Local Memory-Aware Kernel Perforation

Daniel Maier
daniel.maier@tu-berlin.de
Technische Universität Berlin
Germany

Biagio Cosenza
cosenza@tu-berlin.de
Technische Universität Berlin
Germany

Ben Juurlink
b.juurlink@tu-berlin.de
Technische Universität Berlin
Germany

Abstract

Many applications provide inherent resilience to some amount of error and can potentially trade accuracy for performance by using approximate computing. Applications running on GPUs often use local memory to minimize the number of global memory accesses and to speed up execution. Local memory can also be very useful to improve the way approximate computation is performed, e.g., by improving the quality of approximation with data reconstruction techniques. This paper introduces local memory-aware perforation techniques specifically designed for the acceleration and approximation of GPU kernels. We propose a local memory-aware kernel perforation technique that first skips the loading of parts of the input data from global memory, and later uses reconstruction techniques on local memory to reach higher accuracy while having performance similar to state-of-the-art techniques. Experiments show that our approach is able to accelerate the execution of a variety of applications from 1.6× to 3× while introducing an average error of 6%, which is much smaller than that of other approaches. Results further show how much the error depends on the input data and application scenario, the impact of local memory tuning and different parameter configurations.

CCS Concepts • **Computing methodologies** → *Graphics processors*; • **Software and its engineering** → *Software notations and tools*;

Keywords approximate computing, GPU, kernel perforation

ACM Reference Format:
Daniel Maier, Biagio Cosenza, and Ben Juurlink. 2018. Local Memory-Aware Kernel Perforation. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3168814

CGO'18, February 24–28, 2018, Vienna, Austria
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
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 *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18)*, https://doi.org/10.1145/3168814.

1 Introduction

Approximate Computing (AC) exploits the gap between the accuracy provided by a system and the accuracy required by an application. Many applications are resistant to some amount of error and earlier works in this field have proven that there is potential for significant improvements in terms of execution time or energy consumption if a small amount of error can be accepted [3]. The rationale behind AC is that an application can provide acceptable output quality even though the system executing the application was inexact in some way. This property is shared by applications from many domains, including signal processing, machine learning, audio and video processing [20, 21]. Research of approximate techniques has also been conducted from many different perspectives: related work ranges from software-based approaches [10, 15, 19] and programming language support [7, 12, 17, 24] to compiler-based approaches [16, 19] and hardware-based techniques [18].

Applications suited for acceleration by AC provide *inherent application resilience* [3], i.e., they can produce acceptable results despite some of their underlying computations being incorrect or approximate. For example, in a photo, pixels that are in an adjacent location potentially have similar values. This property is well-known and exploited by many applications, e.g., by image/video and data compression. Applications in the context of digital audio signals created by sampling continuous analog signals already introduce some amount of error because of quantization noise. This and similar contexts, therefore, are already required to deal with some amount of error.

As one of the goals of AC is to improve performance, several implementations have combined the high-throughput of massively parallel architectures such as GPUs with approximate computing techniques [5, 8, 15, 16, 23]. The GPU is a very interesting architecture for the application of such techniques. GPU programming models expose different types of memory, which are explicitly selected by the programmer. On the one hand, there is a large amount of global memory[1]. When accessing global memory, a fairly high latency has to be accepted. While this latency can be hidden partly by scheduling a different batch of threads, in fact many applications are memory bound. On the other hand, there is a small amount of local memory, which is accessed with very low

[1]Throughout this paper we use OpenCL terminology. In CUDA terminology *work groups* are also known as *thread blocks*, and *local memory* is also known as *shared memory.*

latency. Additionally, local memory is shared by all threads in a work group.

In previous work, GPU applications have been accelerated by perforating the execution of loops [15, 19]. However, these works are limited in two aspects. First, the *acceptable error* is chosen to be 10% on *average*. That is unacceptable for many applications that cannot tolerate such high amount of error. Therefore, the potential impact of such techniques is limited. Second, most of the used benchmark applications do not use local memory. Among the few applications that actually do use local memory, there is no exploitation of local memory for the approximation technique. This means that existing approximation techniques do not take the model of the specificity of the GPU memory model into account.

This paper presents a novel approach to perform loop perforation on GPUs, namely *kernel perforation*, which is aware of the GPU memory hierarchy and makes use of the fast local memory in order to achieve higher accuracy with the same performance. The central idea of our approach is to save memory accesses by approximating reads from the input buffers of the kernel. Being aware of the GPU memory model, our approach focuses on (1) loading only few data points from (the slower) global memory while (2) using the (faster) local memory to cache adjacent data points and (3) improve accuracy with a local data reconstruction technique. Local reconstruction techniques attempt to reconstruct the input value out of a sparse set of globally-fetched data points, thus exploiting *inherent application resilience* of the applications. The approach also extends existing work on perforation also with novel memory patterns and an analysis of the accuracy on six applications with different input data.

