Brook GLES Pi: Democratising Accelerator Programming

Matina Maria Trompouki
Universitat Politècnica de Catalunya (UPC)
mtrompou@ac.upc.edu

Leonidas Kosmidis
Barcelona Supercomputing Center (BSC)
leonidas.kosmidis@bsc.es

ABSTRACT

Nowadays computing is heavily-based on accelerators, however, the cost of the hardware equipment prevents equal access to heterogeneous programming. In this work we present Brook GLES Pi, a port of the accelerator programming language Brook. Our solution, primarily focused on the educational platform Raspberry Pi, allows to teach, experiment and take advantage of heterogeneous programming on any low-cost embedded device featuring an OpenGL ES 2 GPU, democratising access to accelerator programming.

CCS CONCEPTS

• **Computing methodologies → Graphics processors**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Parallel programming languages**;

KEYWORDS

GPGPU, OpenGL ES 2, Embedded GPUs, Accelerator Programming

ACM Reference Format:
Matina Maria Trompouki and Leonidas Kosmidis. 2018. Brook GLES Pi: Democratising Accelerator Programming. In *HPG '18: High-Performance Graphics, August 10–12, 2018, Vancouver, Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3231578.3231582

1 INTRODUCTION

The pervasive presence of parallel programming in modern and especially future computing systems has increased the public interest, as well as the importance of learning and experimenting with such computing paradigms. However, the cost of current general purpose accelerators (GPUs, FPGAs and many-core processors e.g. Intel's Xeon Phi) is high, since accelerators per se are expensive. Moreover, accelerators cannot be used as standalone devices, but they require a host computer, usually high-end, in order to be programmed and used, increasing further the total cost.

In addition, the traditional educational model has shifted towards self-education; valuable programming skills are acquired nowadays by students practising their homework at home or by self-educated individuals using online resources, Massively Open Online Courses and educational computers. Inevitably, these target groups do not have access to such expensive hardware to experiment in an individual basis. On the other hand, emulators [Brochard 2011] and simulators [Bakhoda et al. 2009] are not an appropriate option for teaching and learning for several reasons: they are either slow, do not provide performance speedups with respect to the CPU algorithm implementation or do not model timing (e.g. for memory transfers, CPU etc), which is essential for understanding trade-offs and design considerations. More importantly however, they cannot be used for realistic projects and real-life applications that can keep the interested parties motivated in order to invest time and keep improving their skills. For all the above reasons, accelerator programming opportunities are very limited.

In this work, we propose Brook GLES Pi, a port of the accelerator programming language Brook [Buck et al. 2004], which enables GPGPU computing on the embedded GPU of the low cost ($25) educational platform Raspberry Pi. Unlike other accelerators, it is a standalone computer, allowing development directly on the target, while it has a large and active collaborative community.

Although our implementation is optimised for this device, it is completely portable, allowing teaching, experimenting and learning GPGPU programming on any embedded device featuring at least an OpenGL ES 2 GPU, which is currently the case of 99% of the devices in mobile market [Khronos 2018], effectively democratising access to heterogeneous programming.

2 RELATED WORK AND BACKGROUND

Modern accelerators are programmed in OpenCL, CUDA or OpenACC. However, those programming models require specific support not present in low-end embedded devices such the ones that we enable with our proposal. Raspberry Pi, our platform of choice, is a low cost educational computer with a VideoCore IV GPU.

VideoCore IV is the only mobile GPU in the market with open hardware specification [Broadcom 2013], enabling low-level assembly level programming [Müller 2015]. However, the increased programming complexity limits the available applications to 3 [Lorimer 2014][Holme 2014][Warden 2014], while its need for root privileges creates potential security and stability issues.

We opted to use the open source stream processing language Brook [Buck et al. 2004], the predecessor of both CUDA and OpenCL languages, originally designed to work over desktop graphics programming APIs. Brook has been very popular, used by several scientific projects as it is indicated the numerous citations of its original publication [Buck et al. 2004], many of which are open source and therefore offer a significant source of reference code.

Brook has been recently revived by researchers in the critical embedded systems community [Trompouki and Kosmidis 2018], who proposed the use of a subset of the language together with an open source implementation of an OpenGL ES 2 backend, called Brook Auto, in order to address the software certification issues created by the use of GPGPU computing in the automotive domain.

