Optimisation Opportunities and Evaluation for GPGPU Applications on Low-End Mobile GPUs

Matina Maria Trompouki*, Leonidas Kosmidis[†,*]

*Universitat Politècnica de Catalunya   [†]Barcelona Supercomputing Center (BSC)

Abstract—**Previous works in the literature have shown the feasibility of general purpose computations for non-visual applications on low-end mobile graphics processors using graphics APIs. These works focused only on the functional aspects of the software, ignoring the implementation details and therefore their performance implications due to their particular micro-architecture. Since various steps in such applications can be implemented in multiple ways, we identify optimisation opportunities, explore the different options and evaluate them. We show that the implementation details can significantly affect the obtained performance with discrepancies up to 3 orders of magnitude and we demonstrate the effectiveness of our proposal on two embedded platforms, obtaining more than $16\times$ speedup over benchmarks designed following OpenGL ES 2 best practices.**

I. INTRODUCTION AND RELATED WORK

The computational power of modern mobile GPUs is approaching the capabilities of high performance systems of the last decade [2]. However, most of the time they are idle, since their full computational power is only unleashed on graphics-intensive applications. In order to leverage their performance for a wider range of applications, most recent high-end (OpenGL ES 3 compatible) GPUs, already support parallel programming models such as OpenCL. These GPUs though are currently powering only the most expensive mobile phones and development kits, denying the rest of the market access to the potential benefits of general purpose computations.

In fact the majority of the mobile market today is based on low and mid-end GPUs [17] supporting only OpenGL ES 2, which are incapable of OpenCL requirements. Moreover, a large segment of the embedded systems market, single board computers, is based on commodity mobile GPUs, to provide very low cost (∼$30) solutions, such as the Raspberry Pi.

Although these low-end GPUs lack OpenCL support, several successful attempts have been done so far towards general purpose computations on them. These efforts can be divided in two groups: a) programming in low-level programming interfaces such as machine code and b) programming in high-level graphics programming languages.

The former category applies only to Broadcom's Video-Core IV, the sole embedded GPU with open documentation so far [1]. Programming is performed in device-dependent assembly code, without official support in development tools. This not only increases the difficulty and the complexity of developing GPGPU programs on this platform, but also prevents portability to other systems. For these reasons only a handful of people have managed to develop small-scale useful applications, resulting in a poor collection of only 3 known programs which take advantage of this GPU [21][8][4]. Furthermore, the portability of this solution is so limited, that those applications don't work on the newer versions of Raspberry Pi, although they are based on the same GPU [15].

The second category, based on graphics APIs to perform general purpose computations has only recently appeared [13]. Despite that this method is very similar to the ones used on the early days of GPGPU programming, it requires special

conversion of data from and towards the GPU memory, due to inherent limitations in the design of the OpenGL ES 2 API. These solutions offer high portability over different platforms, even on high-end GPUs, due to the backwards compatibility of OpenGL ES 3 devices. Moreover, their performance has been shown to be in par with assembly optimised GPGPU applications [5]. However, so far only the functional aspects of those methods have been considered, leaving a lot of room for performance improvements.

In this paper we identify potential performance optimisation opportunities in the design of such applications and we evaluate their effectiveness on two commodity embedded platforms. We show experimentally that despite many optimisations seem straightforward, their performance impact on mobile GPUs can be counterintuitive and we convey the reasons. In total, by combining our optimisations we obtain a speedup of more than $16\times$ over a baseline version designed following OpenGL ES 2 best practices [14][11].

II. OPTIMISATION OPPORTUNITIES

The general structure of a GPGPU application over a mobile graphics API features several optimisation points:

Vertex Processing: Before a kernel is invoked, the vertex data of the kernel are copied to GPU-managed memory, as shown in step 1 in Figure 1. Although in embedded platforms the CPU and the GPU share the same physical memory, the API specification requires this implicit copy and therefore it involves a memory allocation of GPU-managed memory. While the memory and latency cost of the extra copy in the GPU memory might not be significant for an application using a single kernel invocation, most GPGPU algorithms require the repetitive execution of several kernels.

