An experimental study of the Snapdragon 820 using an object detection algorithm

Nuno Miguel Nobre and Graham Riley

Abstract—An object detection algorithm is essentially a robotic system's eyes. Unfortunately, the documentation and tools currently provided to programmers to optimize those algorithms for the low-power processors that can be feasibly embedded into robotic systems are scarce and lack crucial details. Here we delve into the hardware architecture of Qualcomm's Snapdragon 820 processor and disclose some undocumented figures. We show that all cores on its CPU have a 32 KiB L1 data cache and that while the low-power cluster cores use a 512 KiB L2 cache, the high-performance cluster cores boast a larger 1 MiB L2 cache. In addition, we also gather evidence that supports the usage of the clang compiler over gcc when targeting the CPU using the YOLO (You Only Look Once) object detection system as a testbed.

Index Terms—CDT Taster Project, Low-Power Devices, Qualcomm Snapdragon 820, Object Detection, You Only Look Once.

1 Introduction

Humans usually do not need more than a brief look at an image to instantly identify what objects there are, where they are and even infer how they are interacting. Indeed, our visual system, characterized by its quickness and accuracy, plays a fundamental role in allowing us to perform complex tasks like playing tennis or volleyball with little conscious thought. In fact, even if unconsciously, we have to first recognise surrounding objects to be able to interact (at least intentionally) with them. It is thus easily comprehensible why object detection is a key skill for a robot to perform tasks such as grabbing objects, opening doors and avoiding obstacles in human environments [1]. However, unlike humans' fully developed visual system, object detection is still a major challenge in computer vision and the quest for fast and accurate algorithms for object detection that would unlock the potential for general purpose and responsive robots is ongoing [2]. A good and succinct review on the subject can be found in Lu et al. [3].

One of the fastest algorithms available is You Only Look Once (YOLO) [4], [5]. Alternatives such as Deformable Parts Models (DPM) [2] use a sliding window approach where a classifier - a Support Vector Machine (SVM) - is employed at evenly spaced locations over the entire image. Others, such as Regions with Convolutional Neural Network features (R-CNN) [6] and its variants, first generate potential bounding boxes (or regions), then use a CNN for feature extraction and finally apply an SVM classifier on each of those regions. In contrast, YOLO regards object detection as a single regression problem; a CNN predicts bounding boxes and class probabilities from the whole image in one evaluation, hence the name (Fig. 1). This essentially means YOLO is fast, achieving more than 40 FPS for the canonical Pascal Visual Object Classes 2007 dataset on a desktop-based GeForce GTX TITAN X graphics card [7].

N. M. Nobre (nunomiguel.nobre@postgrad.manchester.ac.uk) and G. Riley (graham.riley@manchester.ac.uk) are with the School of Computer Science, The University of Manchester, Oxford Road, Manchester, UK, M13 9PL.

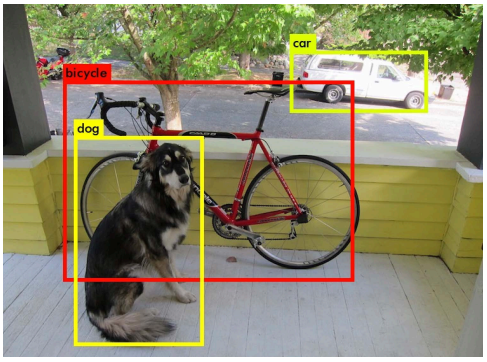

Fig. 1: Sample output of the YOLO object detection system. For big objects relative to the size of the image, the algorithm tends to be fast and accurate.

On robot vision systems, where energy efficiency is a primary requirement, including such a big and power-hungry graphics card is hardly a solution. As such, a less capable but battery-efficient mobile processor like the Qualcomm Snapdragon 820 [8] seems a more reasonable target. Initially, the aim was to harness the full capabilities of the Snapdragon's central processing unit (CPU), graphics processing unit (GPU) and digital signal processor (DSP) and investigate YOLO's temporal performance on such hardware. However, it rapidly became evident that even to exploit solely the ARM-based CPU to its full potential, knowledge of some key aspects of its architectural design would be required. Among these, the cache hierarchy structure and the available CPU vectorization capabilities are important when dealing with programs that include large matrix-matrix multiplications such as YOLO. Although the latter are fully disclosed by ARM, details on the former are not publicly available. In addition, although ARM-based CPU vectorization features are well-known, distinct compilers might exploit them differently and even distinct implementations of the same compiler might activate them at different optimization levels, making the results arduous to interpret and comparisons with temporal performance measurements

on other systems a difficult task. Thus, tackling these issues became the purpose of the experimental study reported here in the hope that it will provide a strong foundation for further work on the topic. The paper makes the following contributions:

