Experiences in Speeding Up Computer Vision Applications on Mobile Computing Platforms

Experiences in Speeding Up Computer Vision Applications on Mobile Computing Platforms

Luna Backes
Barcelona Supercomputing Center,
Barcelona, Spain
Email: luna.backes@bsc.es

Alejandro Rico
Barcelona Supercomputing Center,
Barcelona, Spain
Email: alejandro.rico@bsc.es

Björn Franke
The University of Edinburgh,
Edinburgh, United Kingdom
Email: bfranke@ed.ac.uk

***Abstract*—Computer vision (CV) is widely expected to be the next *big thing* in mobile computing. The availability of a camera and a large number of sensors in mobile devices will enable CV applications that understand the environment and enhance people's lives through augmented reality. One of the problems yet to solve is how to transfer demanding state-of-the-art CV algorithms —designed to run on powerful desktop computers with several GPUs— onto energy-efficient, but slow, processors and GPUs found in mobile devices. To accommodate to the lack of performance, current CV applications for mobile devices are simpler versions of more complex algorithms, which generally run slowly and unreliably and provide a poor user experience. In this paper, we investigate ways to speed up demanding CV applications to run faster on mobile devices. We selected KinectFusion (KF) as a representative CV application. The KF application constructs a 3D model from the images captured by a Kinect. After porting it to an ARM platform, we applied several optimisation and parallelisation techniques using OpenCL to exploit all the available computing resources. We evaluated the impact on performance and power and demonstrate a 4× speed-up with just a 1.38× power increase. We also evaluated the performance portability of our optimisations by running on a different platform, and assessed similar improvements despite the different multi-core configuration and memory system. By measuring processor temperature, we found overheating to be the main limiting factor for running such high-performance codes on a mobile device not designed for full continuous utilisation.**

I. Introduction

Mobile devices represent a new era in computing with a market rapidly growing [1]. At the same time, mobile processor performance has increased by a factor of 25× in 3 years [2]. The challenge now is power: the power envelope is going to remain fixed, at about 2W for smartphones and 5-7W for tablets. This is due to the need for cooling, which is limited in compact mobile devices. This thermal limit equally determines the power limit that stays constant. This also affects performance: a mobile device working at maximum speed for a long time heats up quickly, which forces the system to reduce its operating frequency.

Smartphones and other mobile devices integrate more sensors in each generation. Some examples are GPS, accelerometer, gyroscope, proximity, light and camera. These hardware components make mobile devices an appealing option for computer vision (CV). The possibilities of CV in mobile devices are endless [3], for example: games, improve disabled people's life or new learning techniques. However, demanding state-of-the-art CV algorithms require powerful desktop GPUs able to provide high performance.

Existing CV applications for mobile devices are simpler versions of more complex algorithms in an attempt to provide high frame rates at the expense of functionality and accuracy [4]. We tried a set of existing CV apps on a powerful smartphone, the Samsung Galaxy Note 3, and found that user experience is poor due to erroneous outputs, overheating that led some apps to abort, and slow frame rates. This experience shows that existing attempts of CV on mobile devices are not completely satisfactory, and motivates the work of several projects, such as PAMELA [5] and Google's project Tango [6], towards providing more optimized CV software and specialised hardware.

In this paper, we investigate ways to speed up demanding CV applications on mobile devices. We selected the KinectFusion (KF) application [7] as a representative CV application that requires high performance. KF processes images of objects taken from multiple angles by a Kinect and constructs a 3D model of the observed objects. It usually runs on desktop machines with powerful GPUs because it requires high performance to process the images and construct the model at interactive rates (approximately 30 FPS). We ported KF to run on a mobile system-on-chip (SoC), the Samsung Exynos 5250. We profiled the application and ported it to OpenCL to make use of the embedded GPU available in the chip. We performed a set of optimisations by porting several parts to the application to run on the GPU, parallelised other parts to exploit multiple CPUs, and minimised synchronizations and data transfers.

With this work we attempt to quantify how far current mobile SoCs are capable of running demanding CV applications, such as KF, at interactive rates; anticipate what problems developers may face when porting such applications to mobile devices; and provide a sense of the potential improvements of our optimisations on such applications.

In this context, the contributions of this paper are:

- A profiling of the KF application running on the Samsung Exynos 5250 and description of the optimisations applied based on that profiling. We ported several parts of the application to OpenCL to exploit more parallelism using the embedded GPU, removed data transfers between CPU and GPU using shared memory and removed unnecessary CPU-GPU synchronisations by allowing consecutive computations on the GPU to reuse the data in GPU memory.

This work was funded by the EPSRC grant PAMELA EP/K008730/1 and the Mont-Blanc project (EU FP7 grant agreements 288777 and 610402).

- A temperature evaluation of KF running on Samsung Exynos 5250. We found that overheating is a major issue in terms of stability and performance because the processor applies thermal throttling to avoid temperature exceeding 85 degrees to avoid damaging the CPU. We also evaluated the impact of several cooling solutions. Adding a heat sink and a fan reduced execution time by 20% of the unoptimised KF version. We also found that our optimisations made KF less sensitive to temperature. Moving computation to the GPU significantly reduces overheating and, thus, thermal throttling happens much less frequently than in the unoptimised version. Therefore, the GPU version not only improves performance by exploiting more parallelism, but also due to a reduced thermal throttling activity.
- A performance and power evaluation of KF running on Samsung Exynos 5250 on each optimisation step. We measure the execution time and power consumption of each one of our optimisations. We also break down the execution time on the different stages of the KF algorithm to analyse which of them our optimisations affect. We achieved an overall 4× speed-up with only a 1.38× power increase. This results in a 3× reduction on energy-to-solution which is crucial for battery-restricted environments such as mobile devices. We also repeated the analysis on a different mobile SoC, the Samsung Exynos 5422, and experienced similar improvements that show a good performance portability of our optimisations.