HPG '18, August 10–12, 2018, Vancouver, Canada

ACM ISBN 978-1-4503-5896-5/18/08…$15.00
https://doi.org/10.1145/3231578.3231582

We further enhance their implementation with additional features not implemented in original Brook, while we remove some features which are not appropriate for modern accelerator programming.

3 WHY BROOK INSTEAD OF OPENCL?

The obvious programming choice for accelerators is nowadays OpenCL due to its wide portability. However, for a number of reasons we have decided not to use OpenCL.

First, there are practical limitations that prevent OpenCL to be implemented over graphics APIs for low-end mobile GPUs: despite a previous work [Leskela et al. 2009] claims such an achievement with a limited experimental implementation which was never publicly released, the OpenCL specification [Khronos 2009b], even in its minimum Embedded Profile requires a set of properties that cannot be implemented using OpenGL ES 2 [Khronos 2009a]:

- work groups: the OpenCL programming model organises the computational instances of a kernel in groups. Those groups can share read / write local memory and synchronise their execution through barriers. However, OpenGL ES 2 provides only a single group view of computational instances of a kernel that cannot share any writable memory and does not provide a synchronisation mechanism among them.
- atomics: OpenCL allows the use of atomic operations, however this functionality is not available in OpenGL ES 2.
- scatter output: OpenCL threads can write to any output position, but OpenGL ES 2 allows only a predefined output from each thread executed in the fragment shader.
- multiple outputs: OpenCL kernels can have arbitrary number of inputs and outputs, while OpenGL ES 2 supports only a single output, with up to four 8-bit components.
- IEEE754 floating point compliance: OpenCL implementations should be conformant with the IEEE floating point standard, unless specific compiler optimisations are enabled. Conversely, OpenGL ES 2 hardware and software does not strictly adhere to this standard, for power, performance and transistor count reasons.

In addition to the aforementioned limitations, we believe that OpenCL's programming model is quite low level for the general public, since it requires a significant amount of boilerplate code even for launching a basic kernel, while Brook is as easy to program as CUDA. An overview of the language is provided in its seminal paper [Buck et al. 2004].

Last but not least, Brook is probably the heterogeneous language that supports more devices than any other. Every OpenCL or CUDA capable device supports graphics APIs that are supported by Brook, while Brook offers the advantage of working on top of legacy GPUs without OpenCL/CUDA support. This in conjunction with our OpenGL ES 2 implementation allows Brook to support also the 99% of the devices in mobile market [Khronos 2018].

4 IMPLEMENTATION

We ported Brook on the Raspberry Pi, introducing an OpenGL ES 2 compiler and runtime backend. Both the compiler and the runtime system were ported, in order to enable a completely standalone solution for the Raspberry Pi. Although our code generation and the runtime implementation is optimised for this platform, it is written using only the core OpenGL ES 2 standard [Khronos 2009a], without any vendor specific extensions, in order to ensure maximum portability to any embedded device supporting this standard. Our implementation which is available from [Kosmidis and et al. 2018] has been merged with Brook Auto [Trompouki and Kosmidis 2018], extending it with 2K LOC changed in the compiler and runtime and an additional 4K LOC in regression tests and benchmarks. Next we describe the additions and the dropped features.

Following our baselines, our solution is also based on the proprietary Cg compiler from NVIDIA, which only provides x86 binaries. As one of our primary goals was to enable the development and the deployment of Brook applications directly on the ARM-based Raspberry Pi without requiring any host computer which can increase the cost of our solution, we have employed the emulation of Cg using the qemu-x86 binary translator [Bellard 2005]. The execution time of the emulated compiler is similar to the native compilers on the Raspberry Pi. Note that since we are only using the Cg compiler but not its runtime, no additional emulation takes place when a Brook GLES Pi application is executed and therefore there is no performance penalty at program runtime.

The original Brook only supports floating point (and its vector variants) in kernels [Buck et al. 2004]. We added support for the rest of C data types (char, int) including their signed, unsigned versions and vector extensions up to four components, identical to the ones of CUDA and OpenCL. Due to the limitations described in Section 3, we only support kernels with up to 32 bit output per thread, such as char and vectors of char with up to 4 components, a single integer or a single floating point value.

