Approximating Memory-bound Applications on Mobile GPUs

Daniel Maier, Nadjib Mammeri, Biagio Cosenza, Ben Juurlink
Technische Universität Berlin, Germany
{daniel.maier,mammeri,cosenza,b.juurlink}@tu-berlin.de

***Abstract*—Approximate computing techniques are often used to improve the performance of applications that can tolerate some amount of impurity in the calculations or data. In the context of embedded and mobile systems, a broad number of applications have exploited approximation techniques to improve performance and overcome the limited capabilities of the hardware. On such systems, even small performance improvements can be sufficient to meet scheduled requirements such as hard real-time deadlines. We study the approximation of memory-bound applications on mobile GPUs using kernel perforation, an approximation technique that exploits the availability of fast GPU local memory to provide high performance with more accurate results. Using this approximation technique, we approximated six applications and evaluated them on two mobile GPU architectures with very different memory layouts: a Qualcomm Adreno 506 and an ARM Mali T860 MP2. Results show that, even when the local memory is not mapped to dedicated fast memory in hardware, kernel perforation is still capable of $1.25\times$ speedup because of improved memory layout and caching effects. Mobile GPUs with local memory show a speedup of up to $1.38\times$.**

***Index Terms*—approximate computing, GPU, kernel perforation**

I. Introduction

Many applications have the property of being *inherently resilient* ([1]): they are still capable of producing acceptable results even if part of the computations is incorrect or approximated. *Inherent resilience* [1] is a property that enables a wide number of code transformations and optimization techniques where accuracy is purposely reduced under an acceptable limit in order to gain better performance or to reduce energy consumption. Research in the area of approximate computing aims at studying and exploiting the gap between the accuracy required by an application and the one provided by a system, offering the possibility to have large performance gain out of minimal accuracy degradation. Recent examples of approximate computing come from image processing [2], machine learning [3], object recognition [4] and graph algorithms [5], but the potential application scenarios are countless.

In the context of mobile and embedded systems, approximation techniques can be very important: on one hand, we have systems with limited resources, which can reach, e.g., interactive performance with the huge performance improvement gained by approximation; similarly, approximation techniques may drastically reduce the power consumption of a program. On the other hand, many typical embedded applications already provide error tolerance to some extent. For example, applications in charge of processing real-time audio or video signals are often required to deal with signals acquired under challenging conditions or large sensor networks based on low-cost, unreliable and inaccurate measuring instruments.

A traditional way to approximate applications is through *loop perforation* [6], which skips iterations of a loop with a static pattern. Recently, dynamic skipping of iterations or instructions in loops has also been explored [7]. Loop perforation has been exploited also on GPUs [6], [8].

However, these works are limited in two aspects. The *acceptable error* is very high, 10% on *average*, unacceptable for many applications. The second problem is related to the use of *local memory*, which can be accessed with very low latency and is shared by all threads in a work group. Exploiting local memory is extremely important to get high performance on GPUs, however, most existing approximated GPU applications do not use local memory. These two limitations have been recently addressed by Maier et al. [9] with *local memory-aware kernel perforation*. Their approximation technique is tailored for GPUs and uses the fast local memory for improving the accuracy of perforation. The technique was inspired by loop perforation that skips loop iterations. In contrast, kernel perforation skips the loading of parts of the input data in order to speed up memory-bound applications. Local memory is used in the perforation phase, to cache samples for different threads, and in a successive reconstruction phase, which further improves the accuracy of the approximation.

Embedded GPUs represent a challenging architecture for kernel perforation. Embedded and desktop GPUs share similarities, e.g., a SIMT-like execution model based on SIMD units for vector processing. However, both computing and memory capabilities are fundamentally different. An example is the presence of dedicated local memory: e.g., ARM Mali does not provide local memory, while a Qualcomm Adreno GPU provides dedicated local memory of 32 kB for each compute unit.

This work aims at using advanced kernel perforation techniques on embedded GPUs. In particular, we focus our analysis on how the availability of dedicated local memory impacts the accuracy and performance of kernel perforation. The contributions of this paper are:

1) a first application of local memory-aware kernel perforation to embedded GPUs;
2) an evaluation study of kernel perforation techniques in relation to the availability of local memory on two embedded GPUs (Qualcomm Adreno 506 and ARM Mali T860 MP2) and six test applications.

