Scaling Performance via Self-Tuning Approximation for Graphics Engines

MEHRZAD SAMADI, JANGHAENG LEE, D. ANOUSHE JAMSHIDI, and SCOTT MAHLKE, University of Michigan
AMIR HORMATI, Google Inc.

Approximate computing, where computation accuracy is traded off for better performance or higher data throughput, is one solution that can help data processing keep pace with the current and growing abundance of information. For particular domains, such as multimedia and learning algorithms, approximation is commonly used today. We consider automation to be essential to provide transparent approximation, and we show that larger benefits can be achieved by constructing the approximation techniques to fit the underlying hardware. Our target platform is the GPU because of its high performance capabilities and difficult programming challenges that can be alleviated with proper automation. Our approach—SAGE—combines a static compiler that automatically generates a set of CUDA kernels with varying levels of approximation with a runtime system that iteratively selects among the available kernels to achieve speedup while adhering to a target output quality set by the user. The SAGE compiler employs three optimization techniques to generate approximate kernels that exploit the GPU microarchitecture: selective discarding of atomic operations, data packing, and thread fusion. Across a set of machine learning and image processing kernels, SAGE's approximation yields an average of 2.5× speedup with less than 10% quality loss compared to the accurate execution on a NVIDIA GTX 560 GPU.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers*

General Terms: Design, Performance

Additional Key Words and Phrases: Approximate computing, compiler, GPU, optimization

ACM Reference Format:
Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Scott Mahlke, and Amir Hormati. 2014. Scaling performance via self-tuning approximation for graphics engines. ACM Trans. Comput. Syst. 32, 3, Article 7 (August 2014), 29 pages.
DOI: http://dx.doi.org/10.1145/2631913

1. INTRODUCTION

To keep up with information growth, companies such as Microsoft, Google, and Amazon are investing in larger data centers with thousands of machines equipped with multicore processors to provide the necessary processing capability on a yearly basis. The latest industry reports show that in the next decade, the amount of information will

---

This research was supported by ARM Ltd. and the National Science Foundation under grants CNS-0964478 and CCF-0916689. This article extends an earlier version that appeared in the Proceedings of the 46th International Symposium on Microarchitecture (MICRO 2013) [Samadi et al. 2013].
Authors' addresses: M. Samadi, J. Lee, D. A. Jamshidi, and S. Mahlke, Computer Science and Engineering Department, University of Michigan, 2260 Hayward Street, Ann Arbor, MI, 48109; emails: mehrzads@umich.edu, jhaeng@umich.edu, ajamshid@umich.edu, mahlke@umich.edu; A. Hormati, Google Inc., 651 N. 34th Street, Seattle, WA 98103; email: hormati@google.com.

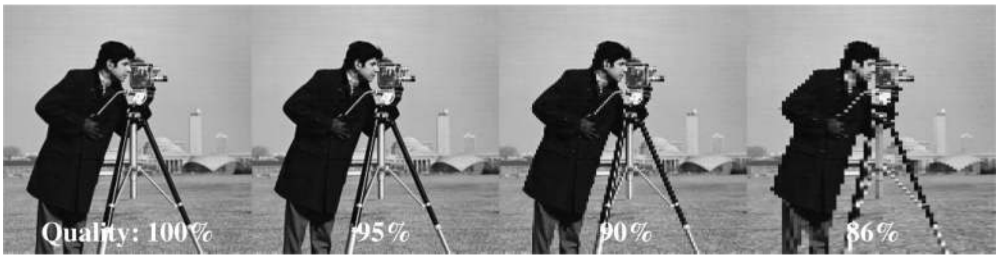


Fig. 1. Application of image blurring filter with varying degrees of output quality. Four levels of output quality are shown: 100%, 95%, 90%, and 86%.

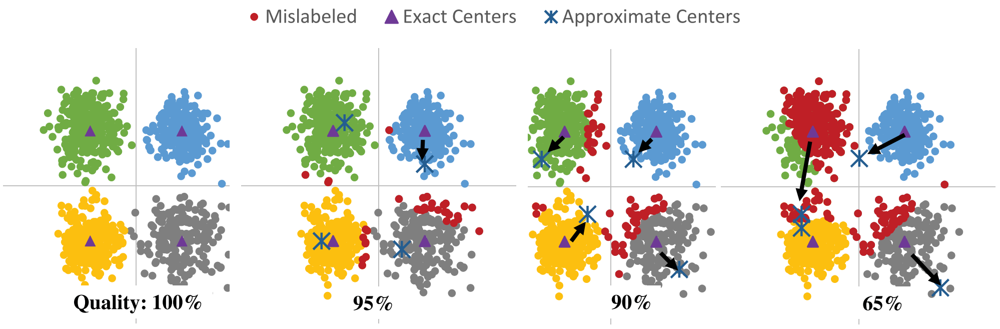


Fig. 2. Clustering of a sample dataset into four clusters using the K-Means algorithm. Exact and approximate clusters' centers are also shown at four levels of output quality: 100%, 95%, 90%, and 65%.

expand by a factor of 50, whereas the number of servers will only grow by a factor of 10 [EMC Corporation 2011]. At this rate, it will become more expensive for companies to provide the compute and storage capacity required to keep pace with the growth of information. To address this issue, one promising solution is to perform approximate computations on massively data-parallel architectures, such as GPUs, and trade the accuracy of the results for computation throughput.

There are many domains where it is acceptable to use approximation techniques. In such cases, some variation in the output is acceptable, and some degree of quality degradation is tolerable. Many image, audio, and video processing algorithms use approximation techniques to compress and encode multimedia data to various degrees that provide trade-offs between size and correctness, such as lossy compression techniques. For example, while trying to smooth an image, the exact output value of a pixel can vary. If the output quality is acceptable for the user or the quality degradation is not perceivable, approximation can be employed to improve the performance. In the machine learning domain, exact learning and inference is often computationally intractable due to the large size of input data. To mitigate this, approximate methods are widely used to learn realistic models from large datasets by trading off computation time for accuracy [Kulesza and Pereira 2008; Shindler et al. 2011]. We believe that as the amount of information continues to grow, approximation techniques will become ubiquitous to make processing such information feasible.

To illustrate this behavior more concretely, consider the two examples shown in Figures 1 and 2. Figure 1 shows the output of a blurring filter applied to an image with varying degrees of quality loss. The left-most image shows the correct output, and the subsequent images to the right show the results with 5%, 10%, and 14% quality loss, respectively. For most people, it is difficult to observe any significant differences between the first three images. Therefore, a range of outputs are acceptable. However, the fourth image is noticeably distorted and would be unacceptable to many users. This illustrates that limited losses in output quality may be unnoticeable, but approximation must be controlled and closely monitored to ensure acceptable quality of results.

Figure 2 provides a similar set of results, but for the K-Means data clustering algorithm. The left-most image shows the correct output: each input data (dot) is clustered into one of four groups, as indicated by the color of the dot with the centroid of each cluster marked by the triangle. The subsequent images show the results with 5%, 10%, and 35% quality loss, respectively. In each image, the dots colored red represent the misclassified data points, and the X's show the approximate centroids. Again, for small amounts of quality loss (three left-most images), the application output largely matches the correct output due to the inherent error tolerance of data clustering. However, the right-most image shows poor results, particularly for one of the clusters (green cluster), with more than half the input data misclassified and the centroid substantially out of position.

The idea of approximate computing is not a new one, and previous works have studied this topic in the context of more traditional CPUs and proposed new programming models, compiler systems, and runtime systems to manage approximation [Rinard 2006, 2007; Agarwal et al. 2009; Baek and Chilimbi 2010; Sampson et al. 2011; Ansel et al. 2011; Esmaeilzadeh et al. 2012a]. In this work, we instead focus on approximation for GPUs. GPUs represent affordable but powerful compute engines that can be used for many of the domains that are amenable to approximation. However, in the context of GPUs, previous approximation techniques have two limitations: (1) the programmer is responsible for implementing and tuning most aspects of the approximation, and (2) approximation is generally not cognizant of the hardware on which it is run. There are several common bottlenecks on GPUs that can be alleviated with approximation. These include the high cost of serialization, memory bandwidth limitations, and diminishing returns in performance as the degree of multithreading increases. Because many variables affect each of these characteristics, it is very difficult and time consuming for a programmer to manually implement and tune a kernel.

SAGE—our proposed framework for performing systematic runtime approximation on GPUs—enables the programmer to implement a program once in CUDA, and depending on the target output quality (TOQ) specified for the program, trade the accuracy for performance based on the evaluation metric provided by the user. SAGE has two phases: offline compilation and runtime kernel management. During offline compilation, SAGE performs approximation optimizations on each kernel to create multiple versions with varying degrees of accuracy. At runtime, SAGE uses a greedy algorithm to tune the parameters of the approximate kernels to identify configurations with high performance and a quality that satisfies the TOQ. This approach reduces the overhead of tuning, as measuring the quality and performance for all possible configurations can be expensive. Since the behavior of approximate kernels may change during runtime, SAGE periodically performs a calibration to check the output quality and performance, updating the kernel configuration accordingly.

To automatically create approximate CUDA kernels, SAGE utilizes three optimization techniques. The first optimization targets atomic operations, which are frequently used in kernels where threads must sequentialize writes to a common variable (e.g., a histogram bucket). The atomic operation optimization selectively skips atomic operations that cause frequent collisions and thus cause poor performance as threads are sequentialized. The next optimization, data packing, reduces the number of bits needed to represent input arrays, thereby sacrificing precision to reduce the number of high-latency memory operations. The third optimization, thread fusion, eliminates some thread computations by combining adjacent threads into one and replicating the output of one of the original threads. A common theme in these optimizations is to exploit the specific microarchitectural characteristics of the GPU to achieve higher performance gains than general methods, such as ignoring a random subset of the input data or loop iterations [Agarwal et al. 2009], which are unaware of the underlying hardware.

In summary, the main contributions of this work are as follows:

—The first static compilation and runtime system for automatic approximate execution on GPUs
—Three GPU-specific approximation optimizations that are utilized to automatically generate kernels with variable accuracy
—A greedy parameter tuning approach that is utilized to determine the tuning parameters for approximate versions
—A dynamic calibration system that monitors the output quality during execution to maintain quality with a high degree of confidence and takes corrective actions to stay within the bounds of target quality for each kernel
—Evaluation of the proposed system and comparison of four different modes of calibrations and monitoring systems

The rest of the article is organized as follows. Section 2 discusses why SAGE chooses these three approximation optimizations. Section 3 explains how the SAGE framework operates. Approximation optimizations used by SAGE are discussed in Section 4. The results of using SAGE for various benchmarks are presented in Section 5. Section 6 discusses different quality monitoring techniques. Section 7 discusses the related work in this area and how SAGE is different from previous works. The summary and conclusion of this work is outlined in Section 8.

