Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems

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

Leonidas Kosmidis
Barcelona Supercomputing Center
leonidas.kosmidis@bsc.es

ABSTRACT

Modern automotive systems require increased performance to implement Advanced Driving Assistance Systems (ADAS). GPU-powered platforms are promising candidates for such computational tasks, however current low-level programming models challenge the accelerator software certification process, while they limit the hardware selection to a fraction of the available platforms. In this paper we present Brook Auto, a high-level programming language for automotive GPU systems which removes these limitations. We describe the challenges and solutions we faced in its implementation, as well as a complete evaluation in terms of performance and productivity, which shows the effectiveness of our method.

1 INTRODUCTION

Modern automotive systems exhibit an increased demand for computational power, in order to implement Advanced Driver Assistance Systems (ADAS), which have become essential in current vehicles. Embedded GPUs can provide the required performance to satisfy this need, due to their massively parallel architecture.

High-end embedded GPUs are programmable using low-level programming models such as CUDA and OpenCL. These programming models are extensions of the C programming language, in order to enhance the programmability of those devices, since this language dominates the software production nowadays. However, this comes at the expense of their software certification against safety standards such as ISO26262 [8], which is imperative in automotive systems. In particular, both CUDA and OpenCL, violate several recommendations found in ISO26262 and other code guidelines for safety critical systems such as MISRA C [3], namely the use of pointers and dynamic memory allocation.

In addition to this, those programming models are only available to high-end embedded devices such as NVIDIA's Drive PX, which have a higher cost and high power consumption, ranging from 10 up to 500W. On the other hand, low-end, low-power embedded GPUs such as the Mali-4xx, the most licensed embedded GPU so far, have been used in the automotive sector for more than a decade, and they are still found in latest safety-certified automotive platforms like Xilinx's Zynq UltraScale+. These low-end GPUs, which support only graphics APIs up to OpenGL ES 2.0 [9], can provide low-cost general purpose algorithm acceleration, using recently introduced GPGPU solutions [16].

In this work, we introduce Brook Auto, an open-source high-level programming language for automotive GPUs [12], which allows the easy certification of GPU accelerated software, while it retains programmer's productivity and execution efficiency. Moreover, Brook Auto is portable across all embedded GPUs found in automotive systems, allowing not only to reduce the software certification cost, but also the overall hardware cost enabling the selection of more cost effective solutions from a wide range of GPU platforms.

Our contributions are the threefold: a) we define a subset of the Brook GPU language, that we call Brook Auto, which is amenable to software certification and portable across every embedded GPU for the automotive domain, b) we demonstrate its compliance with ISO26262 and c) we present an implementation and evaluation of Brook Auto on a low-end embedded platform.

2 SOFTWARE CERTIFICATION AND CUDA/OPENCL

Due to their critical nature, automotive systems need to be certified according to the ISO26262 automotive standard. For this reason, all automotive software needs to comply with a set of rules defined by the standard. Among those rules, we find several rules that are violated by every CUDA and OpenCL program: a) restricted use of pointers, b) no dynamic memory allocation, c) static verification of program properties, d) resilience to faults and e) fault propagation.

By definition, CUDA and OpenCL are based on pointers in order to pass and process data in the GPU. In particular, the user has to maintain both host pointers, pointing to the address space mapped to the CPU, as well as device pointers, pointing to the GPU address space. The data movements between those address spaces have to explicitly managed, ensuring not only the validity of the pointers but also the size of the data to be transferred. This process is error prone, increases the software complexity and it is difficult to maintain, let alone certify.

In addition, the CPU and especially the GPU mapped memory has to be dynamically allocated in an explicit manner. This creates problems to reason about the maximum memory usage. For example, the presence of a memory leak can render the GPU unusable if the GPU memory is exhausted. In a worse scenario, if the allocated memory is precious pinned memory (non-swappable memory on the CPU side), this can jeopardize the stability of the entire system, allowing a fault in one task to affect the whole system (rule e).