- Describes experimental techniques and uses them to provide evidence on the number of cache levels and their sizes on the Snapdragon's CPU (Section 3).
- Provides an in-depth comparison of YOLO's temporal performance on different classes of processors, built with diverse combinations of compilers and compilation flags (Section 4).

2 Experimental set-up

2.1 Hardware environment

We begin this section by summarizing the hardware characteristics of the machines used in the experiments explained and analysed in Sections 3 and 4. These are described in Table 1 and cover the range from fully fledged desktop systems to truly portable devices used in smartphones, covering the current computing landscape. While the phone development board was the sole target of the cache hierarchy analysis in Section 3, the desktop, the laptop and the phone were the targets of Section 4's performance comparisons.

The Snapdragon 820 is an heterogeneous processor with an heterogeneous CPU which, like Intel consumer processors, includes a GPU and, in this particular case, a DSP allowing for CPU offloading to both these devices. Additionally, the CPU itself contains two pairs (clusters) of cores which, while sharing the same Kryo microarchitecture, are clocked at different frequencies. This is an important and common feature of CPUs designed for low-power devices which typically run on the lower-clocked cluster to save battery, reserving the usage of the more powerful cluster for demanding applications[1].

2.2 Software environment

Table 2 outlines the details of the compilers targeting each platform. gfortran was used to compile the code for the cache hierarchy inspection reported in Section 3 while both gcc and clang were employed to build the YOLO binary to be used in Section 4. For consistency among all systems, all binaries were built with 64-bit support.

Furthermore, when targeting the phone's Kryo CPU, Qualcomm's Symphony SDK CPU-only library [19] was included even though none of its features were used. This is to allow direct comparison with future performance measurements which might use this or other Symphony libraries with GPU and DSP support. These libraries allow for task scheduling, heterogeneous offload, and power and thermal management and are, for these reasons, crucial for the end goal of the project this report is setting a foundation for.

Unlike Android NDK's clang, which was found to seamlessly target every platform irrespective of its operating system or instruction set (either x86-64 or ARMv8-A) *as is*, Android NDK's gcc requires, when targeting GNU/Linux x86-64 systems, manual linking to x86-64 non-Android compatible libraries which, as both the desktop and laptop systems include older versions of gcc, were not available. Evidently, this could be solved by updating those systems' gcc but that would still leave two different gcc implementations[2] and, therefore, an inconsistency problem. For this reason, and because Google itself is phasing out the support for gcc, clang should be the compiler of choice in future work. Nevertheless, results with (different versions of) gcc are provided for completeness and comparison purposes.

2.3 Snapdragon's task scheduling

As the Snapdragon's CPU enables the use of all physical cores at the same time[3] and at different clock speeds, it is imperative to control both for reproducibility purposes. With this in mind, Android's built-in `taskset` command was used to bind a program's process to a specific cluster (also known as setting a task's affinity) and a set of tools was written to allow for dynamic frequency scaling [20]. The latter can most simply be done by setting the appropriate governor for each cluster. There are various governors that control how both cluster cores raise and lower their frequency in response to the demands the user is placing on the device. We will make use of the *performance* and the *powersave* governors which statically set the cores to the highest and lowest allowed frequencies respectively [21]. The cluster responsible for running the program will be set to *performance* while the idle cluster will be set to *powersave*, allowing to easily distinguish the cases where the task affinity, as defined by `taskset`, is not honoured by the kernel (when the affinity is not respected, the execution time will be significantly higher). In fact, the GNU/Linux version of `taskset` explicitly guarantees the affinity to be respected, but this is not guaranteed on the Android's implementation usage message. Lastly, it was also possible to verify that interacting with the Android UI would cause the frequencies to (even if briefly) fluctuate. For this reason, the experiments were conducted with the development board solely connected to the power outlet.