2. GPU PROGRAMMING MODEL AND APPROXIMATION OPPORTUNITIES

2.1. GPU Programming Model

The CUDA programming model is a multithreaded single instruction multiple data (SIMD) model that enables implementation of general-purpose programs on heterogeneous GPU/CPU systems. A CUDA program consists of a host code segment that contains the sequential portion of the program, which is run on the CPU, and a parallel code segment that is launched from the host onto one or more GPU devices.

The threading abstraction in CUDA consists of three levels of hierarchy. The basic block of work is a single *thread*. A group of threads executing the same code are combined together to form a *thread block* or simply a *block*. Together, these thread blocks combine to form the parallel segments called *grids*, where each grid is scheduled onto a GPU at a time. Threads within a thread block are synchronized together through a barrier operation (*__syncthreads*()). However, there is no explicit software or hardware support for synchronization across thread blocks. Synchronization between thread blocks is performed through the global memory of the GPU, and the barriers needed for synchronization are handled by the host processor.

The memory abstraction in CUDA consists of multiple levels of hierarchy. The lowest level of memory is *registers*, which are on-chip memories private to a single thread. The next level of memory is *shared memory*, which is an on-chip memory shared only by threads within the same thread block. Access latency to both the registers and shared memory is very low. The next level of memory is *local memory*, which is an off-chip memory private to a single thread. Local memory is mainly used as spill memory for local arrays. Mapping arrays to shared memory instead of spilling to local memory can provide much better performance. Finally, the last level of memory is *global memory*, which is an off-chip memory accessible to all threads in the grid. This memory is used primarily to stream data in and out of the GPU from the host processor. The latency for off-chip memory is 100 to 150× more than that of on-chip memories, but to exploit some locality of data, two levels of caches are introduced to make this latency more tolerable.

The CUDA programming model is an abstraction layer to access GPUs. NVIDIA GPUs use a single instruction multiple thread (SIMT) model of execution where multiple thread blocks are mapped to streaming multiprocessors (SMs). Each SM contains a number of processing elements referred to as streaming processors (SPs). A thread executes on a single SP. Threads in a block are executed in smaller execution groups of threads called *warps*. All threads in a warp share one program counter and execute the same instructions. If conditional branches within a warp take different paths, causing *control path divergence*, the warp will execute each branch path serially, stalling the other paths until all of the paths are complete. Such control path divergences severely degrade the performance.

Because off-chip global memory access is very slow, GPUs support *coalesced memory accesses*. Coalescing memory accesses allows one bulk memory request from multiple threads in a half-warp (full warp in Fermi and Kepler architectures) to be sent to global memory instead of multiple separate requests. To coalesce memory accesses in recent generations of GPUs, all accesses of a warp should be adjacent and in the same cache line. Effective memory bandwidth is an order of magnitude lower than using noncoalesced memory accesses, which further signifies the importance of memory coalescing for achieving high performance.

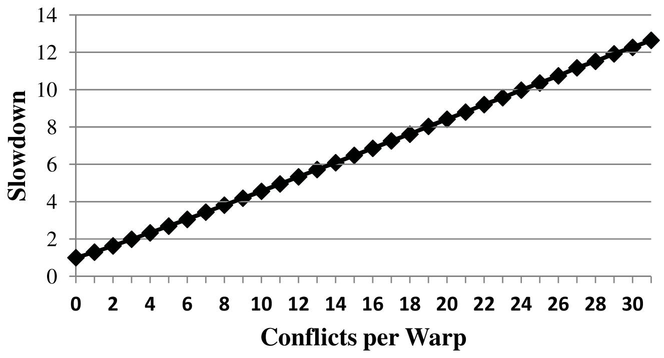
To avoid race conditions, CUDA supports atomic operations. An atomic construct performs a read-modify-write atomic operation on one element residing in global or shared memory. For example, *atomicInc()* reads a 32-bit word from an address in the global or shared memory, increments it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete [NVIDIA 2013].

2.2. Approximation Opportunities

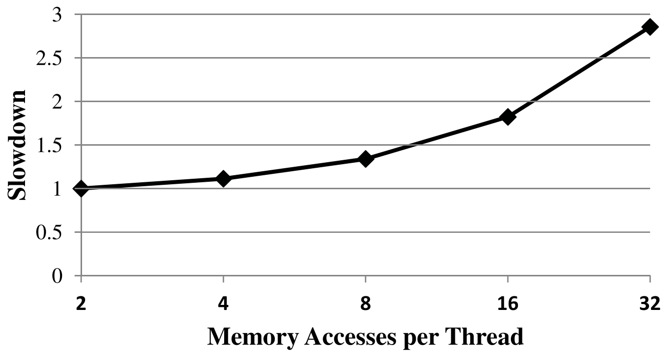
The central idea behind SAGE is to automatically detect and systematically skip or simplify processing of the operations that are particularly expensive to perform on GPUs. To do this, SAGE exploits three specific characteristics of GPUs.

Contention caused by atomic operations has a significant impact on performance. Atomic operations are widely used in parallel sorting and reduction operations [NVIDIA 2013] so that many different threads can update the same memory address in parallel code, as seen in the NVIDIA SDK Histogram application. As the GPU serializes accesses to the same element, performance of atomic instructions is inversely proportional to the number of threads per warp that access the same address. Figure 3(a) shows how the performance of *atomicAdd* decreases rapidly as the number of conflicts per warp increases for the Histogram benchmark. SAGE's first optimization improves performance by skipping atomic instructions with high contention.

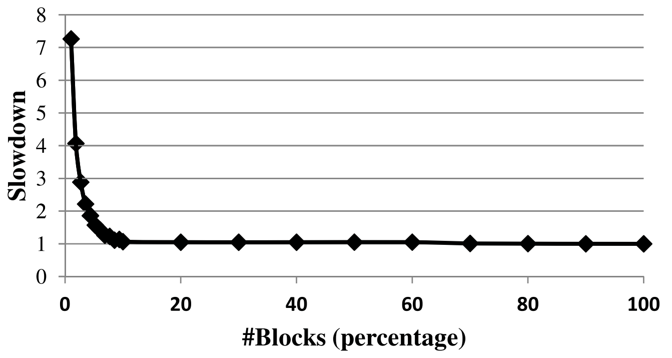
Efficiently utilizing memory bandwidth is essential to improving performance. Considering the large number of cores on a GPU, achieving high throughput often depends on how quickly these cores can access data. Optimizing global memory bandwidth utilization is therefore an important factor in improving performance on a GPU. Figure 3(b) shows the impact of the number of memory accesses per thread on the total performance of a synthetic benchmark. In this example, the number of computational instructions per thread is constant, and only the number of memory accesses per thread is varied. As the relative number of memory accesses increases, performance



(a) High cost of serialization



(b) Memory bandwidth limitation



(c) Diminishing return of multithreading

Fig. 3. Three GPU characteristics that SAGE's optimizations exploit. These experiments are performed on a NVIDIA GTX 560 GPU. (a) The graph shows how accessing the same element by atomic instructions affects the performance for the Histogram kernel. (b) This graph illustrates how the number of memory accesses impacts performance while the number of computational instructions per thread remains the same for a synthetic benchmark. (c) The graph shows how the number of thread blocks impacts the performance of the Blackscholes kernel.

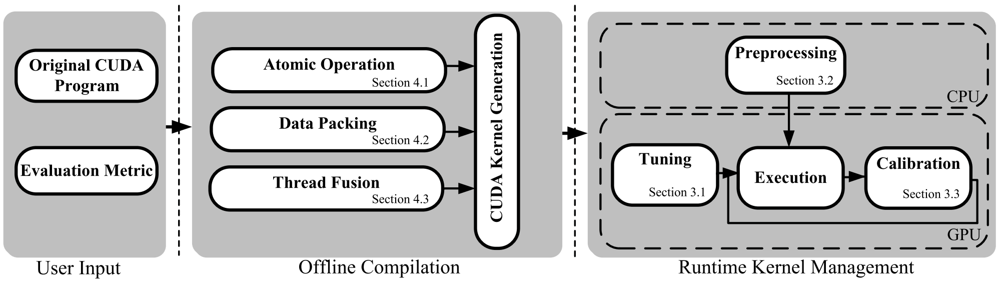


Fig. 4. An overview of the SAGE framework.

deteriorates as the memory bandwidth limitations of the GPU are exposed. The second optimization improves the memory bandwidth utilization by packing the input elements to reduce the number of memory accesses.

As long as there are enough threads, the number of threads does not significantly affect the performance. Since the number of threads running on the GPU is usually more than 10× the number of cores, fewer threads can finish the same job with similar performance. Figure 3(c) illustrates how changing the number of thread blocks in a kernel can affect its performance for the Blackscholes benchmark with 4M options. The baseline is the same kernel using 480 blocks. This figure shows that even 48 blocks (10% of the baseline) can utilize most of the GPU resources and achieve comparable performance. Based on these findings, SAGE's third optimization performs a low overhead thread fusion that joins together adjacent threads. After fusing threads, SAGE computes the output for one of the original, or *active*, threads and broadcasts it to the other neighboring *inactive* threads. By skipping the computation of the inactive threads, SAGE can achieve considerable performance gain.

3. SAGE OVERVIEW

The main goal of the SAGE framework is to trade accuracy for performance on GPUs. To achieve this goal, SAGE accepts CUDA code and a user-defined evaluation metric as inputs and automatically generates approximate kernels with varying degrees of accuracy using optimizations designed for GPUs. The SAGE framework consists of two main steps: offline compilation and runtime kernel management. Figure 4 shows the overall operation of the SAGE compiler framework and runtime.

The *offline compilation* phase investigates the input code and finds opportunities for trading accuracy for performance. This phase automatically generates approximate versions of CUDA kernels using three optimizations that are tailored for GPU-enabled systems. These optimizations systematically detect and skip expensive GPU operations. Each optimization has its own tuning parameters that SAGE uses to manage the performance–accuracy trade-off. These optimizations are discussed in Section 4.

The *runtime management* phase dynamically selects the best approximate kernel whose output quality is better than the user-defined TOQ. The runtime management phase consists of three parts: tuning, preprocessing, and optimization calibration. Using a greedy algorithm, tuning finds the fastest kernel with better quality than the TOQ. The main goal of preprocessing is to make sure that the data needed by these approximate kernels is ready before execution. As the program behavior can change during runtime, SAGE monitors the accuracy and performance dynamically in the calibration phase. If the output quality does not meet the TOQ, calibration chooses a less aggressive approximate kernel to improve the output quality.

