Paraprox: Pattern-Based Approximation for Data Parallel Applications

Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke

Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{mehrzads, ajamshid, jhaeng, mahlke}@umich.edu

Abstract

Approximate computing is an approach where reduced accuracy of results is traded off for increased speed, throughput, or both. Loss of accuracy is not permissible in all computing domains, but there are a growing number of data-intensive domains where the output of programs need not be perfectly correct to provide useful results or even noticeable differences to the end user. These soft domains include multimedia processing, machine learning, and data mining/analysis. An important challenge with approximate computing is transparency to insulate both software and hardware developers from the time, cost, and difficulty of using approximation. This paper proposes a software-only system, Paraprox, for realizing transparent approximation of data-parallel programs that operates on commodity hardware systems. Paraprox starts with a data-parallel kernel implemented using OpenCL or CUDA and creates a parameterized approximate kernel that is tuned at runtime to maximize performance subject to a target output quality (TOQ) that is supplied by the user. Approximate kernels are created 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. Across a set of 13 soft data-parallel applications with at most 10% quality degradation, Paraprox yields an average performance gain of 2.7x on a NVIDIA GTX 560 GPU and 2.5x on an Intel Core i7 quad-core processor compared to accurate execution on each platform.

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

General Terms Design, Performance

Keywords Approximation; Accuracy-aware computing; Data parallel; GPU

ASPLOS'14, March 01–05 2014, Salt Lake City, UT, USA.

ACM 978-1-4503-2305-5/14/03. . . $15.00.
http://dx.doi.org/10.1145/2541940.2541948

1. Introduction

Over the past few years, the information technology industry has experienced a massive growth in the amount of data that it collects from consumers. Analysts reported that in 2011 alone the industry gathered a staggering 1.8 zettabytes of information, and they estimate that by 2020, consumers will generate 50 times this figure [10]. Most major businesses that host such large-scale data-intensive applications, including Google, Amazon, and Microsoft, frequently invest in new, larger data centers containing thousands of multi-core servers. However, it seems that such investments in new hardware alone may not translate to the computation capability required to keep up with the deluge of data. Rather, it may be necessary to consider using alternative programming models that exploit the data parallel computing abilities of existing servers in order to address this problem. This paper focuses on applying one such model, approximate computing, where the accuracy of results is traded off for computation speed, to solve the problem of processing big data.

Approximation is applicable in domains where some degree of variation or error can be tolerated in the result of computation. For domains where some loss of accuracy during computation may cause catastrophic failure, e.g. cryptography, approximation should not be applied. However, there are many important domains where approximation can greatly improve application performance, including multimedia processing, machine learning, data analysis, and gaming. Video processing algorithms are prime candidates for approximation as occasional variation in results do not cause the failure of their overall operation. For example, a consumer using a mobile device can tolerate occasional dropped frames or a small loss in resolution during video playback, especially when this allows video playback to occur seamlessly. Machine learning and data analysis applications also provide opportunities to exploit approximation to improve performance, particularly when such programs are operat-

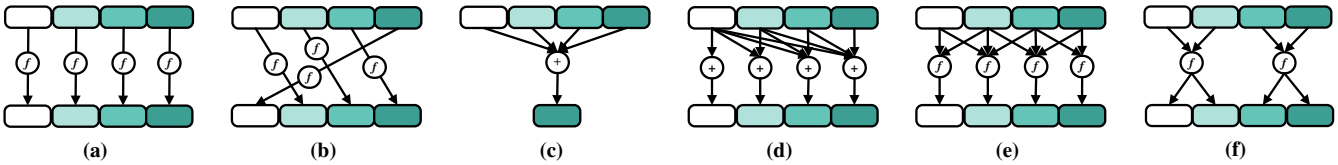

Figure 1: The data parallel patterns that Paraprox targets: (a) Map (b) Scatter/Gather (c) Reduction (d) Scan (e) Stencil (f) Partition.

ing on massive data sets. In this situation, processing the entire dataset may be infeasible, but by sampling the input data, programs in these domains can produce representative results in a reasonable amount of time.

Improving performance by applying approximation has been identified as an important goal by prior works [2, 4, 5, 11, 12, 23, 25, 28]. These works have studied this topic and proposed new programming models, compiler systems, and runtime systems to systematically manage approximation. However, these approaches have three critical limitations. We categorize the prior works based on these limitations:

- **Programmer-based [4, 5]:** In these systems, the programmer must write different approximate versions of a program and a runtime system decides which version to run. Although the programmer may best understand how his code works, writing different versions of the same program with varying levels of approximation is neither easy nor practical to be applied generally.
- **Hardware-based [11, 12, 28]:** These approaches introduce hardware modifications such as imprecise arithmetic units, register files, or accelerators. Although these systems work for general algorithms, they cannot be readily utilized without manufacturing new hardware. Furthermore, having both exact and approximate versions of the same hardware increases the hardware design complexity and the difficulty of validating and verifying such hardware.
- **Software-based [2, 23, 25, 27]:** Previous software-based approximation techniques do not face the problems of the other two categories as they (a) remove the burden of writing several versions of the program from the programmer, and (b) can be used with existing, commodity systems. However, with past approaches, *one solution does not fit all applications*. Each of these solutions works only for a small set of applications. They either cannot achieve a desired amount of performance improvement or generate unacceptable computation errors for applications that they were not explicitly built to handle.

To address these issues, this paper proposes a software framework called *Paraprox*. Paraprox identifies common patterns found in data-parallel programs and uses a custom-designed approximation technique for each detected pattern. Paraprox enables the programmer to write software once and run it on a variety of modern processors, without manually tuning code for different hardware targets. It is applicable to a wide range of applications as it determines the proper approximation optimizations that can be applied to each input program. Because Paraprox does not apply a single solution to all programs, it overcomes the aforementioned limitation of prior software-based approaches.

In this work, we identify different patterns commonly found in data parallel workloads and we propose a specialized approximation optimization for each pattern. We closely study data parallel programs because they are well-fitted for execution on prevalent multi-core architectures such as CPUs and GPUs. Paraprox is capable of targeting any data parallel architecture, provided that the underlying runtime supports such hardware.

Overall, Paraprox enables 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. To control the efficiency, accuracy, and performance of the system, each optimization allows some variables to be dynamically varied. After Paraprox generates the approximate kernels, a runtime system tunes the aforementioned variables to get the best performance possible while meeting the constraints of the TOQ.

To automatically create approximate kernels, Paraprox utilizes four optimization techniques which target six data parallel patterns: Map, Scatter/Gather, Reduction, Scan, Stencil, and Partition. Paraprox applies approximate memoization to map and scatter/gather patterns where computations are replaced by memory accesses. For reduction patterns, Paraprox uses sampling plus adjustment to compute the output by only computing the reduction of a subset of the data. The stencil & partition approximation algorithm is based on the assumption that adjacent locations in an input array are typically similar in value for such patterns. Therefore, Paraprox accesses a subset of values in the input array and replicates that subset to construct an approximate version of the array. For scan patterns, Paraprox only performs the scan operation on a subset of the input array and uses the results to predict the results for the rest of the array.

The specific contributions of this work are as follows:

- Pattern based compilation system for approximate execution.
- Automatic detection of data parallel patterns in OpenCL and CUDA kernels.
- Four pattern-specific approximation optimizations which are specifically designed for six common data parallel computation patterns.
- The ability to control performance and accuracy tradeoffs for each optimization at runtime using dynamic tuning parameters.

The rest of the paper is organized as follows. Section 2 explains how the Paraprox framework operates. Approximate optimizations used by Paraprox are discussed in Section 3. The results of using Paraprox for various benchmarks and architectures are presented in Section 4. Limitations of Paraprox's framework are discussed in Section 5. Section 6 discusses the related work in this area and how Paraprox is different from previous work. Section 7 concludes this paper and summarizes its contributions and findings.