II. Related Work

Computer vision (CV) for mobile devices is becoming a trending topic since the performance of mobile processors is becoming high enough to run simple CV applications. Existing examples of such applications are video stabilisation with rolling shutter correction, face detection and recognition, low-light image/video enhancement, car plate detection and recognition [8]; augmented reality (e.g., placing of furniture on the recorded space [9]); and real-time translation of text [10].

An example is Word Lens [10], which translates text in real time from the video captured by the mobile camera. The application has to recognise the text from the image and translate it, ideally, at an interactive rate. Another example is video stabilisation [8], that processes the scene in real time and recognises the objects to understand how the camera moves relative to the scene and eliminate the trembling. These kinds of video processing are specially difficult when it is dark, there are plenty of similar objects together or the resolution of the camera is low. For this reason, they require complex processing algorithms that are computationally expensive.

An approach to increase the computational capacity of mobile devices is to build specialised hardware. Movidius [11], a vision processor company, develops processors with an architecture specifically tailored for CV. Their latest generation processor, the Myriad 2 [12], comprises 12 specialized vector VLIW processors for high throughput, a wide range of interfaces for high connectivity, a set of imaging/vision accelerators for specific processing tasks, and a high-bandwidth memory fabric that interconnects all the processing resources and peripherals. The architecture is implemented for high throughput rather than latency so it runs at hundreds of megahertz targeting power efficiency.

TABLE I. Percentage of time spent in the different stages of the KF original version.

Stages	Time (%)
Acquisition	2.70%
Preprocessing	23.08%
Tracking	5.50%
Integration	49.87%
Rendering	18.64%
Drawing	0.14%

This chip is integrated in the specialised mobile device developed in Google's project *Tango* [6]. Their objective is to provide a 5" Android phone with highly customized hardware, including Movidius' Myriad chip, and software designed to track the full 3-dimensional motion of the device as you hold it while simultaneously creating a map of the environment. The project also includes the development of a 7" tablet with the NVIDIA Tegra K1 processor, 4GB of RAM, 128GB of storage, motion tracking camera, integrated depth sensing and wireless interfaces.

Another approach to improve the capabilities of CV applications on mobile devices is to optimise the software for existing mobile processors. Previous works [13], [14] ported CV applications to mobile SoCs and provide significant speed-ups mainly by using the GPU. Cheng and Wang [13] ported a face recognition algorithm to OpenGL. The application processes a single image to identify faces using a pattern recognition algorithm. They tested their OpenGL version in Android-powered devices and compared their efficiency in GPUs from Qualcomm, NVIDIA and Imagination Technologies. Wang et al. [14] ported an image inpainting-based object removal algorithm to OpenCL. The application processes a single image to fill holes left by the removal of objects. It uses patterns nearby the hole to build an approximation of the pixels the object was covering. They tested their application in an Android-powered device with a Qualcomm GPU [14].

Our work differs from these previous works in that we port and optimise a highly-demanding application, KinectFusion, that processes a video, instead of a single image, and we provide a comprehensive evaluation of performance, power and also temperature for multiple optimisation steps. As in the work of Cheng and Wang [13], we also compare multiple platforms with different GPUs, but we developed the GPU version of our application in OpenCL rather than in OpenGL. This provides a more maintainable software because it does not require mapping the algorithm to textures and geometric primitives as in OpenGL. Moreover, our optimisations do not only exploit the GPU, as in previous works, but also the multiple CPUs in the SoC.

III. KinectFusion Optimisations

KinectFusion (KF) [7] provides 3D object scanning and model creation using a Kinect for Windows sensor. The user can scan a scene with the Kinect camera and simultaneously see and interact with a detailed 3D model of the scene.

TABLE II. LIST OF OPTIMISATIONS IN IMPLEMENTATION ORDER. EACH OPTIMISATION STEP INCLUDES THE PREVIOUS ONES.

Label	Description
OCL-Initial	Porting of *integrate* kernel (Integration stage)
OCL-SharedMem	Use shared memory (no data copies)
OCL-Raycast	Porting of *raycast* kernel (Integration stage)
OCL-Track	Porting of *track* kernel (Tracking stage)
OCL-OpenMP	Porting of CPU-side code to OpenMP (Preprocessing and Rendering stage)
OCL-Barriers	Removed barriers and unnecessary memory mappings (Integration stage)
OCL-Bilateral	Porting of *bilateral_filter* kernel (Preprocessing stage)

TABLE III. CHARACTERISTICS OF THE ARNDALE BOARD AND THE ODROID-XU3 DEVELOPMENT KITS.

	Arndale Board	Odroid-XU3
Chip	Samsung Exynos 5250	Samsung Exynos 5422
CPU	2×ARM Cortex-A15@1.7GHz	4×ARM Cortex-A15@2GHz 4×ARM Cortex-A7@1.3GHz
L1 Cache	32KB(I)+32KB(D)	32KB(I)+32KB(D)
L2 Cache	1MB unified	2MB + 512KB
GPU	ARM Mali-T604	ARM Mali-T628
Memory	2GB 32-bit 800MHz LDDR3	2GB 32-bit 933MHz LDDR3
Storage	16GB uSD	16GB eMMC v.03
OS	Ubuntu 12.04	Ubuntu 14.04

We ported a C++ version of KinectFusion, that we refer to as *original* version, to run on a mobile processor, the Samsung Exynos 5250 SoC (see Section IV for more details). We profiled the code and identified the time spent on each of the six stages to process each frame [7]. Table I shows the results. Integration accounts for almost 50% of the total execution time, while preprocessing and rendering account for 41%. Therefore, these stages are the main target of our optimisations.

