Simon Rovder[1], José Cano[2], Michael O'Boyle[1]

[1] School of Informatics, University of Edinburgh, UK
[2] School of Computing Science, University of Glasgow, UK

Abstract. In this paper we present effective optimisation techniques for accelerating convolutional neural networks inference on low-powered heterogeneous devices with OpenCL. Using LeNet and VGG-16 as test networks, we implement a custom neural network system in OpenCL and optimise it to minimise their inference times. Our baseline system shows a speedup of 17x for LeNet. We also outline two methods for fast convolution: an iterative vectorised approach and a Morton GEMM based approach. The two approaches demonstrate VGG-16 inference speeds up to 3x faster than current state-of-the-art systems and outperform other custom neural network systems by speedup factors of up to 1.82x.

1 Introduction

Neural networks are currently at the forefront of the field of machine learning. They have shown incredible versatility across a large spectrum of problems ranging from image and sound recognition all the way to natural language processing [4,23,22]. This makes them a desirable tool for all platforms and devices. The primary driving force behind recent neural network research is the progress in development of Graphics Processing Units (GPUs). Until recently, neural networks were too computationally complex to be trained or executed in reasonable amounts of time. GPUs have enabled fast training on affordable hardware and thus brought neural networks into the research spotlight.

At the same time, the rapidly growing popularity of smartphones has attracted GPUs to the world of battery powered devices, opening the doors to utilising neural networks on these devices as well. The main issue faced when attempting to execute neural networks on these low-powered GPUs is a noticeable lack of software support. Considerable amounts of research and development have been dedicated to training complex neural network models on desktop GPUs and CPUs, resulting in numerous neural network frameworks like TensorFlow [1], Caffe [13], Torch [8] or Theano [27]. These frameworks, however, are either not supported on the low-powered devices or are not optimised for the architectural differences between desktop GPUs and low-powered GPUs.

In this paper, we address these issues by investigating recent research into neural network execution on low-powered GPUs, and by benchmarking state-of-the art solutions on the Mali T-628 GPU. We take our research a step further by

implementing custom OpenCL kernels for neural network execution to show how exploiting detailed knowledge of the device architecture (e.g. number of GPU cores, cache line size) can be used to outperform state-of-the-art systems.

The main goal of this work is to evaluate CLBlast [21] (an automatically-tuned deep-learning-enabled library) on the Mali T-628 GPU, investigate its shortcomings and limitations, and attempt to improve upon its performance by hand-crafting kernels for the task. Our hypothesis is that the CLBlast automatic tuning process does not find optimal configurations for the device because it cannot exploit detailed knowledge about the hardware architecture.

To compare CLBlast with our custom-made system, we benchmark the two systems on executing LeNet [18] (one of the very first convolutional models ever published) and VGG-16 [25] (one of the state-of-the-art models for image classification). These two models were chosen to allow us to benchmark the systems against two scenarios: propagating a batch of 100 inputs over a small network (LeNet), and propagating a single input over a large network (VGG-16). The two scenarios are similarly challenging from a computational perspective, yet architecturally differ enough to potentially require different optimisations.

The contributions of this work to neural network execution on low-powered GPUs include:

– Designing a custom system for executing neural networks. We implement the system by hand crafting OpenCL kernels and optimising them for the Mali T-628 GPU. The source code is available online[3].
– Proposing optimisations to outperform other state-of-the-art systems. We demonstrate a 17x speed increase over LeNet with CLBlast, a 3x speed increase over VGG-16 with CLBlast, and a 1.82x speed increase over [19].

2 Background

2.1 OpenCL

OpenCL [26] is a framework for writing and executing code in parallel across one or more heterogeneous platforms, and in this work we use it to offload the computationally expensive neural network operations onto the Mali T-628 GPU. The OpenCL paradigm is designed around splitting a program into:

– **Host-code**: The part of the program intended to run on the CPU. This part can be written in a complex high-level object oriented language (several OpenCL bindings are available[4]) and it runs under the same conditions as any other program we run in every-day computer use.
– **Device-code**: The part of the program intended to run in parallel on the GPU. OpenCL defines its own programming language for device-code (based on C-99) and is compiled by OpenCL at runtime for individual devices. The device-code is designed to perform one particular task very quickly.

[3] https://bitbucket.org/SimonRovder/t628nn
[4] E.g. Java [12], C++ [11], Python [16] and Haskell [10]. We use Python and C++.