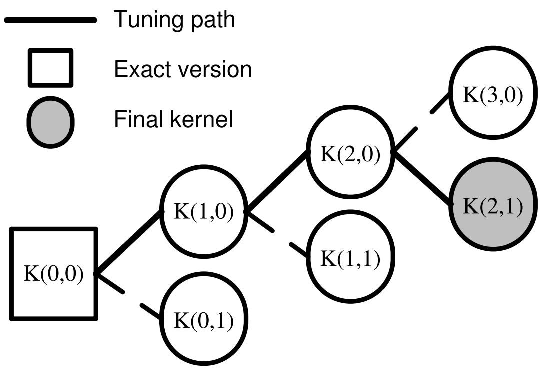


Fig. 5. An example of the tuning process. A node, $K(X, Y)$, is a kernel optimized using two approximation methods. X and Y are the aggressiveness of the first and second optimizations, respectively.

3.1. Tuning

The goal of the tuning phase is to find the fastest approximate kernel whose output quality satisfies the TOQ. Instead of searching all possible configurations, SAGE uses an online greedy tree algorithm to find reasonable approximation parameters as fast as possible to reduce the tuning overhead. Each node in the tree corresponds to an approximate kernel with specific parameters, as shown in Figure 5. All nodes have the same number of children as the number of optimizations used by SAGE, which is two in this example. Each child node is more aggressive than its parent for that specific optimization, which means that a child node has lower output quality than its parent. At the root of the tree is the unmodified, accurate version of the program. SAGE starts from the exact version and uses a steepest-ascent hill climbing algorithm [Russell and Norvig 2009] to reach the best speedup while not violating the TOQ. SAGE checks all children of each node and chooses the one with the highest speedup that satisfies the TOQ. If the two nodes have similar speedups, the node with better output quality will be chosen. This process will continue until tuning finds one of three types of nodes:

(1) A node that outperforms its siblings with an output quality close to the TOQ. Tuning stops when a node has an output quality within an adjustable margin above the TOQ. This margin can be used to control the speed of tuning and how close the output quality is to the TOQ.
(2) A node whose children's output quality does not satisfy the TOQ.
(3) A node whose children have less speedup.

In the example shown in Figure 5, it takes six invocations (nodes) for the tuner to find the final kernel. Once tuning completes, SAGE continues the execution by launching the kernel that the tuner found. SAGE also stores the *tuning path*, or the path from the root to the final node, and uses it in the calibration phase to choose a less aggressive node in case the output quality drops below the TOQ. If this occurs, the calibration phase traverses back along the tuning path until the output quality again satisfies the TOQ.

3.2. Preprocessing

Two of SAGE's optimizations need preprocessing to prepare the data necessary for the generated kernel. For the data packing optimization, preprocessing packs the input data for the next kernel. For the atomic operation optimization, the preprocessor checks input data to predict how much contention occurs during execution of atomic

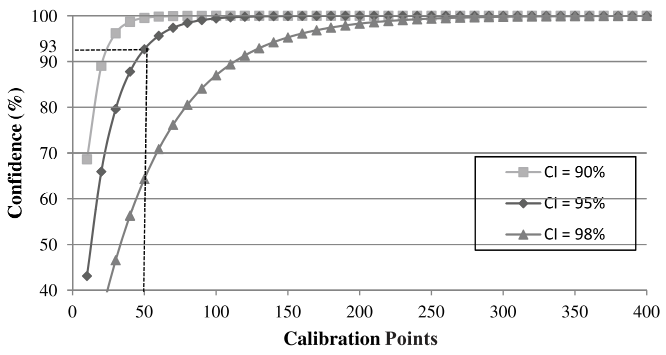


Fig. 6. SAGE's confidence in output quality versus the number of calibrations points for three different confidence intervals (CI).

instructions. Details of preprocessing for these approximation optimizations are described in Section 4.

SAGE runs the preprocessor on the CPU in parallel to GPU execution using synchronous execution. At each time quantum, the GPU runs the selected kernel on a chunk of data while the CPU preprocesses the next chunk before transferring it to GPU memory. This way, preprocessing is completely overlapped by kernel execution and its overhead is negligible.

3.3. Optimization Calibration

As the program behavior can change at runtime, SAGE monitors the accuracy and performance dynamically. After every N invocation of the kernel, the calibration unit runs both the exact and approximate kernels on the GPU to check the output quality and performance. We call N the calibration interval. Computing the output quality is also executed on the GPU in parallel to reduce the overhead of calibration. If the measured quality is lower than the TOQ, SAGE switches to a slower but more precise version of the program. These decisions are based on the tuning path previously described in Section 3.1. By backtracking along the tuning path, SAGE identifies more accurate kernels and calibrates their quality. This process will continue until the output quality satisfies the TOQ. Section 6 discusses different variations of the quality monitoring techniques and their impact on the overall output quality and performance.

Although checking every N^{th} invocation does not guarantee that all invocations satisfy the TOQ, checking more samples will increase our confidence that the quality of the output is acceptable. To compute the confidence, we assume that the prior distribution is uniform. Therefore, the posterior distribution will be $BETA(k+1, n+1-k)$, where n is the number of observed samples and k is the number of samples that satisfies the hypothesis [Tamhane and Dunlop 2000]. In this case, the hypothesis is that the output quality is better than the TOQ. Figure 6 shows how confidence increases as more samples are checked for three different confidence intervals. For example, for a confidence interval equal to 95% and 50 calibration points, confidence is 93%. In other words, after checking 50 invocations, we are 93% confident that more than 95% of the invocations have better quality than the TOQ. If there is an application working on frames of a video at a rate of 33 frames per second and our calibration occurs every 10 kernel invocations, the runtime will be 99.99% confident that more than 95% of output frames will meet the TOQ in less than a minute.

At the beginning of execution, there is low confidence and the runtime management system performs calibration more frequently to converge to a stable solution faster. As confidence improves, the interval between two calibration points is gradually increased so that the overhead of calibration is reduced. Every time the runtime management needs to change the selected kernel, the interval between calibrations is reset to a minimum width and the confidence is reset to zero.

4. APPROXIMATION OPTIMIZATIONS

This section details three GPU optimizations that SAGE applies to improve performance by sacrificing some accuracy: atomic operation, data packing, and thread fusion.

4.1. Atomic Operation Optimization

Idea: An atomic operation is capable of reading, modifying, and writing a value back to memory without interference from any other thread. All threads that try to access the same location are sequentialized to ensure atomicity. Clearly, as more threads access the same location, performance suffers due to serialization. However, if all threads access different locations, there is no conflict and the overhead of the atomic instruction is minimal. This optimization discards instances of atomic instructions with the highest degree of conflicts to eliminate execution segments that are predominantly serial while keeping those with little or no conflicts. As a result of reducing serialization, SAGE can improve performance.

Detection: SAGE first finds all atomic operations inside loops used in the input CUDA kernel and categorizes them based on their loop. For each category, SAGE generates two approximate kernels that will be discussed later.

To make sure that dropping atomic instructions does not affect the control flow of the program, SAGE checks the usage of the output array of atomic operations. It traces the control and data dependence graph to identify the branches that depend on the value of the array. If it finds any, SAGE does not apply this optimization. To detect failed convergence due to dropped atomic operations, a watchdog timer can be instrumented around the kernel launch to prevent infinite loops.

Implementation: The atomic operation optimization performs preprocessing to predict the most popular address for the next invocation of the kernel while the GPU continues execution of the current invocation. To accomplish this, SAGE uses an approach introduced in MCUDA [Stratton et al. 2008] to translate the kernel's CUDA code to C code and then profiles this code on the CPU. To expedite preprocessing, SAGE marks the addresses as live variables in the translated version and performs dead code elimination to remove instructions that are not used to generate addresses. In cases where the GPU modifies addresses during execution, the CPU prediction may be inaccurate. SAGE addresses this by launching a GPU kernel to find the most popular address. The overhead of preprocessing will be discussed in Section 5.

This optimization uses preprocessing results to find the number of conflicts per warp during runtime as follows. First, it uses the CUDA *__ballot* function[1] to determine which threads access the popular address. Next, it performs a population count on the *__ballot*'s result using the *__popc* function[2] to find the number of threads within a warp that access the popular address.

By using runtime conflict detection, the atomic operation optimization generates two types of approximate kernels: ker_M and ker_L. ker_M skips *one* iteration that contains the *most* conflicts. ker_L skips *all* iterations except the one containing the *least* number of conflicts. Both types of kernels contain the code necessary to detect conflicts for each

[1] *__ballot*() takes a predicate as input and evaluates the predicate for all threads of the warp. It returns an integer whose N^{th} bit is set if and only if the predicate is nonzero for the N^{th} thread of the warp [NVIDIA 2013].

[2] *__popc*() sums the number of set bits in an integer input [NVIDIA 2013].

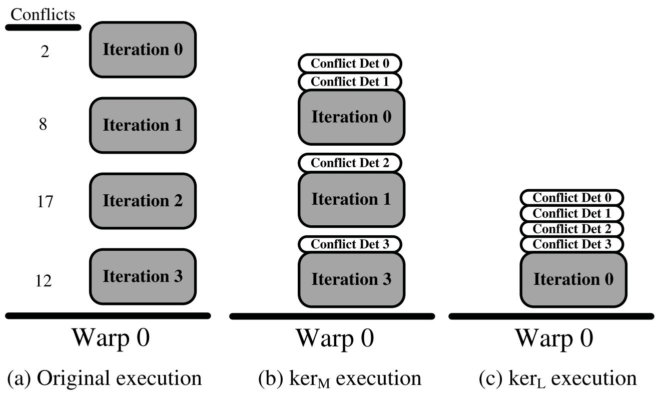


Fig. 7. An illustration of how atomic operation optimization reduces the number of iterations in each thread.

loop iteration at runtime. As all threads within a warp continue to execute the same iterations, no control divergence overhead is added by this optimization. Since ker_L skips more loop iterations than ker_M, ker_L is more aggressive than ker_M.

Figure 7 illustrates how ker_M and ker_L use conflict detection to discard atomic instructions with a large number of conflicts for sample code with four iterations per thread (IPTs). In this example, each iteration contains an atomic operation. The number of conflicts in each iteration is shown on the left in Figure 7(a).

As shown in Figure 7(b), ker_M computes the number of conflicts for the first two iterations and executes the one with fewer conflicts (*Iteration 0*). ker_M continues execution by computing the number of conflicts for *Iteration 2*. Since *Iteration 2* has more conflicts than the previously skipped *Iteration 1*, ker_M executes *Iteration 1* and skips *Iteration 2*. Finally, SAGE executes *Iteration 3* because it has fewer conflicts than *Iteration 2*, which was most recently skipped. At the end of the kernel's execution, ker_M detected and skipped the loop iteration that had the most conflicts (*Iteration 2*) using SAGE's online conflict detection. It accomplished this without needing to run the loop once to gather conflict data, and a second time to apply approximation using this data.