2. Paraprox Overview

In order to generate approximate programs, Paraprox must detect data parallel patterns for optimization. As shown in Figure 1, these patterns have distinct characteristics that require specialized optimizations in order to create fast, approximate versions. In the following list, we describe the characteristics of the six patterns that Paraprox targets:

- ***Map:*** In the map pattern, a function operates on every element of an input array and produces one result per element as shown in Figure 1(a). To process all the input elements in parallel, a map function should be *pure*. A pure function always generates the same result for the same input, and its execution does not have any side-effects, e.g., it cannot read or write mutable state. Since there is no need to synchronize between two threads and no sharing of data is necessary, the map pattern is perfectly matched to data parallel, many-core architectures. In parallel implementations of map patterns, each thread executes one instance of a map function and generates its corresponding result. This pattern is used in many domains, including image processing and financial simulations.
- ***Scatter/Gather:*** Scatter and gather patterns are similar to map patterns but their memory accesses are random as illustrated in Figure 1(b). Based on McCool's definition [15], scatter is a map function that writes to random locations, and gather is the combination of a map function with memory accesses that read from random input elements. The parallel implementations of scatter/gather patterns are similar to map implementations. This pattern is commonly found in statistics applications.
- ***Reduction:*** When a function combines all the elements of an input array to generate a single output, it is said to be performing a reduction (Figure 1(c)). If the function used by the reduction pattern is both associative and commutative, e.g., XOR, the order in which the reduction operation is applied to its inputs is unimportant. In this case, tree-based implementations can be used to parallelize such a reduction. Reductions can be found in many domains, such as machine learning, physics, and statistics.
- ***Scan:*** The all-prefix-sums operation, more commonly termed *scan*, applies an associative function to an input array and generates another array. Every N^{th} element of the output array is the result of applying the scan function on the first N (inclusive scan) or $N-1$ (exclusive scan) input elements. An inclusive scan example is shown in Figure 1(d). The scan pattern is common in the signal processing, machine learning, and search domains.
- ***Stencil:*** In a stencil pattern, each output element is computed by applying a function on its corresponding input array element and its neighbors as shown in Figure 1(e). This pattern is common in image processing and physics applications.
- ***Partition:*** The partition (or tile) pattern is similar to the stencil pattern. The input array is divided into partitions and each partition is processed separately. Each partition is wholly independent of the others as shown in Figure 1(f). Partitioning is commonly used in data parallel applications to efficiently utilize the underlying architecture's memory hierarchy to improve performance. This pattern is common in domains such as image processing, signal processing, and physics modeling.

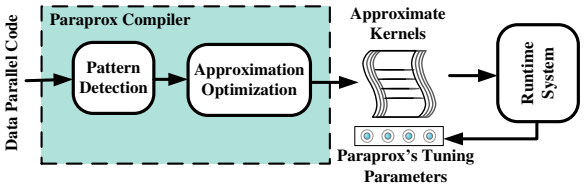

Figure 2: Approximation system framework.

In order to manage the output quality during execution, the Paraprox compilation framework should be used in tandem with a runtime system like Green [5] or SAGE [27] as shown in Figure 2. After Paraprox detects the patterns in the program and generates approximate kernels with different tuning parameters, the runtime profiles the kernels and tunes the parameters so that it provides the best performance. If the user-defined *target output quality (TOQ)* is violated, the runtime system will adjust by retuning the parameters and/or selecting a less aggressive approximate kernel for the next execution.

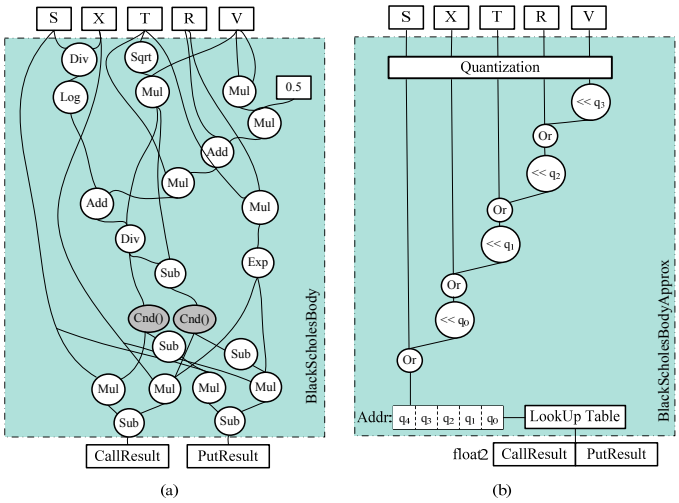

Figure 3: (a) illustrates the dataflow graph of the main function of the *BlackScholes* benchmark. The function $Cnd()$ is a pure function. (b) shows the approximate kernel created using the map and scatter/gather technique described in 3.1.

3. Approximation Optimizations

We will now discuss the approximation optimizations that are applied to each data parallel pattern. For each pattern, we describe the intuition behind the optimization, the algorithm used to detect such a pattern, the implementation of the optimization, and tuning parameters that are used by a runtime to control the performance and accuracy of an approximate kernel during execution.

3.1 Map & Scatter/Gather

3.1.1 Idea:

Paraprox applies *approximate memoization* to optimize map and scatter/gather patterns. This technique replaces a function call with a query into a lookup table which returns a precomputed result. Since the size of this lookup table is limited by the size of memory and by performance considerations, there are situations in which the exact result is not stored in the table. In such cases, Paraprox finds the element nearest to the input present in the lookup table and returns that element instead. Consequently, the quality of the output is inversely proportional to the size of the lookup table (a.k.a. the number of quantization levels). As this optimization replaces the computations done by map and scatter/gather functions with a memory access, the unoptimized code should have more latency due to computation than that of one memory operation in order to achieve speedup.

To fill the lookup table with precomputed data, Paraprox computes the output of the map or scatter/gather function for a number of representative input sets (quantization levels) offline. During runtime, the launched kernel's threads use this lookup table to find the output for all input values.

3.1.2 Detection:

To detect map or scatter/gather patterns, Paraprox checks all functions in the input program to look for functions that can be replaced by a lookup table. There are two requirements for such functions. First, these functions should be pure. Pure functions do not have side effects and their output is only dependent on their inputs. To meet these constraints, pure functions should not:

- read or write any global or static mutable state.
- call an impure function.
- perform I/O.

In addition to being pure, these functions should not access global memory during execution, and their outputs should not be dependent on the thread ID. Therefore, Paraprox looks for functions which do not contain global/shared memory accesses, atomic operations, computations involving thread or block IDs, or calls to impure functions. If a function meets all these conditions, Paraprox marks it as a candidate for approximate memoization.

It should be noted that although Paraprox works at a function granularity, it is possible to find pure sections of code within a function. Detection of such map or scatter/gather sections within a function is left for future research.

As Paraprox will replace computation with memory accesses, this optimization should only be applied to computationally intensive map and scatter/gather patterns in order to achieve high performance improvements. To determine which functions to optimize, Paraprox computes the sum of the latencies of each instruction in the function as a metric to estimate the function's computation cycles as follows:

$$cycles\_needed = \sum_{inst \in f} latency(inst) \quad (1)$$

Instruction latency values are passed to Paraprox in a table based on the target architecture. Paraprox uses this latency table to compute the $cycles\_needed$ for all map and scatter/gather functions found in the program. For GPUs, we used microbenchmarks from Wong et al. [35] to measure the $latency$ of all instructions. We found that if a function's $cycles\_needed$ is at least one order of magnitude greater than the L1 read latency, it can benefit from this approximation. Therefore, Paraprox only applies the approximation on such functions.

3.1.3 Implementation:

Approximate memoization is accomplished in three steps: quantizing the inputs, joining these bit representations of inputs together to create an address, and accessing a lookup

table using that address to get the final result. Figure 3(a) shows the dataflow in the BlackScholesBody function of the *BlackScholes* benchmark. This function meets all the candidacy conditions described in Section 3.1.2. Figure 3(b) shows the approximate version of the same function.