Those memory management overheads can be avoided by allocating and mapping GPU-managed memory through *Vertex Buffer Objects* (VBO), using the `BufferData` API call. Moreover, a hint about the usage of this memory can be given, so that the GPU driver does not need to perform additional costly operations to keep consistency with the CPU.

Texture Loading: Whenever a texture is created using the `glTexImage2D` call, similarly to the vertex case, the GPU driver needs to allocate GPU-managed memory and subsequently copy its data there (step 2 in Fig. 1). Depending on the system and the OpenGL ES implementation, the copy can be performed by DMA, so that the operation is not blocking and subsequent operations can be executed and therefore potentially overlap with GPU operations. However, in applications that need to repetitively transfer data from the CPU, the allocation can consume a significant time portion. In this case, the already allocated GPU memory can be reused, by using the `glTexSubImage2D` call. This way the additional allocation can be avoided for the following kernel invocations.

Texture Writing: After the fragment shader execution, the results need to be copied from the framebuffer to a texture using `CopyTexImage2D` (or `CopyTexSubImage2D` to avoid the extra memory allocation), as can be seen in step 4 in Fig. 1. Although this process can also be accelerated by DMA
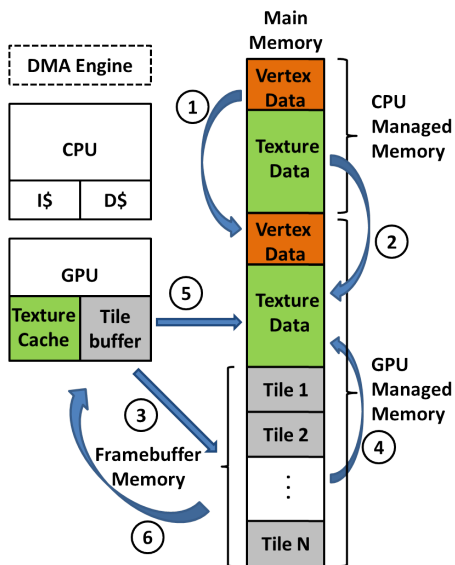
Fig. 1: Memory movement operations in a tiled GPU. Depending on the implementation, the copies are performed by the DMA engine. Only the memory related to graphics operations is depicted. Sizes are not proportionate, for illustration only.

copies, it hides an implicit synchronisation, since the entire rendering needs to be finished, before the transfer is initiated. Moreover, all GPU operations that modify the framebuffer need to be serialised, until the transfer is complete.

In order to appreciate the impact of this operation in embedded GPUs, we need to consider their micro-architecture. These GPUs in their entirety follow a tile-based architecture [9], for bandwidth and power reasons [10]. Rendering takes place in small *tiles* eg. 16x16 pixels, therefore the framebuffer consists of two parts: a small on-chip memory equal to the size of the tile, and a GPU-mapped memory buffer in main memory, which holds the entire framebuffer data, as shown in Fig. 1. In the above situation, despite the tile-rendering, all the data need to be written in the frame buffer in main memory (step 3) before being able to be copied to the texture memory [7].

To avoid this serialisation, *texture rendering* can be employed, by creating a *FrameBuffer Object* and binding the corresponding texture to it with `FramebufferTexture2D`. This way, as soon as a tile finishes rendering in the on-chip memory, it is asynchronously transferred to the corresponding texture, avoiding the extra copy (step 5 in Fig. 1).

Another performance implication of tile-based architectures is that reading back to the on chip framebuffer the previous contents of the framebuffer in memory is expensive (step 6 in Fig. 1), however this is the default operation in most GPUs. To avoid unnecessary traffic and performance penalty when it is not required, some architectures provide an extension (`EXT_discard_framebuffer`). Alternatively an application can invalidate the frame contents before the kernel launch using the `glClear` function, achieving the same effect.

Most mobile GPU micro-architectures, are also *deferred* [18]. This means that rendering is postponed until the next frame, during which it overlaps with the next frame's vertex processing. Whenever dependencies exist between two consecutive frames, the overlapping cannot occur, introducing bubbles in the pipeline. Under specific circumstances, this can happen in multi-pass algorithms, which we examine in the next Section. In that case, texture rendering can produce diminished performance.

