Towards General Purpose Computations on Low-End Mobile GPUs

Matina Maria Trompouki*, Leonidas Kosmidis*,†
*Universitat Politècnica de Catalunya
†Barcelona Supercomputing Center

***Abstract*—GPUs traditionally offer high computational capabilities, frequently higher than their CPU counterparts. While high-end mobile GPUs vendors introduced recently general purpose APIs, such as OpenCL, to leverage their computational power, the vast majority of the mobile devices lack such support. Despite that their graphics APIs have similarities with desktop graphics APIs, they have significant differences, which prevent the use of well-known techniques that offer general-purpose computations over such interfaces. In this paper we show how these obstacles can be overcome, in order to achieve general purpose programmability of these devices. As a proof of concept we implemented our proposal on a real embedded platform (Raspberry Pi) based on Broadcom's VideoCore IV GPU, obtaining a speedup of 7.2× over the CPU.**

I. Introduction

High-end mobile GPUs, i.e. OpenGL ES 3 compliant, are capable of general purpose programming using OpenCL, however the vast majority of the market, still relies on mobile GPUs in the low-end of the spectrum. In particular, as of September 2015 more than 50% of the end users of games designed with the popular multi-platform game engine Unity, use low-end mobile phones [4], which support only the previous version of the graphics API, OpenGL ES 2. This reduces significantly the potential of leveraging the computational power of GPUs on demanding and innovative mobile applications.

Furthermore, most single board computers (ODROID, BeagleBoard, PandaBoard and others) feature an OpenGL ES 2 compliant GPU. Among them, Raspberry Pi, an affordable ($25) educational system for developing countries, relies on the VideoCore IV GPU, capable of 24 GFlops. A general purpose GPU programming model would increase significantly its educational value, educating a new generation of parallel programmers. In addition, due to its price it has been used by millions of hobbyists for a great variety of applications. While this GPU is capable of general purpose computations, this is currently possible only with low level assembly programming, resulting in a handful of people able to do so. For this reason there are only few applications using its GPU [6][1][17]. However, many more powerful applications would be developed, if general purpose programmability in a high level language eased the 24 GFlops of its GPU to be used for computations, instead of staying idle most of the time.

The mobile API supported by these low-end devices, OpenGL ES 2, is a restricted subset of the OpenGL implementation for desktop computers, with similar functionality. However, the programming interface lacks fundamental operations which are required to perform general purpose computations.

In this work, we present a set of solutions to overcome the obstacles created by the OpenGL ES 2 specification [7], to enable general purpose computations on low-end mobile SoCs. This way the programmability of these platforms is enhanced, enabling more powerful applications to be developed. Moreover, the educational value of these systems is multiplied, increasing the turn around on such efforts.

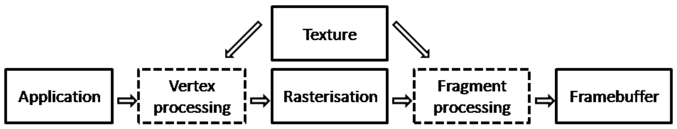

Fig. 1. The graphics pipeline. Dashed stages represent programmable stages.

II. Background and Problem Statement

A. Graphics Pipeline and General Purpose Computations

General purpose computations over graphics APIs are executed on the graphics pipeline, which is depicted in Figure 1. The building block of such applications, is called *kernel*. The programmer maps the data to be processed to geometric primitives, typically to a rectangular *quad* and configures the camera position in a way that the quad covers the screen completely. Each input data is mapped to a texture, whose coordinate corresponds to a specific position in the quad. Constant values are passed to the kernel as *uniforms*. The kernel is described as a vertex or fragment *shader* which operates on the input data to produce the desired output. Next, the output of the kernel is stored in the framebuffer or in a texture. Finally, the output is obtained by reading the texture.

B. Mobile Graphics APIs

Mobile APIs are based on the stable version of the desktop APIs, taking into account their intricacies, such as power, size and bandwidth [5], resulting in stripped lightweight versions of their desktop counterparts. We focus on the second generation, OpenGL ES 2 which introduced programmable vertex and fragment stages (Figure 1), using the OpenGL GLSL ES 1 (GL Shading Language) [8]. As explained in the Introduction, most mobile graphics processors are only compatible with OpenGL ES 2, and will continue to power low-cost solutions where cost is the primary constraint, for products such as Raspberry Pi, for the next 5 years at least. Of course, since new versions of OpenGL ES are backwards compatible with it, the proposed solutions are also valid for newer hardware as well.