On the other hand, ker_L performs conflict detection by running the original loop without any atomic instructions. This finds the iteration with the minimum number of conflicts per warp. After conflict detection, ker_L executes the found iteration, this time running the atomic instruction. In the example in Figure 7(c), ker_L selected *Iteration 0* after it found that this iteration had the minimum number of conflicts (two).

Parameter tuning: To tune how many atomic instructions ker_M or ker_L skip, SAGE modifies the number of blocks of the CUDA kernel. Equations (1) through (3) show the relationship between the number of blocks and the percentage of skipped instructions for both ker_M and ker_L. Since the number of threads per block (TPBs) is usually constant, if the total number of iterations (the trip count of the loop) is constant, more blocks will increase the number of threads and reduce the number of IPTs, which can be derived from Equation (1). A lower number of IPTs results in more dropped iterations, which can be computed by Equation (2). However, even with the highest possible number of blocks (two IPTs), the dropped percentage of iterations is at most 50% for ker_M. To discard more than 50% of iterations, tuning switches to ker_L. With this kernel, the dropped iteration percentage can go from 50% to near 100%, which can be computed using Equation (3). Figure 8 shows how this optimization affects the percentage of dropped iterations by varying the number of blocks per kernel for two different input

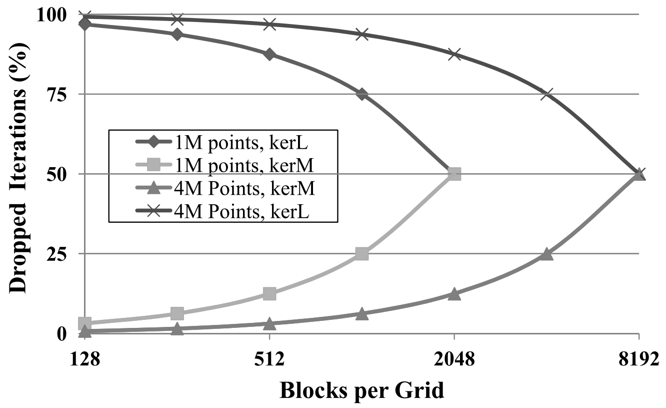


Fig. 8. SAGE controls the number of dropped iterations using Equations (1) through (3) by changing the number of blocks for one and four million data points. ker_M drops only one iteration per thread, and ker_L executes only one iteration per thread. In this case, the threads per block (TPBs) set to 256.

sizes.

$$total = IPT \times TPB \times Blocks \tag{1}$$

$$skipped\_ker_M = \frac{1}{IPT} \times TotalIts \tag{2}$$

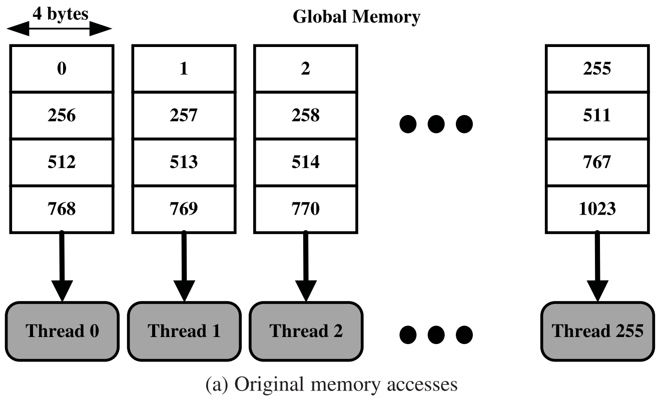
$$skipped\_ker_L = \frac{IPT-1}{IPT} \times TotalIts \tag{3}$$

4.2. Data Packing Optimization

Idea: In GPUs, memory bandwidth is a critical shared resource that often throttles performance, as the combined data required by all of the threads often exceeds the memory system's capabilities. To overcome this limitation, the data packing optimization uses a lossy compression approach to sacrifice the accuracy of input data to lower the memory bandwidth requirements of a kernel. SAGE accomplishes this by reducing the number of memory accesses by packing the input data, thereby accessing more data with fewer requests but at the cost of more computation. This optimization packs the read-only input data in the preprocessing phase and stores it in the global memory. Each thread that accesses the global memory is required to unpack the data first. This approach is more beneficial for iterative applications that read the same array in every iteration. Most iterative machine learning applications perform the same computation on the same data repeatedly until convergence is achieved.

Unlike other approximate data type techniques used for CPUs that are implemented in hardware and target computations [Sampson et al. 2011], this software optimization's goal is to reduce the number of memory requests with an overhead of a few additional computation instructions. All computations are done with full precision after unpacking. The added computation overhead is justifiable because for most GPU kernels that are memory bound, it is more beneficial to optimize memory accesses at the cost of a few extra computation instructions than to optimize the computation of the kernel.

Detection: To apply this optimization, SAGE finds the read-only input arrays of kernels. As unpacking occurs in each thread, the memory access pattern must be known statically so that SAGE is able to pack the data before the kernel executes. In many applications that operate on a matrix, each thread is working on the columns/rows of



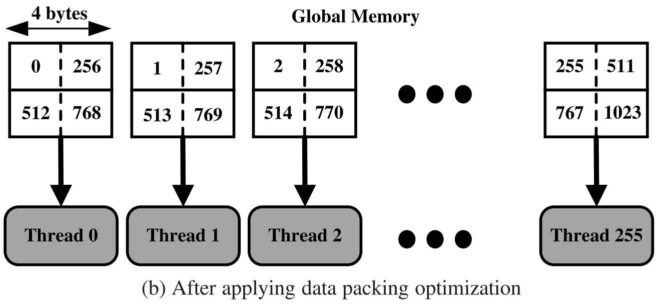


Fig. 9. An example of how the data packing optimization reduces the number of global memory accesses.

the input matrix. Therefore, SAGE packs the columns/rows of the input matrix, and each thread must unpack a column/row before performing the computation. For each candidate input array, SAGE generates an approximate kernel.

It is possible that this optimization causes a divide by zero situation, as the least significant bits are truncated. However, the GPU does not throw divide by zero exceptions. Rather, it produces a large number as the result. Therefore, the program will continue without stopping, and only the output quality may be affected.

Implementation: This optimization performs a preprocessing step that normalizes the data in the input matrix to the range [0,1). The scaling coefficients used are stored in the constant memory of the GPU.

After deciding the number of quantization bits (q_bits), the preprocessor packs $ratio(= \frac{number\ of\ bits\ per\ int}{q\_bits})$ number of floats into one unsigned integer. The packing process is done by keeping the most significant q_bits of each float and truncating the rest of the bits. Figure 9(a) shows the original memory accesses before applying the data packing optimization, and Figure 9(b) illustrates an example of packing two floats in the place of one integer. When a GPU thread accesses the packed data, it reads an unsigned integer. Each thread unpacks that integer and rescales the data by using coefficients residing in the constant memory and uses the results for the computation.

Parameter tuning: To control the accuracy of this optimization, SAGE sweeps the number of quantization bits per float from 16 to 2 to change the memory access ratio from 2 to 16.

4.3. Thread Fusion Optimization

Idea: The underlying idea of the thread fusion optimization is based on the assumption that outputs of adjacent threads are similar to each other. For domains such as image or video processing, where neighboring pixels tend to have similar values, this assumption is often true. In this approach, SAGE executes a single thread out of every group of consecutive threads and copies its output to the other inactive threads. By doing this, most of the computations of the inactive threads are eliminated.

Detection: The thread fusion approach works for kernels with threads that do not share data. In kernels that use shared memory, all threads both read and write data to the shared memory. Therefore, by deactivating some of the threads, the output of active threads might be changed as well, and this will result in an unacceptable output quality. Therefore, SAGE uses this optimization for kernels that do not use shared memory.

Implementation: In this approach, one thread computes its result and copies its output to adjacent threads. This data movement can be done through shared memory, but the overhead of sharing data is quite high due to synchronizations and shared memory latency. It also introduces control divergence overhead, and the resulting execution is too slow to make the optimization worthwhile. Instead, SAGE reduces the number of threads through fusion. Fused threads compute the output data for one of the original threads, which are called *active threads*. Fused threads copy the results of the active threads to the *inactive threads*. Since this data movement occurs inside the fused thread, the transferring overhead is much less than that of using shared memory. However, to copy the data, fused threads should compute output addresses for inactive threads.

To fuse threads, SAGE translates the block ID and thread ID of the fused threads to use in the active threads. For inactive threads, SAGE walks back up the use-def chain to mark instructions that are necessary to compute the output index. Fused threads compute these instructions for all inactive threads to find which addresses they write to and copy the active thread output values to those addresses.

There are two ways to fuse the threads: one involves reducing the number of TPBs, and the other one involves fusing blocks in addition to threads and reducing the number of blocks of the kernel. Since reducing the number of TPBs results in poor resource utilization, SAGE additionally fuses blocks of each kernel. Figure 10(a) shows the original thread configuration before applying the optimization, and Figure 10(b) shows the thread configuration after fusing two adjacent threads and thread blocks. In the new configuration, each thread computes one output element and writes it to two memory locations. Although the thread fusion optimization reduces the overall computations performed by the kernel, reducing more blocks may result in poor GPU utilization, as shown in Figure 3(c). Therefore, at some point, SAGE stops the fusion process, as it eventually leads to slowdown.

Parameter tuning: SAGE changes the number of threads that are fused together to control performance and output accuracy.

5. EXPERIMENTAL EVALUATION

In this section, we show how the optimizations in SAGE affect the execution time and accuracy of different applications. Ten applications from two domains are used: machine learning and image processing. A summary of the application characteristics is shown in Table I. As each optimization targets specific, common performance bottlenecks of GPU applications, each application has usually one or two bottlenecks that SAGE optimizes, as described in Table I.