The contributions of this paper are:

- a novel local memory-aware approximation approach for OpenCL kernels based on loop perforation (*kernel perforation*) that approximates the input data of GPU applications by reducing the amount of data loaded from global memory and reconstructing a high-accuracy approximation with a local reconstruction technique;
- a set of *local* reconstruction techniques that work on local memory and efficiently combine the sparse data fetched by *global* perforation schemes while consistently improving the accuracy of the approximation;
- experimental results on six benchmarks with different input data and different combination of parameters (perforation schemes, reconstruction techniques, local work-group size), where we show speedups of 1.6× to 3× while maintaining a moderate amount of error on an AMD FirePro W5100 GPU.

2 Related Work

Approximate computing has become a hot topic with applications in many different fields and different approaches [4, 6, 8, 11, 24]. A thorough overview can be found in the survey paper of Mittal [14].

Lipasti et al. [9] presented a hardware-based approach 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 [11] overcomes this limitation by not verifying the predicted values, thus not involving the burden of rollbacks.

Yazdanbakhsh et al. [25] presented a similar approach for GPUs that focuses on memory bandwidth, instead of the sole latency. A fraction of cache misses is 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.

Several related works use 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. [3]. 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.

Loop perforation has been presented by Sidiroglou et al. [19]. While many applications are designed to trade-off accuracy for performance using domain-specific techniques, loop perforation is a generalized approximation technique that can be applied in a variety of contexts. Once critical (tunable) loops are identified, the number of iterations of such loops can be decreased. By limiting the amount of error of the final result to 10% at maximum, a typical speedup of 2× can be achieved.

Li et al. [8] introduced a GPU-specific approach based on the *Special Function Unit* (SFU) commonly available in, e.g., NVIDIA GPUs, which provides acceleration for transcendental functions. A tunable approximation is achieved by dividing the work into warps that execute the accurate functions and warps that execute the SFU-based approximate functions.

Samadi et al. [16] presented SAGE, a framework consisting of a compilation step in which a kernel is optimized using approximation techniques and a runtime system that ensures that the target output quality criteria are met. They employed three GPU-specific optimization techniques: discarding of atomic operations, data packing/compression, and thread fusion. Even though SAGE takes into account GPU-specific limitations, it does not exploit the compute unit's local memory to benefit from the low latency and shared memory.

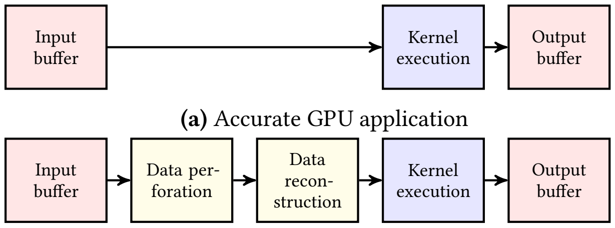
PARAPROX [15] is a framework for transparent and automated approximation of data-parallel applications. Input to the framework is an OpenCL or CUDA kernel, which is parametrized by applying different approximation techniques, depending on the detected data-parallel pattern. A runtime helper is used to 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×.

Mitra et al. [13] 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% on average.

Lou et al. [10] presented image perforation, a technique specifically designed for accelerating image pipelines. By transforming loops so that they skip certain samples that are particular expensive to calculate speedups of 2× up to 10× 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). The pipeline can be modified by a storage optimization that replaces accesses to skipped samples with on-demand reconstruction code.

3 Overview

This paper introduces a novel approximation technique that is specifically designed to approximate general-purpose GPU kernels. The proposed approach extends state-of-art approximation techniques such as the row-/column-based schemes used in Paraprox [15] by exploiting the GPU's fast local memory to deliver more accurate solutions.



(a) Accurate GPU application

(b) Our approach

Figure 1. Accurate GPU application and local memory-aware kernel perforation approach.

In typical GPU applications, as depicted in Figure 1a, a GPU kernel first fetches data from the input buffer in global memory, then it performs its computations, and finally it writes the result to the output buffer in global memory. The penalty 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.

The rationale behind our approach is that fast local memory can also be exploited for more accurate approximation. Figure 1b shows how our local memory-aware kernel perforation approach extends the original application with two additional steps: a data perforation phase that fetches a part of the input data; a data reconstruction phase that reconstructs the missing data elements and works on local memory.

In Section 4 we describe how a kernel is perforated and what part of the input data is selected to be fetched from memory. The successive step implementing the reconstruction phase is described in Section 5. Experimental evaluation and discussion are presented in Section 6. Finally, we conclude our work in Section 7.

4 Kernel Perforation

Sidiroglou et al. [19] introduced *loop perforation*, 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.

In this work, we apply perforation to parallel OpenCL kernels: a loop is a kernel whose iterations correspond to OpenCL work-items. Therefore, we apply *kernel perforation* to a parallel kernel instead of a sequential loop.

4.1 From Loop to Kernel Perforation

When applying perforation to a kernel, there are different aspects to consider. While loop perforation works at loop iteration level, our perforation approach focuses on the data (e.g., buffers) used by a kernel because memory accesses are an important component of GPU performance.

In particular, we distinguish between *input approximation* and *output approximation*: input approximation is the process of approximating data on the input of a GPU kernel while output approximation does the opposite, i.e., approximating data on the output of a GPU kernel.

To illustrate this concept, consider the following accurate program (we use loops for simplicity, but techniques apply to kernels similarly):

```
for(i = 0; i < n; i++) {
    output[i] = calc(input[i]);
}
```

For every input element the function `calc` is called. Its result is written to the output element. In this example, the function `calc` requires the *i*-th input data element and the loop is executed *n* times.