The OpenCL paradigm is based on taking loop-based algorithms and parallelising their execution across a subset of the loops. It is safe parallelise any subset of the loops as long as no dependency criteria are violated, which could lead to memory race conditions. The basic OpenCL terminology is as follows:

- **OpenCL Device**: A device capable of executing OpenCL code.
- **Compute Unit (or Core)**: A hardware component within an OpenCL device capable of executing a *kernel* on a *workgroup*. An OpenCL device will usually contain one or more compute units.
- **Kernel**: A single function within the device-code.
- **NDRange**: The term for the range of ids a kernel is mapped onto.
- **Workgroup**: A subset of the *NDRange* small enough to execute its corresponding *work items* on a single compute unit.
- **Work Item**: A particular instantiation of a kernel.
- **Buffer**: A host-code object that references device memory.

2.2 Neural Networks

Neural networks are a machine learning model inspired by the operation of biological neural networks. The model comprises of mathematical operations chained together to transform input vectors into output vectors. Most common operations include affine transformations, pooling, convolution, normalisation and nonlinearities, the internal parameters of which are learned via Stochastic Gradient Descent (SGD). In this work we focus on two specific networks, LeNet and VGG-16, whose functional requirements are outlined in Table 1.

Table 1. Layer requirements of LeNet and VGG-16.

Layer	LeNet	VGG-16
Fully-connected	✓	✓
ReLU		✓
Sigmoid	✓	
Convolutional	✓	✓
Max-pooling		✓
Subsampling	✓	

LeNet This network Composed of two convolutional layers (C1 and C3) interleaved with subsampling layers (S2 and S4) much like modern networks have max-pooling. These layers are then followed by two fully-connected layers (F5 and F6). A diagram of the architecture can be seen in Figure 1.

VGG-16 There are several different network layouts published in [25], the most commonly used one being Model C (Table 1 of their paper). This is the model we focus on. Its architecture is visualised in Figure 2, which includes thirteen convolutional layers and three fully-connected layers.

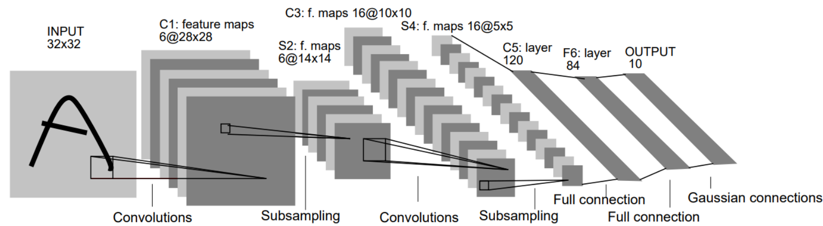

Fig. 1. LeNet architecture from [18].

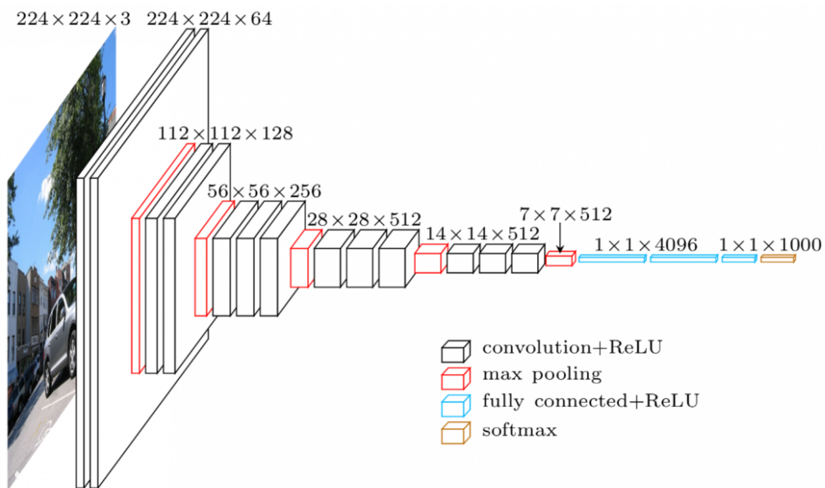

Fig. 2. VGG-16 architecture from [25].

2.3 Mali-T628 GPU

In this work we use the Odroid XU3 board that includes the low power ARM Mali-T628 GPU and 2Gbyte of LPDDR3 RAM. The compute units of the GPU are accessible via two separate devices (i.e. 4+2 compute units), and a relevant metric to it is the **GFLOPS** (amount of billions of floating point operations performed per second) limit, which is 17 GFLOPS per Compute Unit [15].