The paper is organized as follows: Related work is discussed in Section II. Section III gives an introduction of how the concept of kernel perforation can be used to approximate OpenCL kernels and its application to embedded GPUs. Section IV and V discuss, respectively, the experimental settings and the results of our evaluation on two embedded GPUs and a desktop GPU. Finally, Section VI concludes the paper and gives directions for future work.

II. Related Work

Research of approximate techniques has been conducted from many different perspectives, ranging from hardware [10] and compiler approaches [6], [11], to programming language support [12], [13], [14], [15] and software solutions [6], [8], [2], [7]. In this section, we discuss the most relevant related works in the field of approximated computing. For a more detailed overview we indicate Mittal's survey paper [16].

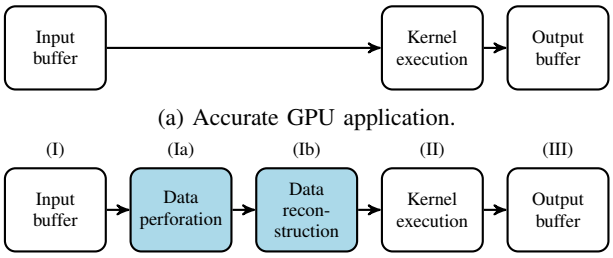
There have been various approaches that are hardware-based. One of these approaches was presented by Lipasti et al. [17] and is called Load Value Prediction, which skips the execution stall due to a cache miss by predicting the value based on locality. However, if the error of a predicted value is too large a rollback is necessary. Load Value Approximation [18] overcomes this limitation by not verifying the predicted values, thus not involving the burden of rollbacks. Yazdanbakhsh et al. [19] presented a similar approach for GPUs that focuses on memory bandwidth, instead of the sole latency. A fraction of cache misses are approximated without any checking for the quality of the predictions. The predictor utilizes value similarity across threads. The programmer must specify which loads can be approximated and which are critical. The fraction to be approximated is used as a knob to control the approximation error. Lal et al. [20] increase the compression rate in memory compression systems by selectively approximating bytes to save extra memory requests and show speedups of up to $1.35\times$.

Compiler approaches have been suggested as well for automatic approximation. Samadi et al. [11] presented SAGE, a framework consisting of (a) a compilation step in which a CUDA kernel is optimized using approximation techniques, and (b) a runtime system that ensures that the target output quality criteria are met. PARAPROX [8] is a framework for transparent and automated approximation of data-parallel applications. Input to the framework is an OpenCL or CUDA kernel, which is parametrized by applying different approximation techniques, depending on the detected data-parallel pattern. A runtime helper is then employed to choose those kernel parameters that meet the specified output quality. For an error budget of 10% they reported an average performance gain of $2.7\times$. Mitra et al. [21] recognized that there are different phases in many applications, each with very different sensitivity to approximation. They presented a framework that detects these phases in applications and searches for specific approximation levels for each of the phases. For an error budget of 5% they report a speedup of 16%. By allowing for an error budget of 20% the speedup increases to 72%.

Several related works have been utilizing software-based approaches for leveraging application's resilience to some amount of error. An analysis of inherent application resilience has been conducted by Chippa et al. [1]. They presented a framework for *Application Resilience Characterization* (ARC) that partitions an application into resilient and sensitive parts, and proposed approximation models to analyze the resilient parts. Lou et al. [2] presented image perforation, a technique specifically designed for accelerating image pipelines. By transforming loops so that they skip certain particular expensive to calculate samples speedups of $2\times$ up to $10\times$ were reported. Subsequent pipeline stages rely on the presence of these samples, and they can be reconstructed using different methods (nearest-neighbor, Gaussian and multi-linear interpolation).

III. Kernel Perforation on Mobile GPUs

This paper evaluates the impact of state-of-the-art approximation techniques on embedded GPUs. We focus on an approach specifically designed to target GPUs [9]. In this section, we provide an overview of the approach (Section III-A), describing how kernel perforation is performed (Section III-B) and which approximation schemes (Section III-C) are used, and presenting how the techniques can be tailored for embedded GPUs (Section III-D).



(a) Accurate GPU application.

(b) Local memory-aware kernel perforation.

Fig. 1: Accurate GPU application and local memory-aware kernel perforation [9] approach.

A. Overview

In typical GPU applications, as depicted in Figure 1a, a GPU kernel first fetches data from the input buffer in global memory (I), then it performs its computations (II), and finally it writes the result to the output buffer in global memory (III). The latency for accessing the global memory is in general very high, although it can be hidden to some extent by the massively parallel architecture of the GPU and its scheduler. A way to improve the performance of GPU kernels is to make use of fast local memory, whose access latency is significantly smaller than the one for global memory.