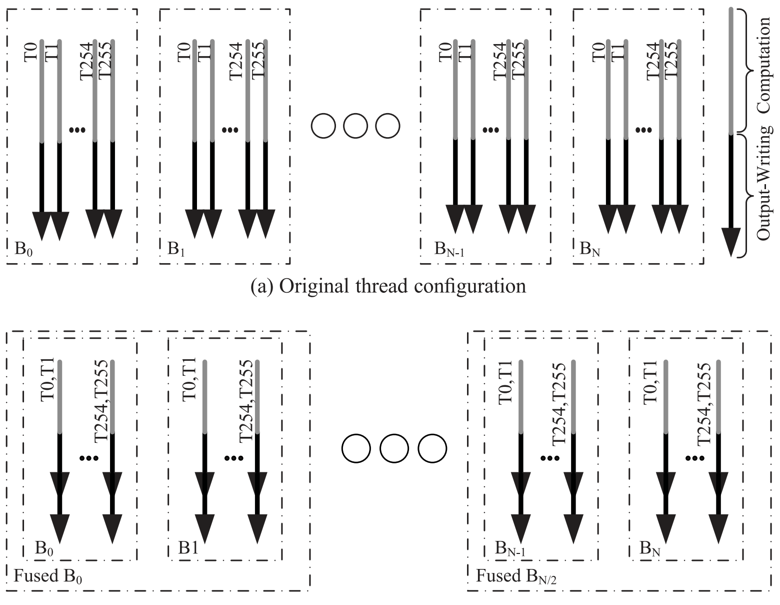


Fig. 10. The thread fusion optimization reduces the computation executed by this kernel by fusing two adjacent threads together and broadcasting the single output for both threads.

Table I. Application Specifications

	Domain	Input Data	Approximation Opportunity	Evaluation Metric
K-Means	ML	1M random points, 32 features	Atomic, Packing	Mean relative difference
Naive Bayes	ML	KDD Cup [Frank and Asuncion 2010]	Atomic	Mean relative difference
Histogram	IP	2,048 × 2,048 images	Atomic	Mean relative difference
SVM	ML	USPS [Frank and Asuncion 2010]	Fusion, Packing	Mean relative difference
Fuzzy K-Means	ML	KDD Cup [Frank and Asuncion 2010]	Packing	Mean relative difference
Mean Shift	ML	KDD Cup [Frank and Asuncion 2010]	Packing	Mean relative difference
Image Binarization	IP	2,048 × 2,048 images	Fusion	1 if incorrect, 0 if correct
Dynamic Range Compression	IP	2,048 × 2,048 images	Fusion	Mean pixel difference
Mean Filter	IP	2,048 × 2,048 images	Fusion	Mean pixel difference
Gaussian Smoothing	IP	2,048 × 2,048 images	Fusion	Mean pixel difference

ML, machine learning; IP, image processing.

5.1. Applications

The *Naive Bayes Classifier* is based on the Bayesian theorem. The training process is done by counting the number of points in each cluster and the number of different feature values in each cluster. To implement Naive Bayes Classifier training, we divide the data points between the threads, and each thread uses an *atomicInc* operation to compute number of points in each cluster. This implementation is based on OptiML's implementation [Sujeeth et al. 2011].

K-Means is a commonly used clustering algorithm used for data mining. This algorithm has two steps that are iteratively executed. The first step computes the centroids of all clusters. The second step finds the nearest centroid to each point. We launch one kernel to compute all centroids. Each thread processes a chunk of data points, and each block has an intermediate sum and the number of points for all clusters. Atomic instructions are used in this kernel because different threads may update the same cluster's sum or number. After launching this kernel, a reduction operation adds these intermediate sums and counters to compute centroids. As each iteration reads the same input data and changes the centroids based on that, SAGE applies both atomic operation and data packing optimizations to this benchmark.

Support vector machines (SVMs) are used for analyzing and recognizing patterns in the input data. The standard two-class SVM takes a set of input data and for each input predicts which class it belongs to from the two possible classes. We used the implementation of Catanzaro et al. [2008] for this application. *Fuzzy K-Means* is similar to K-Means clustering; however, in fuzzy clustering, each point has a degree of belonging to clusters rather than belonging completely to just one cluster. Unlike K-Means, cluster centroids are a weighted average of all data points. Therefore, there is no need to use atomic operations.

Mean Shift Clustering is a nonparametric clustering that does not need to know the number of clusters a priori. The main idea behind this application is to shift the points toward local density points at each iteration. Points that end up in approximately the same place belong to the same cluster. *Histogram* is one of the most common kernels used in image processing applications, such as histogram equalization for contrast enhancement or image segmentation. Histogram plots the number of pixels for each tonal value.

Image Binarization converts an image to a black and white image. This application is used before optical character recognition. *Dynamic Range Compression* increases the dynamic range of the image. *Mean Filter* is a smoothing filter used to reduce noise in images. It smoothes an image by replacing each pixel with the average intensity of its neighbors. *Gaussian Smoothing* is another smoothing filter used to blur images. We used the texture memory to store the input image to improve the performance of these two applications.

5.2. Methodology

The SAGE compilation phases are implemented in the backend of the Cetus compiler [Lee et al. 2003]. We modified the C code generator in Cetus to read and generate CUDA code. SAGE's output codes are compiled for execution on the GPU using NVIDIA nvcc 4.0. GCC 4.4.6 is used to generate the $\times 86$ binary for execution on the host processor. The target system has an Intel Core i7 CPU and an NVIDIA GTX 560 GPU with 2GB GDDR5 global memory.

Output quality: To assess the quality of each application's output, we used an application-specific evaluation metric as shown in Table 1. Since SAGE uses an online calibration, it is limited to a computationally simple metric that minimizes the overhead. In addition, the chosen metric is normalized (0% to 100%) so that we can present results that can easily be compared to one another. It is very easy to modify SAGE so that different metrics, like PSNR or MSE, can be used. In all cases, we compare the output of the original application to the output of the approximate kernel.

Based on a case study by Misailovic et al. [2010], the preferred quality loss range is between 0% and 10% for applications such as video decoding. Other works [Sampson et al. 2011; Baek and Chilimbi 2010; Esmaeilzadeh et al. 2012b] have benchmarks that have quality loss of around 10%. We also used the LIVE image quality assessment database [Sheikh et al. 2006a, 2006b] to verify this threshold. Images in this database have different levels of distortion by white noise and were evaluated by 24 human subjects. The quality scale is divided into five equal portions: Bad, Poor, Fair, Good, and Excellent. We measured output quality of images used in the LIVE study with our evaluation metric. The results show that more than 86% of images with quality loss of less than 10% were evaluated as Good or Excellent by human subjects in the LIVE study. Therefore, we used 90% as the TOQ in our experiments. We also perform our experiments with 95% target quality to show how SAGE trades off accuracy for performance per application.

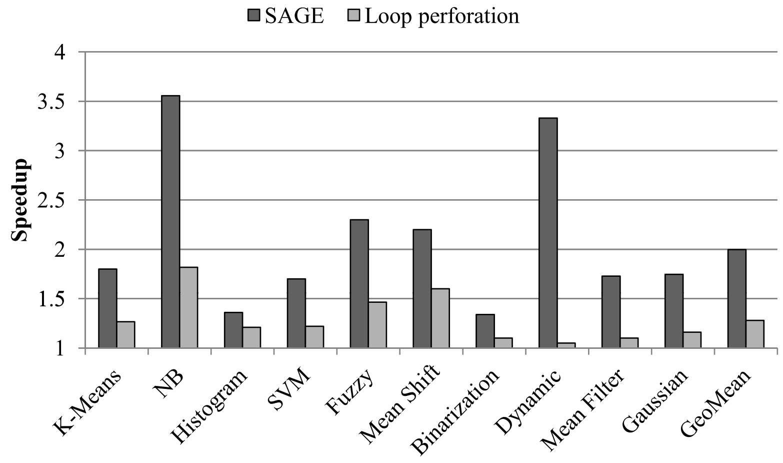
Loop perforation: SAGE optimizations are compared to another well-known and general approximation approach, loop perforation [Agarwal et al. 2009], which drops a set of iterations of a loop. For atomic operation and data packing optimizations, we drop every N^{th} iteration of the loops. For thread fusion optimization, dropping the N^{th} thread results in poor performance due to thread divergence. Instead, we dropped the last N iterations to avoid such divergence. We changed N to control the speedup and output quality generated by loop perforation. The loop perforation technique is only applied to loops that are modified by SAGE to evaluate the efficiency of SAGE's optimizations. Additionally, it should be noted that loop perforation and data packing are orthogonal approaches and can be used together.

5.3. Performance Improvement

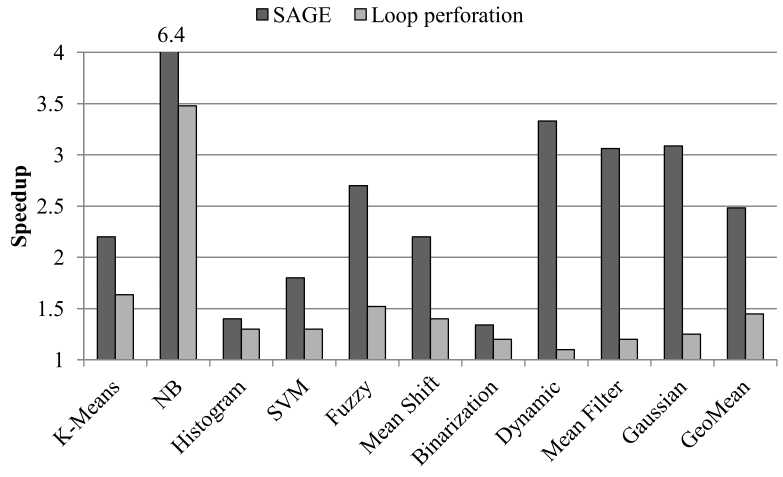
Figure 11(a) and (b) show the results for all applications with a TOQ of 95% and 90%, respectively. Speedup is compared to the exact execution of each program. As computing centroids in K-Means is done by averaging over points in that cluster, ignoring some percentage of data points does not dramatically change the final result. However, since computing the centroids is not the dominant part of K-Means, the K-Means application achieves better performance by using the data packing optimization rather than the atomic operation optimization. In Section 5.4, we show how SAGE gets better speedup by combining these optimizations.

For the Naive Bayes Classifier, computing probabilities is similar to averaging in K-Means. Since the atomic instructions take most of the execution time in this application, SAGE gets a large speedup by using this approximation optimization. On the other hand, loop perforation proportionally decreases the output quality. By decreasing the TOQ from 95% to 90%, the speedup increased from 3.6× to 6.4×.

For the Histogram application, although most of the execution time is dedicated to atomic operations, SAGE gets a smaller speedup than K-Means. The reason is that unlike K-Means, Histogram does not have similar averaging to compute the results, and dropping data points directly affects the output quality. Therefore, quality loss is increased rapidly by dropping more data, and as a result, the speedup is only 1.45× for 90% TOQ. The data packing optimization shows strong performance for memory-bound applications such as Fuzzy K-Means. Fuzzy K-Means is one of the more error-tolerant applications, and ignoring half of the input bits does not affect the output quality significantly. SVM also shows good speedup when using 16 bits and 8 bits per float, but the quality drops below the TOQ at 4 bits per float. For Mean Shift, the speedup does



(a) TOQ = 95%



(b) TOQ = 90%

Fig. 11. Performance for all applications approximated using SAGE compared to the loop perforation technique for two different TOQs. The results are relative to the accurate execution of each application on the GPU.

not increase with more packing. Therefore, the speedup of SAGE is similar for both 90% and 95% TOQ.