The memory allocation problem is a special case of the static verification of program properties in general. Those include maximum stack depth, maximum loop bounds and other limits. Neither language restricts their usage, which

© {ACM} {2018}. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in https://doi.org/10.1145/3195970.3196002

may result in a kernel crash or deadlock, depending on the kernel inputs. In a similar way, emulation for the cases where a kernel resources exceed the available GPU resources, can lead to multiple implicit GPU calls for a single kernel.

Finally, a memory violation in a GPU kernel or memory transfer between the GPU and the CPU, often crashes the driver and requires a system restart, violating both d) and e). Therefore, it is evident that these programming models cannot be used for the highest integrity level (ASIL-D) according to ISO26262.

3 JUSTIFICATION OF THE LANGUAGE SELECTION

Similar problems have been identified in the past on CPU automotive code written in C, which have been addressed by the introduction of programming guidelines and the definition of subsets of the C language such as MISRA C [3]. However, this cannot be applied in the case of CUDA and OpenCL, due to the fact that the above problematic characteristics are indispensable parts of their programming model. Therefore, instead of devising a new GPU language with the desired characteristics to be compliant with ISO26262 and taking the risk that it might not be embraced by end users, we leverage Brook [4], a widely used language for GPUs in the past and a success story proven in practice.

Brook [4] is an open source language developed at Stanford University, introduced during the early stage of the heterogeneous programming era to leverage the computational power of multiple CPUs and GPUs, in a period that GPU computing was only possible using low-level graphics APIs, such as the desktop OpenGL and DirectX.

Brook is the basis of NVIDIA's CUDA programming language, while it has also been employed by their rival at that moment, ATI (currently AMD).The latter further improved Brook for use with their GPU's low level programming interfaces, initially CTM (Close-to-Metal) and later CAL (Compute Abstraction Layer) [11], and they released it also with an open source license as Brook+ [1]. This resulted to an important amount of GPGPU software written in this language, mainly for scientific simulations.

Brook abstracts the programmer from the graphics complexities so that she/he has only to identify the algorithmic part to be offloaded (kernel) and its input/output. This procedure is very similar to CUDA. A short but complete description of the language features can be found in [4].

Brook kernels are written in a restricted subset of C (no recursion, no `goto`, no pointers), general enough to support several backends which might have different types of limitations. The language also features vector extensions similar to OpenCL, making the transition of a programmer from Brook to CUDA/OpenCL and vice versa straightforward.

Finally, Brook is one of the most universally supported languages for heterogeneous computing. Before our implementation, which is described in the Section 5 and extends Brook's support to almost any accelerator device, Brook supported 4 different backends, one targeting CPUs, based on OpenMP and three desktop GPU backends: a) desktop version of OpenGL for Windows and Linux, b) DirectX 9 for Windows and c) Close-to-Metal for AMD graphics cards. Moreover, AMD's Brook+, included a CAL [11] backend for AMD graphics cards, on both Windows and Linux machines, while keeping the front-end compatibility with Brook.

4 BROOK AUTO AND COMPLIANCE WITH ISO26262

By default Brook does not support pointers. Instead it uses *streams* in order to allow the GPU to process data in a kernel. In Brook there is no possibility for a GPU thread to access beyond the allocated GPU memory. Moreover, the correct element is guaranteed to be accessed (the one corresponding to that thread), minimizing the possibility of the programmer to make a mistake. When the array index notation is used to access the GPU memory, even in the case that the programmer may erroneously access beyond the limits of the GPU memory, our implementation over the OpenGL ES 2.0 graphics API ensures that there is no crash, which can impact the availability of the GPU or the entire system. This is because when the texture unit is used for accessing memory, memory violations do not raise exceptions.