3 Related Work

3.1 Deep Learning Frameworks

TensorFlow [1] is one of the most popular neural network and machine learning frameworks used today. It provides an interface for expressing machine learning algorithms in terms of Tensor operations and maps these operations onto optimised GPU kernels. TensorFlow often makes use of existing highly optimised kernels which are parts of other libraries, including cuBLAS [9], cuda-convnet [17]

and cuDNN [7]. These libraries are optimised for the NVIDIA CUDA architecture and optimisations that improve performance on NVIDIA GPUs often have a negative impact on the Mali architecture. The Mali developer guide recommends removing some of the optimisations tailored to other architectures [3]. Other widely used frameworks are Caffe [13], Torch [8] and Theano [27].

3.2 CLBlast

On of the most recent general-purpose machine-learning-enabled library optimised for the Mali T-628 GPU was released in May 2017 under the name CLBlast [21]. Inspired by the clBLAS library [2], CLBlast is an automatically tuned BLAS (Basic Linear Algebra Subroutines) library, which implements all BLAS operations as well as batched versions of some of them. Apart from BLAS operations, CLBlast also provides us with an *im2col* subroutine, which can be used to perform convolution. With this functionality, CLBlast can be used for implementing a deep neural network execution system.

There are several reasons why CLBlast is very relevant to our research. Firstly, it tunes itself to the device it runs on and the Mali T-628 is one of the 40 devices used to test the library during its creation [21]. Secondly, it is capable of running matrix multiplication on the Mali T-628 GPU at 8 GFLOPS [21], which is extremely fast compared to other research [14] if it is accomplished without rearranging matrices in memory. These facts make CLBlast as close to state-of-the-art as we can get with our device. A disadvantage of using CLBlast is its lack of pooling layer support and activation function support, which are important and popular neural network features.

3.3 Optimising GPU kernels

Memory access patterns are the main deciding factor of how fast a GPU kernel will be, and as such, have been a target of many studies [5] [20] [24]. It is so crucial to optimise memory patterns, that APIs were designed to assist programmers in designing kernels with optimal memory access patterns [6]. However, most of this research focuses on CUDA architectures and is not applicable to OpenCL low-powered GPUs like the Mali T-628. It is worth noting, however, that improvements in kernel performance in these studies have been gained by remapping elements in memory, in order to have a memory access pattern better suited for the particular hardware optimisations. In our work we shall make use of the same general optimisation techniques, yet we shall target them directly at the hardware optimisations of the Mali T-628 GPU.

Similar research to ours was recently done in [19], where the primary issue is its focus on convolutional layers only. Research by Nocentino and Rhodes [20] into using Z-Morton memory layouts to provide faster access to individual regions of 2-dimensional data has provided the inspiration for using variants of these layouts for optimising matrix multiplication in Section 5.2 of this work. Finally, in [29] it is performed an across-stack investigation of different techniques to determine their impact on inference and runtime performance.

4 Executing LeNet with Custom System

The first step is to create a very simple baseline system that can execute LeNet, which we can then build on top of. According to Table 1, we implement the required four layers in the following three single kernels:

- **Fully-Connected Layers**: A baseline matrix multiplication kernel amended with bias addition and sigmoid activation.
- **Convolutional Layers**: A loop-based variant of convolution that avoids any memory blowup introduced by alternative methods such as *im2col*.
- **Subsampling Layers**: A separate kernel that avoids having to simulate subsampling e.g. using convolution, which adds redundant computation.

Combining the three custom kernel implementations we can fully execute LeNet on the Mali T-628 GPU. To compare the performance against CLBlast, we forward propagate a batch of 100 input images and measure the run time. We timed both kernel time and total wall time[5]. Timing both is important because CLBlast does not give us access to the underlying OpenCL `Event` objects of the executed kernels, meaning the measurements for CLBlast are wall time. Comparing CLBlast's wall time to our kernel time would not be a representative comparison. The reason for timing kernel time as well is to give us insight into the kernel initialisation overhead. The resulting times can be seen in Figure 3 and Table 2. There are a number of observations we may take from these results.

First of all, we now have concrete measurements of kernel initialisation times. We can find this value by subtracting the kernel time from the wall time, which gives us the infrastructural overhead of executing the GPU kernel in the first place. This overhead varies with the kernel itself yet remains between 16 and 31 milliseconds for all of them, which is substantially less than the 76 millisecond overhead observed for CLBlast (note that we benchmarked the individual library subroutines relevant to neural networks, i.e. xCOPY, xIM2COL, xGEMM, xGEMMBATCHED). The results from Table 2 indicate that kernel initialisation is the primary component of our baseline's inference time, meaning we cannot meaningfully improve upon these results. Optimising the layer kernels would result in a reduction of kernel time, which would have insignificant impact on the overall runtime of many of the layers. Take F6 as an example: the overall run time of F6 is 30.15 milliseconds, 0.77 of which is kernel time.