Based on this profile, we ported a set of code sections, referred to as *kernels*, to run on the GPU using OpenCL, and parallelised some other parts of the application using OpenMP to run on the multiple CPUs on the chip. We also applied optimisations to avoid memory copies of kernel data between the CPU and the GPU, and reduced the amount of synchronizations in between kernel executions.

Unlike heterogeneous systems with discrete GPUs, the Samsung Exynos 5250 has a unified memory shared between the CPU and the GPU. The implementation of our first KF kernel in OpenCL used memory copies between the CPU and the GPU for simplicity. This implies to create memory objects before calling the kernel by allocating a buffer on the GPU of the same size of the data to be copied from the CPU (therefore, it uses double the memory than necessary) and copying the data to the GPU before the kernel call and back to the CPU after kernel completion.

In the case of shared memory, we can allocate the data in the CPU using *clCreateBuffer*, instead of *malloc*, with a parameter (CL_MEM_ALLOC_HOST_PTR) that allows the memory to be created and accessed by both CPUs and GPUs. In OpenCL, if you create a buffer, it cannot be accessed from the CPU unless it is *mapped* to the CPU using *clEnqueueMapBuffer*. Also, before calling the GPU kernel, the buffer must be mapped to the GPU using *clEnqueueUnmapMemObject*. Therefore, to use shared memory in our KF OpenCL version, we first allocate the data in the CPU using *clCreateBuffer* and immediately map it to the CPU to initialise it. Then, before calling the kernel for execution on the GPU, we map the data to the GPU so the kernel can access it, and whenever the kernel completes, it is mapped back to the CPU.

As we ported kernels to OpenCL, we identified several points were CPU synchronisation between kernels was not necessary. Some kernels execute back-to-back on the GPU, so the mapping of the data structures from/to the GPU in between them is not necessary. The data generated by a kernel to be used by a following kernel does not need to be mapped to the CPU after the first kernel finishes and back to the GPU before the second one starts, if the CPU is not going to process that data in between. Then, the data can remain mapped to the GPU from the start of the first kernel until the finalisation of the second one.

Also, to ensure correctness, the CPU waits using a barrier primitive (*clFinish*) to block until a kernel (or set of kernels) finishes, before calling other kernels that require the output of the previous ones. We removed this barrier synchronisation by using a single *command queue* for all kernels. When a CPU enqueues a kernel execution in a command queue, these kernels will be executed in the GPU keeping the order in which they were enqueued. This way, by enqueuing the kernels in program order, we ensure that sequential order is maintained and there are no data races, and therefore, no need for barrier synchronisations.

Table II shows the list of kernel portings and optimisations in implementation order, and the label used in the results to refer to them. All optimisation steps include the previous optimisations. All kernels ported after OCL-SharedMem use shared memory (no data copies).

IV. EXPERIMENTAL SETUP

Our hardware testbed for evaluating performance, power and temperature is the Arndale board development kit that features the Samsung Exynos 5250 SoC. This SoC includes two ARM Cortex-A15 cores, an ARM Mali-T604 GPU and 2GB of main memory. We repeated our experiments in a second development kit, the Odroid-XU3 [15], to assess the performance portability of our code. This board features the Samsung Exynos 5422 SoC with a big.LITTLE architecture including four ARM Cortex-A15, four ARM Cortex-A7 cores, an ARM Mali-T628 and 2GB of main memory. Full specifications of both boards are shown in Table III.

We executed the KF application with four different video input files in raw format as benchmarks. The four benchmarks are: *chairs*, *desktop*, *person* and *weird*, named after the objects appearing in each video. These videos were recorded by moving a camera around the objects to capture as many different angles as possible. This way, the application can recognise the depth from the camera to the object and use it to create the 3D model. The execution of the benchmarks take between 15 and 45 minutes in the original C++ version.

Figure 1 shows a frame of the output video for an execution of the *chairs* benchmark. The KF application outputs a video with three images. The left image is the depth map showing a color depending on the distance to the camera; the middle one is the tracking showing the status of the recognition for each pixel; and the right one is a raycast of the constructed 3D volume.

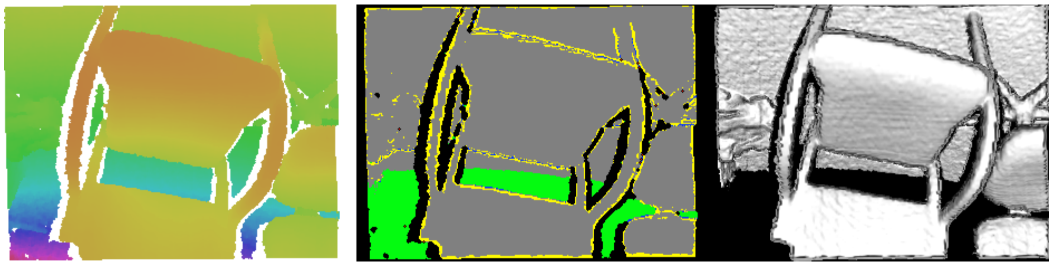


Fig. 1. Images of the output video obtained by executing KF with the *chairs* benchmark.