Similarly, Brook does not use pointers neither dynamic memory allocation to manage GPU memory and its transfers. In particular, it defines stream handles, which do not allow low level access into the GPU memory. The size of each stream is encapsulated in the object size, therefore it is not possible to have out of bound violations and crashes during data movements from and to the GPU memory. In Brook Auto, we force each stream handle to be statically sized, to allow the static determination of the maximum GPU memory usage. In the same way, in Brook Auto we enforce upperbounds to the loop constructs in the kernels, so that the maximum trip count can be deduced, while the recursion is already forbidden in Brook. In order to avoid the emulation cost on certain GPUs, Brook Auto restricts the number of inputs and outputs to the ones supported by the target platform.

5 IMPLEMENTATION

Next, we describe the implementation of an OpenGL ES 2 backend in Brook Auto, to enable ISO26262 friendly high-level GPGPU programming on automotive GPU platforms.

Our starting point has been the open source subversion repository of Brook [7] and its OpenGL backend. The repository has been converted to git preserving all history and original authors and can be accessed along with our implementation in github [12].

5.1 Compiler

We modified the Brook compiler front-end to enforce our ISO26262 compliant subset, Brook Auto. The original Brook implementation for desktop GPU kernels is based on source-to-source transformations performed by the high-level Cg [13] language compiler from NVIDIA.

Similar to the desktop OpenGL backend, our implementation is based on the Cg compiler. We upgraded the Brook compiler to interface properly with its latest version (3.1.0013) [14] which features a hidden undocumented option to generate OpenGL ES 2 shader (kernel) code (GLSL ES 1.0 [10]).

5.2 Array Indexing and `indexof`

The OpenGL ES 2 standard [9] only allows memory accesses in textures using *normalized coordinates*, values between 0 and 1 that are scaled according to the texture size. However, they were particularly challenging to provide full compatibility with the rest of the backends, especially with the CPU one. That is, the same Brook kernel to be executed in the same way independently of the target device. The source of the problem comes from the fact that Brook allows the programmer to use array indices to access kernel inputs, similarly to C. However, this means that the array indices, which are the texture coordinates in our backend, need to be integer values. In the desktop version, the straightforward option is to use non-normalized coordinates, which are not available in the embedded case. To support this functionality in our Brook Auto backend, we pass the texture dimensions as extra hidden arguments in the kernel invocation, and scale appropriately the indices in the generated code. This way the conversion is completely transparent to the user.

We used the same solution to implement the index of the current element with the `indexof` operator. This Brook operator is equivalent to `threadId` in CUDA, which returns the identifier of the current thread and it is used to access certain elements of the provided input. The only difference in this case is that an implicit texture coordinate is generated in the GLSL ES 1.0 kernel code, which is further transformed as any other user-declared array index.

5.3 Texture Size

Several OpenGL ES 2 implementations support only power of two textures or square only textures. Those cases are automatically detected in our implementation and they are appropriately handled in the allocations, in a transparent to programmer manner. The runtime keeps the sizes of textures internally so that they can be used for correct scaling e.g. for array indexing or in `indexof` implementation. Note that Brook supports both single and multidimensional inputs with up to 4 dimensions, which are again transparently used using a translation scheme, even though the underlying memory is always 2-dimensional as OpenGL ES 2 requires [9].

5.4 Numerical Formats

Regarding the numerical format interoperability, the numerical transformations proposed by [16] have been incorporated in our backend. The implementation of this computationally intensive and performance-critical code has been optimized in both CPU and GPU. The input reconstruction and output encoding GPU part has been optimized with GLSL vector operations [10], to take advantage of the underlying implementation of OpenGL ES 2 GPUs, which are based on a vector microarchitecture in their majority. On the other hand, the CPU code that sets the data in the textures and reads them back is implemented in portable performance-oriented C code. In both cases, we paid special attention to retain portability across different OpenGL ES 2 platforms.

5.5 Reductions

Brook supports kernel reductions over input data, similar to OpenMP, Intel's Threading Building Blocks (TBB) and Cilk Plus. This functionality consists of an iterative application of certain associative operations (e.g. addition) over the input elements in order to reduce the size of the input.