Paraprox quantizes the function's inputs to generate an address into the lookup table. For a quantized input i, Paraprox can control the output quality of the approximate function by altering the number of bits (q_i) used to represent that input. If a pattern has multiple input variables, e.g. i and $i+1$, each input has its own quantization bits (q_i and q_{i+1}). When concatenated together, these quantization bits form the address into the lookup table. The table's size is thus equal to 2^Q, where $Q = \sum_{i=0}^{n} q_i$ for all n inputs.

Using fewer bits reduces the number of quantization levels (2^{q_i}) that represent an input value, thus limiting the input's accuracy. Conversely, increasing the number of bits will permit more quantization levels, which will increase the accuracy of the input representation. If the pattern's output is very sensitive to small changes in the input and there are not enough bits allocated to adequately represent this, Paraprox detects that the output quality is deteriorating and increases q_i. On the other hand, if the output is not very sensitive to changes in the input or the input's dynamic range is very small, Paraprox can reduce q_i.

Bit tuning: The process of determining q_i for inputs is called *bit tuning* and is performed offline. For each input argument to the function, Paraprox computes the range of the function's output by applying training data to the function and storing the results in memory. If an input at runtime is not within this precomputed range, it will map to the nearest value present in the lookup table.

If a function has multiple inputs, naively dividing the quantization bits equally amongst all inputs does not necessarily yield ideal results, so Paraprox can unevenly divide the bits of the quantized input to favor some inputs over others. For example, in Figure 3, the BlackScholesBody function has five inputs, two of which (R and V) are always constant during profiling. When Paraprox detects this, it chooses to allot all quantized bits to represent the other three variable inputs.

Our experiments show that the overall speedup of this optimization is dependent on the size of the lookup table but not the number of bits in q_i assigned to each input. However, the quantization bits still need to be distributed carefully amongst inputs to guarantee satisfactory output quality.

To reduce output quality loss for a given lookup table size, bit tuning uses a tree algorithm. Each node in the tree corresponds to an approximate kernel with a specific q_i bits per input. The root node divides bits equally between the inputs. Figure 4 shows the tree for the example shown in Figure 3(b). In this example, the lookup table size is 32768, which implies that the address into the table is 15 bits wide. The root of the tree shows that this address initially is evenly

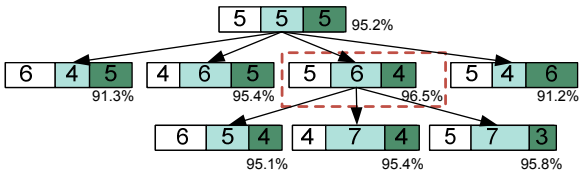

Figure 4: An example of how Paraprox's bit tuning finds the number of bits assigned to each input for the BlackScholesBody function. The lookup table has 32768 entries and its address is 15 bits wide. The output quality is printed beside each node. Bit tuning's final selection is outlined with a dotted box.

split into five bits each for the three variable inputs. Each child node is different from its parent such that one bit is reassigned from one input to an adjacent input.

The bit tuning process starts from the root and uses a steepest ascent hill climbing algorithm to reach a node with the highest output quality. Paraprox checks all the children of each node and selects the one with the best output quality. This process will continue until it finds a node for which all of its children have lower output quality than itself. In the example shown in Figure 4, Paraprox starts from node ($q_1 = 5$, $q_2 = 5$, $q_3 = 5$) and checks all its children. Among them, node (5,6,4) has the best output quality. Since all children of node (5,6,4) have a lower output quality than itself, node (5,6,4) is selected, and Paraprox assigns 5, 6, and 4 quantization bits to the first, second, and third inputs, respectively. Paraprox uses this process to find a configuration that returns the highest output quality for the specified lookup table size.

As bit tuning aims to control quality loss, it needs to determine how much error is introduced for each bit configuration it considers. To do so, bit tuning first quantizes the inputs using the division of bits specified by the current tree node under inspection. It then calculates the results of the exact and approximate functions and compares the two to compute a percent difference. Figure 4 shows these quality metrics for the BlackScholesBody example. It should be noted that bit tuning does not need to use an actual lookup table as it computes the approximate result that it is currently investigating.

To determine the size of the lookup table, Paraprox starts with a default size of 2048. For each lookup table size, Paraprox performs bit tuning to find the output quality. If the quality is better than the TOQ, Paraprox decreases the size of lookup table to see if it can further improve performance. If the quality is worse than the TOQ, Paraprox doubles the lookup table's size, as larger tables improve accuracy. This process stops when Paraprox finds the smallest table size that has an output quality that satisfies the TOQ.

After computing the size of the lookup table and assigning quantization bits for each input, Paraprox populates the lookup table. For each quantization level of each input, Paraprox computes the output and stores it in the lookup table. After filling the lookup table, Paraprox passes the approxi-

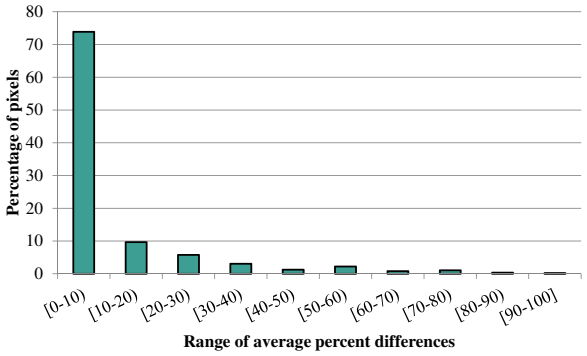

Figure 5: The average percent differences between adjacent pixels in ten images. More than 75% of pixels are less than 10% different from their neighbors.

mate kernel a pointer to the lookup table. The lookup table can be allocated in the global memory, or if a target has fast access memories, like the constant cache or shared memory in GPUs, those can be utilized instead of the global memory. Section 4.4.2 investigates these different options and compares their impacts on performance. Should the output quality change during runtime, Paraprox can accelerate the process of switching between different sized lookup tables by storing multiple tables in memory and changing the pointer passed to the kernel at runtime to reflect this decision. Paraprox can generate as many tables as it can fit in memory. However, in our experiments we found that no more than three tables are needed for our benchmarks.

3.1.4 Tuning Parameter:

To tune the output quality and performance, Paraprox allows the runtime to select amongst lookup tables of different sizes.

3.2 Stencil & Partition

3.2.1 Idea:

The stencil and partition approximation algorithm is based on the assumption that adjacent elements in the input array usually are similar in value. This is often the case for domains such as image and video processing, where neighboring pixels tend to be similar if not the same. To evaluate this assumption, Figure 5 shows the average percent difference of each pixel with its eight neighbors, which constitute a tile, for all pixels in 10 different images. As the figure shows, on average, more than 70% of the each image's pixels have less than 10% difference from their neighbors. Therefore, most of the neighbors of each pixel have similar values.

Under this assumption, rather than access all neighbors within a tile, Paraprox accesses only a subset of them and assumes the rest of the neighbors have the same value.

3.2.2 Detection:

To detect stencil/partition patterns, Paraprox checks the load accesses to the arrays and looks for a constant number of

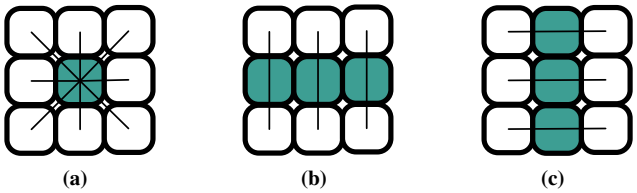

Figure 6: The three different schemes Paraprox uses to approximate the stencil pattern. (a) illustrates how the value at the center of the tile approximates all neighboring values. (b) and (c) depict how one row/column's values approximate the other rows/columns in the tile.

affine accesses to the same array, indicating a tile size. These accesses can be found in loops with a constant loop trip or in manually unrolled loops. After finding these accesses, Paraprox computes the tile's size and dimensionality. Paraprox detects stencil/partition patterns based on the array access indices ($(f + i) * w + g + j$). Parameters f, g, and w are the same (loop invariant) for all accesses that are examined. Parameters i and j can be hand-coded constants or loop induction variables. The size of a tile can be determined by looking at the dynamic range of i and j.