By applying loop perforation, e.g., every three iterations, we are actually approximating the output array, which contains the results of the computations. Therefore, we are performing an output perforation of the loop.

```
for(i = 1; i < n; i += 3) {
    output[i] = calc(input[i]);
    output[i+1] = output[i];
    output[i+2] = output[i];
}
```

The output is calculated for the *i*-th element, while it is approximated for `output[i+1]` and `output[i+2]`. The loop is executed $\frac{n}{3}$ times. Approaches such as Paraprox are output approximation, because the output of the kernel execution is approximated.

While output approximation grants high performance improvements, it has two limitations: It usually introduces a very high error, and it does not take into account the possible memory reuse of the input data. A way to overcome this problem is to implement the data perforation in the input data of the loop, e.g.:

```
for(i = 0; i < n; i += 3) {
   x0 = input[i];        // data perforation
   x2 = input[i+2];
   x1 = (x0+x2)/2;       // data reconstruction

   output[i]   = calc(x0);
   output[i+1] = calc(x1);
   output[i+2] = calc(x2);
}
```

In this example, first `x0` and `x2` are loaded from the input array. Then `x1` is approximated by calculating the linear interpolation between `x0` and `x2`. Finally, program execution continues and the result for the input data at position $i, i+1, i+2$ is calculated analogous to the accurate program.

The approximation schemes presented in this paper perform input approximations, where the input buffers are approximated before they serve as an input to the OpenCL kernel computation.

While this is a simple one-dimensional application of the approach, it can be easily extended to two- and three-dimensional kernels, where perforation is performed, e.g., at row or column level. Two-dimensional approximation schemes are further described in Section 4.3.

4.2 Local Input Perforation

The approach of input approximation is motivated by two observations: Many applications are inherently resilient to the input as well as the output and, therefore, they can tolerate small errors. Memory accesses on GPUs have a very long latency, and approximation of the input may take advantage of low-latency local memory to improve the approximation. Most real-life data contain redundancy, for example there is a spatial locality in digital images. Additionally, this data often contains noise, (e.g., quantization noise), and hence is inaccurate. Such data is the input to many applications. 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. Figure 2 shows an example of row-based approximation. The error introduced by data perforation is visible as black lines in Figure 2b; Figure 2c shows the input data reconstruction. In general, input approximation can be a suitable acceleration technique for any application that processes data with redundancy and 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. Although it has been shown that output approximation can be used for many applications, this is a conceptual limitation.

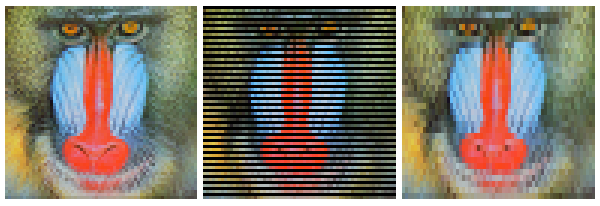

(a) original **(b)** perforated **(c)** approximated

Figure 2. Original, perforated and approximated data [22].

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 significant *reuse* of data across threads. Data reuse means the data loaded by a thread is also used by other threads which in turn also load data.

In the OpenCL programming model, this is usually implemented using the 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 rather limited. Therefore, we use the local memory to implement the steps (Ia) data perforation and (Ib) data reconstruction shown in Figure 1b.

4.3 Perforation Schemes

Paraprox first presented an implementation of kernel perforation [15] by detecting different data-parallel access schemes and generating new kernels that use scheme-specific approximation techniques. For instance, when a stencil access scheme is detected, three approximated kernel versions are generated, each implementing a different approximation scheme. The scheme determines which elements are computed and which elements are to be approximated, and the approximation of elements is accomplished by copying adjacent (calculated) values. Figure 3 shows a visual representation of the schemes on a 2D kernel. Colored elements

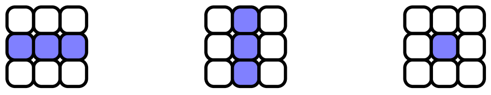

(a) Row **(b)** Column **(c)** Center

Figure 3. Approximation schemes used in Paraprox.

are calculated and white elements are copied from adjacent calculated values. Scheme (a) calculates a row of results and uses this row to approximate the adjacent rows on the top and on the bottom of this row. Scheme (b) proceeds analogous but with columns instead. The most aggressive scheme is (c), which only calculates the value in the center and approximates all adjacent values.

In Paraprox, approximation is accomplished by copying the calculated result to adjacent result values. As described later in Section 5, we introduce more accurate ways to reconstruct these values with different reconstruction techniques.

The reported speedup for these schemes ranges from more than 1.7× for `ConvolutionSeparable` to more than 3× for `Gaussian` and `Mean`. The speedup can be mainly attributed to a reduced number of global memory accesses, as the applications contain only a few multiplications and additions for each calculated output element. However, in the case of the Gaussian filter, a 3×3 filter kernel leads to 9 data elements read. If we assume that scheme (c) is applied, every third data element in x and y direction is calculated and the remaining data elements are approximated. Nonetheless, every data element is loaded once from global memory, as the calculation of the not approximated data elements depend on them (assuming a 3×3-sized filter kernel). Finally, Paraprox assumes a very high maximum error of 10%.