Below we list the differences between the desktop and the embedded OpenGL versions, which pose limitations for performing general purpose computations on the mobile version.

1) In the desktop version, the programmer can only use one programmable stage, while relying on the fixed functionality on the other. However, in the ES 2 version, the programmer is forced to program both, even if he/she doesn't apply any custom operations.

2) The desktop version provides many options for geometry primitives as triangles, quads and polygons, as opposed to the embedded version which supports only triangles.
3) Textures are typically two dimensional structures to store images for graphics. Desktop OpenGL versions provide also single dimensional variants, which were widely used for GPGPU applications, when the kernel required single dimensional arrays as inputs, in contrast with the ES version, which supports only traditional 2D textures.
4) The texture coordinates which are used to obtain data from the textures as input, in the desktop version can either be *normalised* - ranging from 0 to 1 - or as is, that is taking values up to the dimension of the texture. The latter have been most popular in GPGPU applications, due to the ease in programmability they offered. In the ES version though, only normalised coordinates are allowed.
5) Despite most low-end GPUs (Mali-4XX, PowerVR SGX, VideoCore IV, Adreno 2XX) support 32-bit floating point operations, the API does not provide means to use floating point textures, for bandwidth reasons. While some vendors provide extensions for half floats, in general it is not enough for general purpose computations.
6) In both specifications the pixel colour values, which are the output of the fragment shader, are converted in the range [0,1] when they reach the framebuffer. Non-image processing general purpose computing applications, relied on vendor extensions which allowed floating point values for framebuffer data. However, in the embedded domain, very few vendors provide a similar extension, to allow the use of half float, which is neither enough nor portable.
7) While the embedded version supports *texture rendering* (writing framebuffer data directly to a texture), it doesn't provide any mechanism to read a texture's data back to client's (CPU) memory.
8) Desktop GL versions support multiple outputs from each shader "thread", e.g. `gl_FragColor`, and `gl_FragData[N]`. Conversely, the ES limits the output to a single array element, either `gl_FragColor` or the first element of `gl_FragData[0]`.

III. Dealing with the challenges

Below we present how the identified challenges can be overcome in a system based on OpenGL ES 2, in order to perform general purpose computations.

1) The GPGPU computations can be either implemented in the vertex or the fragment processing stage (or both), with the fragment one being the most popular. In this case, the vertex computation remains unchanged. Since the mobile API requires this functionality to be implemented instead of relying on fixed functionality, the programmer needs to implement a pass-through vertex shader. Note that the processing requires the shading of a screen covering geometry primitive and the camera viewing directly on that primitive. If this wasn't the case, the appropriate projection should also be implemented in the vertex shader, however the simple camera position allows to override this need. In fact the only use of this pass-through vertex shader is to pass all the required parameters (*varyings*) to the fragment shader which implements the general purpose computation.
2) The absence of complex geometric primitives such as quads can be overcome by using two triangles covering the same region.
3) Previous works [11][9] in the early times of GPGPU computations, presented transformations of single dimensional array accesses, to two dimensional texture coordinates and vice versa. Therefore we can reuse those techniques, which have been developed for the desktop versions of OpenGL.
4) Similarly this problem has been addressed in the literature [9], for desktop GPUs which don't allow non-normalised texture coordinates. Therefore the equivalence of the mobile API allows the reuse of these techniques.
5) The lack of support for 32-bit floating point and other numerical formats except single byte for texture values, requires the development of an appropriate mechanism which can allow the transfer and representation of those values for input in the kernel.
6) Similarly, the lack of the aforementioned numerical formats in the framebuffer, requires also a mechanism to allow saving the kernel's output in a format other than normalised byte values.
7) The limitation of reading a texture back in the CPU memory, can be overcome by using two complementary ways. The first is to use a pass-through fragment shader to copy the texture in the framebuffer, which can be subsequently read to the system memory. However, with careful kernel ordering the texture to be read can be already mapped into the framebuffer, so that there is no need for the additional shader.
8) The fact that a fragment shader cannot output more than a single array means that if a GPGPU kernel does so, it needs to be split in more than one shaders, one per output. However, this is not a real limitation, since most GPGPU kernels, provide a single output. In fact all benchmarks of Rodinia[15] suite fit in these two cases.