3.2.3 Implementation:

To approximate stencil/partition patterns, Paraprox uses three different approximation schemes: center, row, and column based. For each approximation, a *reaching distance* parameter controls the number of memory elements that Paraprox accesses. In the center based approach, the element at the center of a tile is accessed and Paraprox assumes that all its neighbors have the same value. When Paraprox accesses an element, its neighbors, whose distances from the accessed element are less than the reaching distance, will not be accessed as shown in Figure 6(a).

Figures 6(b) and 6(c) illustrate the row and column based approximation schemes. In these schemes, one row/column within a tile is accessed, and all other rows/columns within a reaching distance from it are assumed to be the same and are left unaccessed.

3.2.4 Tuning Parameter:

To control performance and output quality, Paraprox allows a runtime to select from various approximate kernels and tune each kernel's reaching distance.

3.3 Reduction

3.3.1 Idea:

To approximate reduction patterns, Paraprox aims to predict the final result by computing the reduction of a subset of the input data in a way similar to loop perforation [2]. Figure 7 illustrates how this concept is applied. The assumption here is that the data is distributed uniformly, so a subset of the

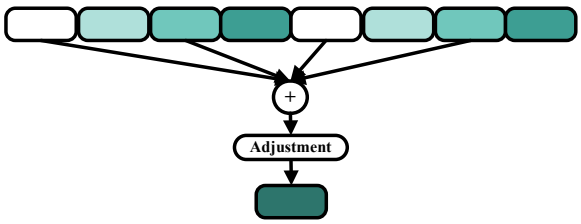

Figure 7: An illustration of how Paraprox approximates the reduction pattern. Instead of accessing all input elements, Paraprox accesses a subset of the input array and adds adjustment code to improve the accuracy.

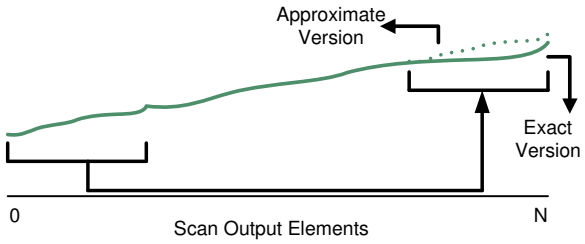

Figure 8: An example of how Paraprox uses the first elements of the scan results to approximate the end of the output array.

data can provide a good representation of the entire array. For example, instead of finding the minimum of the original array, Paraprox finds the minimum within one half of the array and returns it as the approximate result. If the data in both subarrays have similar distributions, the minimum of these subarrays will be close to each other and approximation error will be negligible.

Some reduction operations like addition need some adjustment to produce more accurate results. For example, after computing the sum of half of an array, if the result is doubled it more closely resembles the results of summing the entire array, thus the output quality is improved. In this case of addition, Paraprox assumes that the other half of the array has the exact same sum as the first half, so it doubles the approximated reduction result.

3.3.2 Detection:

Reduction recognition has been studied extensively by previous works [26, 36]. To detect reduction patterns, Paraprox searches for accumulative instructions that perform an operation like $a = a + b$, where a is called the reduction variable and addition is the reduction operation. Reduction loops have the following two characteristics: *a)* they contain an accumulative instruction; and *b)* the reduction variable is neither read nor modified by any other instruction inside the loop.

In order to parallelize a reduction loop for a data parallel architecture, tree-based reduction implementations are often used. These reductions have three phases. In the first phase (Phase I), each thread performs a reduction on a chunk of input data. In the next phase (Phase II), each block accumulates the data generated by its threads and writes this result to the global memory. The final phase (Phase III) then accumulates the results of all the blocks to produce the final results. All of the phases contain a reduction loop that Paraprox optimizes, creating approximate kernels for each loop. The runtime determines which approximate version to execute.

Atomic operations can also be used to write data parallel reductions. An atomic function performs a read-modify-write atomic operation on one element residing in global or shared memory. For example, CUDA's *atomicInc()* and OpenCL's *atomic_inc()* both read a 32-bit word at some address in the global or shared memory, increment it, and write the result back to the same address [13, 19]. Among atomic operations, the atomic add, min, max, inc, and, or, and xor operations can be used in a reduction loop. Paraprox searches for and marks loops containing these operations as reduction loops.

3.3.3 Implementation:

After detecting a reduction loop, Paraprox modifies the loop step size to skip iterations of the loop. In order to execute every N^{th} iteration and skip the other $N - 1$ iterations, Paraprox multiplies the loop step by N. We call N the *skipping rate*. For example, if Paraprox multiplies the loop step size by four, only a quarter of the original iterations are executed and the rest are skipped.

If the reduction operation is addition, Paraprox inserts adjustment code after the loop. This code multiplies the result by the skipping rate. To make the adjustment more accurate, the reduction variable's initial value should be equal to zero before the reduction loop. Otherwise, by multiplying the result, the initial value is multiplied as well which produces an unacceptable output quality. In order to address this, Paraprox replaces the loop's reduction variable with a temporary variable set to zero just before the loop's entrance. After adjustment, Paraprox then adds the scaled temporary variable back to the original reduction variable to produce the final result.

3.3.4 Tuning Parameter:

Paraprox allows a runtime to change the skipping rate in order to tune the speedup and accuracy of the kernels.

3.4 Scan

3.4.1 Idea:

To approximate scan patterns, Paraprox assumes that differences between elements in the input array are similar to those in other partitions of the same input array. Parallel implementations of scan patterns break the input array into subarrays and computes the scan result for each of them. In order to approximate, Paraprox only applies the scan to a sub-

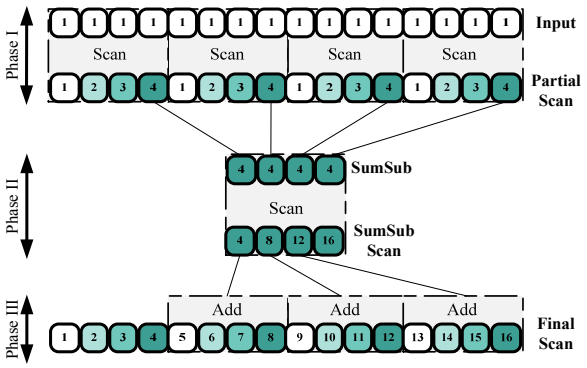

Figure 9: A data parallel implementation of the scan pattern has three phases. Phase I scans each subarray. Phase II scans the sum of all subarrays. Phase III then adds the result of Phase II to each corresponding subarray in the partial scan to generate the final result. This figure depicts how the scan is computed for an input array of all ones.

set of these subarrays and uses its results for the rest of the subarrays.

As the N^{th} element of the scan result is the sum of the first N elements of its input array, any change to the N^{th} element modifies the N^{th} output element and all elements afterwards. Therefore, if Paraprox applies approximation to one of the early input elements, any approximation error will propagate to the results for all the following elements, resulting in an unacceptable output quality. This effect is studied in Section 4.4.3.

In order to avoid this cascading error, rather than uniformly skipping loop iterations, Paraprox predicts the last elements of the scan results by examining the first output elements. Figure 8 presents an example of how Paraprox copies the first elements of the result to the end of the array to approximate the last elements.

3.4.2 Detection:

The data parallel implementation of the scan pattern is traditionally composed of three phases as illustrated in Figure 9. As an example, this figure shows how these phases compute the scan results for an input array containing all ones. In the first phase, the input is divided into many subarrays and each block of threads performs a scan on one subarray and stores results in a partial scan array. The sum of each subarray is also written to another array called *sumSub*. The second phase then runs a scan on the *sumSub* array. The i^{th} element of *sumSub*'s scan result is equal to the sum of elements in subarrays 0 to i. In the third phase, every i^{th} element of *sumSub*'s scan result is added to the scan results of the $i + 1$ partial scan subarray to produce the final scan results.