Windowing Subsystem properties: Applications with visual output, whether performing GPGPU computations or not, rely on the `eglSwapBuffers` call to synchronise between two consecutive frames and prevent half-rendered images. However, this call forces a wait until all the submitted work in the GPU has been finished, limiting performance. In addition, even when the GPU work has finished before this call, EGL by default synchronises the GPU output with the refresh rate of the screen, to avoid performing more work that the human eye can detect. This is called `vsync` and limits the GPGPU kernel invocations to that rate, typically 60 per second.

Applications without visual output can avoid this call completely, achieving the maximum kernel launching rate. However this is not always possible, eg. in applications that use the display but perform GPGPU computations in the background. In this case, the synchronisation can avoid being limited by the display's refresh rate, using the `eglSwapInterval(0)` call. This still prevents distorted display, by waiting until the current frame is rendered, but the following GPU work is executed as soon as this synchronisation is complete.

Kernel Code: The actual arithmetic operations and precision play a significant role in the performance. Writing code in a way that is easy to identify multiply and add (MAD) operations, which are ubiquitous in embedded GPU ISAs, can help the compiler to generate appropriate instructions. Moreover, the use of built-in functions such as `dot` for computing inner products or `clamp` to saturate results in a range, can accelerate computations since many vendors directly implement those functionalities in hardware, so that the compiler can map them easier to assembly instructions.

A specific use of this optimisation can be applied when GPGPU is implemented as described in [13]. The authors of that paper, show that the output of GPGPU computations achieved in that way is between 24 and 32 bits of precision, mainly due to floating point format limitations. In that case, instead of performing multiplications in full 32-bit precision which is going to be lost after the execution of the shader, the `mul24` built-in function can be used, which performs a faster multiplication with 24 bits only. Moreover, input and output can be restricted in reading/writing only 3 out of the 4 bytes of each element, reducing the bandwidth requirements by 25%.

III. MULTI-PASS ALGORITHMS

Multi-pass algorithms are used to implement complex GPGPU computations. These algorithms consist of multiple kernel invocations, executed in a software pipelined fashion. Multi-pass shaders have been employed from the interactive graphics community for many years, in order to produce sophisticated graphics effects [3]. However their main difference with GPGPU computations is that typically those computations write their output to the same target. This is not the case for GPGPU tasks, where the output of one kernel is used as an input to the next one. Due to the OpenGL ES 2 specification, a texture cannot be used at the same time for both input and output, because this can result in unexpected behaviour. As a consequence, an intermediate buffer is needed to keep intermediate results between kernel invocations.

Multi-pass algorithms can be used to solve problems related to exceedance of implementation limits in kernel code, such as the number of instructions or texture accesses in a shader. Moreover, multi-pass implementations can improve the memory usage of the individual kernels, so that the memory hierarchy can serve more effectively the memory requests.

Despite its numerous advantages, this optimisation not only is more complicated, but also has significant performance implications related to the micro-architecture of the mobile GPUs we consider, as we show in the Evaluation Section.

```
void main(){
  float acc=0.0, A, B, i;
  for(i=0.0; i<(1.0/(M/BLOCK_SIZE)); i+=1.0/M) {
    reconstr_in(A, text0, vec2(i+blk_n, Coord0.y))
    reconstr_in(B, text1, vec2(Coord1.x, i+blk_n))
    acc+=A*B;
  }
  reconstr_in(interm, text2, Coord2)
  encode_out(acc+interm)
}
```

Fig. 2: Multi-pass kernel implementation of *sgemm*. `reconstr_in` and `encode_out` represent the transformation functions proposed by [13], to transform kernel inputs and outputs from and to normalised values. Before the kernel invocation the uniform variable `blk_n` needs to be set to $current\_block * BLOCK\_SIZE/M$.

IV. A MULTI-PASS CASE STUDY

In this Section we show the case of multi-pass conversion in a frequently used kernel, the single precision matrix-matrix multiplication or *sgemm*. In Figure 2 we show a multi-pass implementation of this benchmark, where the computation is performed in steps of N elements at each kernel invocation.