Second, the results also indicate that for small models like LeNet even an unoptimised basic OpenCL implementation can outperform CLBlast by a significant margin. We discovered that the xGEMM subroutine decides whether to perform direct or indirect multiplication based on the size of the matrices. When multiplying an $m \times k$ matrix with a $k \times n$ matrix, if the product mnk exceeds a certain threshold (which is called XGEMM_MIN_INDIRECT_SIZE and is automatically found during the tuning stage), the indirect approach is selected and the matrix is re-shaped in memory, changing layouts to perform multiplication

[5] Wall time being the actual amount of time the operation appeared to take.

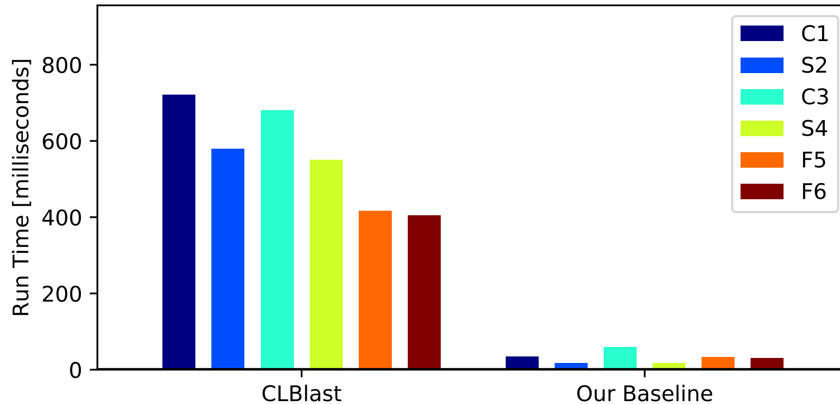

Fig. 3. Comparison of CLBlast runtime to our baseline custom implementation runtime at forward propagating 100 inputs through LeNet.

Table 2. Comparison of CLBlast runtime to our initial baseline implementation runtime at forward propagating 100 inputs through LeNet (times in milliseconds).

Layer	CLBlast	Our custom system			Wall Speedup
		Wall time	Kernel time	Overhead	
C1	721.14	34.81	14.49	20.32	**20.72**
S2	579.78	17.30	1.02	16.28	**33.52**
C3	681.30	60.04	40.27	19.77	**11.35**
S4	550.48	17.35	0.63	16.72	**31.72**
F5	416.53	33.90	4.20	29.70	**12.29**
F6	405.32	30.92	0.77	30.15	**13.11**
Total	3354.55	194.31	61.37	132.94	**17.26**

faster. We also discovered that the xGEMMBATCHED subroutine doesn't apply this memory optimisation at all.

We conclude that our baseline outperforms CLBlast's inference speed by a factor of 17, which is a significant improvement over state-of-the-art for small models like LeNet, and we shall now move on to a larger model: VGG-16.

5 Executing VGG-16 with Custom System

Before we execute VGG-16 with our custom system we must adapt it to support the additional requirements of this more complex network (Table 1). The first major requirement of VGG-16 is feature map size preserving convolution (using zero padding) and ReLU activation instead of the sigmoid used with LeNet. The second requirement is a max-pooling layer implementation. While max-pooling layers are very straightforward to implement, we must ensure that padding produced by the preceding convolutional layer or required by the subsequent convolutional layer is properly handled.

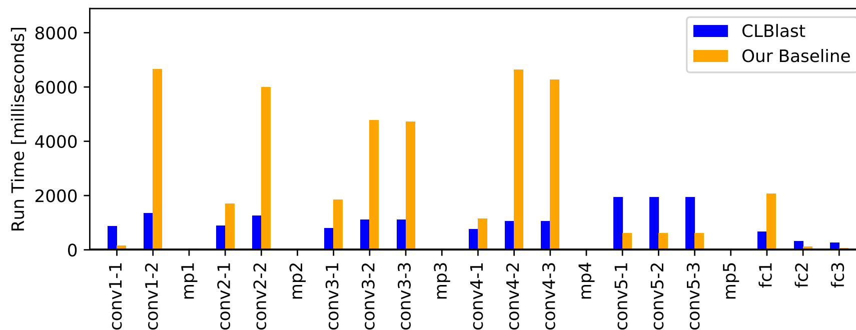

Fig. 4. Comparison of CLBlast runtime to our baseline layer implementation run time of forward propagating a single input through VGG-16