A. Performance Measurement

To measure the impact on execution time of each optimisation, we profiled the application at each optimisation step. All the executions were done in exclusive mode: no other applications running at the same time. The profiling was done with *clock_gettime()* between the six different stages of the KF algorithm for each frame: acquisition, preprocessing, tracking, integration, rendering and drawing.

The metrics are the execution time in seconds of each computation stage, and frames per second (FPS) as a measure of whole application performance. The results are the average from 5 executions.

The application binaries for each optimisation step are generated by compiling the sources with the following CPU flags: *-O3 -mcpu=cortex-a15 -mtune=cortex-a15 -mfloat-abi=hard -mfpu=neon-vfpv4*; and the GPU flags: *-cl-fast-relaxed-math -cl-mad-enable -cl-no-signed-zeros -cl-denorms-are-zero -cl-single-precision-constant*. The frequency governor of the chip is set to *performance*.

B. Temperature Measurement

We measured temperature using the available sensors in the chip. The *sensors* command of the *lm-sensors* package outputs the temperature of the chip measured by the hardware sensors in Celsius degrees. We measured temperature every two seconds. This interval is a good compromise to let enough time for temperature changes while avoiding missing temperature peaks.

We measured chip temperature for the original KF version without cooling, with heat sink, and with heat sink and fan. After that, all measurements are done with heat sink plus fan for stability except the last one. As mobile devices do not have space to include cooling solutions such as heat sinks and fans, we did the final comparison without any cooling to get a real insight of the performance that can be achieved by both original and optimised versions in a real device.

C. Power Measurement

We measured the power consumption of the board using a Yokogawa WT230 power meter [16]. Figure 2 shows the connection scheme. The meter is connected to act as a bridge between the power socket and the board, and it measures the power of the entire platform, including the power supply and the fan. The power meter has a sampling frequency of 10Hz and 0.1% precision. There is no measurement overhead, as all readings are captured in the laptop through the serial port

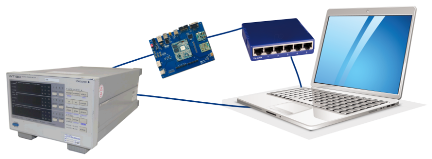


Fig. 2. Power meter connection scheme.

without affecting the execution on the board. The start and end of the readings is triggered by the running script in the board that connects through SSH to the laptop.

We also calculated energy-to-solution [17] to evaluate how our optimisations affect battery life.

V. Results

In this section, we present the results of our temperature, performance and power evaluation of KF running on mobile processors. First, we show the impact of different cooling solutions on the original C++ version, second, the impact on performance and power of our optimisations, third, the comparison of the original and optimised versions without any cooling, and, finally, the evaluation of the performance portability of our code.

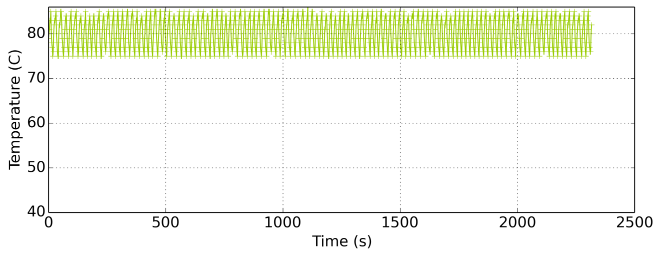
A. Original Version Evaluation

In this section we present the results of the original C++ version in terms of temperature, performance and power.

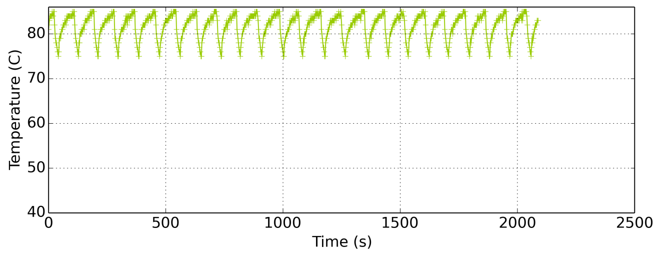
1) Temperature: We decided to measure chip temperature because we were experiencing a large variation for different executions. We found that thermal throttling caused frequency downscaling that varied from run to run. In an attempt to achieve consistent performance results, we evaluated different cooling solutions: without cooling, with a heat sink, and a heat sink plus a fan.

Figure 3.a shows the temperature of the chip throughout time during the execution of the person benchmark. To avoid chip damage due to overheating, the operating system monitors temperature and when it reaches 85°C, the maximum allowed, it downscales frequency to lower temperature to 75°C. Then, frequency is increased back and temperature increases again to 85°C because it is still running a compute-intensive workload. This up-and-down frequency scaling degrades performance due to the overhead of periodically changing frequency and due to the reduced amount of time working at maximum speed.

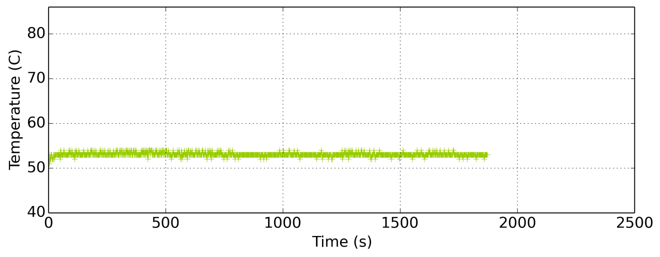
Figure 3.b shows the same measurements after covering the chip with a heat sink. There is still thermal throttling



(a) Original version without cooling



(b) Original version with heat sink



(c) Original version with heat sink and fan

Fig. 3. Temperature in Celsius degrees over time of the original version (person benchmark). Without cooling (a) there is intense thermal throttling activity; adding a heat sink (b) reduces the frequency of thermal throttling; and adding a fan (c) completely avoids overheating and thermal throttling disappears.