Reductions in Brook are implemented internally as multipass kernels over two intermediate buffer textures. The size of the input is constantly reduced until the output contains the desired number of elements. In order to reduce memory overheads, the same textures are reused for the reduction steps. The fact that the amount of data are reduced from each reduction step does not create any complication in the OpenGL backend, where non-normalized coordinates were used. However, this is not the case for our OpenGL ES 2 backend. The normalized coordinates forced by API limitations imply that their actual value is directly related to the allocated texture size. In order to overcome this problem, we had to keep track internally of the actual data size for reduction operations as well, similar to the solution we applied for array indices and the `indexof` operator.

6 EVALUATION

Without commonly accepted GPU benchmarks (Rodinia [5], Parboil [15] or SHOC [6]) written in Brook and considering that porting any of these suites is beyond the scope of our work, we had to find appropriate alternatives that allow us to evaluate our implementation and show its potential.

For this reason, we use the reference Brook applications included in the Brook+ version, released by AMD [1] in order to show examples of how Brook can be used, as well as to demonstrate the performance advantages compared to CPUs. Some of the applications (SpMV, bitonic sort, binary search, image filtering, mandelbrot, flops) are also present in the original Brook release [7]. However, their Brook+ version is enhanced with additional features also present in the rest of the reference applications, which make them suitable for performance evaluation. Each benchmark is parametrized, so that the size of its input set is configurable as well as the seed of the random generator that is used to generate it, in order to achieve reproducibility. Moreover, a CPU implementation of each algorithm is included in the application, allowing to validate the GPU output against the CPU results. Finally, a time measurement functionality and statistics reporting is integrated in each application, so that the performance of both GPU and CPU is measured and reported, along with the obtained speedup.

The Brook+ provided set of applications is a rich collection of mainly computational intensive algorithms that are amenable to GPU parallelization and which make use of

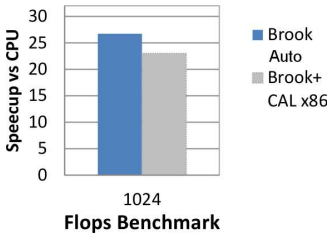


Figure 1: Relative GPU/CPU capabilities between our target platform and a reference x86 platform.

the various language features. Among them we find financial algorithms (Binomial Option Pricing and Black Scholes), matrix operations (Sparse Matrix Vector Multiplication (SpMV) and single precision matrix matrix multiplication (sgemm)), sorting and binary searching, image filtering and fractal generation (mandelbrot), prefix sum and a graph processing algorithm (Floyd Warshall shortest path computation).

When possible, the applications are used unmodified. However, some applications violate our Brook Auto specification. In that case, the application is trivially modified, e.g. by enforcing maximum loop counts, splitting the kernel in as many versions as the outputs and converting vector types to scalar. In all cases, the correctness of the GPU implementation is retained by validating it with the CPU output.

We execute our experiments on an ARM based platform with a VideoCore IV GPU.To facilitate the explanation of the scalability trends for the various benchmarks and compare the effectiveness of our implementation with the desktop GPU backend, we have selected a reference Brook+ capable platform with similar GPU/CPU performance ratio, an Intel Core 2 Duo CPU T9400 equipped with an AMD Mobility Radeon HD 3400 Series GPU.

As we can see in Figure 1, the Flops benchmark which consists of 2 billion floating point operations over 1MB of data, shows that the relative capabilities of the GPUs compared to the respective CPUs in both systems are in the same order of magnitude. In particular in our target system the GPU is 26.7 times faster than the CPU for the computation and the transfer of a very computationally intensive task, while in the reference x86 system, the GPU is 23 times faster. Therefore the scalability trend followed by all applications as well as the order of magnitude of their speedup is expected to be similar in both systems.