Maier et al. [9] proposed a novel way to perform kernel perforation where the fast local memory is exploited for more accurate approximation. Figure 1b shows their approach that is also used in this paper. Kernel perforation extends the accurate application with two additional steps: a data perforation phase (Ia) that fetches a part of the input data; a data reconstruction

phase (Ib) that reconstructs the missing data and works on local memory.

B. Kernel Perforation

Loop perforation [6] is an approximation technique that improves the performance of a loop execution by skipping some iterations. Loop perforation has been originally applied to sequential code and can be easily parametrized through tunable loops in order to trade accuracy for performance.

Maier et al. [9] discuss how perforation can be implemented on GPUs by taking care of memory accesses, and introduce the concepts of *input approximation* opposed to *output approximation* for kernel perforation. The general idea is that many applications are inherently resilient to the input as well as the output, i.e., they can tolerate small errors, but because of the long latency of memory access on GPUs, an approximation of the input may take advantage of low-latency local memory to improve the approximation. Furthermore, fast local memory is used for reconstruction of the not loaded data in order to minimize the error.

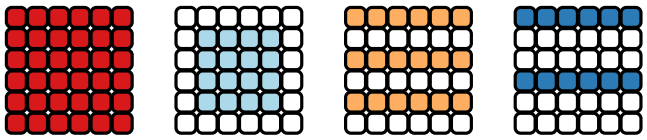
Input approximation works by skipping the loading of some of the input data. If the input data is two-dimensional, e.g., an image, a possible input perforation scheme may skip every other row. In general, input approximation can be a suitable acceleration technique for any application that (a) processes data with redundancy and (b) is resilient to some amount of error in its input data. This is an advantage over output approximation techniques that require spatial locality in the *output* data set. Although it has been shown that output approximation can be used for many applications, this is a conceptual limitation.

The usage of local memory to prefetch data from global memory is a well-known technique to accelerate GPU kernels. Applications' execution time usually benefits from the usage of local memory if there is enough reuse of data, i.e., data loaded by a thread is also re-used by the same or other threads, who in turn also load data.

In the OpenCL programming model, this is implemented using local memory, which is shared among all threads in a work group. On GPUs, the latency of local memory is far lower compared to the latency of accessing the global memory, but its size is limited. Therefore, local memory is used to implement the steps of (Ia) data perforation and of (Ib) data reconstruction, as shown in Figure 1b.

C. Perforation Schemes

The perforation schemes determine which parts of the input data are loaded from memory and which parts are approximated. Figure 2 shows three perforation schemes used in this work. Each cell represents a value (e.g., a float) in the input data. Colored cells represent data that is loaded from memory and white cells represent data that is approximated. The cells map to threads in a work group and the local memory used by the threads. In a typical OpenCL program, each thread might load one data element to local memory and the other threads reuse the same data. When designing the perforation schemes the GPU's memory architecture needs to be taken into account. Especially the cache organization is important to avoid loading data into the cache and then not taking advantage of the data, because it is actually perforated and approximated.



(a) Accurate (b) Stencil (c) Rows1 (d) Rows2

Fig. 2: 2D Perforation schemes for threads in a local group.

In this work we have used three different perforation schemes depicted in Figure 2. In Figure 2 (a) we show the accurate scheme where no data is approximated (red). Figure 2 (b) is a scheme that perforates the boundaries of a tile. It can be used to approximate stencil applications where the data elements on the edges need to be fetched additionally while they have only a very small impact on the result. However, the perforation scheme is application-specific and therefore cannot be used with all applications. The schemes shown in Figure 2 (c) and (d), by contrast, are generally applicable. They perforate every other row and two out of three rows, respectively. The schemes align very well with the memory architecture of GPUs, as they take into account the cache organization.

The parts of the input data that were not loaded from memory need to be reconstructed. We use nearest-neighbor-approximation to reconstruct the missing data, as our main target is speedup and more sophisticated reconstruction techniques also affect the performance.

D. Application to Embedded GPUs

It is important to remark that while the OpenCL programming model exposes local memory (software) and a series of functions to understand type and size, this does not necessarily mean that it is always mapped into fast (hardware) dedicated local memory. This is generally the case for desktop GPUs, which typically expose as local memory their internal physically shared memory of 16 kB or 32 kB.