Iterators are an unusual Brook feature, which is syntactic sugar for creating and initialising streams of indices. However, they do not map to any related concept in modern accelerator or parallel programming languages. In particular, the Brook+ [AMD 2009] (AMD's GPGPU language prior to OpenCL) applications found in the AMD Stream SDK [AMD 2009] which we use in the Results Section, uses the operator `indexof` instead. This is consistent with the use of the thread identifier in modern accelerator programming languages. For those reasons, we decided to drop support for this feature without sacrificing performance or programmability.

Besides iterators, `GatherOp` and `ScatterOp` operators do not have an equivalent in modern accelerator computing languages. `GatherOp` performs indirect reads from a stream, which is also the functionality implemented in array indexing. The reason behind this operator was to allow Brook to be executed also on older graphics hardware without this capability. As a consequence, the operator was implemented on the CPU and therefore resulted in poor performance. Since this functionality, also known as *dependent texture read* is supported in all OpenGL ES 2 compliant hardware, we consider it obsolete and we dropped its support in our backend.

`ScatterOp` emulates read-modify-write accesses in kernels, since GPU fragment shaders lacked this functionality until the appearance of CUDA and OpenCL compatible hardware. For this reason this operator is also implemented in the CPU and yields low performance. Several OpenGL ES 2 compliant GPUs also lack this feature, however, even when supported by the hardware, the scatter parallel programming pattern is discouraged in modern accelerator programming, due to its inefficient memory use [McCool et al. 2012][Kirk and Hwu 2016][Jeffers and Reinders 2013]. As a solution,

the algorithm transformation to the gather pattern is recommended. For this reason, we decided to drop the support for this operator as well, in order to make sure that the end user acquires this skill. It is worth to note that AMD in its Brook+ implementation kept those operators for backwards compatibility reasons, but never provided any example, application or documentation in its SDK [AMD 2009] that made use of it, which confirms our decision.

Finally, Brook supports streams of structures for kernel input and output in order to enhance programmability. However the use of *arrays of structures* (AoS) in accelerators and in particular in modern GPUs and many core accelerators like Intel's Xeon Phi is discouraged for performance reasons, because it limits vectorisation opportunities and leads to underutilisation of memory bandwidth due to inefficient caching and memory coalescing. As an alternative, memory layout changes in the application in order to use *structures of arrays* (SoA) are recommended [McCool et al. 2012][Kirk and Hwu 2016][Jeffers and Reinders 2013].

For this reason, the support for structs in our implementation has been rated as a low priority and it has not been implemented yet, requiring AoS to SoA reorganisation. However, the significant performance difference when AoS is used has an educational value, so it is going to be implemented in the future.

5 RESULTS

Brook GLES Pi passes all the regression tests of Brook code base [Ian Buck et al 2007] for the supported features, including the ones we added for our language extensions. Our evaluation shows that our implementation exhibits the same performance trends for an application executed on the Raspberry Pi using our backend, as the same application executed on state-of-the-art GPU systems. In other words, an application that its GPU performance scales with the input size (provides speedup) on a state-of-the-art system with a GPU, it also scales on the Raspberry Pi's GPU. On the contrary, the applications which provide diminished performance (slowdown) in a desktop system also have the same behaviour on the Raspberry Pi's GPU. This allows to experiment with accelerator programming on a low cost platform. All parallel programming patterns that exist for accelerators can be used also on our system, with the exception of the ones based on scatter as discussed earlier.

We perform our evaluation with AMD's Brook+ SDK Applications [AMD 2009], using the version of the applications released by [Trompouki and Kosmidis 2018], which is compatible with Brook's OpenGL ES 2 backend. We execute Brook GLES Pi on a $25 Raspberry Pi B+, featuring a Broadcom BCM2835 SoC with an ARM CPU and 512 MB of memory, out of which we have configured 256 to be dedicated to the VideoCore IV GPU.