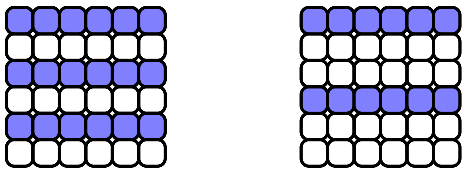
4.4 GPU Perforation Schemes and Halos

The implementation of an input perforation scheme determines which data is loaded from global memory to local memory. Three important aspects are considered. First, the scheme needs to match with the memory architecture in a way that preferably no data that is loaded from memory is discarded by the scheme. For GPUs the memory access granularity depends on different factors but in general is at least 32 bytes. Second, the scheme also needs to match the applications input data structure. For example, if the input data contains a line-shaped structure, skipping lines while loading the data increases the error much more than skipping, e.g., columns. And third, the scheme needs to take into account the organization of threads in work groups.

We present two types of schemes that overcome such limitations. Both of them assume that the input data contains spatial locality, which means that adjacent data elements have a high similarity. This is often the case for video or image data. The approach is not limited to image processing applications. Furthermore, the redundancy does not need to come from spatial locality. Any input data that contains a known redundancy structure can be used as a template for designing a perforation scheme.

From a statistical point of view, a scheme with randomly selected data elements to be approximated would be the best choice, because then the error due to approximation is equally distributed over the input data. Furthermore, a random scheme is less likely to hide structures in the input data. However, such a random scheme would interfere with the way memory is accessed on a GPU, where whole lines of memory are fetched in one transaction.

Row Approximation Scheme Fetching one data element from memory also induces the fetching of data elements in the same row. As global memory accesses are affected by a relatively long latency, it is clear that any data that is actually fetched from global memory should also be used to improve the approximation. Our row approximation scheme (Figure 4) skips the loading of rows in a work group tile and, therefore, adjacent elements in a row are always used. As in adjacent work groups the same approximation scheme is applied, the schemes match each other.



(a) Row scheme 1 **(b)** Row scheme 2

Figure 4. Row approximation scheme.

Stencil approximation scheme Figure 5 shows a stencil approximation scheme for a tile size of 6×6 and a stencil kernel size of 3×3. To compute the accurate result, an extra row on top and on the bottom as well as an extra column left and right need to be fetched additionally. This approximation scheme only fetches the block in the center (drawn in blue) and approximates the extra data elements based on their neighbors. The data elements on the boundaries have influence on a smaller number of the stencil calculations than data elements, that are not on the border. This property is leveraged by the approximation scheme.

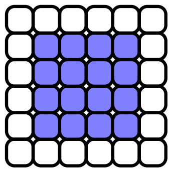


Figure 5. Stencil approximation scheme.

5 Reconstruction

By skipping elements in the input data, an error is introduced. The purpose of the reconstruction step is to minimize this error. Such reconstructions may use very different techniques, depending on the type of application and the input that is targeted. The simplest approach is to approximate the missing elements by copying the values of adjacent data elements. Depending on the data access scheme, that determines which elements from the input data is needed for the calculation of one element in the output data, this approach can be quite effective.

In general, the usage of local memory can accelerate the execution of applications, if there is data reuse across different threads. By using local memory, which is shared among all threads in a work group, data reuse can be accomplished. This approach is well-known and can be considered a standard optimization technique for GPU applications.

However, if output approximation is applied to an application that already uses local memory and there is data reuse, the advantage of the approximation is very small, as the whole input data needs to be loaded (because of data reuse) and there is global memory access that can be *saved* by approximating it.

An example of this situation can be taken from Paraprox: Consider a filter kernel size of 3×3 and the proposed stencil approximation techniques that skip the calculation of either every other row, every other column, or both. The last option calculates the result for only 1 out of 9 elements in the output data, as Figure 3 shows. However, *all* data elements in the input data are accessed at least once. If we assume that local memory is used to prefetch the data plus the surrounding elements, the number of approximated memory accesses is zero and hence the speedup, that was due to approximated global memory accesses, declines.

5.1 Reconstruction Techniques

After loading the incomplete data to the local memory (data perforation), the missing data needs to be reconstructed. Ideally, a perfect reconstruction of the missing data is desired. However, as there is information missing, a perfect reconstruction is not possible. Therefore, we aim to minimize the error. For the reconstruction different options are possible. In this work we compare two different types of data reconstruction techniques on the sparsely fetched data: nearest-neighbor interpolation and linear interpolation.

The data reconstruction method depends on the approximation scheme that was used and not all combinations are possible.

Nearest-neighbor Interpolation A straight-forward approach for the completion of the perforated data is nearest-neighbor interpolation. Data elements that were not loaded are approximated by picking the nearest value that was loaded as a replacement value.

Linear Interpolation Another well-known technique is linear interpolation. For this method, it is necessary that the element to be approximated has adjacent elements on both sides. This requirement is not always true, see for example the edges of the stencil scheme in Figure 5. In this case we employ nearest-neighbor interpolation.