However, this may be very different on embedded GPUs. In Table I, we report the three devices used in this paper (two embedded and a desktop for comparison). All devices report 32 kB of local memory. For the Adreno GPU and the AMD GPU the memory type is local; however, for the Mali GPU the type is global. This indicates that there is no dedicated, local memory for Mali.

IV. Experimental Evaluation

In our evaluation, we compare the performance of two different kernel perforation techniques with different parameters on two mobile systems and one desktop system. We apply the kernel perforation approach to benchmark applications and measure performance and error. As the applications perform

TABLE I: HARDWARE PLATFORMS DETAILS.

	Qualcomm Adreno 506	ARM Mali 860 MP2	AMD FirePro W5100
Class	mobile	mobile	desktop
OpenCL version	2.0	1.2	1.2
Global memory size	1.39 GB	3.72 GB	3.5 GB
Unified memory	yes	yes	no
Local memory type	local	global	local
Local memory size	32 kB	32 kB	32 kB

the same approximations on different platforms the performance differs while the error introduced by the approximation is identical on all platforms.

A. Experimental Platforms

There are three important mobile GPU vendors: Imagination Technologies (PowerVR), Qualcomm (Adreno) and ARM (Mali). While all of them support OpenCL in a way or another, OpenCL support is not generally built into mobile operating systems. In our study, we compare an Adreno GPU and a Mali GPU with a desktop GPU, with a focus on the results on mobile platforms equipped with and without local memory.

Adreno is a mobile GPU from Qualcomm that was originally developed by ATI. It is exclusively integrated within Qualcomm's Snapdragon SOC family. Our test device for the Adreno platform is a Qualcomm MSM8953 Snapdragon 625 with an Adreno 506 GPU. It is equipped with 3 GB of memory. The device is running Android 7.1.2 and Linux 3.18.31. This mobile GPU has 32 kB of dedicated local memory.

ARM's own GPU available for licensing with ARM cores is Mali. Mali is integrated by many SOC manufactures, e.g., MediaTek and Samsung. Our device for the Mali platform is a MediaTek Helio P10. This mobile GPU has no dedicated local memory, even though it reports being equipped with 32 kB of local memory through the OpenCL API.

Both mobile platforms are equipped with a SOC that integrates both CPU and GPU. The mobile GPUs do not have dedicated memory but share the same memory with the CPU. While the shared global memory comes with all negative effects of shared memory (e.g., lower bandwidth) it can also provide the advantage of no necessity to transfer data between the host's memory and the GPU's global memory. The amount of main memory that the GPU can use may be different from the available amount of memory. In our case the Adreno platform is equipped with 3 GB of memory, while OpenCL reports only less than 1.5 GB global memory. For the Mali platform OpenCL reports nearly the full main memory.

For OpenCL, we rely on the operating systems support for OpenCL drivers. While OpenCL is not officially supported by the Android operating system, there are devices available that have the relevant libraries and drivers included. This enables us to run OpenCL programs in a real portable way without the usage of any Android-specific software layers. We employ the same source code for our applications on all three platforms. However, the quality of the software stack is unclear. Some of our applications failed to execute due to driver issues and this led to missing data in our results.

We measure two values for each experiment: execution time of the kernel using the OpenCL API and the error of the approximated kernel with respect to an accurate kernel. To ensure stable results and to minimize any effects from the operating system, voltage scaling, etc., we repeated each experiment 100 times with the first 50 as a warm-up time. We use the average execution time of the kernel measured using the OpenCL API of the last 50 executions. Moreover, we disabled Dynamic Voltage and Frequency Scaling (DVFS).