Because of its complicated implementation, detecting a scan pattern is generally difficult. A programmer can mark scan patterns for the compiler using pragmas, or the compiler can use template matching to find scan kernels used in benchmarks [20]. Paraprox uses the second approach by performing a recursive post order traversal of the abstract syntax tree of the kernel and comparing it with the template. If they match, Paraprox assumes that the kernel contains a scan pattern.

3.4.3 Implementation:

The first phase of the scan pattern takes the longest time to execute, so approximation techniques should target this phase. As mentioned before, Paraprox approximates the results for the last subarrays to prevent the propagation of error through all of the results. In this approximation, Paraprox assumes that last subarrays have similar scan results to the first subarrays. Therefore, instead of computing scan results for all subarrays, Paraprox *skips* some and uses the first multiple subarrays' scan results in place of the scan results for the skipped subarrays.

In order to skip the last N subarrays, Paraprox skips some of the computations in Phases I and II. In Phase I, Paraprox launches fewer blocks to skip the last N subarrays. In Phase II, Paraprox changes the argument containing the number of subarrays that is passed to the kernel.

In Phase III, threads that are responsible for adding to generate the first N subarrays add their scan results to the last element of Phase II's results (the *sumSub* scan array) and write these results as the scan's output for the last skipped subarrays. Figure 8 shows how these threads copy an early portion of the results to generate the result's last elements.

3.4.4 Tuning Parameter:

A runtime can control the number of subarrays Paraprox skips in order to tune output quality and performance.

4. Experimental Evaluation

4.1 Methodology

The Paraprox compilation phases are implemented in the Clang compiler version 3.3. Paraprox's output codes are then compiled into GPU binaries using the NVIDIA nvcc compiler release 5.0. GCC 4.6.3 is used to generate the x86 and OpenCL binaries for execution on the host processor. To run OpenCL code on the CPU, we used the Intel OpenCL driver. We evaluated Paraprox using a system with an Intel Core i7 965 CPU and a NVIDIA GTX 560 GPU with 2GB GDDR5 global memory. We selected 13 applications from various domains and different patterns as benchmarks. We ran each application 110 times with different input sets. We ran the first 10 executions to train and detect the best kernel, and then we measured and averaged the runtimes of the next 100 executions. A summary of each application's characteristics is shown in Table 1.

Compilation flow in Paraprox: This section describes Paraprox's compilation flow as illustrated in Figure 10. First,

Applications	Domain	Input Size	Patterns	Error Metric
BlackScholes [20]	Financial	4M elements	Map	L1-norm
Quasirandom Generator [20]	Statistics	1M elements	Map	L1-norm
Gamma Correction	Image Processing	2048x2048 image	Map	Mean relative error
BoxMuller [20]	Statistics	24M elements	Scatter/Gather	L1-norm
HotSpot [9]	Physics	1024x1024 matrix	Stencil-Partition	Mean relative error
Convolution Separable [20]	Image Processing	2048x2048 image	Stencil-Reduction	L2-norm
Gaussian Filter	Image Processing	512x512 image	Stencil	Mean relative error
Mean Filter	Image Processing	512x512 image	Stencil	Mean relative error
Matrix Multiply [20]	Signal Processing	2560x2560 matrix	Reduction-Partition	Mean relative error
Image Denoising [20]	Image Processing	2048x2048 image	Reduction	Mean relative error
Naive Bayes [32]	Machine Learning	256K elements with 32 features	Reduction	Mean relative error
Kernel Density Estimation [16]	Machine Learning	256K elements with 32 features	Reduction	Mean relative error
Cumulative Frequency Histograms	Signal Processing	1M elements	Scan	Mean relative error

Table 1: Details of applications used in this study.

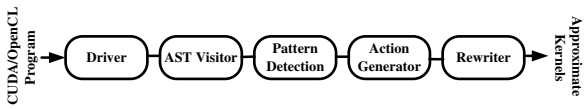

Figure 10: Paraprox's compilation flow.

Clang's *driver* generates the abstract syntax tree (AST) of the input code and sends it to the AST visitor. The *AST visitor* traverses the AST and runs the *pattern detector* on each kernel. The *pattern detector* identifies the parallel patterns within each kernel, and informs the action generator which kernels contain what patterns. The *action generator* then creates a list of actions for each approximate kernel, where an action represents a modification to the output CUDA code for each optimization applied. These actions include: adding, deleting, and substituting an expression in the final code. For each list of actions, the *rewriter* copies the input kernel and applies all actions on the copied version and generates the approximate kernel. To evaluate the impact of Paraprox's optimizations on the CPU, we created a CUDA-to-OpenCL script that converts Paraprox's generated CUDA code to an equivalent OpenCL version.

4.2 Results

In this section, we analyze how Paraprox's optimizations affect the execution time and accuracy of different applications. Figure 11 presents the results for the benchmarks run separately on a CPU and a GPU. The speedup is relative the exact execution of each program on the same architectures. As seen in the figure, Paraprox achieves an average speedup of ~2.5x for approximated code run on either the CPU or GPU with a target output quality of 90%.

Output Quality: To assess the quality of each application's output, we used application-specific evaluation metrics as listed in Table 1. For benchmarks that already contained a specific evaluation metric, the included metric was used. Otherwise, we used the mean relative error as an evaluation metric. For all benchmarks, we compare the output of the unmodified, exact application to the output of the approximate kernel created by Paraprox.

A case study by Misailovic et al. [18] shows that users will tolerate quality loss in applications such as video decoding provided it does not exceed ~10%. Similar works [5, 12, 27, 28] cap quality losses for their benchmarks at around 10%. SAGE [27] verified this threshold using the experiments in the LIVE image quality assessment study [31]. Images in LIVE's database have different levels of distortion and were evaluated by 24 human subjects, who classified the quality of the images using a scale equally divided amongst the following ratings: "Bad," "Poor," "Fair," "Good," and "Excellent." SAGE [27] showed that more than 86% of images with quality loss less than 10% were evaluated as "Good" or "Excellent" by human subjects in the LIVE study. Therefore, we used 90% as the minimum target output quality (TOQ) in our experiments.

4.3 Performance Improvement

Paraprox applied map approximation to the *BlackScholes*, *Quasirandom Generator*, *Gamma Correction*, and *BoxMuller* benchmarks. For *BlackScholes*, Paraprox detects two map functions: *Cnd()* and *BlackScholesBody()*. Since the estimated cycle count for Cnd() is low, Paraprox only applies the optimization on BlackScholesBody() which has a high estimated cycle count. As a result, *BlackScholes* achieves ~60% improvement in performance with $<$10% loss in output quality. *BoxMuller* has a scatter/gather functionwith two inputs and two outputs. *Gamma Correction* is very resilient to quality losses caused by approximation, as its output quality remains at 99% while it achieves $>$3x speedup on the GPU. When reducing the lookup table size, however, its output quality drops suddenly to $<$90%. *BlackScholes* and *Quasirandom Generator* get better results on the CPU but *Gamma Correction* and *BoxMuller* perform better on the GPU. The reason is that for benchmarks that can retain good output quality with smaller lookup tables, the GPU achieves better performance. However, as the size of lookup table increases, the number of cache misses increases. In such cases,