6 Experimental Evaluation

To evaluate our approach, we have reproduced the approximation schemes used by Paraprox (as described in Section 4.3) and compared them with our approach in terms of error as well as speedup. Furthermore, we extended Paraprox's schemes with a more aggressive perforation scheme that approximates 4 instead of 2 rows or columns. We apply their approach to a variety of benchmarks. However, we are not able to reproduce the numbers that were reported in the original work. This can be explained by the usage of different hardware and different benchmark implementations. Moreover, some benchmarks are more sensitive to different input data, as we show in Figure 6.

Our results were conducted on an AMD FirePro W5100 GPU with 3.5 GB memory using OpenCL driver AMD-APP version 17.10-414273 supporting OpenCL version 1.2.

Table 1. Details of the applications that have been used in the evaluation.

Application	Domain	Error Metric
Gaussian	Image processing	Mean relative error
Median	Medical imaging	Mean relative error
Hotspot	Physics simulation	Mean relative error
Inversion	Image processing	Mean relative error
Sobel	Image processing	Mean error

6.1 Benchmarks

We manually applied our approach to six benchmarks. An overview can be found in Table 1. For all except one benchmark we use the mean relative error (MRE) as metric. The MRE is determined by calculating the difference of result and the value and then dividing by the true value: $\frac{x_{true}-x_{test}}{x_{true}}$. However, when x_{true} is zero or close to zero the MRE is either very high or undefined. The Sobel3/Sobel5 applications are particularly prone to such situations. Therefore, we opt to use the mean error as metric for these two applications instead, which does not suffer from this limitation. Input and output are grayscale images sized 1024×1024 for Gaussian, Inversion, Median and Sobel3/Sobel5. For Hotspot we used the 1024 sized input data sets provided by Rodinia [2].

The Gaussian filter is a well-known low-pass filter. Low-pass filtering has many applications, e.g., in electronics and signal processing. A reduction in noise and detail is an important preprocessing step in image processing, e.g., for building edge detection applications, as they are particularly sensitive to noise. The Gaussian has data-reuse across threads and therefore benefits from the use of local memory in general. The filter kernel size is 3×3.

The Inversion filter is an application that computes the *digital negative* of an image. We use this artificial benchmark to assess the performance of applications with 1×1 filter kernels. As there is no data reuse across threads, such applications usually do not benefit from the use of local memory.

The Median filter is a nonlinear spatial filter with applications in medical imaging and image processing. It is particularly effective in reducing *salt-and-pepper noise*, which are sudden and sharp signal disturbances. By applying the filter to a signal, each sample is replaced by the median of the samples in the neighborhood. To calculate the result of the filter, first all values of the filter mask need to be sorted by their value. The median value is selected and used as result. Our implementation is using local memory for prefetching of data elements from global memory. Additionally, we are using private memory to load all samples in the current filter kernel. Then we follow approach of Blum et al. [1] to determine the *median of medians*. Therefore, our implementation is already highly optimized. The filter kernel size is 3×3.

Hotspot is a thermal simulation tool and part of the Rodinia benchmark suite [2]. The application consists of a 2D transient thermal simulation kernel that iteratively solves differential equations. Input to the hotspot application are two square matrices: The first matrix represents power data and the seconds represents temperature data. Output is a matrix of the same size that contains the temperature.

The Sobel operator is used in image processing and computer vision to build edge-detection applications. It computes an approximation of the gradient that emphasizes on the edges in an image. The calculation is done in a horizontal and a vertical convolution step. We use the Sobel operator in two applications. Sobel3 is using a 3×3 filter kernel mask and Sobel5 is using a 5×5 filter kernel mask.

6.2 Input Data Sensitivity

The results of our work show that the amount of error that is introduced by the approximation depends on the input to the applications. In contrast, the speedup only depends on the selected approximation scheme. We apply our technique to six benchmarks to show the sensitivity to input data. Gaussian, Inversion, Median and Sobel3/Sobel5 are executed on a set of 100 input data sets taken from the USC-SIPI Image Database [22] and consisting of a subset of the misc and pattern catalogue. For each of the applications we selected one of the Pareto-optimal configurations. For Hotspot and Inversion *row scheme 1* was used. For the other applications *stencil scheme* was used. Figure 6 shows the results. The upper part of the figure shows the distribution of the error for the applications. The average error is almost always less than 5%. Only Sobel5 shows a higher average. For all image-based applications, there are some outliers that have an error of up to 20%, except Sobel3 and Sobel5 which have a higher error.

The Gaussian application is speedup by 2.2× by our approach. The median error is less than 4% and the variance of the error is small, even considering that there are some outliers in the error distribution of up to 17%.

The Inversion application has a speedup of 1.59×. The median error is about 5%. The variance of the error is larger and there are outliers of up to 20% error.

A speedup of 1.62× is shown for the Median application. This is particularly interesting considering the application is working also with *private* memory in the implementation of median of medians as explained in Section 6.1. The reported speedup is therefore on top of an already optimized application. The median error is about 5%. The variance of the error is about the same as for the Inversion application.