SAGE uses the thread fusion optimization for four applications: Dynamic Range Compression, Image Binarization, Mean Filter, and Gaussian Smoothing. We used 2,048 × 2,048 pixel images to compute the quality for these applications. Dynamic Range Compression and Image Binarization performances are reduced after fusing more than four threads. This is mainly because of the memory accesses and fewer numbers of blocks needed to fully utilize the GPU. Therefore, tuning stops increasing the aggressiveness of the optimization because it does not provide any further speedup. As seen in the figures, these two applications show the same speedup for both quality targets. However, for Mean Filter and Gaussian Smoothing, increasing the number of fused threads results in better performance. By decreasing the TOQ, the speedup of Mean Filter goes from 1.7× to 3.1×.

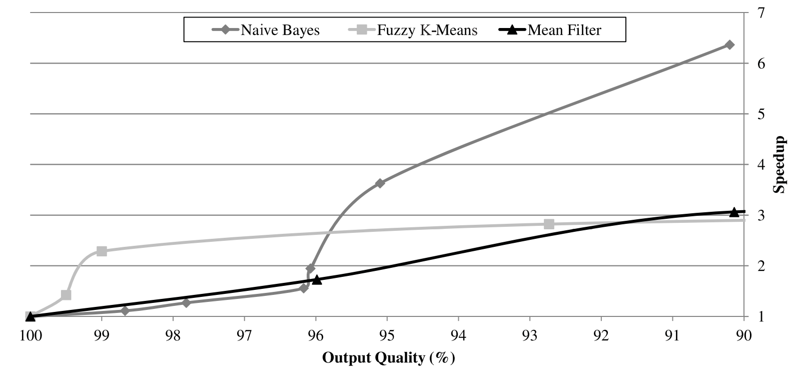


Fig. 12. Performance-accuracy curves for three sample applications. The atomic operation optimization is used for Naive Bayes Classifier. The data packing is used for Fuzzy K-Means application, and the thread Fusion is applied to Mean Filter.

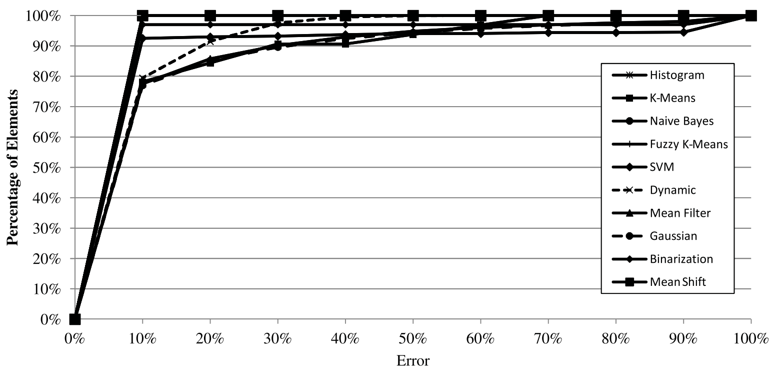
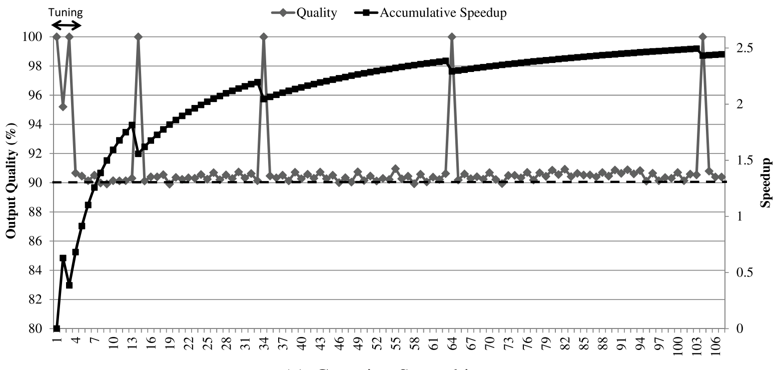


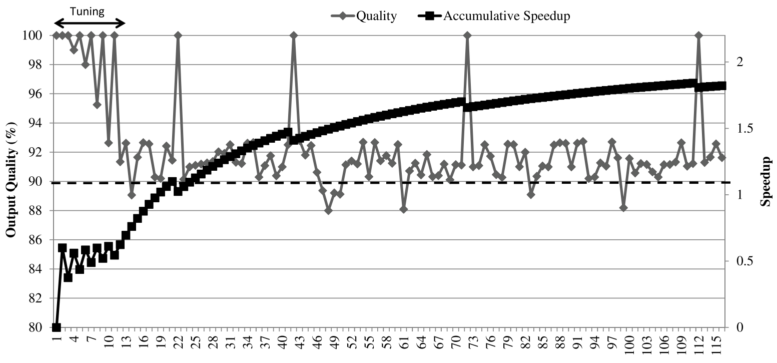
Fig. 13. CDF of final error for each element of all applications' output with the TOQ equal to 90%. Most output elements (more than 78%) have less than 10% error.

Performance-accuracy curve: Figure 12 shows a performance-accuracy curve for three sample applications to show how SAGE manages the speedup-accuracy tradeoff. Here, the atomic operation optimization is used for Naive Bayes. As the percentage of dropped iterations is increased, both quality loss and speedup are increased. Since SAGE changes the approximate kernel from $kernel_{max}$ to $kernel_{min}$, there is a performance jump between 96% and 95% output qualities. SAGE uses the data packing optimization for Fuzzy K-Means and controls the speedup by changing the number of floats that are packed. The performance-accuracy curve for Mean Filter is also shown in Figure 12 to show how the thread fusion optimization impacts speedup and quality. By fusing more threads, both speedup and quality loss are increased. After fusing more than eight threads, speedup decreases because of poor GPU utilization.

Error distribution: To study the application-level quality loss in more detail, Figure 13 shows the cumulative distribution function (CDF) of final error for each element of the application's output with a TOQ of 90%. The CDF shows the distribution of output errors among an application's output elements and shows that only a modest number of output elements see large error. The majority (78% to 100%) of each transformed application's output elements have an error of less than 10%. As can be seen in this figure, for Image Binarization, most of the pixels have 0% error but others



(a) Gaussian Smoothing



(b) K-Means

Fig. 14. Performance and output quality for two applications for 100 invocations with different input sets. The horizontal dashed line represents the TOQ.

have 100% error. These pixels correspond to the edges of objects in the picture where adjacent threads outputs are dissimilar.

5.4. Case Studies

This section describes how SAGE uses the tuning and calibration phase to control the accuracy and performance for two example applications. In both cases, the tuning margin is set to 1%, which means that tuning stops if it finds a kernel with output quality 1% better than the TOQ. In these examples, we assumed that we have enough confidence about the output quality to show how the calibration interval changes during runtime.

In the first example, we run the Gaussian Smoothing application on 100 consecutive frames of a video. Figure 14(a) shows the accumulative speedup and accuracy for this example. For this experiment, we assume that the TOQ is 90%. SAGE starts with the exact kernel and increases the aggressiveness of the optimization until the output quality is near 90%. To compute the output quality for tuning, SAGE runs the approximate and exact versions one after the other. Therefore, during tuning, the output quality that the user observes is 100%, and there is a temporary slowdown. As seen in Figure 14(a), it takes three different invocations to tune this application. After tuning, SAGE uses the final kernel that is found by tuning to continue the execution, and the speedup starts to increase. The initial interval between two calibrations is set to 10 invocations. SAGE runs both versions (exact and approximate) to compute the output quality for calibration. The first calibration happens at the 10th invocation after tuning, and the output quality is now better than the TOQ. Therefore, there is no need to change the kernel. As shown in the figure, each calibration has a negative impact on the overall performance. Therefore, after each calibration where output quality is better than TOQ, SAGE increases the interval between two consecutive calibrations to reduce the calibration overhead.

The second example is K-Means. We run K-Means with 100 random input datasets of 1M points each and 32 dimensions. Each data set has 32 clusters with a random radius, and one of the clusters is dominant. Figure 14(b) shows the accuracy and speedup for all invocations. K-Means starts with the exact kernel, after which SAGE increases the aggressiveness of both optimizations: atomic operation and data packing. Since data packing is more effective than atomic operation, SAGE continues tuning the kernel by packing two floats. Again, SAGE checks both child nodes, and packing still provides the best speedup for the next tuning level. At the end of tuning, packing more data does not further improve performance. Therefore, SAGE increases the dropped iterations by using the atomic operation optimization. Tuning is stopped because the output quality is between 91% and 90%. As seen in the figure, it takes six different invocations to tune. The first calibration happens 10 invocations after tuning is finished. In subsequent calibrations, accuracy is in the margin of the TOQ, and SAGE begins to gradually increase the interval between two calibrations.

As mentioned in Section 3.3, SAGE does not guarantee that output quality is always better than the TOQ. As seen in Figure 14, at some point, quality drops below 90%. This is because we sample invocations for calibration.

5.5. Runtime Overhead

Preprocessing overhead: For the data packing optimization, SAGE transfers the packed data instead of the actual data. For the atomic optimization, preprocessing sends the most popular address as a new argument to the approximate kernel. Therefore, there is no additional transferring overhead for these two optimizations.

Since preprocessing is done on the CPU in parallel to GPU execution, the overhead is negligible for all benchmarks except K-Means. In K-Means, addresses used by atomic operations are changed dynamically during GPU execution, and SAGE finds the most popular address on the GPU. However, the preprocessing overhead is less than 1% to find the cluster with maximum number of points.

Tuning overhead: Tuning overhead depends on how many invocations are needed to tune the application for the specified TOQ. When it is 90%, all of our applications take three to six invocations to tune. These results are considerably better than checking all configurations. For example, searching the whole configuration for K-Means needs 20× more invocations. This gain will be larger for benchmarks with more kernels and approximation opportunities.