2.4 Statistical considerations

To avoid making dubious statistical assumptions and approximations regarding the data's underlying distribution, plotted data points (whether in the form of line or bar graphs) represent minimum execution times whereas the respective error bars denote maximum execution times among 10 runs performed under the same conditions. Also note that all graphs depicted here were produced with MATLAB [22].

3 Cache hierarchy

3.1 Workload

In order to probe the cache system in the Snapdragon's CPU, we adapted a small piece of Fortran code written in 1997 by Mark Bull:

1. ARM's own big.LITTLE [16] technology closely resembles this arrangement whilst using ARM-designed Cortex-A cores.

2. Either Android NDK's gcc and the laptop and desktop systems' gcc or, in the case of using Android NDK's gcc with the newly updated libraries, libraries compiled with different compiler implementations.

3. This is also known as heterogeneous multi-processing (HMP).

TABLE 1: Targeted hardware configurations used in the experiments [8]–[13]. For the two Intel processors listed, in addition to their base frequencies, the maximum single-core Turbo Boost frequencies are also noted. Note that, contrary to DDR DRAM which is usually connected over a 64-bit wide bus, LPDDR modules commonly use 16- or 32-bit wide channels [14]. In addition, in this paper, binary notation is adopted to avoid the 2^x vs. 10^y ambiguity [15]. As such, a gigabyte (GB) is 10^9 B whilst a gibibyte (GiB) is 2^{30} B.

Machine type/model	Processor package	Main memory package/interface
Desktop HP EliteDesk 800 G1 SFF	Intel Core i7-4770 CPU: Intel Haswell :: 4× @ 3.40 GHz (up to 3.90 GHz) :: x86-64 GPU: Intel HD Graphics 4600 @ 350 MHz	16 GiB DDR3 @ 800 MHz Dual 64-bit channels (25.6 GB/s)
Laptop HP Pavilion dv6-6196ep	Intel Core i7-2630QM CPU: Intel Sandy Bridge :: 4× @ 2.00 GHz (up to 2.90 GHz) :: x86-64 GPU: Intel HD Graphics 3000 @ 650 MHz	8 GiB DDR3 @ 667 MHz Dual 64-bit channels (21.3 GB/s)
Phone (development kit) Open-Q 820 (APQ8096)	Qualcomm Snapdragon 820 CPU: Qualcomm Kryo :: 2× @ 1.59 GHz + 2× @ 2.15 GHz :: ARMv8-A GPU: Qualcomm Adreno 530 @ 624 MHz DSP: Qualcomm Hexagon 680 :: 4× @ 500 MHz (scalar) / 2× @ 500 MHz (vector)	3 GiB LPDDR4 @ 1866 MHz Dual 32-bit channels (29.9 GB/s)

TABLE 2: Brief description of the relevant software related to each of the hardware configurations used. Whereas in some cases the compilers are part of the respective systems, in others they are part of the Android NDK toolchain [17]. In particular, Android NDK's gfortran is, in fact, an extension to the original toolchain [18]. Note that the compilers targeting the phone development kit are in reality cross-compilers installed on the laptop computer.

Machine type/model	Operating system/Linux kernel	Compiler front ends		
		gfortran	gcc	clang
Desktop HP EliteDesk 800 G1 SFF	Scientific Linux 7.1 3.10.0-327.22.2.el7.x86_64	-/-	4.8.3 20140911	3.8.275480 (Android NDK r14)
Laptop HP Pavilion dv6-6196ep	Ubuntu 14.04.5 LTS 4.4.0-67-generic	-/-	4.8.4 20141219	3.8.275480 (Android NDK r14)
Phone (development kit) Open-Q 820 (APQ8096)	Android 6.0 3.18.20-g641747c-00004-g54ca4b4	4.9.x 20150123 (Android NDK r13b)	4.9.x 20150123 (Android NDK r14)	3.8.275480 (Android NDK r14)

```
1  iops=ceiling(n*1.0/stride)
2  eops=nint(nops*1.0/iops)
3
4  do i=1,eops
5    do j=1,iops
6      k=(j-1)*stride+1
7      res=res+x(k)
8    end do
9  end do
```