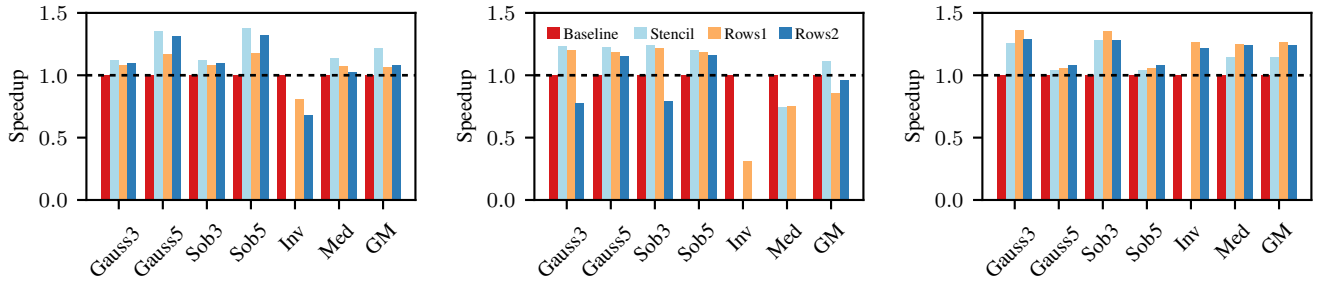
B. Benchmark Applications

Our benchmark set consists of medical, signal processing and image processing applications. Table II gives an overview of the applications. We evaluate the execution time and the accuracy when approximating the applications. We calculate the speedup using the execution time of the accurate application with optimal work group size as baseline. The error is measured by first generating the true result as reference using the accurate application. Then we calculate the error of the result of approximated applications. We use the mean relative error (MRE) as a metric for the error, except for the SOBEL application. In this case the mean error (ME) is a more suitable metric, as the MRE is undefined when the true value is zero.

TABLE II: APPLICATIONS USED IN THE EVALUATION.

Application	Domain	Error Metric
GAUSSIAN	Signal processing	Mean relative error
MEDIAN	Medical imaging	Mean relative error
INVERSION	Image processing	Mean relative error
SOBEL	Image processing	Mean error

The GAUSSIAN filter is a well-known low-pass filter, and has applications in electronics and signal processing. The GAUSSIAN has data-reuse across threads and, therefore, benefits from the use of local memory in general. The filter kernel width is either 3 or 5. The MEDIAN filter is a nonlinear spatial filter with applications in medical imaging and image processing. The filter replaces each sample by the median of adjacent samples. Our optimized implementation uses private memory, i.e., registers, to load all samples in the current filter kernel and compute the median. The filter kernel size is 3×3. The INVERSION filter is an application that maps each input value to its inverse. The filter has a kernel size width of 1 and, therefore, there is no data reuse across threads. The SOBEL operator is used in edge detection applications. We use two versions: SOBEL3 with a filter kernel size of 3×3 and SOBEL5 using a 5×5 kernel size. Previous work [9] has shown that the error depends on the input data and can vary a lot depending on both application and input data. In all our tests we use input data that yields an approximate average error for the application. Input data dimensions are 512×512. All applications operate on single precision floating-point data.



(a) Qualcomm Adreno 506. (b) ARM Mali T860 MP2. (c) AMD FirePro W5100.

Fig. 3: Comparison of speedup with respect to applications.

V. Results

In our experimental evaluation we show the performance and the accuracy on three platforms. We analyze in particular how the perforation schemes perform on the different platforms with a consideration of the availability of dedicated local memory. First, we examine the optimal execution times on each platform. Then, we have a detailed look on the performance of the applications on each platform. We compare the performance of each approximated application across the three platforms. Finally, we show the error introduced by the different perforation schemes.

A. Performance on Three Platforms

In general, it can be noted that the technique can speed up applications on all three platforms. In Figure 3 we compare the three perforation schemes for six applications. Furthermore, we show the geometric mean of the speedups (GM).

1) Qualcomm Adreno 506: The speedup for Qualcomm's GPU is between 7% and 37%, except for the Inversion application where there is actually a slowdown of 20%. Applications with a larger filter kernel like Gaussian5 and Sobel5 benefit more from approximation. The stencil perforation scheme (light blue) always yields the largest speedup. The Rows2 perforation scheme performs the second best, except for the Median application. The approximated kernels of the Inversion application are slowed down. The geometric mean (GM) of all perforation schemes is positive.

2) ARM Mali T860 MP2: The ARM GPU has no dedicated local memory, still kernel perforation is able to accelerate the execution on four out of six applications between 15% and 24%. The highest speedup can be always observed for the Stencil scheme, followed by the Rows1 scheme and Rows2 scheme. The Rows2 scheme is not beneficial for Gaussian3 and Sobel3 application while it is for Gaussian5 and Sobel5 which have a larger filter kernel size. The geometric mean of the speedup for the stencil scheme is positive while it is negative for the Rows1 and Rows2 scheme.

For the Inversion application—that has no actual algorithmic data reuse—we can see the effect of the missing dedicated local memory, as all accesses to local memory are in fact not local but global and therefore the overall latency is increased. A similar situation can be observed for the Median application that is implemented using a highly optimized algorithm using private memory.

