CPU-GPU Collaboration for Output Quality Monitoring

Mehrzad Samadi and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{mehrzads, mahlke}@umich.edu

Abstract

In this paper, we proposed a new low overhead collaborative technique of output quality monitoring for approximate computing on GPUs. In this technique, the CPU is responsible for performing quality monitoring while the GPU executes approximate kernels. For two image processing applications, we showed that this technique outperforms previous quality monitoring techniques.

1. Introduction

Heterogeneous systems that combine both traditional processors with powerful GPUs have become standard in all systems ranging from servers to cell phones. GPUs represent affordable but powerful compute engines that can be used for many of the domains that are amenable to approximation. Approximation is applicable in domains where some degree of variation or error can be tolerated in the result of computation. There are many important domains where approximation can greatly improve application performance, including multimedia processing, machine learning, data analysis, and gaming. This paper focuses on applying approximate computing, where the accuracy of results is traded off for computation speed, to accelerate programs on GPUs.

Recent works, Paraprox [3] and SAGE [4], provide static compilers to generate approximate data parallel kernels. These systems enable the programmer to implement a kernel once using the OpenCL or CUDA data parallel languages and, depending on the target output quality (TOQ) specified for the kernel, tradeoff accuracy for performance. At run-time, these systems use a greedy algorithm to tune the parameters of the approximate kernels to identify configurations with high performance and a quality that satisfies the TOQ.

As the program behavior can change at run-time, there should be a runtime system to monitor the quality and performance dynamically. After every N invocations of the kernel, SAGE, similar to the Green framework [2], runs both the exact and approximate kernels on the GPU , one after each other, to check the output quality and performance. Comparing the exact and approximate outputs to compute the quality is also executed on the GPU in parallel to reduce the overhead of quality monitoring. If the measured quality is lower than the TOQ, SAGE switches to a slower but more precise version of the program. This process will continue until the output quality satisfies the TOQ.

However, since each time SAGE computes the quality running both versions sequentially, this quality monitoring has a high overhead. To reduce this overhead, in this paper, we propose a new collaborative CPU-GPU quality monitoring technique (CCG) which runs the quality monitoring code on the CPU while the GPU executes approximate data-parallel kernels.

2. Quality Monitoring Techniques

In this section, we explain different techniques to compute the output quality. We compare the accuracy and performance of these techniques in Section 3.

Conservative Fixed Interval (CFI): This technique checks the output quality every N_{th} invocation. Computing the output quality is done by running the approximate and exact versions sequentially. After that, the runtime system computes the output quality by comparing the exact and approximate outputs. Since this process has a high overhead, quality checking has a high impact on the overall performance of this technique.

At the checking point, if the measured quality is lower than the $TOQ - delta$, the runtime system switches to a slower but more precise version of the program. Since this technique just reduces the aggressiveness of the approximate versions, we call this technique conservative.

Conservative Adaptive Interval (CAI): This approach is similar to CFI but to reduce the monitoring overhead of the CFI, this technique performs quality checking more frequently to converge to a stable solution faster at the beginning of the execution. If the output quality is higher than $TOQ - delta$, the interval between two checking points is gradually increased so that the overhead of quality checking is reduced. Every time the run-time management needs to change the selected kernel (output quality is lower than $TOQ - delta$), the interval between checking points is reset to a minimum width. This technique is conservative too (like the CFI) so the overall performance might be lower than ideal.

Non-Conservative Fixed Interval (NFI): To improve the performance, unlinke aformentioned techniques, this technique is looking for opportunities to increase the aggressiveness of approximate versions. At the checking point, if the output quality is higher than $TOQ + delta$, the runtime system increases the aggressiveness of the approximate versions. On the other hand, if the output quality is lower than $TOQ - delta$, the runtime system decreases the aggressiveness of the approximate versions.

Non-Conservative Adaptive Interval (NAI): This technique is similar to the NFI but with adaptive intervals. Since this technique is adaptive and non-conservative, its overall performance should be higher than the last three mentioned techniques.

Collaborative CPU-GPU (CCG): Instead of checking every N_{th} invocation, this technique checks the quality of all invocations and runs the quality checking on the CPU in parallel to the GPU execution using synchronous execution. At each time quantum, the GPU runs the selected kernel for an invocation while the CPU computes the output quality for the next invocation. To make the overhead of quality checking almost zero, it should be completely overlapped by the kernel execution.