Using this adapted baseline to forward propagate one image over VGG-16 yields the results shown in Figure 4. As expected, our baseline does not perform very well at executing large convolutional layers, taking a total of 44.24 seconds to forward propagate the image (note that CLBlast needs 17.40 seconds). As opposed to the LeNet results in Table 2, however, the VGG-16 run times are mostly convolutional kernel time, so we may now proceed with optimising this kernel to drive the total inference time down by meaningful amounts.

5.1 Convolution with Workgroup Optimisation and Vectorisation

We first focus on an iterative approach by configuring workgroup sizes and vectorising the kernels. Our goal is to avoid the *im2col* memory blowup and ensure maximum overlap across the data used by all work items across all work groups executing together at any given point in time. This can be done by optimising memory layouts and specifying workgroup sizes. By specifying workgroup sizes, OpenCL can be forced to execute chunks of the NDRange in a particular order. The memory layouts of data can then be modified to achieve the desired data overlap under the forced chunk ordering. We found that the NHWC memory layout (Number of inputs, Height, Width, Channels) and a workgroup size of (1, 4, 4) provides the best performance results, reducing the time for the convolutional layers from 41.81 to 20.83 seconds. Note that computing a tightly packed cube of values in the convolution output will yield maximum data overlap across the work items required to compute that cube of output values. Since there are four workgroups executing in parallel on the GPU, they must compute the tightly packed cube of outputs together, generating a cube of 4x4x4 values.

However, the Mali T-628 GPUs primary strength lies in its SIMD instructions, specifically with the OpenCL dot function which executes on dedicated hardware. This means that we can perform the dot products concurrently with any other regular floating point multiplication, thus fully utilising the GPU hardware. Due to the NHWC memory layout, adding vectorisation to our code is trivial and further reduces the time of the convolutional layers to 6.16 seconds.

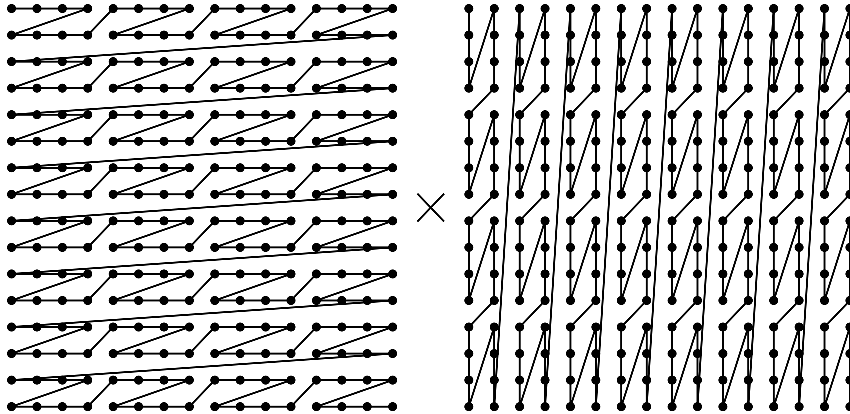

Fig. 5. Memory layouts used by the Morton GEMM kernel for the left and right matrices of the operation respectively (referred as R_2_4_R and C_4_2_C).

5.2 Convolution with Morton GEMM kernel

Our iterative convolution kernels do not fully utilise the computational power of the GPU, which we know to be 17 GFLOPS. To push performance up even further, we investigate the im2col-GEMM convolution method proposed in [30] and attempt to make it perform better than the CLBlast implementation.

The performance of matrix multiplication is highly dependent on whether data elements are accessed in the same order in which they are stored in memory. Accessing the elements in optimal order yields significant increases in performance and, contrary to that, accessing them with large strides has significant negative effects [28]. However, we have (in a sense) a combined access pattern, where data is accessed sequentially in strides (due to vectorisation).

Morton Order layouts are a middle-ground approach that minimise the cost of using a combination of row-wise and column-wise access patterns. They do this by reducing the cache miss bulk (i.e. the number of cache misses that occur simultaneously) at the cost of increasing the cache miss latency (prefetching is a common technique used to mitigate cache miss latencies by pre-emptively fetching data likely to be requested in the near future into the cache). The main idea behind Morton Order layouts is to cache lines cover a region of the matrix, instead of having cache lines linearly stretch out over either the rows or columns of a matrix. One of the most popular variants of the Morton order layouts is the **Z-Morton**, which is useful for situations in which it is unclear what order memory accesses will happen in. In our case, we have certain control over this using workgroup size specification.