The texture `interm` is used to keep intermediate results between kernel invocations. The next kernel needs to add the intermediate values it computed to the ones produced from the previous kernel to its output. Since a kernel cannot read and write in the same texture, we need an additional intermediate texture for this. In each kernel invocation we swap one with the other, implementing a double-buffering scheme.

Each kernel launch computes the dot product of a chunk or *block* of N elements. For this reason, this type of computation is called blocking [19] or tiling [22]. Assuming that the size of the matrices is $M \times M$, and M is multiple of N, we need to perform M/N passes to perform the multiplication. Each kernel requires to know which block needs to be processed. This information is passed to the kernel as uniform before each kernel launch.

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

We evaluate our proposed optimisations on a Raspberry Pi with a VideoCore IV GPU and a mobile platform with an Imagination PowerVR SGX 545 GPU, both implementing a tile-based deferred rendering (TBDR) micro-architecture [18].

In the absence of GPGPU benchmarks over graphics interfaces and considering that the effort of porting known OpenCL GPGPU benchmark suites such as Parboil [12] exceeds the scope of this article, we use two representative benchmarks, which exhibit significantly different behaviour, similar to [13].

The first application *sum*, implements streaming addition over two matrices. The second benchmark implements the *sgemm* benchmark, which was presented in the use case of Section IV. Both kernels represent common linear algebra operations, used in abundance in many applications such as machine learning [21], computer vision [15] or numerical solvers [16][20]. The baseline versions of both applications have been developed from scratch, following the performance guidelines by [14] and implementing the kernel input/output conversions proposed by [13]. The benchmarks are executed on random 1024x1024 matrix inputs.

To highlight small improvements in some optimisations (asynchronous data transfers and synchronisations), results are obtained with the entire benchmark body executed 10000 times. Due to the large exploration space, we follow an incremental approach, starting from one configuration and applying the next optimisation on the best performing one, in the order they are presented.
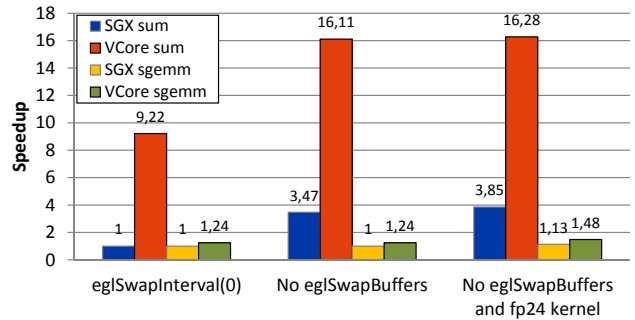

Fig. 3: Effect of Vsync for *sum* and *sgemm*

B. Results

Figure 3 shows the performance impact of *vsync*. Setting the Swap Interval to 0 gives a significant improvement on VideoCore, on which the default interval is 60Hz. On SGX it has no effect, since synchronisation keeps taking place at the default rate which is much higher. Without swap buffers, both platforms experience a boost but in a different degree. Because *sum* has low arithmetic intensity, each kernel invocation finishes very fast, much faster than the refresh rate. Therefore, disabling *vsync* on VideoCore skyrockets its performance to $9.2\times$. Moreover, when *sum* is used in an application without graphics output, eliminating `eglSwapBuffers` results in a speed up of $16\times$ on the VideoCore and $3.47\times$ on the SGX.

On the other hand, *sgemm*'s computational intensity is so high, that each kernel execution takes longer than the refresh rate. Therefore disabling vsync improves performance less, 24% on the VideoCore. Applications such as *sgemm* are called fragment-shading bound in the graphics terminology [3]. Finally, reducing kernel I/O and arithmetic operations to 24 bits, provides an additional improvement of 10% on each platform.

Vertex Buffer Objects (VBO) improve *sum* performance in both platforms up to 1.5% depending on the memory hint provided, however the plot is omitted for space limitations.