Calibration overhead: Calibration overhead is a function of how much speedup SAGE can achieve by using approximation and the interval between two consecutive calibration phases. Equation (4) shows the calibration overhead for calibration interval N. t_e is the execution time of one exact invocation of the application, G is the gain achieved by SAGE using approximation, and t_c is the time that SAGE spends to compute the output quality. Since this quality-checking phase is done in parallel on the GPU, it is negligible compared to the actual execution of the application:

$$overhead_{calibration} = \frac{t_{calibration}}{t_{total}} = \frac{t_e + t_c}{N \times t_e/G + t_e + t_c} \tag{4}$$

Figure 15 illustrates the calibration overhead for the two case studies (TOQ is 90%) for different calibration intervals. This overhead is about 1% for calibration intervals more than 100. Gaussian Smoothing has a higher overhead compared to K-Means

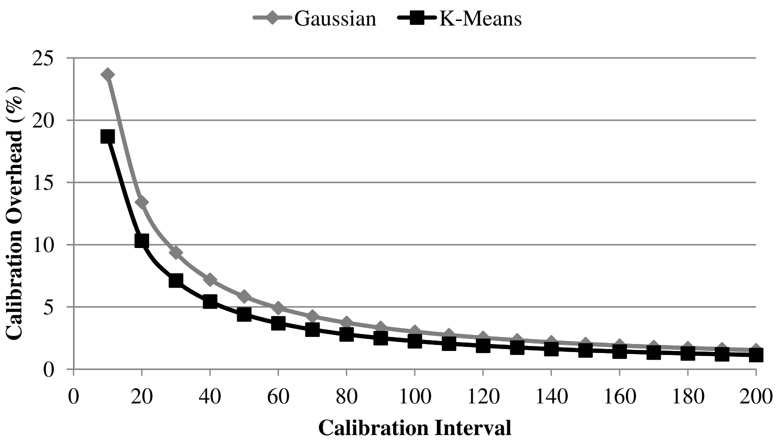


Fig. 15. Calibration overhead for two benchmarks for different calibration intervals.

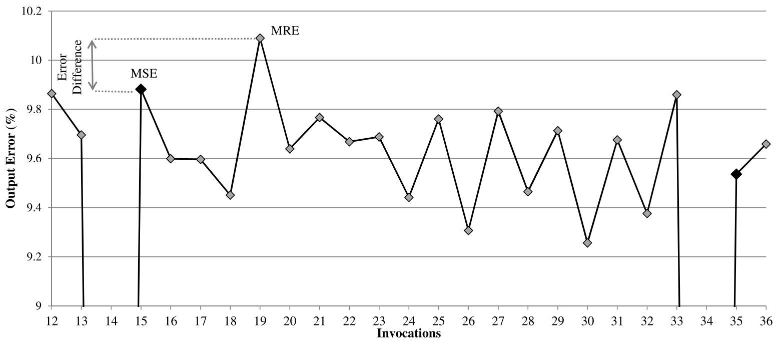


Fig. 16. The difference between MRE and MSE for one calibration interval of Gaussian Smoothing from Figure 14(a). Since SAGE runs the exact version to compare output quality, the output error is zero for invocations 14 and 34.

because SAGE enables a better speedup for Gaussian Smoothing. Therefore, for Gaussian Smoothing, the difference between execution time of the exact and approximate versions is larger.

6. QUALITY MONITORING STUDY

This section describes different variations of the quality monitoring and calibration technique proposed in this article. As mentioned in Section 3, to reduce the calibration overhead, SAGE only checks every N^{th} invocation of the program. This approach works efficiently for applications that have *temporal similarity*, where two consecutive input sets shows similar output quality using the same configuration of approximation methods. One example of this type of application is applying the Gaussian filter on different frames of a video as shown in Section 5.4. *Maximum sampled error* (MSE) is the maximum error that SAGE observes during runtime, and *maximum real error* (MRE) is the actual maximum error. There is a difference between MSE and MRE because SAGE does not check the output quality of all invocations. However, this difference is small for the Gaussian Smoothing application because of its temporal similarity. Figure 16 illustrates the difference between the MSE and the MRE for one calibration interval of Gaussian Smoothing from Figure 14(a). To illustrate this difference for various input sets, we applied Gaussian Smoothing to all frames of 10 different videos. Figure 17

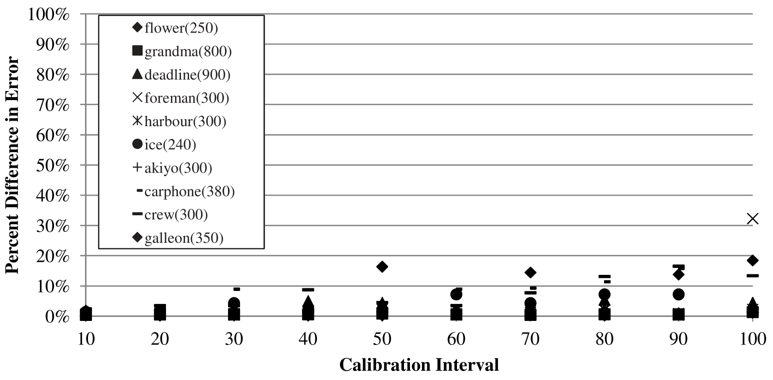


Fig. 17. The percentage difference between MRE and MSE during calibrations for 10 videos. The number of frames is displayed next to the video's name in parentheses.

shows the percentage difference between MSE and MRE. As the interval between two calibrations increases, this difference increases. However, even with a calibration interval of 100 invocations, the difference between these errors is less than 10% for most of the videos.

To show the calibration efficiency, we considered four alternatives to SAGE's calibration technique.

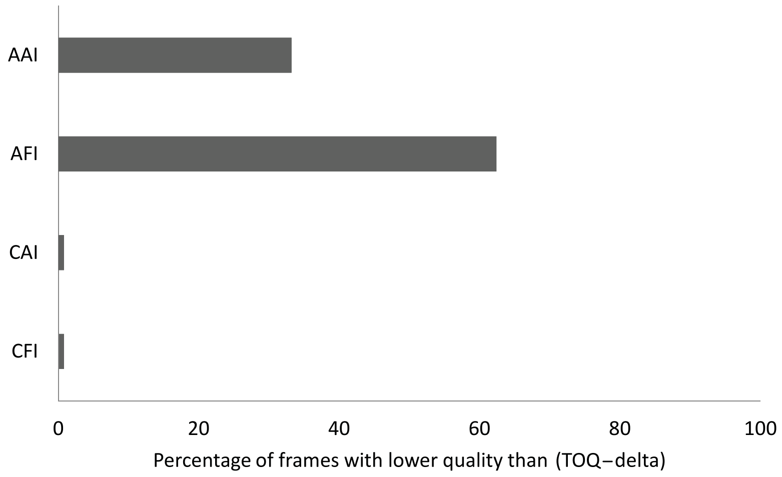
Conservative fixed interval (CFI): This technique checks the output quality every N^{th} invocation. The output quality is computed by running the approximate and exact versions sequentially. After that, the runtime system computes the output quality by comparing the exact and approximate outputs. Since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

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

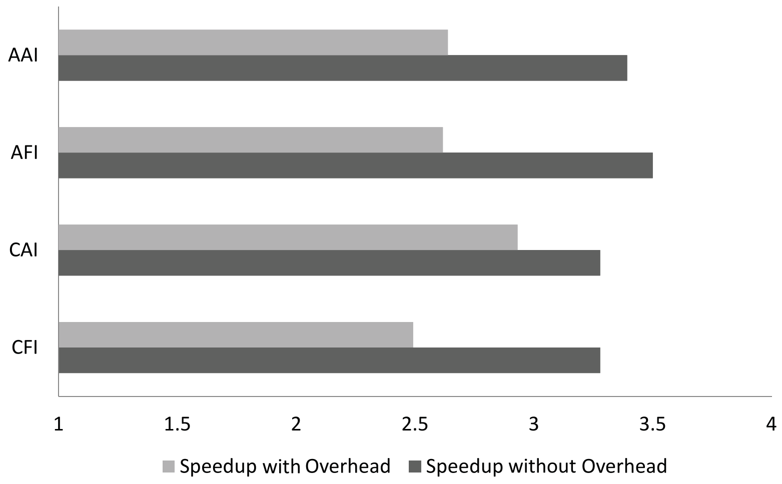
Conservative adaptive interval (CAI): This approach reduces the monitoring overhead of CFI by performing quality checking more frequently to converge to a stable solution faster at the beginning of kernel execution. If the output quality is higher than $TOQ - delta$, the interval between two checking points is gradually increased so that the overhead of quality checking is reduced. Every time the runtime 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. Like CFI, this technique is conservative, so the overall performance might be less than ideal. This approach is the same as the calibration technique used in SAGE if $delta = 0$.

Aggressive fixed interval (AFI): To improve performance, unlike the aforementioned techniques, AFI looks for opportunities to reduce the output quality. 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.

Aggressive adaptive interval (AAI): This technique is similar to AFI but with adaptive intervals. Since this technique is both adaptive and nonconservative, its overall performance should be higher than the last three mentioned techniques.



(a)



(b)

Fig. 18. (a) Percentage of frames with unacceptable quality (lower than $TOQ - delta$) for different quality monitoring techniques. (b) Overall speedup of applying Gaussian filter on all frames of flower video with and without considering the calibration overhead.

To evaluate these four techniques, we applied Gaussian Smoothing on all frames of the flower video. Figure 18(a) shows the percentage of frames for which their output quality is not acceptable (lower than $TOQ - delta$), and Figure 18(b) shows the overall speedup. As expected, the output qualities of the conservative techniques (CFI and CAI) are always better than those of the aggressive techniques. Aggressive techniques provide better speedups if we ignore calibration overhead. However, the quality of more than 30% of frames is not acceptable using such techniques. Since aggressive techniques increase the aggressiveness of the approximation during runtime, the performance for processing each frame is higher compared to the conservative approaches. However, at the same time, aggressive techniques more frequently make incorrect decisions and SAGE calibrates more often to catch and correct these

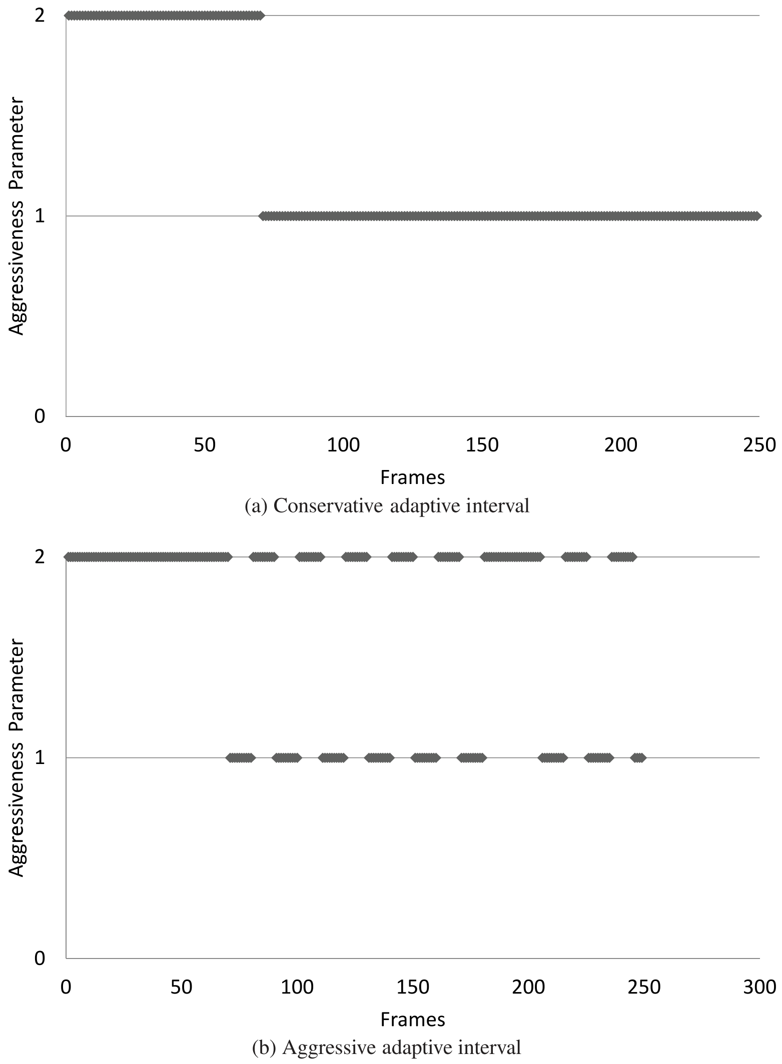


Fig. 19. The aggressiveness of the approximation method used for each frame using two calibration intervals (CAI and AAI).

decisions. Due to the large number of incorrect decisions, aggressive techniques suffer higher calibration costs.