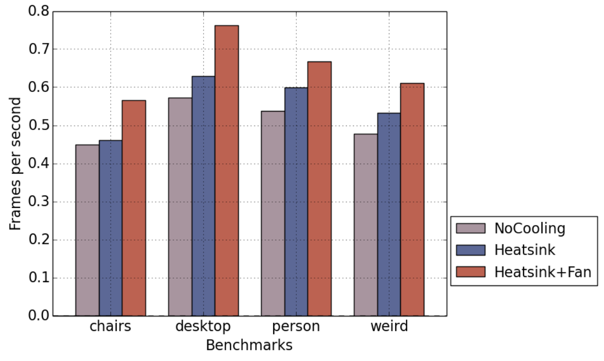


Fig. 4. Performance in FPS of the original version for the different benchmarks and cooling configurations.

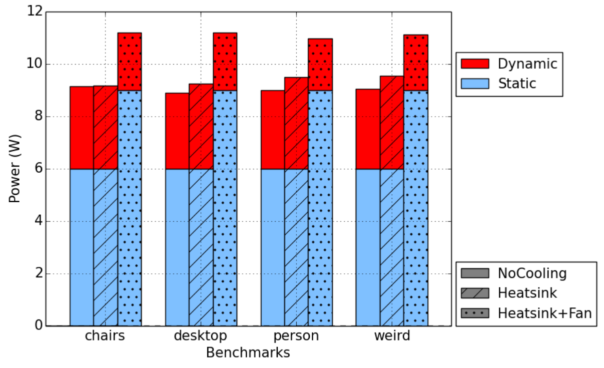


Fig. 5. Average power consumption in Watts comparing the original version (for all benchmarks) without cooling, with heat sink and with fan.

but it happens less frequently. The time to reach 85°C is longer, and it consequently lowers the number of times that it is necessary to change the frequency. In the same time the frequency changed ten times without cooling, it changes two times with the heat sink. This way, the chip works more time at higher speed and the overhead of changing frequency is paid less times.

Figure 3.c shows the same measurements after adding a fan on the top of the heat sink. This finally avoids thermal throttling: the temperature never reaches the maximum and stays almost constant. This way, the chip works at the maximum frequency all the time so there is no overhead associated to frequency scaling.

2) Performance: Figure 4 shows the FPS rate of the four different benchmarks of the original version with the three cooling setups. Thermal throttling has a clear impact in performance that leads the heat-sink-only version to outperform the execution without cooling by 11%. The setup with heat sink and fan experiences a 12% speed-up compared to the execution with heat sink and is 24% better than the execution without cooling.

3) Power: Figure 5 shows the static and dynamic power of the original version for the different cooling setups and benchmarks. We consider the static power as the idle power. The static power consumption is the same for the original and heat sink setups (6W), but dynamic power is higher with heat sink. This is because there are fewer changes of frequency so it remains more time at the highest frequency. The setup with fan consumes more power for two reasons: the fan adds extra power so static power is higher (9W); and dynamic power is also higher because it runs at maximum frequency during the whole execution. The dynamic power increase of setup with fan over the setup with heat sink is 23% for chairs; 20% for desktop; 4% for person; and 8 for weird.

Including a heat sink and a fan into a mobile device is not feasible and the performance improvement in these experiments is approximately the same as the power increase. Nevertheless, we perform our optimisations evaluation using the heat sink and fan so we avoid the interference of thermal throttling from reasoning about the effect of each optimisation, and also to have a lower variability in our results.

B. Optimisation Results

In this section we present the performance and power results of the multiple optimisation steps listed in Table II.

1) Performance: Figure 6 shows the execution time for all benchmarks and optimisations decomposed for the different program stages. The first three optimisations, OCL-Initial,

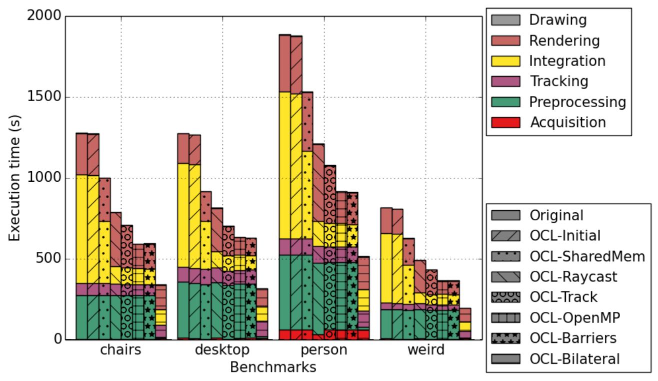


Fig. 6. Execution time breakdown of each stage in seconds of all the optimisations for the different benchmarks.

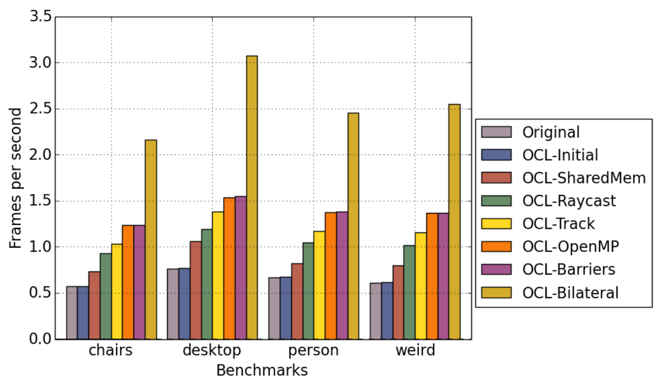


Fig. 7. Performance in FPS of all the optimisations for the different benchmarks.