In this work we use **Hybrid** variants of the Z-Morton layout and we propose a Hybrid Morton Order based general matrix multiplication kernel (*Morton GEMM* kernel for short) with varying cache miss bulk sizes and cache miss periods. The memory layouts of this kernel are visualised in Figure 5. The spots

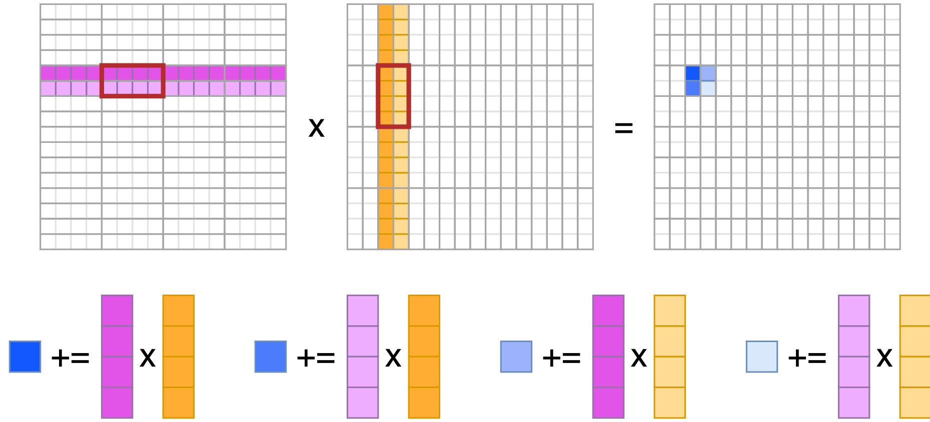

Fig. 6. Visualisation of the work done within a single work item of the Morton GEMM kernel. Values are loaded from matrices in quadruplets into the `float4` OpenCL type in a sliding window fashion. Pairwise `float4` dot products are computed using the `dot` function and added to the cumulative result.

in this Figure represent the elements in the matrix, while the lines connecting them represent the layout of the elements in memory. This kernel is inspired by the memory blocking GEMM kernel presented in the ARM reference literature specifically for the Mali T-628 GPU [14], which makes use of the OpenCL `dot` function and computes a 2-by-2 patch of the resulting matrix in a single work item (Figure 6). We improve the performance of the ARM kernel by 12% using optimal memory layouts, which enables better prefetching on the device. We optimise the Z-Morton layout for single threaded access along both rows and columns of a matrix by exploiting the control over OpenCL workgroup sizes and the order in which workgroups are executed to predict exactly which rows and columns of which matrix will be traversed at what time.

The optimal workgroup size for the Morton GEMM kernel is 4 rows by 16 columns of the NDRange. As such, a single workgroup computes an 8 row by 32 column patch of the resulting matrix (because each work item computes a 2-by-2 patch). This means the output matrix has to be padded such that its dimensions are multiples of these constants, leading to some memory redundancy. This redundancy is, however, relatively small and is a price worth paying for the substantial speed increase gained from using the Morton GEMM kernel. Note that the Morton GEMM kernel performs matrix multiplication at 13.5 GFLOPS at the *least* and exhibits no performance diminishing as matrix sizes increase.

Note that in order to execute convolution using the Morton GEMM kernel, we must also implement the following operations as OpenCL kernels: i) Padding, we need an operation that is capable of adding or removing padding from feature maps in the NCHW layout; ii) Zeroing, to avoid the memory blowup *im2col* introduces, we will reuse a single pair of OpenCL buffers across the entire network; iii) Im2col, we need an implementation of *im2col*, much like CLBlast has.

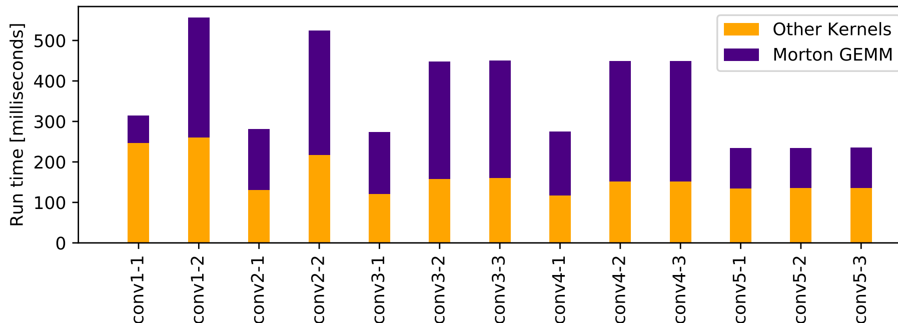

Fig. 7. A breakdown of how computation time was distributed between Morton GEMM and other infrastructure kernels for our Morton GEMM based convolution.