However, since GPU's the performance is higher than the CPU's for data parallel kernels, the CPU cannot keep up with the GPU while computing the output quality for the whole input data set. We solved this problem by two means: First, we parallelized the output quality computing on the CPU with four threads. Second, instead of performing full quality checking, this technique runs partial quality checking, which applies the exact and approximate kernels to a subset of input data and compares the results to estimate the overall output quality. To perform partial quality monitoring, we identified three central challenges that must be solved.

First, it is not straight-forward how to generate partial quality checking code for general applications automatically. In this paper, we wrote these codes manually. However, it is possible to use the same pattern-based compilation method as used in Paraprox [3]. Paraprox creates approximate kernels by recognizing common computation idioms found in data-parallel programs (e.g., Map, Scatter/Gather, Reduction, Scan, Stencil, and Partition) and substituting approximate implementations in their place. It is possible to generate pattern-based partial checking codes too.

Second, the subset of the input data that is used for partial quality monitoring should be chosen carefully to be a representative for the whole input data set. For now, we chose a uniformly distributed data from the input array and applied partial quality monitoring to that.

Third, the method of choosing the aggressiveness of approximation for the next kernel based on the partial output quality is important to get the best accuracy. In this work, we checked the output for three levels of approximation for each invocation (the current level, one level more aggressive, and one level less aggressive). After computing the partial output quality for three approximate versions, the CPU will decide which one to use for the next kernel.

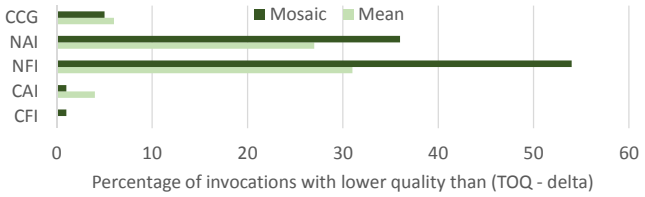

Figure 1: Percentages of images with unacceptable quality (lower than $TOQ - delta$) for different quality monitoring techniques.

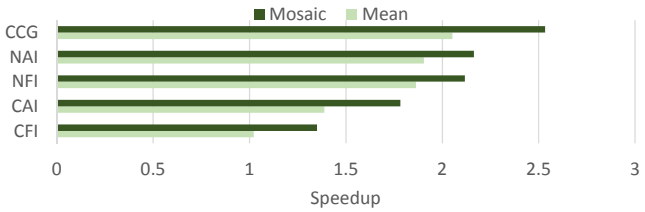

Figure 2: Overall speedup of applying two programs on all 1600 images.

3. Experimental Evaluation

In this section, we show how these quality monitoring techniques affect the execution time and accuracy of different applications. We used two applications from the image processing domain: Mosaic and Mean filter. To approximate these benchmarks, we used the approximation methods described in Paraprox [3]. For Mosaic application, we used loop perforation [1]. For Mean filter, we assumed that neighbor pixels have similar values. Based on this assumption, rather than accessing all neighbors within a tile, we access only a subset of them and assumes the rest of the neighbors have the same value. We applied these applications on 1600 500×500 images of different flowers. In these experiments, we set the TOQ to 95% and delta is 1%.

Figure 1 shows the percentages of images for which their output quality is not acceptable (lower than $TOQ - delta$) for all quality monitoring techniques. In other words, this figure shows how accurate these techniques are. Figure 2 shows the overall speedup of different techniques for applying Mosaic and Mean filter applications on all 1600 images. As expected, conservative techniques' (CFI and CAI) output quality is always better than other techniques. However, they do not show great performance. Non-conservative techniques provide great speedups but the quality of more than 25% of images is not acceptable with these techniques. The problem is that these techniques are based on the assumption that it is possible to predict the quality of images by computing the quality of every N_{th} invocations. As seen in

these figures, the CCG outperforms other techniques mostly because its monitoring overhead is negligible.

References

[1] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, Mar. 2009.

[2] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 198–209, 2010.

[3] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: pattern-based approximation for data parallel applications. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–50, 2014.

[4] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning approximation for graphics engines. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 13–24, 2013.