In this code, **ceiling** returns the least integer greater than or equal to its argument, **nint** rounds its argument to the nearest whole number and the **do**-loop corresponds to what is known as a **for**-loop in other languages. In essence, the code executes repeated sums over equally spaced elements, as determined by the value of `stride`, of a vector `x` of length `n`. The procedure is then repeated and timed for increasingly bigger vectors (this is not shown in the excerpt) to expose the access latencies to different levels of the memory hierarchy. As the total number of operations performed by the CPU, `iops`, changes with the size of the vector, the program does `eops` $\approx$ `nops`/`iops` passes through the vector to keep the number of operations (approximately) constant and equal to `nops` across the different sized vectors. Additionally, as `nops` is chosen to be fairly large (50 000 000), spurious effects should average out, contributing to an overall fair comparison.

As corroborated by the results to be presented in Subsection 3.2, this code has a fundamental flaw when applied to today's CPUs. In fact, as the vector elements are fetched successively using some constant stride, the access pattern can be easily discerned by optimizing compilers and the hardware prefetcher[4], hiding cache access latencies vital to this experiment [24]. The following code excerpt tries to circumvent this by accessing the vector elements in a mirror image style scheme:

```
1  iops=ceiling(n*1.0/stride)
2  eops=nint(nops*1.0/iops)
3
4  do i=1,eops
5    do j=1,iops
6      k=merge((iops-j/2)*stride+1,
7    $         (j-1)/2*stride+1,
8    $         mod(j,2) .eq. 0)
9      res=res+x(k)
10   end do
11 end do
```

4. As opposed to software prefetching which uses instructions explicitly included by the programmer or the compiler into the code [23].

Where **merge** selects its first or second argument based on whether a logical mask, given as its third argument, is true or false, respectively, and **mod** computes the remainder of the division of its first by its second argument. As the relational operator **.eq.** means equality, the logical expression is testing whether the variable of the inner loop, `j`, is an even number. With this approach, if the K accessed elements `x(k)` are imagined as being arranged in ascending order of `k` and labelled $1, 2, \ldots, K-1, K$ the access ordering turns out to be $1, K, 2, K-1, 3, \ldots$

The complete source code as well as a compilation guide for both versions of the code are available online [20].

3.2 Results

The results obtained for both versions of the code with `stride` $= 16$ on both the low-power and high-performance clusters of the Snapdragon's CPU are depicted in Fig. 2. It is clear that the last level cache is $512\,\mathrm{KiB}$ on the low-power cluster and $1\,\mathrm{MiB}$ on the high-performance cluster since the large and sudden increase in execution time that follows is characteristic of the need to repeatedly access main memory. Besides, an hypothetical larger cache would have to be greater than $40\,\mathrm{MB}$ (the upper limit of the horizontal axis), which is highly unlikely for a processor of this class.

When the access is performed with the mirror image style scheme another noticeable and permanent increase in the execution times is visible at around $32\,\mathrm{KiB}$. Caches are normally sized in powers of $2\,\mathrm{B}$ making this a reasonable estimate for the size of the previous level cache. Thus, this evidence points towards a two-level cache system with a L1 data cache of $32\,\mathrm{KiB}$ irrespective of the type of core and a L2 cache of $512\,\mathrm{KiB}$ and $1\,\mathrm{MiB}$ for the low- and high-performance clusters respectively, corroborating the assertions made elsewhere [25].

4 YOLO TEMPORAL PERFORMANCE

4.1 Workload

YOLO was compiled both using gcc and clang with and without vectorization passes enabled. On both compilers these consist of a block vectorizer, which merges multiple similar scalar instructions into vector instructions, and a loop vectorizer, which widens instructions in loops to operate on multiple consecutive iterations [26], [27]. Compilers typically do this by exploring the SIMD (Single Instruction, Multiple Data) instruction set extensions available on x86-64 (e.g. AVX, MMX, SSE) and on ARM (e.g. NEON) processors.