We observe a speedup of 1.98× for the Hotspot application. As we rely on the input data provided by the application we have 8 different input data sets, that differ in their size. The input data is generated using a tool shipped with the Hotspot application. The results show that in general executing the application with a perforated data set introduces only a very small error in the result. Furthermore, compared with other applications the variance of the error is very small.

A speedup of 1.79× can be seen for the Sobel3 application. While the median error is 5% the variance of the error is larger than for the previous applications. For about 75% of the measurements, the error is less than 15%.

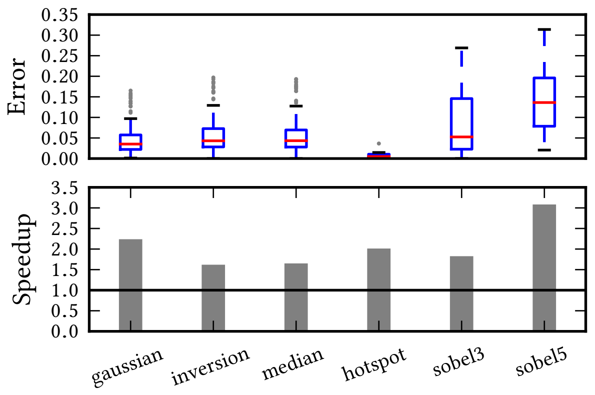


Figure 6. Error distribution on different input data. On the bottom is the speedup of our approach compared to the state-of-the-art baseline implementation depicted.

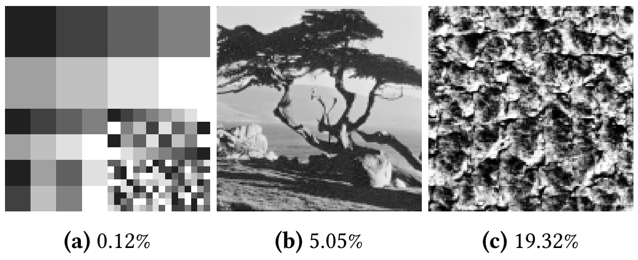


Figure 7. Input data and corresponding error [22].

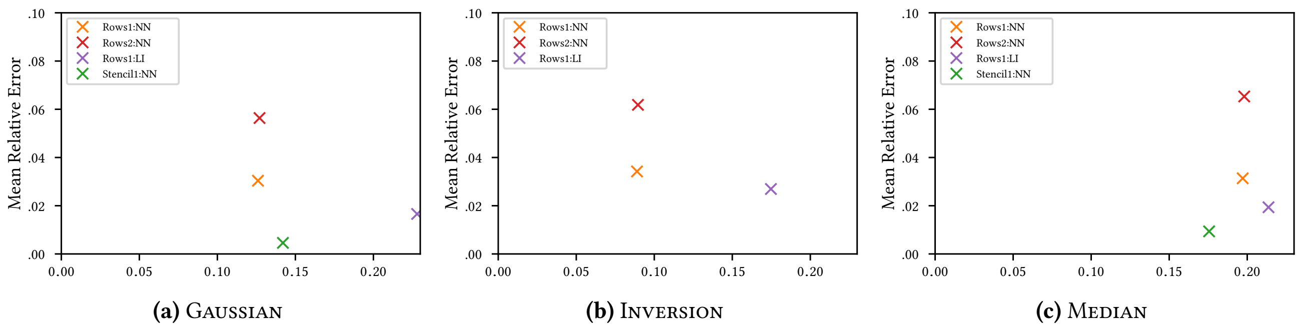


Figure 8. Perforation schemes with different parameters.

The highest speedup in our study is 3.05× for the Sobel5 application. The higher speedup in comparison to Sobel3 can be partly attributed to the larger filter kernel size and therefore to more data reuse across threads. While the median error is 15% and therefore significantly higher than for Sobel3, the distribution of the error is more dense and 75% of the measurements have an error smaller than 20%.

These results show that the amount of error introduced by our approach can differ by orders of magnitude depending on the input. To illustrate this further we show exemplary input and corresponding error for the Median application in Figure 7. Input data that contains larger areas of the same color can be approximated with a very small error of only 0.12% (Figure 7a). Countryside photographs (Figure 7b) produce an error of 5.05% that is about the median of our test input data set. Pattern-images (Figure 7c) contain a lot of high frequency and therefore are prone to perforation. They yield a larger error, in this case 19.32%.

Applications that execute filter kernels with no or small halo areas (Gaussian, Inversion, Median) and therefore also a small area of prefetching, have a smaller variance in the error. These applications also have center-weighted filter kernels. Compared to that, Sobel3 has as larger variance but still a comparable low median error of about 5%. Sobel5 has a median error of 13% and an even higher variance.

6.3 Parametrization

Perforation Schemes In Figure 8, we compare different perforation approaches. We conducted our study for the applications Gaussian, Inversion and Median. On the x-axis is the runtime of the applications in 1/100 seconds depicted while the y-axis shows the mean relative error.

We compare four configurations: `Rows1:NN` is perforation of every other row and reconstruction using nearest-neighbor interpolation. `Rows2:NN` is perforation of 3 out of 4 rows and reconstruction using nearest-neighbor interpolation. `Rows1:LI` is perforation of every other row and reconstruction using linear interpolation. `Stencil1` is perforation of the boundaries of a work group tile and uses nearest-neighbor interpolation for reconstruction.