To study the impact of conservative/aggressive methods on performance, Figure 19 shows the aggressiveness parameter that different techniques choose for each frame. Figure 19(a) shows the aggressiveness of approximation when calibrating using the CAI technique. Since this approach is conservative, the speedup just decreases over time, and it will choose a pessimistic approximation method to make sure that fewer frames have unacceptable quality. On the other hand, Figure 19(b) shows the aggressiveness of approximation for the AAI. This technique switches between two most aggressive approximation methods for all of the frames. Therefore, AAI provides better performance than conservative techniques without considering the calibration overhead. However, using this technique, many frames have unacceptable output quality like that shown in Figure 18(a).

These results show that conservative techniques work better than aggressive techniques for applications with temporal similarity. Although aggressive techniques might achieve better performance than conservative techniques, their overall output quality is unacceptable.

7. RELATED WORK

Trading accuracy for other benefits such as improved performance or energy consumption is a well-known technique [Rinard 2007; Agarwal et al. 2009; Sidiroglou-Douskos et al. 2011; Hoffmann et al. 2011; Baek and Chilimbi 2010; Sampson et al. 2011; Ansel et al. 2011; Esmaeilzadeh et al. 2012a; Samadi et al. 2014; Samadi and Mahlke 2014]. Some of these techniques are software based and can be utilized without any hardware modifications [Rinard 2007; Sidiroglou-Douskos et al. 2011; Hoffmann et al. 2011; Agarwal et al. 2009; Sartori and Kumar 2012; Samadi et al. 2014; Samadi and Mahlke 2014]. Agarwal et al. [2009] used code perforation to improve performance and reduce energy consumption. They perform code perforation by discarding loop iterations. Instead of skipping every N^{th} iteration or random iterations, SAGE skips iterations with the highest performance overhead, which results in the same accuracy loss but better performance gain. Rinard [2007] terminates parallel phases as soon as there are too few remaining tasks to keep all of the processors busy. Since there are usually enough threads to utilize the GPU resources, this approach is not beneficial for GPUs. Sartori and Kumar [2012] also use a software approach that targets control divergence that can be added to the SAGE framework. Samadi et al. [2014] introduced pattern-based approximation framework, Paraprox, which detects common patterns in the input data-parallel program and applies pattern-specific approximation methods.

Green [Baek and Chilimbi 2010] is another flexible framework that developers can use to take advantages of approximation opportunities to achieve better performance or energy consumption. The Green framework requires the programmer to provide approximate kernels or to annotate their code using extensions to C and C++. In contrast to these techniques, this work automatically generates different approximate kernels for each application. Another difference is that SAGE's framework is specially designed for GPUs with thousands of threads, and its approximation techniques are specially tailored for GPU-enabled devices. In addition, the process of finding the candidate kernel to execute is different from that of the Green framework. Instead of offline training, SAGE uses an online greedy tree algorithm to find the final kernel more quickly. Ansel et al. [2011] also propose language extensions to allow the programmer to mark parts of code as approximate. They use a genetic algorithm to select the best approximate version to run. Unlike these approaches, Paraprox chooses the approximation optimization based on the patterns detected in the input code and generates approximate versions automatically for each pattern without programmer annotation. Paraprox, however, can be utilized by the runtime systems introduced in these works to optimize performance. Samadi and Mahlke [2014] proposed CPU-GPU collaborative monitoring technique, which predicts the quality for all kernel invocations instead of the time sampling method used in SAGE.

EnerJ [Sampson et al. 2011] uses type qualifiers to mark approximate variables. Using this type system, EnerJ automatically maps approximate variables to low power storage and uses low power operations to save energy. Esmaeilzadeh et al. [2012a] used the same approach to map approximate variables to approximate storage and operations. Whereas these approximate data type optimizations need hardware support, our data packing optimization is applicable to current GPU architectures and does not require any hardware modification. Another work by Esmaeilzadeh et al. [2012b] designs neural processing unit (NPU) accelerators to accelerate approximate programs. Li and Yeung [2007] used approximate computing concept to design a lightweight recovery mechanism. Relax [De Kruijf et al. 2010] is a framework that discards the faulty computations in fault-tolerant applications. Sampson et al. [2013] show how to improve memory array lifetime using approximation.

Finally, there exists a large variety of work that maps machine learning and image processing applications to the GPU, such as SVM [Catanzaro et al. 2008] and K-Means [Li et al. 2010]. OptiML [Sujeeth et al. 2011] is another approach that proposes a new domain-specific language (DSL) for machine learning applications that target GPUs.

Besides approximate systems, there are a number of systems [Ansel et al. 2009; Samadi et al. 2012] that support adaptive algorithm selection to evaluate and guide performance tuning. PetaBricks [Ansel et al. 2009] introduces a new language and provides compiler support to select among multiple implementations of algorithms to solve a problem. Adaptic [Samadi et al. 2012] is a compiler that automatically generates different kernels based on the input size for GPUs.

8. CONCLUSION

Approximate computing, where computation accuracy is traded for better performance or higher data throughput, provides an efficient mechanism for computation to keep up with exponential information growth. For several domains, such as multimedia and learning algorithms, approximation is commonly used today. In this work, we proposed the SAGE framework for performing systematic runtime approximation on GPUs.

Our results demonstrate that SAGE enables the programmer to implement a program once in CUDA and, depending on the TOQ specified for the program, automatically trade accuracy for performance. Across 10 machine learning and image processing applications, SAGE yields an average of 2.5× speedup with less than 10% quality loss compared to the accurate execution on a NVIDIA GTX 560 GPU. This article also shows that there are GPU-specific characteristics that can be exploited to gain significant speedups compared to hardware-incognizant approximation approaches. We also discussed how SAGE controls the accuracy and performance at runtime using optimization calibration in two case studies.

ACKNOWLEDGMENTS

Much gratitude goes to the anonymous referees who provided excellent feedback on this work.

REFERENCES

Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffmann. 2009. *Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures*. Technical Report MIT-CSAIL-TR-2009-042. Massachusetts Institute of Technology, Cambridge, MA. Available at http://hdl.handle.net/1721.1/46709.

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A language and compiler for algorithmic choice. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation*. 38–49.

Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 2011 International Symposium on Code Generation and Optimization*. 85–96.

Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 Conference on Programming Language Design and Implementation*. 198–209.

Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. 2008. Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*. 104–111.

Marc De Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 497–508.

EMC Corporation. 2011. Extracting Value from Chaos. http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012a. Architecture support for disciplined approximate programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–312.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012b. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 45th Annual International Symposium on Microarchitecture*. 449–460.

Andrew Frank and Arthur Asuncion. 2010. UCI Machine Learning Repository. Retrieved July 29, 2014, from http://archive.ics.uci.edu/ml.

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–212.

Alex Kulesza and Fernando Pereira. 2008. Structured learning with approximate inference. In *Advances in Neural Information Processing Systems*. 785–792.

Sang Ik Lee, Troy Johnson, and Rudolf Eigenmann. 2003. Cetus—an extensible compiler infrastructure for source-to-source transformation. In *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing*. 539–553.

Xuanhua Li and Donald Yeung. 2007. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*. 181–192.

You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 2010. Speeding up K-means algorithm by GPUs. In *Proceedings of the 2010 10th International Conference on Computers and Information Technology*. 115–122.

Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE Conference on Software Engineering*. 25–34.

NVIDIA. 2013. NVIDIA CUDA C Programming Guide, Version 5.5. Retrieved July 29, 2014, from https://developer.nvidia.com/cuda-toolkit-55-archive.

Martin Rinard. 2006. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 2006 International Conference on Supercomputing*. 324–334.

Martin C. Rinard. 2007. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Systems and Applications*. 369–386.

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. 2012. Adaptive input-aware compilation for graphics engines. In *Proceedings of the 2012 Conference on Programming Language Design and Implementation*. 13–22.

Mehrzad Samadi, D. 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*. 35–50.

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 International Symposium on Microarchitecture*. 13–24.

Mehrzad Samadi and Scott Mahlke. 2014. CPU-GPU collaboration for output quality monitoring. In *Proceedings of the 1st Workshop on Approximate Computing across the System Stack*. 1–3.

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. *ACM SIGPLAN Notices* 46, 6, 164–174.

Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 25–36.

John Sartori and Rakesh Kumar. 2012. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 427–428.

Hamid R. Sheikh, Muhammad F. Sabir, and Alan C. Bovik. 2006a. A statistical evaluation of recent full reference image quality assessment algorithms. *IEEE Transactions on Image Processing* 15, 11, 3440–3451.

Michael Shindler, Alex Wong, and Adam W. Meyerson. 2011. Fast and accurate k-means for large datasets. In *Advances in Neural Information Processing Systems*. 2375–2383.

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*. 124–134.

John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. 2008. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 16–30.

Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. 2011. OptiML: An implicitly parallel domain specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*. 609–616.

Ajit C. Tamhane and Dorothy D. Dunlop. 2000. *Statistics and Data Analysis*. Prentice Hall.

Hamid R. Sheikh, Zhou Wang, Lawrence Cormack, and Alan C. Bovik. 2006b. LIVE Image Quality Assessment Database Release 2. Available at http://live.ece.utexas.edu/research/quality.

Received April 2014; revised April 2014; accepted May 2014