The robust performance of Morton GEMM results in almost identical run times of the kernel across all convolutional layers, as seen in Figure 7. The Figure also shows how much time was spent performing the surrounding operations facilitating Morton GEMM (Padding, Zeroing, Im2col), which is for some layers more than the run time of Morton GEMM itself, 55% on average.

5.3 Results and Discussion

Figure 8 shows the inference time breakdown across layers of our two fully-optimised systems (i.e. vectorised and Morton GEMM), comparing their performance to that of CLBlast and the custom OpenCL kernels published by Loukadakis et al. [19]. These results clearly show the superiority of our custom implementations over the other current state-of-the-art systems across all layers of VGG-16. Overall, our best approach (Morton GEMM) outperforms them with speedups of 3x and 1.82x respectively. Again, we have successfully demonstrated that exploiting detailed knowledge of the device architecture outperforms an automatically tuned system. The results, however, do *not* conclusively place one of our systems over the other. Each system has specific benefits.

There is one important downside to the Morton GEMM approach, we cannot fit the entirety of VGG-16 into memory at one time if we also want to allocate memory for the *im2col* buffers. This is a peculiar downside because the Morton GEMM approach does not use much more memory than the iterative vectorised approach, and it would seem that the iterative approach was ever so slightly below the device memory cap all along. This means the network had to be timed in two parts, once to propagate over the convolutional layers and once to propagate over the fully-connected layers. It is worth noting that the largest chunk of the network is the weight matrix used in the *fc1* layer, which contains 74% of the trainable parameters of VGG-16. The convolutional layers are hence only a small part of the network, and varying the parameter counts there will not help us reduce the memory demands to a consequential degree.

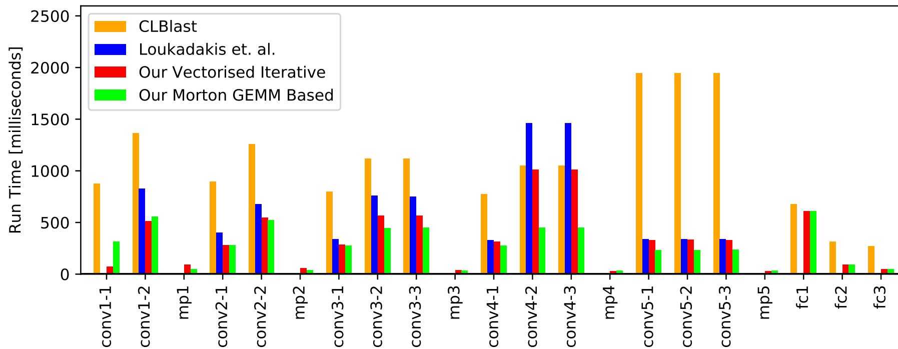

Fig. 8. Comparison of VGG-16 inference times by layer of our two proposed systems and two other state-of-the-art systems (missing or unsupported metrics omitted).

An advantage of the Morton GEMM approach is that it scales very well to deeper layers as can be seen on *conv4* layers in Figure 8. This means it is more likely to work fast for larger networks in future research. Since the actual matrix multiplication only contributes to 55% of the Morton GEMM convolution (Figure 7), there also remains great amount of optimisation potential to this implementation. Furthermore, since the approach reuses memory, the memory overhead of *im2col* becomes negligible with the amount of convolutional layers, as there will only ever be two buffers to store intermediate results in.

6 Conclusions

We have successfully shown that a hand-tuned OpenCL neural network implementation can outperform competing state-of-the-art automatically tuned systems on the Mali T-628 GPU. Our custom system showed a speedup of 17x for LeNet and 3x for VGG-16 when benchmarked against the recently published CLBlast library. Our best optimised system also showed a speedup of 1.82x over the most recent research into hand-tuning OpenCL kernels for VGG-16 inference on the same GPU [19]. Future research could attempt to minimise the overhead of infrastructural kernels we used to support performing convolution using Morton GEMM. The overheads are significant (Figure 7), and removing the overhead could lead to a further 1.6x speedup over our fastest implementation.