As expected, the error is higher if the approximation scheme is more aggressive, e.g., if more input data is perforated. The error for `Stencil1` is very low and always less than 1%, as this perforation scheme is approximating only a small amount of data. The error of `Rows1:NN` is about half of the error of `Rows2:NN`. However, the runtime is for all applications the same. This might be attributed to the specific implementation or the memory architecture. The error for `Rows1:LI` is smaller than for `Rows1:NN` (Gaussian: -45%, Inversion: -21%, Median: -34%). However, the error of `Rows1:NN` is already small and less than 4% for all three applications. The error of `Stencil1` is less than 1%. This is due to the small amount of data that is approximated.

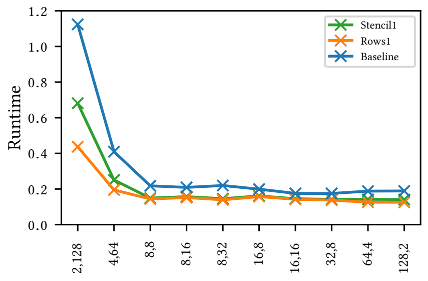
The runtime of `Rows1:NN`, `Rows2:NN` and `Rows1:LI` is similar for the Gaussian and Inversion application. However, it is different for the Median application, which is explained by the use of private memory.

Local Work Group Size We compare performance with local work group size in Figure 9. We use nearest-neighbor interpolation for all applications. The baseline applications use local memory for Gaussian and Median. Median is also using private memory as described in Section 6.1. The accurate Inversion application does not use local memory as a prefetching step would increase runtime.

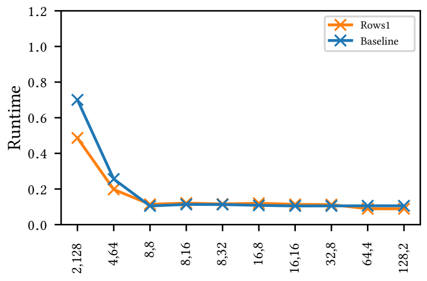
Two properties are remarkable: First, all configurations have a larger or equal x than y component. This is due to the better alignment of these configurations to the memory interface. Second, the optimal work group configuration for an application is different for the accurate `Baseline` and for the approximated versions. Therefore, work group configuration needs to match the approximation scheme.

6.4 Pareto Optimality

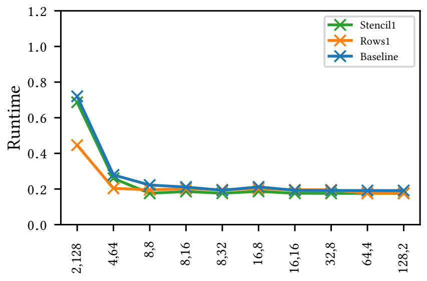
We compare our approach with Paraprox' state-of-the-art solution [15] in Figure 10. The state-of-the-art solution is plotted using a • marker and our approach is plotted using a × marker. The Pareto-optimal solutions are connected using a gray dashed line. Center, rows, and cols are three output approximation schemes, see Figure 3. The numbers



(a) Gaussian

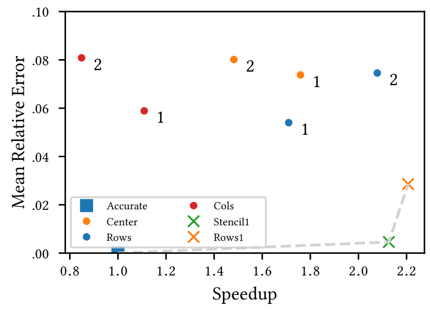


(b) Inversion

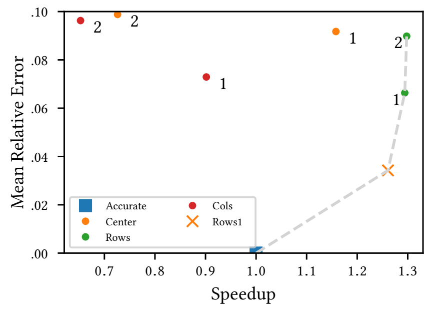


(c) Median

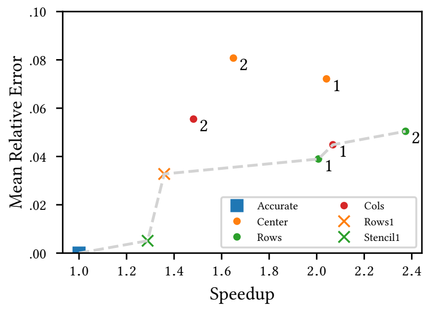
Figure 9. Local work group size tuning.



(a) Gaussian



(b) Inversion



(c) Median

Figure 10. Pareto-optimal solutions of the proposed and Paraprox' state-of-the-art solutions.

next to the points indicate the perforation scheme: (1) approximate 2 rows/columns; (2) approximate 4 rows/columns. Our approach is compared using two perforations schemes. `Stencil1` is approximating the work group boundaries, and `Rows1` is approximating every second row, see Figure 4 and Figure 5. The speedup of all applications is calculated with respect to the baseline implementation from Paraprox.