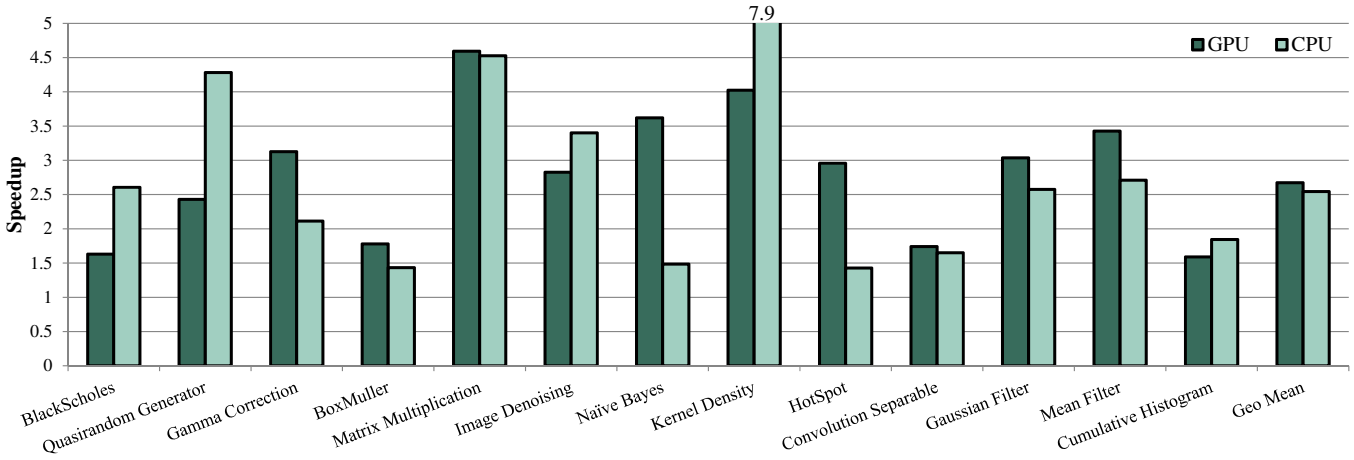

Figure 11: The performance of all applications approximated by Paraprox for both CPU and GPU code. The baseline is the exact execution of each application on the same architecture. In these experiments, the target output quality (TOQ) is 90%.

execution on a CPU is preferable to that on a GPU as cache misses have a lower impact on the performance for CPUs.

The reduction approximation is applied to the *Matrix Multiplication*, *Naive Bayes trainer*, *Image Denoising*, and *Kernel Density Estimation* applications. *Matrix Multiplication* and *Image Denoising* show similar performance on both the CPU and GPU. On the other hand, *Naive Bayes* achieves better speedup on the GPU. The approximated *Naive Bayes* performs very well on a GPU (>3.5x vs ~1.5x on a CPU) since this benchmark uses atomic operations, which are more expensive for GPU architectures with many threads running concurrently. By skipping a subset of atomic operations, great speedups in execution time are achieved on a GPU. Since the main component of *Kernel Density Estimation* is an exponential instruction and there is hardware support for such transcendental operations on a GPU (i.e. the special function unit on a CUDA device), skipping these operations provides better performance improvements for CPUs than it does for GPUs.

The *HotSpot*, *Convolution Separable*, *Gaussian filter*, and *Mean Filter* applications contain stencil patterns. *HotSpot*, *Gaussian filter*, and *Mean filter* use 3x3 tiles and *Convolution Separable* has two stencil loops with 1x17 tiles. Since the loop in *Mean Filter* is unrolled manually by the programmer and memory accesses are kept outside the function while computations are inside, there is no reduction loop and the reduction optimization is not applied. Paraprox just applies the stencil optimization on this application. On the other hand, *Convolution Separable* has both stencil and reduction patterns. Paraprox applies both optimizations on this application. The stencil optimization returns a 1.7x performance speedup while maintaining 90% output quality while the reduction optimization results in a 1.6x speedup. On the other hand, the reduction optimization's performance is better than stencil optimization for CPU. Therefore, when targeting a GPU Paraprox only used the results of the stencil optimization in its final kernel, and when targeting a CPU it used the reduction optimization. Because the partition and stencil optimizations primarily optimize memory accesses, speedups are greater for GPU approximated code as memory accesses are more costly on this platform.

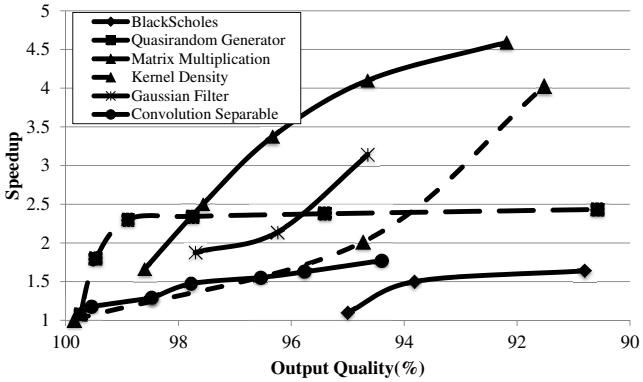

Figure 12: Controlling the speedup and output quality by varying an optimization's tuning parameters for six benchmarks.

The *Cumulative Histogram* benchmark contains a scan pattern. This application is another resilient application — even when skipping half of the subarrays, the output quality stays at ~99%. For this pattern, the speedup is similar for both CPU and GPU approximated kernels.

Performance-Quality Tradeoffs: Figure 12 illustrates how Paraprox manages the performance-accuracy tradeoff for six benchmarks. The map approximated *BlackScholes* starts with 95% output quality and performance similar to the exact version, but as the size of the lookup table decreases, the speedup increases to 1.6x speedup with only ~4% more loss

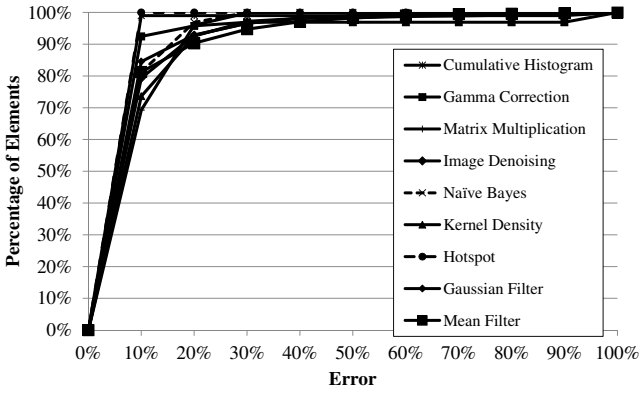

Figure 13: The CDF of final error for each element of an application's output with the $TOQ = 90\%$. The majority of output elements (>70%) have <10% error.

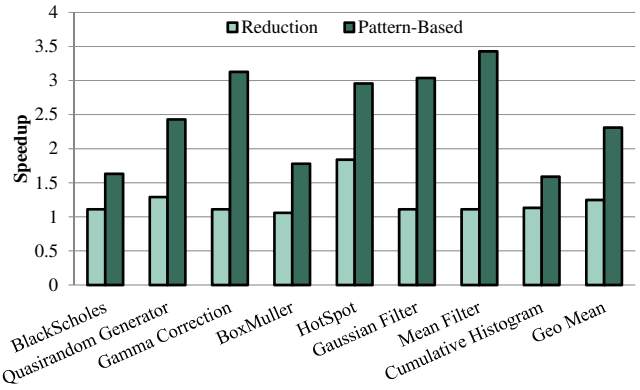

Figure 14: A performance comparison of the reduction optimization vs. specific pattern-based optimizations on benchmarks that do not contain a reduction pattern. In these experiments, the code targets a GPU, and the $TOQ = 90\%$.

in quality. Similar behavior is observed for the *Quasirandom Generator*. When the table size is small enough to fit in the cache, the speedup gains begin to saturate for these map optimized kernels. Both *Matrix Multiplication* and *Kernel Density Estimation* contain reduction patterns. As Paraprox doubles the skipping rate for these kernels, the difference between two consecutive nodes grows, thus causing both the speedup and quality loss to grow. The performance of *Gaussian Filter* and *Convolution Separable* rises as output quality degrades. For *Convolution Separable*, Paraprox changes the reaching distances of both loops in the kernel to control the output quality. Since *Gaussian Filter* applies a 2D filter to an image, Paraprox uses row, column, and center stencil patterns to control the output quality. For this benchmark, Paraprox gets >2x speedup with <4% quality loss.

Error Distribution: To study each application's quality losses in more detail, Figure 13 shows the cumulative distribution function (CDF) of the error for each element of the application's output with the $TOQ = 90\%$. The CDF illustrates the distribution of output errors amongst an application's output elements. The figure shows that only a modest number of output elements see large output error. The majority (70%-100%) of each approximated application's output elements have an error of <10%.