For comparison, we selected two more powerful state-of-the-art (Q4 2017) systems. The first system is based on an AMD RYZEN 7 1800X CPU,with 32 GB of memory and total cost of $2500. It uses an NVIDIA GeForce GTX 1050 Ti with 4 GB of memory and retail price $250. The second system is the latest embedded development platform from NVIDIA for advanced driver assistance systems (ADAS) and autonomous driving, Jetson TX2. It contains 6 ARM CPUs, 8 GB of memory and an NVIDIA Pascal GPU with 256 CUDA cores, with retail price $600.

In both high-end systems the benchmarks are executed using Brook's OpenGL backend on the GPU [Buck et al. 2004], while their CPU version is executed without OpenMP, in order to have a straightforward comparison with our single-core Raspberry Pi system. Note that this setup plays against us, since otherwise the speedups of the state-of-the-art systems would have been lower.

Table 1: Relative performance and data transfer bandwidth of the GPU of the considered platforms, compared to their respective CPU as reported by the flops benchmark.

Platform (GPU/CPU)	Perf. Ratio	Bandwidth Ratio
Raspberry Pi (VideoCore IV vs ARM)	23×	1/33×
NVIDIA GTX 1050 Ti vs AMD	19×	1/11×
NVIDIA Jetson TX2 (Pascal GPU vs ARM)	11×	1/4×

The metric we use is the relative performance (speedup) of each benchmark when it is executed on the GPU, compared to its CPU version. Table 1 shows the relative capabilities of each system's GPU compared to its CPU, using the flops benchmark. According to this benchmark, the Raspberry's Pi GPU has the highest relative performance over its CPU compared to the other platforms (23 times faster), but the lowest relative data transfer ratio to the accelerator.

For each of the benchmarks we show the speedups for each platform while the input size varies up to the maximum supported by the Raspberry Pi texture size, 2048. We notice that our implementation follows the same trend on the Raspberry as the OpenGL backend on the state-of-the-art systems, while the speedup or slowdown is within the same or within an order of magnitude. [Trompouki and Kosmidis 2018] distinguishes the applications in scalable and non-scalable benchmarks. However, in this article we use the terms benchmarks with strong scalability and benchmark with weak scalability, which better describe their behaviour.

The applications (a)-(d) in Figure 1 exhibit weak scalability, therefore their relative performance does not increase much for input sizes up to 2048. In fact, they run slower on the GPU, since they are mainly constrained by the memory bandwidth for the examined range of input sizes. As shown for this class of applications, our Brook implementation on the Raspberry Pi provides the same behaviour of weak scalability with the two state-of-the-art reference systems, obtaining relative performance of the GPU compared to the CPU execution of each system in the same order of magnitude.

In Figure 1, the applications (e)-(j) exhibit a significant increase in their relative GPU performance when the input size is increased. We observe that our solution is able to provide speedups in the same order of magnitude or in two cases (Floyd Warshall and sgemm) within an order of magnitude with the other two systems, for applications with strong scalability, too.

Combining the results of both application categories, we showed that Brook GLES Pi can be used in order to develop and evaluate heterogeneous algorithms on the low-cost Raspberry Pi, observing similar performance trends with state-of-the-art accelerator-based systems which are orders of magnitude more expensive. This means that our solution can be used, for example, in order to reduce significantly the educational costs for an introductory course in accelerator programming.