In Figures 2 and 3 we present the results for the Brook+ applications for various input sizes. We omit data sizes smaller than 128×128 elements because for such small data sets the CPU is always faster than the GPU, since the latency for transferring the data to the GPU memory dominates the total execution time. The results of interest are indicated with the blue line, while the results for the x86 reference platform are depicted with light grey, since their only purpose is to show the scalability trend of each application in the desktop Brook+ backend implementation, in other words whether it provides a speedup over the CPU and its order of magnitude. Note that the x86 results are obtained without our OpenGL ES 2 modifications and they use AMD's Brook+ runtime, which internally uses CAL [2] (Compute Abstraction Layer, a type of low-level API for AMD GPUs, similar to PTX for NVIDIA's CUDA enabled GPUs) instead of OpenGL, which is more optimized for AMD GPUs. Therefore, it is supposed to be more efficient than our backend.

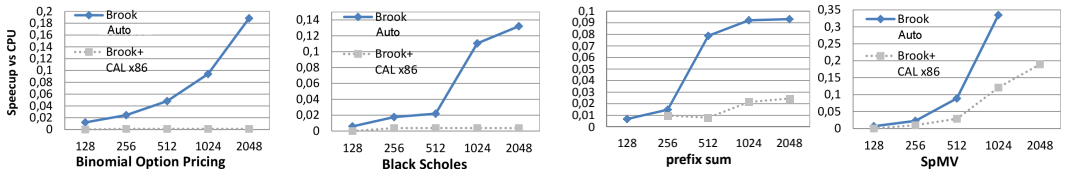
We divide our results in two categories, depending on the scalability of the benchmarks within the explored input sizes.

6.1 Non-scalable GPU Programs

In Figure 2 we see the benchmarks which do not provide a speedup over the CPU. The two financial applications, Binomial Option Pricing and Black Scholes, although they are both quite computationally intensive, they do not provide any performance advantage over the CPU for the considered input sizes. An important reason for this is that the applications have a streaming processing pattern: reading a few input values, do some complex computations and produce the output. In the CPU version, the small inputs and intermediate results between the heavy computations reside in the CPU cache, which turns to be very effective for the CPU implementation compared to the GPU one. Interestingly, the Brook+ version executed on the x86 platform does not show any performance increase with the input size, unlike our Brook Auto implementation. The reason for this difference is that the Brook+ version is highly vectorized, making it compute bound, therefore the kernel performance is saturated even with small input sets. On the other hand, the Brook Auto version on our target platform is scalar and therefore there is a small performance improvement when the input size is increased, although in both cases the GPU version achieves less than 20% performance compared to the CPU. However, if it was possible to provide inputs larger than 2048^2 elements (which is the limitation in our backend, due to the hardware limits of OpenGL ES 2 GPUs), the scalability trend of these applications for our backend shows that larger inputs would provide a benefit over the CPU version, especially in the case of Binomial Option Pricing.

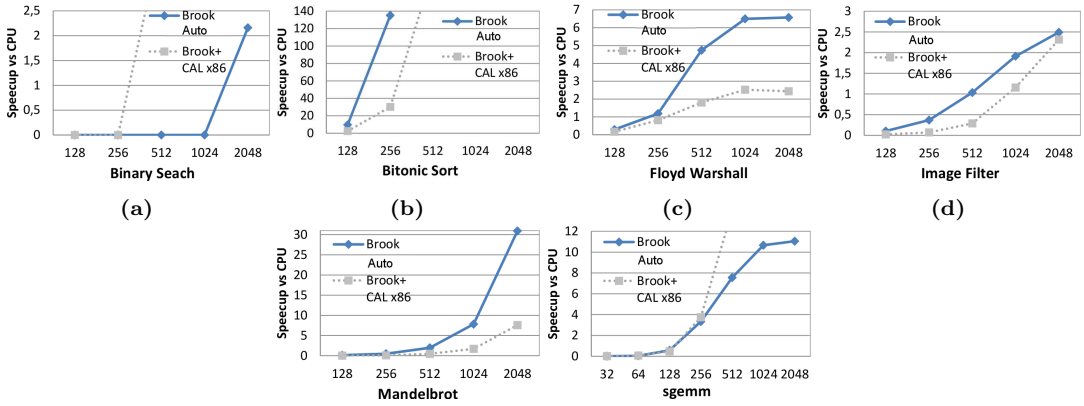
The Prefix sum application in Brook+ is implemented as a multipass kernel invocation with low arithmetic intensity, therefore the data movement dominates the execution time. However the CPU version is extremely efficient in both systems, as it is implemented as a simple accumulation loop.