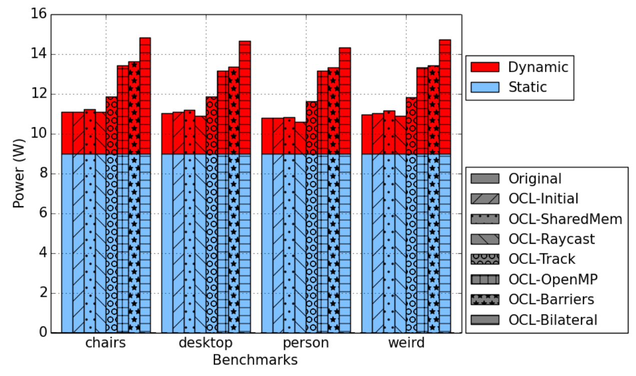


Fig. 8. Average power consumption in Watts comparing all optimisations and benchmarks.

OCL-SharedMem, OCL-Raycast, affect the Integration stage. Even though OCL-Initial executes faster the *integrate* kernel on the GPU, the overhead of the copies cancels that benefit and get the same performance as the Original version. Removing data copies, OCL-SharedMem, avoid that overhead and represents a 45% execution time reduction of the Integration stage. OCL-Raycast get a further 71% over OCL-SharedMem. Both together reduce execution time of the Integration stage by 84%. This reduces overall execution time by 30%.

The porting of another function (*track*), in OCL-Track, reduces the execution time of the Rendering stage by 9%. Parallelising some parts of the application to use the multiple cores in the chip with OpenCL, optimisation OCL-OpenMP, represents a further overall execution time reduction of 14% versus OCL-Track. The reduction of synchronisations in OCL-Barriers has a negligible impact. It appears the synchronisations we removed were having little or not overhead. This optimisation would probably be more important for discrete GPUs without shared memory, where the overhead of these calls is larger.

Last, porting the *bilateral_filter*, OCL-Bilateral, dramatically reduces the execution time of the Preprocessing stage by 96%. This represented an execution time reduction of 46% versus OCL-Barriers.

Comparing the original version to the last version with all optimisations, we achieved a 74% execution time reduction, which translates to a speed-up of 3.93×. This translates to an improvement in the FPS rate.

Figure 7 shows the achieved FPS for each version. From the original 0.6–0.7 FPS, combining all optimisations we achieved between 2.2–3.1 FPS.

2) Power: Figure 8 compares the power consumption for each optimisation. The power consumption of the first four versions is similar. It seems the additional power of running on the GPU is compensated by the power saved by keeping the core idle meanwhile.

The first optimisation with a significant impact on power is OCL-OpenMP. This is because not only the GPU is used extensively, but also both cores in the CPU side are working in parallel. Then, OCL-Bilateral also implies an increase in power consumption, but in a much lower proportion than the reduction in execution time of almost half mentioned before.

Comparing the original version with the last optimisation (OCL-Bilateral), power consumption increased in 38%. If we compare the power consumption increase with the performance increase, we see a clear benefit: power increased by 38% while performance grew by 293%. This large difference is mainly due to the large portion of static power, which is paid regardless of the performance achieved.

In terms of energy-to-solution for all benchmarks, we get that the original version consumes 58409 J and the optimised version 19850 J, which represents a 66% reduction in energy. For other metrics such as energy-delay (ED) and energy-delay-square (ED^2) products, the optimised version reduces them in 91% and 98%, respectively. This means 3× better energy-to-solution, 11× better ED and 44× better ED^2.

C. Final Results

Considering only the optimisations in the Integration stage, the speed-up for that stage is 7.13×. For the whole application, this means a speed-up of 1.60×. We calculated, using Amdahl's law, the maximum theoretical speed-up assuming infinite resources and the result was 1.99×. With our optimisations and considering the limited resources of an embedded platform, achieving 1.60× over 1.99× is a very good result.

Overall, comparing the setups with fan (original and optimised), the speed-up is 4×. Since having these cooling solutions in a mobile device is not feasible, we also ran our optimised version in the board without any cooling to get a

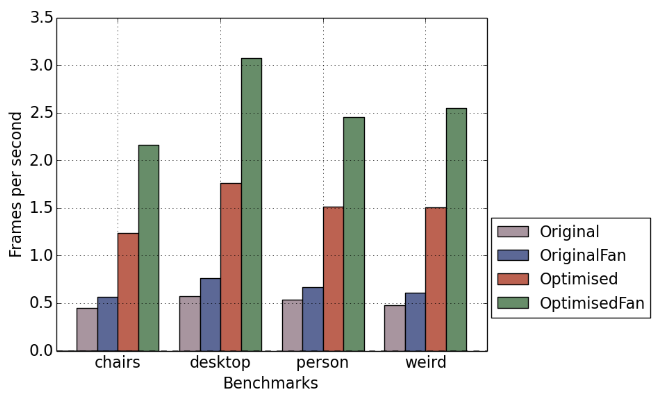


Fig. 9. Performance comparison in FPS of the original version and the optimised version for the different benchmarks and cooling configurations.