Most of the limitations can be easily addressed. In fact the most challenging ones are 5) and 6) which require a way to support any input and output numerical format for the kernels. Next we present a detailed solution for this problem.

IV. Numeric transformations for kernel I/O

The OpenGL ES 2 specification supports only the unsigned byte format for texture and framebuffer values. The texture values c, ranging from [0,255], are interpreted in the shader as floating point values f in the range [0,1] as follows [7]:

$$f = \frac{c}{2^8 - 1} \tag{1}$$

Similarly, before the data are ready to be written in the framebuffer, the values are first clamped in the interval [0,1] and are subsequently converted to unsigned bytes as follows:

$$i = \lfloor f \times (2^8 - 1) \rfloor \tag{2}$$

Based on these properties, we are after a set of transformations able to allow any numeric format to be represented as input and output for compute kernels. The formats we want to enable are the ones supported in the C language: unsigned and signed variants of char and integer, as well as floating point.

A. *Unsigned char*

In order to use in the shader exactly the input values passed, as values between [0,255], we define a bijective mapping M:

$$M : [0, 1] \longmapsto [0, 255]$$

Based on (1) the interval [0,1] is quantised in 256 uniformly distributed sub-intervals, representing values multiple of 1/255. However, an unsigned byte represents multiples of 1/256.

The quantity δ by which 1/255 and 1/256 differ is given by:

$$\frac{1}{2^8 - 1} + \delta = \frac{1}{2^8} \Longrightarrow \delta = -\frac{2^8 - 1}{2^8} \quad (3)$$

Therefore the reconstructed byte value in the shader is:

$$b_u = M(f) = \lfloor f + \delta \rfloor \cdot 255 \quad (4)$$

Conversely for the output in the framebuffer we need first to divide by 255, in order to normalise the value in the range [0,1] and then increase the value by this quantity.

$$M^{-1}(u_b) = \frac{u_b}{255} - \delta \quad (5)$$

Note that neither for the input nor for the output of the kernel, the values require any CPU intervention. The transformation is applied in its entirety by the shader.

B. *Signed char*

For signed byte inputs, we define a bijective mapping M2:

$$M2 : [0, 1] \longmapsto [-128, 127]$$

Building on the transformations for unsigned bytes and on the 2-complement representation's properties we have:

$$b_s = M2(b_u) = \begin{cases} b_u & \text{if } b_u < 128, \\ b_u - 256 & \text{if } b_u \geq 128 \end{cases}$$

In the same way, the inverse transform for the output is:

$$M2^{-1}(b_s) = \begin{cases} \frac{b_s}{255} - \delta & \text{if } b_s \geq 0, \\ \frac{b_s + 256}{255} - \delta & \text{if } b_s < 0 \end{cases}$$

C. *Unsigned Integers*

Integers are represented in the CPU memory as a contiguous set of bytes, each with a different significance. For an integer i_u consisting of 4 bytes $b_{u_i}, i \in [0, 3]$ with 0 representing the least significant byte, its numeric value can be computed as:

$$i_u = \sum_{i=0}^{3} b_{u_i} \cdot 256^i \quad (6)$$

building on the above transformations. Depending on the hardware capabilities this can be either computed in integer or floating point. In the latter case, we need to consider that mobile GPU vendors support 32-bit floating point, with 8 bits exponent and 23 bits mantissa (see next Section). Therefore the maximum integer number which can be represented before starting to skip numbers which are not multiples of powers of 2, is 2^{24}. Consequently the obtained precision is equivalent to a 24-bit integer, enough for most integer operations in an embedded system. For scientific computations it is not a problem either, as they are based entirely on floating point.

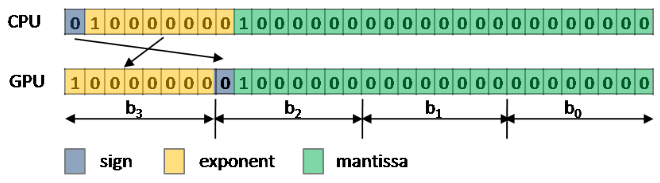

Fig. 2. Floating point representation in CPU and GPU, with corresponding byte values.