Finally, SpMV is implemented as a series of 3 small, low arithmetic intensity kernels (O(n)), which means that data transfers are more expensive than the computations for the small sizes we consider. However, the scalability behavior of this benchmark indicates that for larger input sets GPU might be beneficial compared to the CPU. Note that the maximum input value for our implementation is 1024 instead of 2048, since this is the size of each dimension of the compressed matrix, which when it is decompressed, it reaches the maximum texture limit on the OpenGL ES 2 GPUs. In the Brook+ implementation we provide results for 2048^2 elements since in the AMD desktop GPU this limit is 4096.



(a) (b) (c) (d)

Figure 2: Non-scalable GPU programs



(a) (b) (c) (d) (e) (f)

Figure 3: Scalable GPU programs

6.2 Scalable GPU programs

In Figure 3 we present the applications which provide a speedup compared to the CPU for at least some input size within the hardware allowed explored limits. As in the case of the applications examined in the previous section, these benchmarks also follow the same trend with their Brook+ counterparts on Brook Auto. That is, a program that benefits from the GPU and has a speedup over the CPU under x86 with Brook+, also benefits from the mobile GPU in our implementation in Brook Auto and vice versa.

Binary search for input sets up to 1024^2 elements is more efficient on the CPU, however for 2048^2 elements the GPU is 2.16 times faster. This is because from that size and up, the input set does not fit in the L1 cache of the CPU anymore. Moreover, the CPU is able to perform all the 2048^2 searches in parallel compared to the CPU, which mitigates the extra cost of transferring the data between the GPU.

Bitonic sort, which implements a data-independent sorting algorithm, provides an impressive 135× speedup over the CPU for a vector of 256^2 elements. This is because the Brook implementation consists of a kernel called repetitively over the same data without any transfers in between. Note that we only provide speedup results for sizes up to 256^2 elements, because for larger inputs although the GPU finishes fast, the CPU version takes several hours to finish.

The Floyd Warshall algorithm finds the shortest path in a weighted directed graph, by exploring all possible paths between a pair of vertices. Although the kernel has low arithmetic intensity and needed to be split in two – since it produced two outputs – the Brook Auto version achieved increasing speedups for any graph size larger than 256^2 vertices, until it reached a plateau at 6.5× for larger graphs.

The image processing application has relatively low arithmetic intensity since it applies a 3×3 filter to the input image. Therefore the GPU starts to pay-off for image sizes larger than 512×512 pixels reaching a 2.5× speedup.

The Mandelbrot set is another example of a task that the GPU excels, yielding a speedup up to 31×. This kernel, which computes a fractal image, has a significant arithmetic intensity and its value does not depend on input. Therefore, it only transfers the output data from the GPU.

Finally, sgemm also achieves significant speedups up to 11×. Note that in this case, the vectorized x86 Brook+ implementation achieves better scalability than our scalar version for matrix sizes larger than 256×256 elements.

To sum up, the performance of our developed backend based on OpenGL ES 2 is in par with AMD's highly optimized Brook+ CAL-based backend. Moreover, the introduced limitations of our backend due to the hardware limitations of Open GL ES 2 GPUs do not seem to impact the obtained performance.

6.3 Runtime Efficiency and Productivity

So far we have evaluated our Brook Auto runtime over OpenGL ES 2 by comparing the performance of various

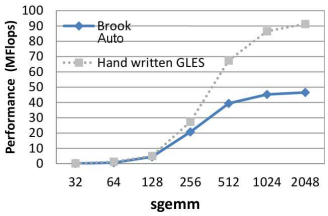


Figure 4: Brook Auto code generation and runtime efficiency vs hand-written OpenGL ES 2.

benchmarks on the CPU and GPU. However, there is no indication about the efficiency of our proposal in terms of code generation and runtime overhead, compared to a hand-written GPGPU application over OpenGL ES 2.