The approach with gcc was to start with the standard `-O2` optimization flag and then build on top of that with the vectorization flag, `-ftree-vectorize`. According to gcc's documentation [28] this should not be enabled at `-O2` but only at `-O3` and `-Ofast`. To keep the measurements as simple as possible we only examine the influence of the vectorization pass relative to all the other additional optimizations that `-Ofast` enables over `-O2`. Unfortunately, experiments established that all gcc implementations used deviate from GNU's documented recommendations in some way. In fact, it turns out that block vectorization is enabled by default, even at `-O2`, on both the desktop and the laptop GNU/Linux systems, whereas in the case of Android NDK's gcc, both vectorization optimizations are enabled at `-O2`. To circumvent this, we negate the flag: `-fno-tree-vectorize`[5]. Evidently, only a thorough manual exploration would reveal if the documented recommended behaviour is being violated in other ways. This adds to the list of arguments against gcc usage in this context.

With clang we start at `-O1` which, as the results show, seems to perform fairly similarly to `-O2` in gcc (at least on x86-64). Then we introduce `-Ofast` and compare this with `-O2` which adds vectorization to `-O1` but does not include the unsafe maths optimizations of `-Ofast`.

This arrangement should allow for two things: in the gcc case, for inferring the influence of vectorization and, in the clang case, for a clean comparison between different platforms and different `-O` level optimization flags.

The complete source code as well as a compilation guide are available online [20].

4.2 Results

Fig. 3 summarizes the results obtained on the three machines specified in Table 1. These are non-parallel (single-core) execution results for the image produced by YOLO presented earlier on Fig. 1. Separate measurements for the low-power and high-performance cores are provided. From Fig. 3a, it is clear that vectorization is the main optimization responsible for reduced execution times, with greater relative impact on the Snapdragon's CPU. However, some care has to be taken, as, in Fig. 3a, we are not comparing results obtained with the same gcc version. Indeed, Fig. 3b does not seem to display the same relative improvement in execution time. In fact, in Fig. 3b, where the same version of clang is used throughout, the proportion between bars corresponding to different optimization flags on the same platform looks reasonably similar across different platforms. For example, `-O2` execution times seem about the same fraction of `-O1`'s on the laptop and on the Snapdragon's high-performance cluster (@ $2.15\,\mathrm{GHz}$). Put simply, while the collected data shows that vectorization is the main responsible for faster execution times, it does not suffice to determine if vectorization is more relevant on the Snapdragon's CPU than on the other tested platforms.

Apart from the above considerations, Fig. 3b depicts exactly what we would naively expect: higher clocked CPUs and higher numbered optimization flags perform better. Surprisingly, however, in contrast with the behaviour on x86-64 machines, clang did not manage to provide a significantly faster binary than gcc on the Snapdragon's CPU. Presumably, as the Android NDK migration from gcc to clang is still under development [17], some libraries might not have been fully optimized yet and this discrepancy may disappear once this migration is complete.

5 RELATED WORK

Achieving high-performance on machine learning and computer vision applications on mobile and low-power devices

5. For both the desktop and the laptop, which are using gcc 4.8.x, we actually need to add `-fno-tree-slp-vectorize` to explicitly disable the block vectorization as, in this older version, `-ftree-vectorize` only controls the activation status of the loop vectorizer.