For the inverse transformation, we use the remainder of the corresponding power of 256, depending on the byte's significance, to transform back to the host format:

$$b_{u_i} = i_u \; mod \; 256^i \quad (7)$$

D. *Signed Integers*

Signed integers, are reconstructed as unsigned and adjusted:

$$i_s = \begin{cases} i_u & \text{if } b_{u_3} < 128, \\ i_u - 256^3 & \text{if } b_{u_3} \geq 128 \end{cases}$$

Similarly for the reverse transform:

$$b_{s_i} = \begin{cases} i_s \; mod \; 256^i & \text{if } i_s \geq 0, \\ (i_s + 256^3) \; mod \; 256^i & \text{if } i_s < 0 \end{cases}$$

E. *Floating Point Numbers*

The general form of floating point numbers is the following: $(-1)^{sign} \cdot 1.mantissa \cdot 2^{exponent}$. The CPU uses the IEEE 754 floating point format, which defines for single precision floating point numbers 1 bit for sign, 23 for mantissa and 8 bits for the exponent. The exponent is stored in *biased* form, using $bias = -127$, which is the number to be added to its value, in order to obtain the actual value of the exponent.

While the internal GPU format is vendor dependent, the number of bits used for mantissa and exponent can be obtained by the `glGetShaderPrecisionFormat` API call. Most mobile GPUs (VideoCore IV, PowerVR SGX, Adreno 2XX, Mali-4XX[1]), match the IEEE 754 number of bits, which allows to retain the same precision for both CPU and GPU.

Unlike the previously described formats that use the same memory representation for CPU and GPU, this is not the case with the floating point format, which requires reversing the order of the exponent and the sign from the CPU as shown in Figure 2, to have the exponent bits packed in a single byte.

In order to reconstruct the floating point number in the GPU we reconstruct first its components:

$$exponent(u_{b_3}) = \begin{cases} b_{u_3} - 127 & \text{if } b_{u_3} < 128 \\ b_{u_3} - 256 - 127 & \text{if } b_{u_3} \geq 128 \end{cases}$$

$$sign\_value(u_{b_2}) = \begin{cases} 1 & \text{if } b_{u_2} < 128, \\ -1 & \text{if } b_{u_2} \geq 128 \end{cases}$$

$$mantissa(u_{b_i}) = (\sum_{i=0}^{2} b_{u_i} \cdot 255^i) \cdot 2^{-24}$$

Therefore:

$$f = (1 + mantissa) \cdot 2^{exponent} \cdot sign\_value$$

[1]in vertex processor only

For the reverse transformation we decompose them:

$$exponent(f) = \lfloor log2(f) \rfloor$$

$$sign\_value(f) = sign(f)$$

$$mantissa(f) = (f \cdot 2^{-exponent} - 1) \cdot 2^{24}$$

Then we only need to pack the exponent with the sign bit in a single byte, and store each byte of the significand:

$$u_{b_3}(f) = exponent - 127 + sign\_value \cdot 128$$

$$u_{b_i} = mantissa \; mod \; 256^i \; , \; i \in [0, 2] \tag{8}$$

These transformations can optionally preserve special values such as infinities and not-numbers (`NaNs`), which are required in high performance and scientific computing, by checking the exponent value and using the corresponding constant.

V. Results

For the experimental demonstration of our proposal, we use a widely used low-end mobile GPU, VideoCore IV, capable of 24 GFlops [2], featured in the Raspberry Pi. We use 2 benchmarks with one configuration per input type (integer and floating point) described in Section IV. The applications are developed from scratch to show that GPGPU computations are possible on low-end GPUs over graphics API, while providing performance benefit over CPU computations.

The first application, *sum*, applies a simple streaming operation (addition) on two arrays, while the second implements the *sgemm* benchmark. We use matrix sizes of 1024 random-value elements and validate the results with the CPU. The measurements compare application wall times, including time spent in data transfers and kernel compilations.

For the floating point versions, the GPU output is accurate with respect to the fp32 format used by the CPU, within the 15 most significant bits of the mantissa. This results in precision higher than half-float (fp16) supported by extensions in some other GPUs and between fp24 that desktop GPUs used in the early days of GPGPU computing and fp32. This difference comes from the GPU platform (hardware and software), since the same transformations on the CPU are precise.