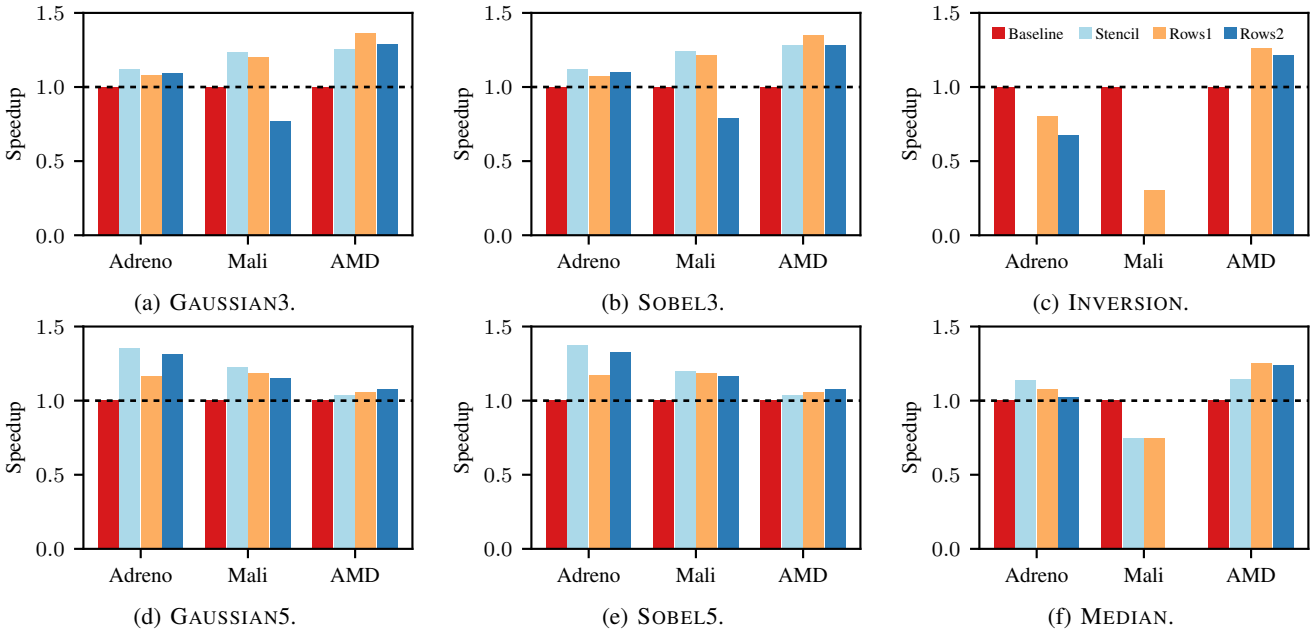
3) AMD FirePro W5100: The acceleration achieved with the desktop AMD GPU is between 3% and 36%. None of the applications is actually slowed down by the approximation. Gaussian3 and Sobel3 show the largest speedups of 25% to 36% while the applications with a larger kernel size (Gaussian5/Sobel5) only show a speedup of 3% to 7%. For most applications, the Rows1 scheme leads to the largest speedup, except for applications with larger filter kernel size. The Rows2 scheme is less beneficial.

To conclude this section we would like to point out that the Stencil perforation scheme, while yielding always the lowest speedup on the desktop GPU, is able to always produce the highest speedup on both mobile applications. This observation is contrary to the results shown by the desktop GPU. In our experiments we learned that synchronization of threads (in a work group) is more expensive on mobile platforms. This explains in part these results. Furthermore, applications with a larger filter kernel size benefit more from approximation on mobile GPUs than on the desktop GPU. The difference in between Gaussian3 and Gaussian5 on AMD is 16%, while it is 227% on the Qualcomm GPU and 182% for the ARM GPU. Applications with smaller filter kernel size (Gaussian3/Sobel3) benefit more from approximation when there is no dedicated local memory.

B. Application Performance on Different Platforms

In Figure 4 we compare the performance per application on the different platforms in order to highlight application-specific differences of the perforation schemes. For the Gaussian3 and Sobel3 applications, all perforation schemes on all platforms, except for Rows2 on the Mali GPU, are able to accelerate the execution of the kernel. The speedup on the desktop GPU is larger than on the mobile platforms: Adreno is accelerated by 7% to 12% and Mali is 19% to 22% faster. The higher speedup of Mali can be explained by the improved data locality.