Figure 4a compares texture and framebuffer (FB) rendering performance. Surprisingly, direct texture rendering is not always the most efficient rendering choice, regardless of the suggestions from hardware vendors [11] and language guides [14]. In fact, which one is superior depends not only on the platform but on the application, too.

sgemm benefits from rendering to the framebuffer in both platforms, due to its multi-pass implementation and the mobile GPU micro-architecture. Recall that the multi-pass implementation requires the output texture of one kernel to be fed as an input texture to the next one. The framebuffer is by default double-buffered, therefore it contains two surfaces which can be used in alternate fashion, as opposed to off-screen textures which are by default single-buffered, creating a dependence between two consecutive frames. Since both architectures follow a deferred rendering architecture, which suffers a significant penalty when two consecutive frames depend on each other as explained in Section II, the performance is reduced when double buffering is not used. On the other hand, the default implementation of *sum* does not have dependencies between two consecutive kernels and therefore texture rendering is more effective (1/0.000447=2237×), due to the most efficient tile copying mechanism we described in the same section. Note that this is a significant difference of 3 orders of magnitude which highlights the importance of selecting the appropriate rendering target, especially in the absence of dependencies between consecutive kernels. Similarly, texture rendering on VideoCore is around an order
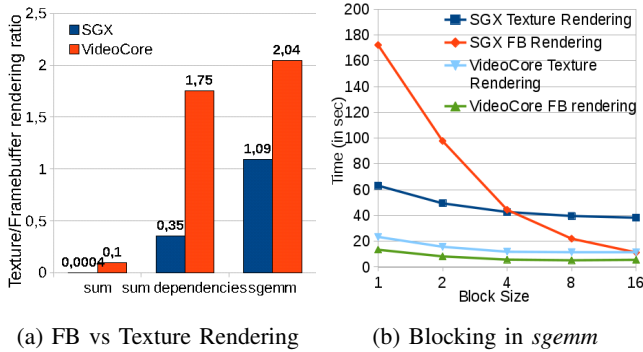
(a) FB vs Texture Rendering   (b) Blocking in *sgemm*

Fig. 4: Performance comparison for different rendering targets and block sizes. Results compare optimised versions.

(a) Texture Rendering   (b) Framebuffer rendering

Fig. 5: Performance improvement with texture memory reuse for different rendering targets using block size of 16.

of magnitude faster than the framebuffer.

When we introduce artificial dependencies between consecutive kernel invocations, it continues to execute more efficiently on the SGX using texture rendering, while on VideoCore with the framebuffer. This difference is due to the larger tile size (64×64) and the exploitation of the DMA controller in the OpenGL ES 2.0 driver in VideoCore, which offloads the overhead of the copy from the framebuffer memory to the texture buffer (operation 4 in Figure 1) at a very high transfer rate, around 1GB/s [6], hiding its latency.

In Figure 4b we see the performance of the multi-pass implementation of *sgemm*. We use a block size up to 16 since in both platforms higher values lead to crashes and shader compilation failures either due to exceeding GLSL implementation limits, such as the maximum number of instructions or the maximum number of texture accesses for a kernel.

The performance increases with the block size in both platforms. On the SGX small block sizes with framebuffer rendering (FB in the figure) deteriorate performance significantly. This happens because the kernel performs a small amount of work and finishes in relatively short amount of time, and therefore the memory copying from the framebuffer dominates the execution time. However, when the block size is equal or exceeds 4, the kernel execution time becomes longer, and thus the copy to texture memory can be efficiently overlapped with computation in the framebuffer's second buffer, reducing significantly the overall execution time. On the other hand, the DMA acceleration in the VideoCore, gives always an advantage to the framebuffer rendering, for any block size.

Finally, in Figure 5a we notice that texture reusing is beneficial for direct texture rendering, mainly for input textures and can give a speedup of 15% to a non-frame rendering bound kernel. However on SGX the reuse causes a small degradation 2-7%. In contrast, framebuffer rendering (Figure 5b) is not improved by texture reuse in either platform, while in the case of *sgemm* on SGX the performance drops noticeably to 70%. This performance degradation is due to the false sharing caused by reusing the same output texture, which makes the dependency between the consecutive kernel invocations more prominent, since on SGX this operation takes long in the absence of DMA assistance. This phenomenon is an excellent example of the trade-off between memory consumption and performance that may arise in a tiled-based GPU architecture.