Figure 10a shows the Gaussian application. The Pareto-optimal solutions are `Stencil1` and `Rows1`. The error for `Stencil1` is with 0.45% very low and the speedup is 2.1×. The error for `Rows1` is 2.9%. The increased error is explained by the larger amount of approximated input data. This also explains the higher speedup of 2.2×. The second highest speedup is 2.08× by the state-of-the-art approach `Rows` that approximates 2 out of 3 rows. This is also the explanation of the much higher error of 7.5%.

The Inversion application is shown in Figure 10b. Pareto-optimal solutions are `Rows` and `Rows1`. `Stencil1` cannot be used as the application has a filter kernel size of 1×1. `Cols` becomes slower, which is explained by the improper alignment of column-shaped perforation and memory data layout.

The results of the Median application are shown in Figure 10c. Pareto-optimal configurations are `Stencil1`, `Rows1`, `Rows` and `Cols`. Speedup and error of `Stencil1` and `Rows1` are 1.29× and 0.5%; 1.36× and 3.3%. As the baseline implements the *median of medians* and therefore uses *private memory* which is faster than local memory.

In general, the error of our approach is improved significantly compared with Paraprox while we reach a similar speedup. Our approach is not limited to applications that generally benefit from the use of local memory as shown by the results of the Inversion application.

7 Conclusion

We introduce local memory-aware kernel perforation, a novel technique for the acceleration of GPU kernels using approximate computing. Our approach first skips loading part of data from global memory and later uses local memory enabled reconstruction methods. We present a general perforation scheme that skips loading rows of input data. Additionally, we present a stencil perforation scheme that skips loading the data elements close to the borders of the work group tiles.

The experimental evaluation shows that our approach is able to accelerate a variety of application from 1.6× to 3× while maintaining an average error of 6%. Our results show that the amount and distribution of the error depends on the input data. We were able to significantly lower the error while keeping similar performance than the state-of-the-art approach Paraprox.

In a parameter exploration study, we show that, depending on the employed perforation approach and reconstruction technique, the error can be tuned from 0.5% to 7% depending on the input data. We show that the optimal local work group size for the baseline kernel and approximate kernels are different. Therefore, a system optimized for the baseline kernel will not perform optimal for approximate kernels. We investigate the Pareto-optimality of our approach. Our experiments show that our approach can improve the error and the speedup significantly with respect to state of the art.

In a following work we will implement our currently manual approach in a fully automatic compiler-based framework. As we have shown, that the technique gives promising results for a set of general-purpose kernels, a library can automatically apply and tune the technique to approximable kernels and memory regions and accelerate a large set of applications.

References

[1] Manuel Blum, Robert W Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E Tarjan. 1973. Time bounds for selection *. *Journal of Computer and System Sciences* 7, 4 (1973).

[2] 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 *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE.

[3] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Design Automation Conference (DAC)*. ACM/EDAC/IEEE.

[4] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[5] Beayna Grigorian and Glenn Reinman. 2015. Accelerating Divergent Applications on SIMD Architectures Using Neural Networks. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 1 (2015).

[6] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. 2011. IMPACT: Imprecise Adders for Low-power Approximate Computing. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design (ISLPED)*. IEEE.

[7] Melanie Kambadur and Martha A Kim. 2016. Nrg-loops: Adjusting Power from Within Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*. ACM.

[8] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-Driven Transparent Approximation Acceleration on GPUs. In *Proceedings of the 2016 International Conference on Supercomputing (ICS)*. ACM.

[9] Mikko H. Lipasti, Christopher B Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. *ACM SIGPLAN Notices* 31, 9 (1996).

[10] Liming Lou, Paul Nguyen, Jason Lawrence, and Connelly Barnes. 2016. Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples. *ACM Transactions on Graphics (TOG)* 35, 5 (2016).

[11] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[12] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *ACM SIGPLAN Notices*, Vol. 49. ACM.

[13] Subrata Mitra, Manish K. Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-aware Optimization in Approximate Computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO)*. IEEE.

[14] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys (CSYR)* 48, 4 (2016).

[15] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[16] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning Approximation for Graphics Engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM.

[17] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *ACM SIGPLAN Notices*, Vol. 46. ACM.

[18] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. 2015. A Low Latency Generic Accuracy Configurable Adder. In *Design Automation Conference (DAC)*. ACM/EDAC/IEEE.

[19] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. 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 (ESEC/FSE)*. ACM.

[20] Renée St Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose Code Acceleration with Limited-precision Analog Computation. *ACM SIGARCH Computer Architecture News* 42, 3 (2014).

[21] Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. 2015. Scalable-effort Classifiers for Energy-efficient Machine Learning. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*. ACM.

[22] Allan G. Weber. 2006. The USC-SIPI Image Database. http://sipi.usc.edu/database/database.php. (2006). Accessed August 2018.

[23] Amir Yazdanbakhsh, Divya Mahajan, Hadi Esmaeilzadeh, and Pejman Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design & Test* 34, 2 (2017).

[24] Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, et al. 2015. Axilog: Language Support for Approximate Hardware Design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE.

[25] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2016. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016).