The situation for the Gaussian5 and Sobel5 is quite different. All perforation schemes are able to yield a speedup on both desktop and mobile GPUs. However, the speedup on mobile platforms is larger compared to the desktop GPU. The speedups on the desktop GPU are between 3% and 7%. The speedups on the mobile GPUs are larger and between 16% and 35% for the Adreno GPU and between 15% and 22% for the Mali GPU. This observation can be explained by the



(a) GAUSSIAN3. (b) SOBEL3. (c) INVERSION.

(d) GAUSSIAN5. (e) SOBEL5. (f) MEDIAN.

Fig. 4: Comparison of speedup with respect to platform.

much larger memory bandwidth of desktop GPUs. Therefore, mobile applications benefit more from a reduced number of memory accesses.

The INVERSION application is accelerated by the AMD GPU and slowed down by both mobile GPUs. The stencil scheme is not applicable here. The execution failed due to a driver issue for the Rows2 perforation scheme on Mali. The slowdown can be attributed to a comparable large number of synchronization operations which are in particular expensive.

The MEDIAN application is already highly optimized using private memory. Still, the application can be accelerated by approximation on Adreno and AMD. However, for Mali, a slowdown can be observed, that can be explained by the absence of dedicated local memory.

C. Execution Times

We list the kernel execution times for the optimal work group configuration for accurate and approximated applications in Table III on the different platforms. As the three platforms are different, executions times are not directly comparable. While the desktop GPU's execution time is around 30-60 μs, the mobile GPU's execution time is much larger: the Adreno GPU takes up to ~6 ms and the Mali GPU takes up to ~14 ms to execute the kernel.

D. Analysis of the Error

While the speedup of the different perforation schemes is different across the different architectures the error introduced by the approximation is—for the same application and input data—almost platform agnostic. We show a representative error for all applications and perforation schemes in Figure 5. The approximation techniques are in general sensitive to the input data. Different types of input data lead to a different accuracy. Maier et al. [9] provide more detailed insights.

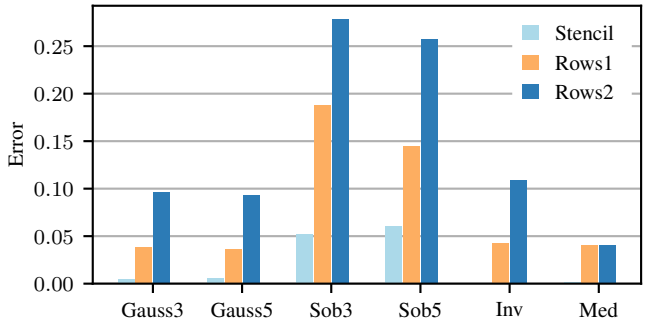


Fig. 5: Error with respect to application and approximation.

The stencil scheme always yields the best accuracy. However, the stencil scheme is not applicable to the INVERSION application. The error is less than 1% for GAUSSIAN3, GAUSSIAN5 and MEDIAN. For SOBEL3 and SOBEL5 it is around 5%, whereat these two applications are very sensitive to approximation. The Rows1 scheme introduces a larger error between 4% and more than 5% for SOBEL3. With the Rows2 scheme the accuracy degrades further for all applications: Less than 10% error for most applications except an error of more than 20% for SOBEL3/SOBEL5.

E. Summary

For the mobile applications, the best performance is always achieved using the Stencil scheme. This is advantageous as this perforation scheme also introduces the smallest error (almost always 5% or less). In cases where the Stencil scheme is not applicable, an alternative scheme can be employed. However, for the INVERSION application, other schemes have not been proven useful in terms of speedup.

The achieved speedup on mobile platforms is generally smaller than the speedup on the desktop platform. However,

TABLE III: RUNTIME (1/100 S) FOR OPTIMAL CONFIGURATION ON DIFFERENT PLATFORMS.

	Qualcomm Adreno 506				ARM Mali T860 MP2				AMD FirePro W5100			
	Baseline	Stencil	Rows1	Rows2	Baseline	Stencil	Rows1	Rows2	Baseline	Stencil	Rows1	Rows2
GAUSSIAN3	2.707	2.415	2.515	2.471	7.556	6.133	6.300	9.802	0.049	0.039	0.036	0.038
GAUSSIAN5	6.165	4.560	5.288	4.703	13.801	11.250	11.659	11.963	0.057	0.055	0.054	0.053
SOBEL3	2.711	2.419	2.522	2.471	7.594	6.113	6.256	9.616	0.050	0.039	0.037	0.039
SOBEL5	6.177	4.488	5.275	4.670	13.751	11.468	11.621	11.832	0.057	0.055	0.054	0.053
INVERSION	0.487	—[1]	0.605	0.721	0.445	—[1]	1.460	—[2]	0.027	—[1]	0.021	0.022
MEDIAN	3.027	2.668	2.821	2.953	7.013	9.427	9.395	—[2]	0.062	0.054	0.049	0.050

[1] Perforation scheme not applicable. [2] Kernel execution not successful.

the speedup achieved by our approach can make the difference between meeting a real-time deadline and missing it.