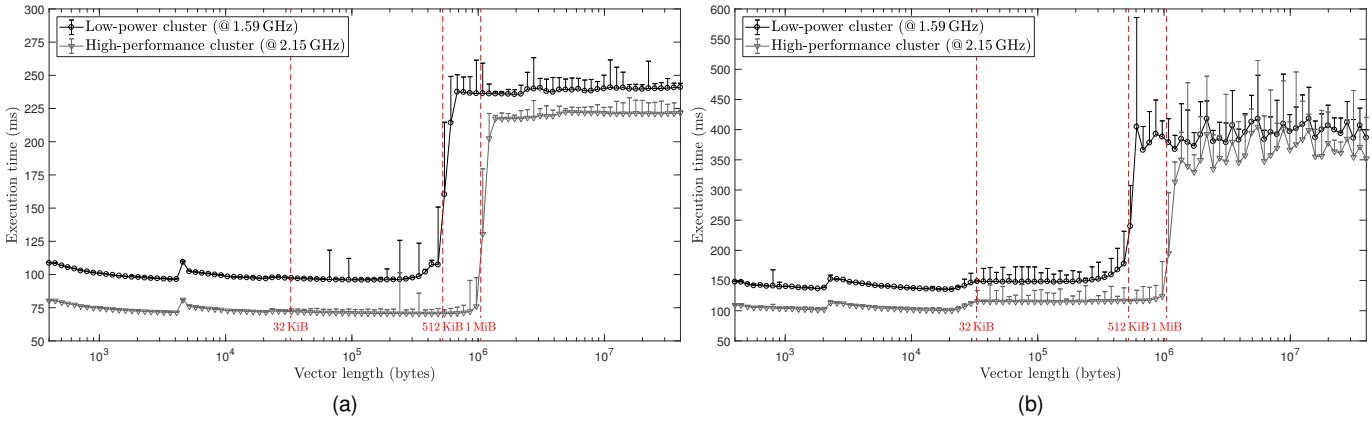

Fig. 2: Execution times for the original (a) and modified (b) versions of the memory hierarchy detection Fortran code. The latter hints at a 32 KiB L1 data cache for both cluster cores while both graphs present evidence of a 512 KiB L2 cache for the low-power cluster and a 1 MiB L2 cache for the high-performance one.

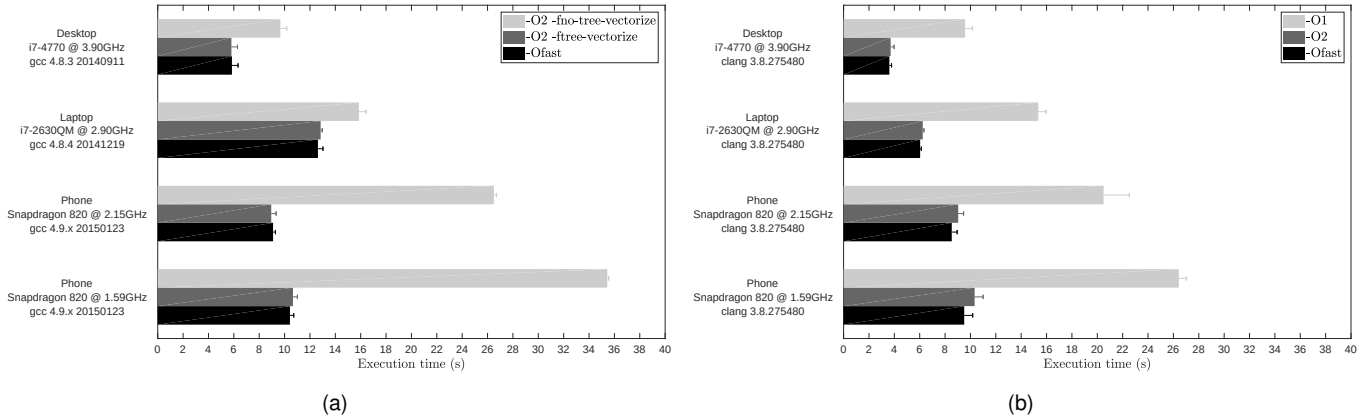

Fig. 3: Single-core execution times for the YOLO detection system after being compiled with gcc (a) and clang (b). Although the former provides an insight on what to expect from gcc, only the latter, by using the exact same version and implementation of the clang compiler, provides a fair comparison between the different platforms.

is, nowadays, a popular topic in both academia and the hardware industry. The following gives an example of each.

Boda-RTC is an OpenCL approach to Convolutional Neural Network (CNN) computations [29]. As OpenCL is a vendor-neutral platform it allows for targeting a multitude of hardware configurations including the Snapdragon 820 accelerators (GPU and DSP). This contrasts with YOLO's CUDA-only approach which might bring better performance on NVIDIA's own GPUs, but also limits its applicability to, and only to, those GPUs. Early performance results on the Snapdragon 820 using the Boda framework seem promising. However, the authors do note that progress was hindered due to lack of documentation and tools.

On the hardware end of the spectrum, ARM's DynamIQ technology [30], [31], the evolutionary step forward for ARM's big.LITTLE, brings increased multi-core flexibility by allowing each core on the same cluster to have different performance and power characteristics. In addition, the instruction set is being extended with dedicated machine learning and artificial intelligence instructions which, if compatible with existing software techniques, should produce visible performance gains over current solutions.