4.4 Case Studies

4.4.1 Specialized Optimizations Achieve Better Results:

To show that one optimization does not work well when generally applied, we apply only the reduction optimization to benchmarks that do not contain such a pattern. We chose this optimization as it is most similar to a well-known approximation technique, loop perforation [2], where loop iterations are skipped to accelerate execution. Figure 14 compares the reduction optimization's performance with Paraprox's results on a GPU with the $TOQ = 90\%$. For benchmarks containing map and stencil patterns, skipping iterations results in unmodified output elements. Therefore, the output quality rapidly decreases, severely limiting the speedup. For benchmarks with scan patterns, the cascading error will reduce the output quality and speedup is similarly limited. On average, the reduction optimization alone achieves only ~25% speedup, compared to the 2.3x speedup that Paraprox achieves by matching patterns to specialized optimizations.

4.4.2 Design Considerations for the Map Optimization:

To fully investigate the impact of map approximation on both accuracy and performance, we used four common computationally intensive map functions from different domains:

- **Credit card balance equation [1]:** This equation finds the number of months it will take to pay off credit card debt.

$$N(i) = \frac{-1}{30}\frac{ln(1+\frac{b_0}{p}(1-(1+i)^{30}))}{ln(1+i)} \quad (2)$$

- **Shifted Gompertz distribution [21]:** This equation gives the distribution of the largest of two random variables.

$$F(x) = (1-e^{-bx})e^{-\eta e^{-bx}} \quad (3)$$

- **Log gamma [6]:** This equation calculates the logarithm of the gamma function. To implement this equation, we used the CUDA *lgammaf* [19] function.

$$LG(z) = log(\Gamma(z)) \quad (4)$$

- **Bass diffusion model [7]:** This equation describes how new products get adopted as an interaction between users

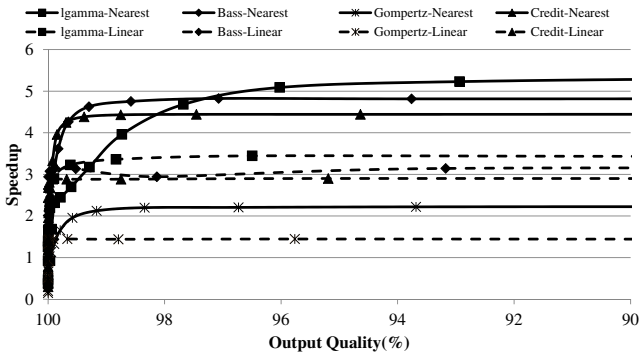

Figure 15: The impact of approximate memoization on four functions on a GPU. Two schemes are used to handle inputs that do not map to precomputed outputs: *nearest* and *linear*. *Nearest* chooses the nearest value in the lookup table to approximate the output. *Linear* uses linear approximation between the two nearest values in the table. For all four functions, *nearest* provides better speedups than *linear* at the cost of greater quality loss.

and potential users.

$$S(t) = m\frac{(p+q)^2}{p}\frac{e^{-(p+q)t}}{(1+\frac{p}{q}e^{-(p+q)t})^2} \quad (5)$$

For all of these equations, all parameters other than the input variable are constant.

Selecting an Output for an Unrepresented Input: As discussed in Section 3.1.3, there are a limited number of quantization levels based on the size of the lookup table. It is possible that there are inputs that do not directly map to a precomputed output. In such cases, Paraprox can either select the *nearest* precomputed output, or it can apply *linear* approximation to the two nearest values in the table to generate a result in between these values. Figure 15 shows the performance-quality curve for all four equations using the *nearest* and *linear* methods on the GPU. For all four equations, *nearest* gives better performance compared to *linear* but with lower output quality. Even though the same lookup table size is used, *linear* generates more accurate output, but the overhead of adding another memory access and more computation is overwhelming. On the other hand, *linear* is better at achieving higher output quality (~99%). In this experiment, the lookup table is allocated in the GPU's global memory.

As seen in Figure 15, the shifted Gompertz distribution achieves a lower speedup than the other functions. This is due to it having many low latency instructions. Both the Bass and Credit equations execute floating point divisions, which translate to subroutine calls to code with high latency and low throughput for GPUs [35]. On the other hand, the Gompertz equation uses several exponential instructions, which are not high latency as they are handled by a special functional unit on a GPU [19].

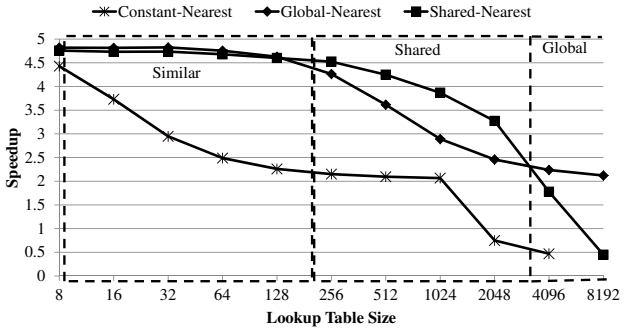

Figure 16: A comparison of the performance of approximate memoization when the lookup table is allocated in the constant, shared, and global memories on a GPU.

Location of the Lookup Table: To find which memory location is best for storing lookup tables, we created approximate versions of the Bass function that used the constant, shared, and global memories of the GPU to store the lookup table. Figure 16 shows the performance versus the table size for these three versions of Bass on the GPU. For lookup tables stored in global and constant memory, we set the L1 cache size to 32KB and size of the shared memory to 16KB. When the lookup table is stored in shared memory, we set the size of the shared memory to 32KB and the L1 cache to 16KB.

Using constant memory never gives optimal results regardless of the cache size. The reason is that for larger table sizes, using shared memory or the global L1 cache will have a lower read latency [35].

To compare global and shared memory, we divided the figure into three regions. When the cache size is small, both global and shared memory show similar speedups. Since it takes time to warm up the L1 cache for global memory, shared memory outperforms the global memory in the second region. In the third region, however, by increasing the size of the lookup table, the overhead of transferring data from global to shared memory is increased and the global memory outperforms the shared memory.

Based on these results, Paraprox generates both shared and global approximate kernels, and the runtime system will choose one based on the performance, output quality, and the lookup table size. If the lookup table is larger than the size of the shared memory, the lookup table must be stored in global memory.

Lookup Table Size vs. Performance: Figure 16 shows that speedup drops when increasing the size of the lookup table. Using the CUDA profiler, we found that the number of uncoalesced memory accesses is primarily responsible for this. As the lookup table's size increases, the number of uncoalesced accesses also increases, thus resulting in lower speedups as shown in Figure 17. This figure shows that the number of instructions that get serialized increases as the

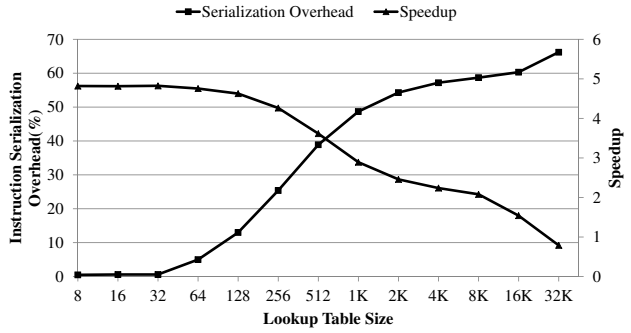

Figure 17: The impact of the lookup table size on the percentage of uncoalesced accesses on the GPU.