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. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), `http://tensorflow.org/`, software available from tensorflow.org
2. AMD: clblas, `https://github.com/Yangqing/caffe/wiki/Convolution-in-Caffe:-a-memo`
3. ARM: Mali-T600 Series GPU OpenCL Version 1.1.0 Developer Guide (2012), `http://infocenter.arm.com/help/topic/com.arm.doc.dui0538e/DUI0538E_mali_t600_opencl_dg.pdf`
4. Bengio, Y., Courville, A.C., Vincent, P.: Unsupervised feature learning and deep learning: A review and new perspectives. CoRR **abs/1206.5538** (2012), `http://arxiv.org/abs/1206.5538`
5. Bialas, P., Strzelecki, A.: Benchmarking the cost of thread divergence in cuda (2015), `https://arxiv.org/pdf/1504.01650.pdf`
6. Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: Optimizing memory access patterns for heterogeneous systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 13:1–13:11. SC '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/2063384.2063401
7. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cudnn: Efficient primitives for deep learning. CoRR **abs/1410.0759** (2014), `http://arxiv.org/abs/1410.0759`
8. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)
9. Corporation, N.: Cublas library, `https://developer.nvidia.com/cublas`
10. Gaster, B.R., Morris, J.G.: Embedding opencl in ghc haskell. In: 6th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2013) (2013)
11. Gaster, B.R.: The opencl c++ wrapper api, `https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.1.pdf`
12. Google: javacl, `https://code.google.com/archive/p/javacl/`
13. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
14. Johan Gronqvist, A.L.: Optimising opencl kernels for the arm mali-t600 gpus. In: In GPU Pro 5: Advanced Rendering Techniques. A K Peters/CRC Press (2014)
15. Karthik Hariharakrishnan, Anthony Barbier, H.F.: Opencl on mali faqs (2013), `https://developer.arm.com/graphics/resources/tutorials/opencl-tutorials`
16. Klockner, A.: Pyopencl, `https://mathema.tician.de/software/pyopencl/`
17. Krizhevsky, A.: Cuda-convnet, 2014, `code.google.com/p/cuda-convnet/`
18. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. In: Proceedings of the IEEE. pp. 2278–2324 (1998)

19. Loukadakis, M., Cano, J., O'Boyle, M.: Accelerating deep neural networks on low power heterogeneous architectures. In: 11th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2018) (January 2018)
20. Nocentino, A.E., Rhodes, P.J.: Optimizing memory access on gpus using morton order indexing. In: Proceedings of the 48th Annual Southeast Regional Conference. pp. 18:1–18:4. ACM SE '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1900008.1900035
21. Nugteren, C.: Clblast: A tuned opencl blas library. In: Proceedings of the International Workshop on OpenCL. pp. 5:1–5:10. IWOCL '18, ACM, New York, NY, USA (2018)
22. Ravanelli, M., Brakel, P., Omologo, M., Bengio, Y.: A network of deep neural networks for distant speech recognition. 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) pp. 4880–4884 (2017)
23. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M.S., Berg, A.C., Li, F.: Imagenet large scale visual recognition challenge. CoRR **abs/1409.0575** (2014), `http://arxiv.org/abs/1409.0575`
24. Siegel, J., Ributzka, J., Li, X.: Cuda memory optimizations for large datastructures in the gravit simulator. Journal of Algorithms & Computational Technology **5**(2), 341–362 (2011)
25. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR **abs/1409.1556** (2014), `http://arxiv.org/abs/1409.1556`
26. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (May 2010). https://doi.org/10.1109/MCSE.2010.69
27. Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints **abs/1605.02688** (May 2016), `http://arxiv.org/abs/1605.02688`
28. Thiyagalingam, J., Beckmann, O., Kelly, P.H.J.: Is morton layout competitive for large two-dimensional arrays yet? Concurrency and Computation: Practice and Experience **18**, 1509–1539 (2006)
29. Turner, J., Cano, J., Radu, V., Crowley, E.J., OBoyle, M., Storkey, A.: Characterising across-stack optimisations for deep convolutional neural networks. In: 2018 IEEE International Symposium on Workload Characterization (IISWC). pp. 101–110 (September 2018). https://doi.org/10.1109/IISWC.2018.8573503
30. Vasudevan, A., Anderson, A., Gregg, D.: Parallel multi channel convolution using general matrix multiplication. In: 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp. 19–24 (July 2017). https://doi.org/10.1109/ASAP.2017.7995254