VI. CONCLUSION

In this work we identified optimisation opportunities in GPGPU applications built on top of the OpenGL ES 2 API, targeting low-end mobile GPUs. We analysed for the first time the effect of the mobile GPU micro-architecture on these optimisations, na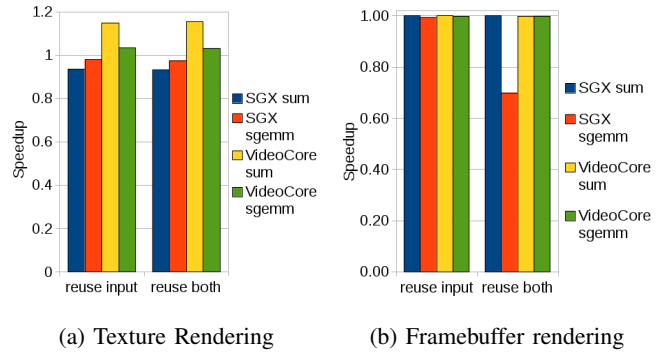mely the tiled-based deferred architecture as well as the utilisation of the DMA controllers in the driver implementation. Moreover we demonstrated that the implementation implications cannot be neglected, since they have a paramount impact on performance which can differ up to 3 orders of magnitude. Finally, we showed their effectiveness on two embedded low-end GPUs obtaining improvements more than 16× over their baseline version build in accordance with OpenGL ES 2 best practices [14][11].

REFERENCES

[1] A birthday present from Broadcom. http://www.raspberrypi.org/blog/a-birthday-present-from-broadcom.
[2] Bringing Console Quality Lighting to Mobile (Presented by Imagination Technologies), Game Developer Conference 2014.
[3] GPU Gems Series, Volumes 1-3. Addison-Wesley.
[4] Hacking the gpu for fun and profit. https://rpiplayground.wordpress.com.
[5] Optimized GEMM performance on Raspberry Pi. https://www.raspberrypi.org/blog/more-qpu-magic-from-pete-warden.
[6] *Question about Zero Copy on VideoCore.* https://github.com/raspberrypi/firmware/issues/85.
[7] Andrew Garrard. Moving pictures: Making the most of the mobile. In *ACM SIGGRAPH 2014 Courses*.
[8] Andrew Holme. GPU_FFT. http://www.aholme.co.uk/GPU_FFT/Main.htm.
[9] I. Antochi. *Suitability of Tile-Based Rendering for Low-Power 3D Graphics Accelerators*. PhD thesis, Delft University of Technology, 2004.
[10] I. Antochi et al. Memory bandwidth requirements of tile-based rendering. In *SAMOS*, 2004.
[11] Imagination Technologies. *PowerVR 3D Application Development Recommendations*.
[12] J. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, March 2012.
[13] M.M. Trompouki and L. Kosmidis. Towards General Purpose Computations on Low-End Mobile GPUs. In *DATE*, 2016.
[14] Aaftab Munshi, Dan Ginsburg, and Dave Shreiner. *OpenGL ES 2.0 Programming Guide*. Addison-Wesley Professional.
[15] Peter Warden. Image Recognition on the Raspberry Pi 2. http://petewarden.com/2015/05/10/image-recognition-on-the-raspberry-pi-2.
[16] Strzodka R. *Hardware Efficient PDE Solvers in Quantized Image Processing*. PhD thesis, University of Duisburg-Essen, 2004.
[17] R. Zioma, Unity. Optimizing PBR for Mobile. In *SIGGRAPH 2015*.
[18] Rys Sommefeldt. A look at the PowerVR graphics architecture: Tile-based rendering. http://blog.imgtec.com/powervr/a-look-at-the-powervr-graphics-architecture-tile-based-rendering
[19] Robert Schreiber. Block algorithms for parallel machines. In *Numerical Algorithms for Modern Parallel Computer Architectures*. 1988.
[20] Turek, Göddeke et al. UCHPC – Unconventional high-performance computing for finite element simulations. ISC'08.
[21] Peter Warden. Deep Learning on the Raspberry Pi. http://petewarden.com/2014/06/09/deep-learning-on-the-raspberry-pi.
[22] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Parallel Processing for Scientific Computing*, 1989.