Writing an OpenGL ES 2 GPGPU application by hand is a titanic endeavor. For this reason, we have implemented from scratch only a single application, the commonly used sgemm benchmark. In Figure 4 we can see the performance comparison between the hand-written application and our Brook runtime. Both implementations use a blocked algorithm similar to [17] and the results are reported for the optimal tile size for each version (16×16 for Brook Auto and 8×8 for the hand-written one). We observe that the performance of the sgemm under our Brook Auto backend is between 50 and 90% of the performance exhibited by the handwritten application depending on the input size. This difference in performance comes from the runtime overhead of Brook, and it is consistent with the overhead of the original Brook implementation over the desktop version of OpenGL [4].

However, in terms of complexity and productivity, there is a tremendous difference between the two versions. The Brook version has been written in less than 2 hours and contains 70 lines of code. For comparison, the hand optimized OpenGL ES 2 version has been written and optimized in more than one year and contains 1500 lines of C code.

Therefore, it is evident that our Brook Auto backend achieves a quite high performance on the embedded GPU, while at the same time it offers leaps in GPU programmability.

7 CONCLUSION

In this work we presented Brook Auto, a high-level programming language for automotive GPU-based systems, compliant with ISO26262. We presented an implementation of a portable Brook Auto backend that can be executed in any embedded GPU. The performance of our implementation is fast, delivering speedups over its CPU backend, similar to the one of other Brook backends for desktop GPUs when it is compared with hand-written implementations. Finally, we show that there is a significant reduction in complexity and an immense increase in productivity.

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

[1] AMD. 2009. AMD Brook+ Subversion Repository. (2009). https://sourceforge.net/projects/brookplus/.
[2] AMD. 2009. AMD Stream Computing SDK and Driver. (2009). http://developer.amd.com/wordpress/media/files/atistream_1.4.0_beta-lnx32.tgz.
[3] Motor Industry Software Reliability Association. 2013. *MISRA-C-2012. Guidelines for the Use of the C Language in Critical Systems*. MISRA, Warwickshire.
[4] 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 Transactions on Graphics* 23, 3 (2004), 777–786.
[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*. 44–54.
[6] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3)*. 63–74.
[7] Ian Buck et al. 2007. Brook Subversion Repository. (2007). https://sourceforge.net/projects/brook/.
[8] International Organization for Standardization. 2009. *ISO/DIS 26262. Road Vehicles – Functional Safety.*
[9] Khronos Group. 2009. *OpenGL ES Common Profile Spec. V2.0.*
[10] Khronos Group. 2009. *The OpenGL ES Shading Language V1.0.*
[11] Aaron Lefohn, Mike Houston, Chas Boyd, Kayvon Fatahalian, Tom Forsyth, David Luebke, and John Owens. 2008. Beyond Programmable Shading: Fundamentals, Introduction to the AMD Stream SDK by Mike Houston. In *ACM SIGGRAPH 2008 Classes (SIGGRAPH '08)*. Article 9, 9:1–9:21 pages.
[12] Leonidas Kosmidis, Matina Maria Trompouki et al. 2018. Brook Auto. (2018). http://github.com/lkosmid/brook.
[13] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph.* 22, 3 (2003), 896–907.
[14] NVIDIA. 2012. Cg Toolkit. (2012). http://developer.nvidia.com/object/cg_toolkit.html.
[15] John A. Stratton, Christopher D A Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Liu, and Wen mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign.
[16] Matina Maria Trompouki and Leonidas Kosmidis. 2016. Towards General Purpose Computations on Low-end Mobile GPUs. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE '16)*. 539–542.
[17] Matina Maria Trompouki and Leonidas Kosmidis. 2017. Optimisation Opportunities and Evaluation for GPGPU applications on Low-End Mobile GPUs. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE '17)*. 950–953.