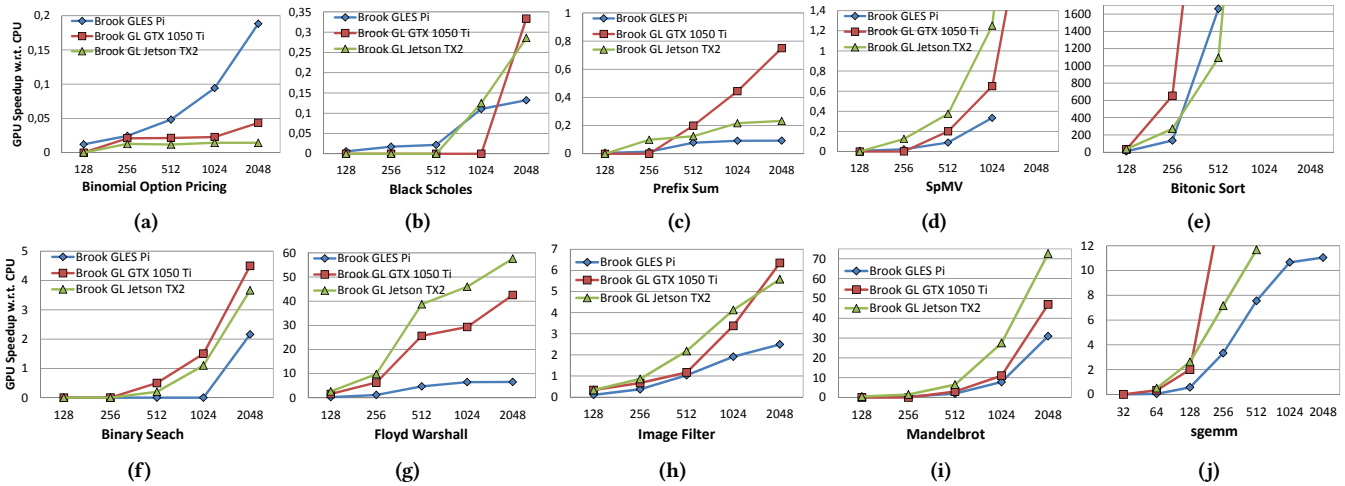

Figure 1: (a)-(d): Benchmarks with Weak Scaling, ie. small performance increase with input size. For the explored sizes, their GPU vs CPU speedup is lower than 1, therefore their CPU version is faster.
(e)-(j): Benchmarks with Strong Scaling. Note the consistent behaviour for both classes across the three platforms (low-end embedded, high-end embedded, high-end desktop), although with variations in magnitude.

6 CONCLUSION

In this paper we presented Brook GLES Pi, an OpenGL ES 2 backend particularly developed for the Raspberry Pi platform, in order to democratise access to accelerator programming using low-cost hardware. Our implementation offers correct functionality, and similar speedup trends with other Brook backends for state-of-the-art systems with significantly higher cost, therefore providing a reliable demonstration of the capabilities of heterogeneous computing.

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.

REFERENCES

AMD. 2009. Brook+ Subversion Repository. (2009). https://sourceforge.net/projects/brookplus/.

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174.

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*. 41–46.

Broadcom. 2013. VideoCore IV 3D Architecture Reference Guide. (2013). https://docs.broadcom.com/docs-and-downloads/docs/support/videocore/VideoCoreIV-AG100-R.pdf.

Roland Brochard. 2011. FreeOCL. (2011). https://github.com/zuzuf/freeocl.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 777–786.

Andrew Holme. 2014. Accelerating Fourier transforms using the GPU. (2014). http://www.raspberrypi.org/blog/accelerating-fourier-transforms-using-the-gpu.

Ian Buck et al. 2007. Brook Subversion Repository. (2007). https://sourceforge.net/projects/brook/.

Jim Jeffers and James Reinders. 2013. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann.

Khronos. 2009a. OpenGL ES Common Profile Spec. V. 2.0. (2009).

Khronos. 2009b. The OpenCL Specification V. 1.0. (2009).

Khronos. 2018. OpenGL ES Overview. (2018). http://www.khronos.org/opengles.

David Kirk and Wen-Mei Hwu. 2016. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

Leonidas Kosmidis and Matina Maria Trompouki et al. 2018. Brook GLES Pi. (2018). http://github.com/lkosmid/brook.

Jyrki Leskela, Jarmo Nikula, and Mika Salmela. 2009. OpenCL Embedded Profile Prototype in Mobile Device. In *2009 IEEE Workshop on Signal Processing Systems*.

Eric Lorimer. 2014. Hacking The GPU For Fun And Profit. (2014). https://rpiplayground.wordpress.com.

Michael McCool, James Reinders, and Arch Robison. 2012. *Structured Parallel Programming: Patterns for Efficient Computation*. Addison-Wesley Professional.

Marcel Müller. 2015. New QPU Assembler. (2015). http://www.raspberrypi.org/blog/new-qpu-macro-assembler.

Matina Maria Trompouki and Leonidas Kosmidis. 2018. Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems. In *Proceedings of the 2018 Design Automation Conference (DAC '18)*.

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