The *sum* shows a speedup of 7.2$\times$ over the CPU for integer and 6.5$\times$ for floating point, while *sgemm* 6.5$\times$ and 6.3$\times$ respectively. This is in line with speedups reported for a hand-optimised assembly code for this benchmark on this GPU [3]. The current implementation of our proposal is not optimised, which would increase performance further. Fp versions have lower speedups, since in the CPU the integer operations are faster than the fp ones. In general we see that both kernels are able to provide faster execution times than the CPU, even with the extra burden of packing and unpacking inputs and outputs for the computation on the GPU, and the partial bit re-arrangements for the floating point data on the CPU.

VI. Related Work

Several works performed general purpose computations over graphics APIs for desktop systems. Strzodka and Göddeke, pioneers at the early days of GPGPU computing, developed several methods and applications for accelerating scientific computations on GPUs[14][12][13][16]. An exhaustive survey of GPGPU literature on desktop GPUs can be found in [10].

The only work of that era similar to ours is [12], which is centred on emulating 16-bit integer operations on GPU hardware supporting 8-bit fixed-point numbers. Consequently the suggested representation of the 16-bit integer numbers in memory, is a custom format, not the common 2's complement that we use. This reduces the interoperability between CPU and GPU, while our integer solution utilises unmodified 32-bit integers. The biggest and most important difference though, is that our solution covers not only integer formats, but floating point, too, which are indispensable for GPGPU computations.

Although GPGPU is nowadays common place, no work in the literature has shown general-purpose computations on low-end GPUs (e.g. supporting OpenGL ES 2). The main reasons for this, were the challenges described in the Section II and specifically the lack of support for any numeric formats as input and output in graphics shaders for mobile APIs.

VII. Conclusion

In this work we explain the need for general purpose computation on low-end mobile GPUs and the limiting factors that prevent such an implementation over their graphics APIs. We propose solutions for each one and we specifically address the need to pass data in and out from GPU shaders in any numerical format. We have demonstrated the feasibility of performing GPGPU computations on a real mobile platform and we have shown that it can offer significant speedups. We expect this solution to enable developers designing new performance-demanding applications taking advantage of mobile GPUs for general purpose computations.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Leonidas Kosmidis is also funded by the Spanish Ministry of Education under the FPU grant AP2010-4208.

References

[1] Hacking the gpu for fun and profit. https://rpiplayground.wordpress.com.

[2] How powerful is it? https://www.raspberrypi.org/help/faqs/#performanceSpeed.

[3] Optimized GEMM performance on Raspberry Pi. https://www.raspberrypi.org/blog/more-qpu-magic-from-pete-warden.

[4] Unity Mobile Hardware Stats. http://hwstats.unity3d.com/mobile/gpu.html.

[5] Akenine-Möller T. and Strom J. Graphics processing units for handhelds. *Proceedings of the IEEE*, 96(5):779–789, May 2008.

[6] Andrew Holme. GPU_FFT. http://www.aholme.co.uk/GPU_FFT/Main.htm.

[7] Khronos Group. OpenGL ES Common Profile Specification Version 2.0.

[8] Khronos Group. The OpenGL ES Shading Language Version 1.0.

[9] Aaron Lefohn, Joe Kniss, and John Owens. Implementing efficient parallel data structures on gpus. In *GPU Gems2*. Addison-Wesley.

[10] Owens, Luebke, Govindaraju et al. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005*.

[11] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. SIGGRAPH '02.

[12] Strzodka R. Virtual 16 bit precise operations on RGBA8 textures. In *Proceedings of Vision, Modeling, and Visualization (VMV'02)*, 2002.

[13] Strzodka R. *Hardware Efficient PDE Solvers in Quantized Image Processing*. PhD thesis, University of Duisburg-Essen, 2004.

[14] Rumpf M. and Strzodka R. Using Graphics Cards for Quantized FEM Computations. In *Visualization, Imaging and Image Processing Conference*, 2001.

[15] Shuai Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09*.

[16] Turek, Göddeke et al. UCHPC – Unconventional high-performance computing for finite element simulations. ISC'08.

[17] Peter Warden. Deep Learning on the Raspberry Pi. http://petewarden.com/2014/06/09/deep-learning-on-the-raspberry-pi.