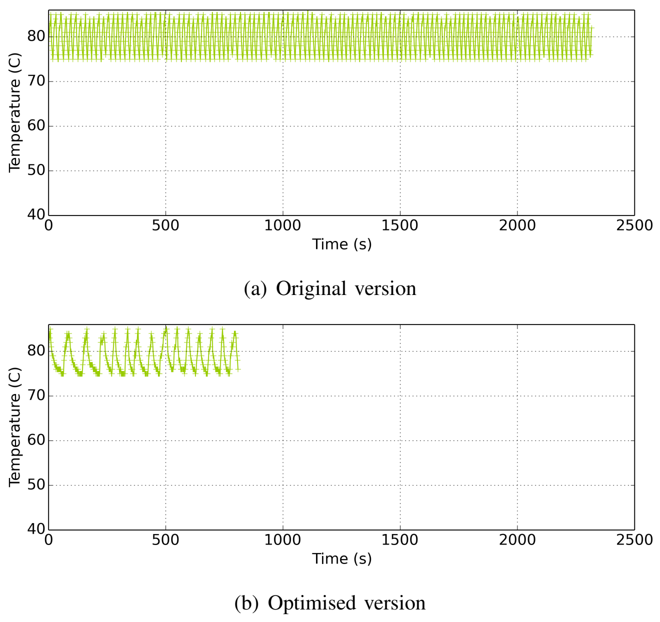


Fig. 10. Temperature over time in Celsius degrees (person benchmark) without cooling. In the original version (a) there is intense thermal throttling activity, but with our optimisations (b), it is less frequent. Execution time is shorter thanks to the better exploitation of parallelism of our optimisations and also due to the lower thermal throttling activity resulting from a reduced CPU activity.

sense of what would be the improvement on a real device. Figure 9 shows the FPS rate of both original and optimised (OCL-Bilateral), for the setups without cooling and with heat sink plus fan. The average speed-up of the optimised without cooling compared to the original without cooling is 3×.

Figure 10(a) shows the temperature for the original version, and Figure 10(b) shows the temperature for the optimised one. With our optimisations, not only we improved performance, but also we reduced the frequency of thermal throttling. In the optimised version, the GPU is active for long periods of time leading the CPU to be idle waiting the GPU to finish. Our hypothesis is that the heating produced by the CPU is lower in the optimised version than in the original due to its lower activity. Although the GPU is active in the optimised version and not in the original, it seems that this activity does

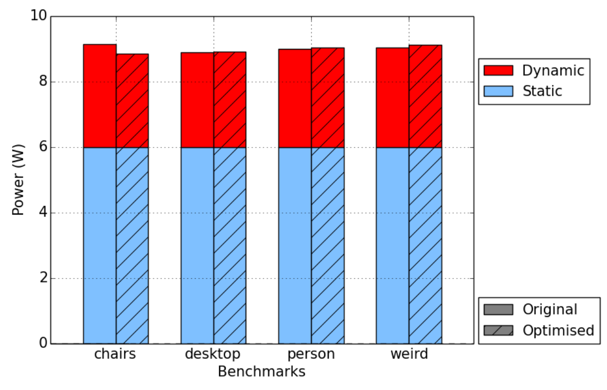


Fig. 11. Average power consumption in Watts comparing the original and optimised versions without cooling.

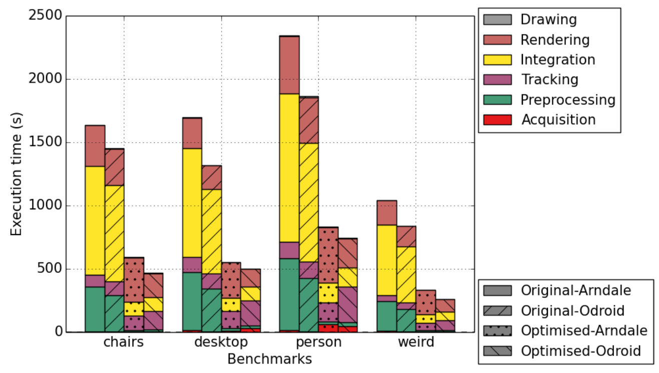


Fig. 12. Execution time breakdown of each stage in seconds of the original and optimised versions on the Arndale board without cooling, and the optimised version on the Odroid-XU3.

not generate an amount of heat that compensates the lower utilisation of the CPU. This would result in overall lower heating in the optimised version and therefore explain the lower thermal throttling activity.

Figure 11 shows that the power consumption of the original and optimised versions without cooling is practically the same. So, for the same power, we achieve better performance and more stable results as the frequency of thermal throttling is lower. This shows the benefits of the better energy efficiency of the GPU.

D. Performance portability

We also evaluated our optimised code with a different development board, the Odroid-XU3. This board integrates a more powerful SoC featuring the next generation of the GPU.

Figure 12 shows the execution time breakdown of our optimised version on the Odroid-XU3 compared to the execution of the original and optimised versions on the Arndale board. The main differences between the optimised versions are the Rendering and Tracking stages, all other stages take a similar time to execute. We executed the program in the Odroid-XU3 with two CPUs that are exactly the same as the ones in the Arndale board but running at a higher speed. In the optimised version, the *track* kernel from the Tracking stage runs in the GPU and the Rendering stage is completely executed on the

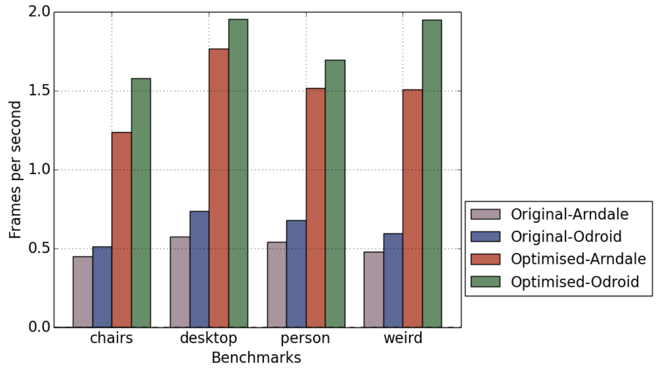


Fig. 13. Performance comparison in FPS of the original and the optimised versions on Arndale and Odroid-XU3 for the different benchmark.