6 Conclusions and Further Work

The experiments summarized here revealed that the low-power cluster cores of the Snapdragon's CPU have a 32 KiB L1 data cache and a 512 KiB L2 cache whilst the high-performance cluster cores, while sharing the same design for the L1 data cache, have a bigger 1 MiB L2 cache. Additionally, the mobile-friendly clang seems clearly the way forward at least for workloads targeting the CPU: it produces faster code, it effortlessly supports a range of platforms and it will also continue to be actively supported on Android.

We see essentially two paths forward. Firstly, and following directly from this work, it would be of interest to test YOLO with Qualcomm's own LLVM compiler [32] and their own maths libraries [33]. Nevertheless, as the former is essentially a slightly optimized version of the clang compiler back end used here and the latter are CPU-only numerical

libraries with the standard BLAS and LAPACK interfaces, we do not expect dramatic improvements in performance. Secondly, using the Symphony SDK to offload the CPU should now be easier given the developed techniques and tools for task affinity and frequency control [20], and the information collected here on cache sizes and access latencies. At least, it should allow for more convincing rationales on what and when to offload.

REFERENCES

[1] Z. Jia, A. Saxena, and T. Chen, "Robotic object detection: Learning to improve the classifiers using sparse graphs for path planning," in *IJCAI*, 2011, Conference Proceedings, pp. 2072–2078.

[2] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, and D. Ramanan, "Object detection with discriminatively trained part-based models," *Ieee Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 9, pp. 1627–1645, 2010.

[3] K. Lu, X. An, J. Li, and H. He, "Efficient deep network for vision-based object detection in robotic applications," *Neurocomputing*, 2017.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *ArXiv e-prints*, vol. 1506, 2015. [Online]. Available: http://adsabs.harvard.edu/abs/2015arXiv150602640R

[5] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," *ArXiv e-prints*, vol. 1612, 2016. [Online]. Available: http://adsabs.harvard.edu/abs/2016arXiv161208242R

[6] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, Conference Proceedings, pp. 580–587.

[7] NVIDIA Corporation, *GeForce GTX TITAN X*. 2701 San Tomas Expressway, Santa Clara, CA 95050, USA: NVIDIA Corporation, 2015. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x

[8] Qualcomm Technologies, Inc., *Snapdragon 820*. 5775 Morehouse Drive, San Diego, CA 92121-1714, USA: Qualcomm Technologies, Inc., 2015. [Online]. Available: https://www.qualcomm.com/products/snapdragon/processors/820

[9] Intel ARK, *Intel® Core™ i7-4770 Processor*. 2200 Mission College Blvd, Santa Clara, CA 95054-1549, USA: Intel Corporation, 2013. [Online]. Available: http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz

[10] Intel ARK, *Intel® Core™ i7-2630QM Processor*. 2200 Mission College Blvd, Santa Clara, CA 95054-1549, USA: Intel Corporation, 2011. [Online]. Available: http://ark.intel.com/products/52219/Intel-Core-i7-2630QM-Processor-6M-Cache-up-to-2_90-GHz

[11] M. Humrick, *Snapdragon 820 Performance Preview*. Tom's Hardware, 2015. [Online]. Available: http://www.tomshardware.com/reviews/snapdragon-820-performance-preview,4389-2.html

[12] L. Codrescu, "Architecture of the Hexagon™ 680 DSP for mobile imaging and computer vision," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, Conference Proceedings, pp. 1–26.

[13] Intrinsyc Technologies Corporation, *Open-Q™ 820 Development Kit*. 885 Dunsmuir Street, 3rd Floor, Vancouver, BC, Canada: Intrinsyc Technologies Corporation, 2015. [Online]. Available: https://shop.intrinsyc.com/products/open-q-820-development-kit

[14] Texas Instruments Wiki, *LPDDR*. 13588 North Central Expressway, Dallas, TX 755243, USA: Texas Instruments Inc., 2011. [Online]. Available: http://processors.wiki.ti.com/index.php/LPDDR

[15] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, 5th ed. Amsterdam; Boston: Elsevier, 2014.