VI. CONCLUSION

Local memory-aware kernel perforation is a novel approximation technique tailored for GPUs that uses the fast local memory for improving the accuracy of the approximated OpenCL kernels. The technique exploits local memory in two ways: in a perforation phase, to cache perforated data for different threads; in a successive reconstruction phase, which further improves the accuracy of the approximation.

We present the first implementation and evaluation of local memory-aware kernel perforation on embedded GPUs. While the technique makes explicit use of OpenCL local memory, some embedded GPUs (e.g., ARM Mali and Imagination Technologies PowerVR) do not map local memory to dedicated hardware memory. To analyze the impact of dedicated local memory, we study the aforementioned approximation techniques on two embedded GPUs with very different memory layouts: Qualcomm Adreno 506 (32 kB of local memory) and ARM Mali T860 (no dedicated local memory).

Results show that, even when the OpenCL local memory is not mapped on a dedicated fast memory in hardware, kernel perforation is still capable of accelerating memory-bound applications. In many cases even a small speedup is enough to ensure real-time applications to meet their deadlines.

Future work can investigate the impact of kernel perforation on energy and power consumption. Furthermore, detailed insights in the latency introduced by reconstruction can be guiding the design of new perforation schemes and reconstruction techniques. Compiler-based approximations can enable a wide application of the approach with less application-specific knowledge.

REFERENCES

[1] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and Characterization of Inherent Application Resilience for Approximate Computing," in *Design Automation Conference*, ser. DAC. ACM/EDAC/IEEE, 2013.

[2] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes, "Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 5, 2016.

[3] S. Venkataramani, A. Raghunathan, J. Liu, and M. Shoaib, "Scalable-effort Classifiers for Energy-efficient Machine Learning," in *Proc. of the 52nd Annual Design Automation Conf.*, ser. DAC. ACM, 2015.

[4] T. W. Chin, C. L. Yu, M. Halpern, H. Genc, S. L. Tsao, and V. J. Reddi, "Domain Specific Approximation for Object Detection," *IEEE Micro*, vol. PP, no. 99, 2018.

[5] H. Omar, M. Ahmad, and O. Khan, "GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms," in *2017 IEEE 35th International Conference on Computer Design (ICCD)*. IEEE, 2017.

[6] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs With Loop Perforation," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE. ACM, 2011.

[7] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018.

[8] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of the 19th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS. ACM, 2014.

[9] D. Maier, B. Cosenza, and B. Juurlink, "Local Memory-Aware Kernel Perforation," in *International Symposium on Code Generation and Optimization (CGO)*, ser. CGO. ACM, 2018.

[10] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A Low Latency Generic Accuracy Configurable Adder," in *Design Automation Conference*, ser. DAC. ACM/EDAC/IEEE, 2015.

[11] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning Approximation for Graphics Engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. ACM, 2013.

[12] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-power Computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011.

[13] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi *et al.*, "Axilog: Language Support for Approximate Hardware Design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015.

[14] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014.

[15] M. Kambadur and M. A. Kim, "NRG-loops: Adjusting Power from Within Applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO. ACM, 2016.

[16] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys (CSYR)*, vol. 48, no. 4, 2016.

[17] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," *ACM SIGPLAN Notices*, vol. 31, no. 9, 1996.

[18] J. S. Miguel, M. Badr, and N. E. Jerger, "Load Value Approximation," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO. IEEE, 2014.

[19] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "Rfvp: Rollback-free value prediction with safe-to-approximate loads," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, 2016.

[20] S. Lal, J. Lucas, and B. Juurlink, "Slc: Memory access granularity aware selective lossy compression for gpus," 2019.

[21] S. Mitra, M. K. Gupta, S. Misailovic, and S. Bagchi, "Phase-aware Optimization in Approximate Computing," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO. IEEE, 2017.