CPUs. So, the stages that run mostly on the GPU take similar time on both platforms. However, the stages that run mostly or completely on the CPU run faster on the Odroid because the CPUs run at a higher frequency.

Figure 13 shows the comparison of the performance achieved by the Arndale board and the Odroid-XU3. The optimised version runs at a speed between 1.6 and 1.9 FPS. This is 2.87× speed-up compared to the original run on the Odroid-XU3 and 1.2× faster than the optimised in the Arndale board. The speed-up of the optimised versus the original in the Arndale board was 2.95×. This similar performance improvement demonstrates the performance portability of our optimisations.

VI. Conclusion

Mobile devices are truly heterogeneous SoCs with specialised processing elements or accelerators for faster and more energy-efficient computation. Initially, GPUs were used only to accelerate graphics. Nowadays, the performance of embedded GPUs, and mobile SoCs in general, is improving, and embedded GPUs are becoming general purpose GPUs. This context and the many sensors mobile devices have, creates the perfect environment for CV on mobile devices.

CV applications consist of applying several computations on a large amount of data, commonly to a matrix of pixels. These algorithms are parallel by their input files and its own nature. GPUs are very useful for speeding up graphics but also for speeding up this kind of computer vision applications.

We optimised the KF application, representative for other state-of-the-art CV applications, for an specific SoC commonly found in mobile devices using the Arndale development board. The main optimisations consisted on porting parts of the application to the GPU on which they execute faster and requiring less energy thanks to the higher energy efficiency compared to the CPU for compute-intensive operations. We also executed our optimised version on a newer and more powerful platform, the Odroid-XU3 development board. Just by recompiling and rerunning the optimised code, we achieved better performance, which demonstrates the performance portability of our optimisations. The optimised version resulting from our work serves as a base code that can be easily adapted to other platforms with embedded GPUs.

Regarding hardware constraints, apart from the compute-capable support, by the nature of the packaging of a mobile device, it is difficult to dissipate heating. Current mobile chips are not ready for sustained full utilisation because the chip overheats and starts throttling, slowing down performance. Due to experiencing large variation in the performance of different executions, we decided to measure the temperature of the chip and we realised that there was thermal throttling. We put a heat sink to solve this issue but it just helped to reduce the frequency of thermal throttling. Adding a fan removed thermal throttling and kept temperature below the limit. Nevertheless, these cooling elements cannot be integrated in a mobile device as they do not fit. Also, the performance benefit of using the fan was cancelled by the extra power the fan consumes.

Despite all our optimisations, we learnt that mobile devices are not yet ready for demanding CV applications such as KinectFusion. Combining all our optimisations using the GPU, we obtained between 2.2 and 3.1 FPS which represented a 4× speed-up compared to the original CPU version. Although this is still far from an interactive rate (30 FPS), this work is one step forward towards this target.

References

[1] J. Rivera and R. van der Meulen, "Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments to Grow 4.2 Percent in 2014." http://www.gartner.com/newsroom/id/2791017, 2014.

[2] N. Trevett, "Accelerating mobile augmented reality." Keynote at InsideAR Conference, 2012.

[3] B. Smith et al., "More iPhone Cool Projects," 2010.

[4] P. Hasper et al., "Remote Execution vs. Simplification for Mobile Real-time Computer Vision," in *9th International Conference on Computer Vision Theory and Applications (VISAPP)*, 2014.

[5] "PAMELA project." http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/K008730/1. Last accessed: Sep 2014.

[6] "Project Tango." https://www.google.com/atap/projecttango/. Last accessed: May 2014.

[7] S. Izadi et al., "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011.

[8] "Irida Labs." http://www.iridalabs.gr. Last accessed: Oct 2014.

[9] "IKEA catalogue app." http://www.ikea.com/gb/en/catalogue-2015/index.html. Last accessed: Oct 2014.

[10] "Word Lens." http://questvisual.com/. Last accessed: Oct 2014.

[11] "Movidius." http://www.movidius.com. Last accessed: Oct 2014.

[12] D. Moloneya, B. Barry, R. Richmond, C. Brick, D. Donohoe, *et al.*, "Myriad 2: Eye of the Computational Vision Storm." HotChips 26: A Symposium on High Performance Chips, 2014.

[13] K. Cheng and Y. Wang, "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pp. 1–4, IEEE, 2011.

[14] G. Wang et al., "Computer Vision Accelerators for Mobile Systems Based on OpenCL GPGPU Co-Processing," *J. Signal Process. Syst.*, vol. 76, no. 3, pp. 283–299, 2014.

[15] "Odroid-XU3." http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127&tab_idx=1. Last accessed: Jan 2015.

[16] Yokogawa, "Yokogawa WT210/WT230 Digital Power Meters." Bulletin 7604-00E: http://tmi.yokogawa.com/files/uploaded/BU7604_00E_050_1.pdf.

[17] E. Padoin et al., "Time-to-solution and energy-to-solution: a comparison between arm and xeon," in *Applications for Multi-Core Architectures (WAMCA), Third Workshop on*, pp. 48–53, IEEE, 2012.