[16] ARM Holdings plc, *big.LITTLE Technology: The Future of Mobile*. 110 Fulbourn Road, Cambridge, GB-CB1 9NJ, UK: ARM Holdings plc, 2013. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[17] Android Developers, *Android NDK*. 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA: Google Inc., 2017. [Online]. Available: https://developer.android.com/ndk/index.html

[18] P. Mustière, *GitHub repository*. 88 Colin P Kelly Jr St, San Francisco, CA 94107, USA: GitHub, Inc., 2017. [Online]. Available: https://github.com/buffer51

[19] Qualcomm™ Developer Network, *Symphony System Manager SDK v1.1.2*. 5775 Morehouse Drive, San Diego, CA 92121-1714, USA: Qualcomm Technologies, Inc., 2017. [Online]. Available: https://developer.qualcomm.com/software/symphony-system-manager-sdk

[20] N. M. Nobre, *GitHub repository*. 88 Colin P Kelly Jr St, San Francisco, CA 94107, USA: GitHub, Inc., 2017. [Online]. Available: https://github.com/nmnobre

[21] D. Brodowski and N. Golde, *CPU frequency and voltage scaling code in the Linux™ kernel*. Google Git repositories on Android, 2016. [Online]. Available: https://android.googlesource.com/kernel/common/+/android-4.9/Documentation/cpu-freq/governors.txt

[22] The MathWorks, Inc., *MATLAB® R2017a*. 1 Apple Hill Drive, Natick, MA 01760-2098, USA: The MathWorks, Inc., 2017. [Online]. Available: https://uk.mathworks.com/products/matlab.html

[23] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 40–52, 1991.

[24] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, Conference Proceedings, pp. 63–74.

[25] R. Smith and A. Frumusanu, *The Qualcomm Snapdragon 820 Performance Preview: Meet Kryo*. AnandTech, 2015. [Online]. Available: http://www.anandtech.com/show/9837/snapdragon-820-preview/2

[26] Free Software Foundation, Inc., *GCC, the GNU Compiler Collection: Auto-vectorization in GCC*. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1335, USA: Free Software Foundation, Inc., 2016. [Online]. Available: https://gcc.gnu.org/projects/tree-ssa/vectorization.html

[27] LLVM Team, *The LLVM Compiler Infrastructure: Auto-Vectorization in LLVM*. Thomas M. Siebel Center for Computer Science, 201 North Goodwin Avenue, Urbana, IL 61801-2302, USA: University of Illinois at Urbana-Champaign, 2017. [Online]. Available: http://llvm.org/docs/Vectorizers.html

[28] Free Software Foundation, Inc., *GCC, the GNU Compiler Collection: Options That Control Optimization*. 51 Franklin Street, Fifth Floor, Boston, MA 02110-1335, USA: Free Software Foundation, Inc., 2017. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[29] M. W. Moskewicz, F. N. Iandola, and K. Keutzer, "Boda-RTC: Productive generation of portable, efficient code for convolutional neural networks on mobile computing platforms," in *2016 IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2016, Conference Proceedings, pp. 1–10.

[30] G. Wathan, *ARM DynamIQ: Technology for the next era of compute*. ARM Community: Processors blog, 2017. [Online]. Available: https://community.arm.com/processors/b/blog/posts/arm-dynamiq-technology-for-the-next-era-of-compute

[31] N. Nayampally, *ARM DynamIQ: Expanding the possibilities for artificial intelligence*. ARM Community: Processors blog, 2017. [Online]. Available: https://community.arm.com/processors/b/blog/posts/arm-dynamiq-expanding-the-possibilities-for-artificial-intelligence

[32] Qualcomm™ Developer Network, *Snapdragon LLVM Compiler for Android v3.8.8*. 5775 Morehouse Drive, San Diego, CA 92121-1714, USA: Qualcomm Technologies, Inc., 2017. [Online]. Available: https://developer.qualcomm.com/software/snapdragon-llvm-compiler-android

[33] Qualcomm™ Developer Network, *Snapdragon Math Libraries v0.15.2*. 5775 Morehouse Drive, San Diego, CA 92121-1714, USA: Qualcomm Technologies, Inc., 2017. [Online]. Available: https://developer.qualcomm.com/software/snapdragon-math-libraries