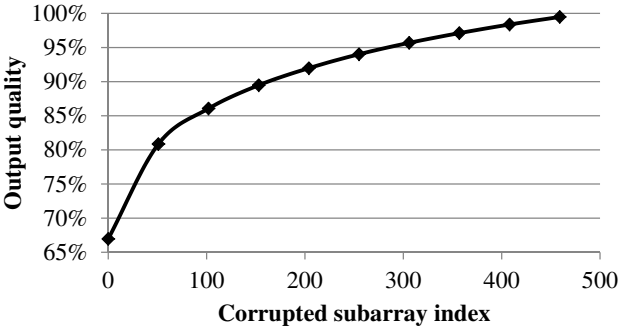

Figure 18: The impact of the starting point of the data corruption on an approximated scan pattern's final output.

size of the lookup table grows. This serialization is caused by more uncoalesced accesses.

4.4.3 Cascading Error in Scan Patterns:

Paraprox approximates the last subarrays of the results for scan patterns. To illustrate why this is done, we use the *Cumulative frequency histogram* benchmark with one million random input data points. For our first run, we "corrupt" the first input subarray (10% of the input elements) by setting its elements to zero. We then move this section of all zeroed data to the next subarray of elements, and rerun the scan. For each test modifying the first to the last input subarray, we record the output quality. Figure 18 shows the impact of the starting point of the data corruption on the final output result. When the first subarray of the input is zeroed, the overall output quality will be ~67%. This is caused by the error propagating through the rest of the results. However, if the error happens at the end of the input array, the output quality will be ~99%. Therefore, Paraprox only approximates the last elements of the final scan results to ensure a high output quality.

5. Limitations

Paraprox is a research prototype and it has some limitations. Below, we discuss some of these limitations which will be addressed in future work.

Runtime System: In this work, our main focus is automatically generating approximate kernels and providing tuning knobs for a runtime system. Paraprox generates approximate kernels, and a separate runtime system will decide which one to use and how tune the selected kernel's parameters. In our results, we did not consider runtime overhead. However, as shown in SAGE [27] and Green [5], it is not necessary to constantly monitor the quality, so checks are performed every N^{th} invocation. Based on the experiments done in [27], checking the output quality every 40-50 invocations during runtime has less than 5% overhead. This would reduce our reported performance but only by a modest level.

Pattern Recognition: Since we used the AST to detect patterns, variations in code can make the pattern recognition process difficult, especially when detecting scan patterns. However, pattern recognition for other patterns like reduction or detecting pure functions for map and scatter-gather patterns are stable techniques which can detect patterns across a wide variety of implementations. It is also possible to enhance pattern detection by getting hints from the programmer or using higher level languages.

Compiler Optimizations: It is possible that approximation eliminates some other compiler optimization opportunities such as auto-vectorization. In these cases, an approximate kernel might not perform as well as expected. Fortunately, the runtime system chooses which approximate kernel to run based on their speedup and quality. Therefore, if the approximate kernel does show great performance improvement, the runtime system will choose the original kernel which is highly optimized.

Safety of Optimizations: It is possible that execution of approximate code causes raising exceptions or segmentation faults. There are compiler analyses that detect the possibility of crashing to prevent the compiler from applying the optimizations. For example, for a division that uses an approximated output and may raise a divide by zero exception, it is possible to instrument the code to skip this calculation where the approximated divisor is zero. However, improving the safety of approximation techniques is beyond the scope of this work and it is left for future study.

6. Related Work

Pattern-based programming is well-explained by McCool [15]. This book introduces various parallel patterns. Our focus is on the detection and approximation of data parallel patterns.

The concept of trading accuracy for improved performance or energy consumption is well-studied [2–5, 11, 12, 14, 17, 23, 25, 27, 28]. Previous approximation techniques can be categorized in three categories:

Software-based: Using software approximation, SAGE's [27] framework accelerates programs on GPUs. SAGE's goal is to exploit the specific microarchitectural characteristics of the GPU to achieve higher performance. Although these optimization performs better than general methods, their ap-

plicability is limited compared to Paraprox's approximation methods. SAGE also has a runtime system which Paraprox can use to tune and calibrate the output quality during runtime.

Rinard et al. [23, 25] present a technique for automatically deriving probabilistic distortion and timing models that can be used to manage the performance-accuracy tradeoff space of a given application. Given a program that executes a set of tasks, these models characterize the effect of skipping task executions on the performance and accuracy. Agarwal et al. [2] use code perforation to improve performance and reduce energy consumption. They perform code perforation by discarding loop iterations. Paraprox uses a similar method for reduction patterns, but while loop perforation is applied only to sequential loops, Paraprox applies it to *all* loops in such patterns. Skipping iterations, however is not suitable for all data parallel patterns, so Paraprox only applies it to loops with reduction patterns. For example, by skipping iterations of a map loop, a subset of the output array will be left unmodified which results in an unacceptable output quality. A variation of approximate memoization is utilized in a work by Chadhuri [8] for sequential loops. Our approach is different in that it is designed for data parallel applications and it detects when to apply memoization to achieve performance improvement. Previous work by Sartori et. al. [30] targets control divergence on the GPU. Rinard et. al. [23] also proposes an optimization for parallel benchmarks that do not have balanced workloads. Misailovic et. al. [17] propose probabilistic guarantees for approximate applications using loop perforation. Relaxed synchronization is also used as an approximation method to improve performance [22, 24]. Although these approaches perform well for their target applications, their applicability is far more limited than tools that can identify and finely optimize kernels based on the varied data parallel patterns they may contain, which is one of Paraprox's key contributions.

Programmer-based: Green [5] is a flexible framework that developers can use to take advantage of approximation opportunities to improve performance or energy efficiency. This framework requires the programmer to provide approximate kernels or annotate their code using C/C++ extensions. In contrast to these techniques, Paraprox automatically generates different approximate kernels for each application. Ansel et. al. [4] 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.

Hardware-based: EnerJ [28] 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. EnerJ also guarantees that the approximate part of a program cannot affect the precise portion of the program. Esmaeilzadeh et al. [11] demonstrated dual-voltage operation, with a high voltage for precise operations and a low voltage for approximate operations. Another work by Esmaeilzadeh [12] designs a neural processing unit (NPU) accelerator to accelerate approximate programs. Alvarez et al. [3, 14] introduced hardware-based fuzzy memoization and tolerant region reuse techniques for multimedia applications. Other works [33, 34] also designed different approximate accelerators. Sampson et. al. [29] show how to improve memory array lifetime using approximation. These approximate datatype optimizations and special accelerators require hardware support. Our approach, however, can be used by current architectures without hardware modification.

7. 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 the exponential growth of information. However, approximation can often be time consuming and tedious for programmers to implement, debug, and tune to achieve the desired results. This paper proposes a software-only framework called *Paraprox* that identifies common computation patterns found in data-parallel programs and uses a custom-designed approximation template to replace each pattern. Paraprox enables the programmer to write software once and run it on a variety of commodity processors, without manual tuning for different hardware targets, input sets, or desired levels of accuracy.

For 13 data-parallel applications, Paraprox yields an average of 2.7x and 2.5x speedup with less than 10% quality degradation compared to an accurate execution on a NVIDIA GTX 560 GPU and Intel Core i7 965 CPU, respectively. We also show that Paraprox is able to control the accuracy and performance by varying template configuration parameters at run-time. Our results show that pattern-specific optimizations yield nearly twice the performance improvement compared to naively applying a single, well-known approximation technique to all benchmarks.

8. Acknowledgement

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. This research was supported by ARM Ltd. and the National Science Foundation under grants CNS-0964478 and CCF-0916689. We would like to thank Armin Alaghi and Gaurav Chadha for their comments.

References

[1] The credit card equation, 2013. http://faculty.bennington.edu/˜jzimba/CreditCardEquation Derivation.pdf.

[2] 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.

[3] C. Álvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, pages 922–927, 2005.

[4] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for autotuning variable-accuracy algorithms. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, pages 85 –96, 2011.

[5] 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.

[6] D. H. Bailey. *Experimental mathematics in action*. A.K. Peters, 2007.

[7] F. Bass. A new product growth for model consumer durables. *Management Science*, pages 215–227, 1969.

[8] S. Chaudhuri, S. Gulwani, R. L. Roberto, and S. Navidpour. Proving programs robust. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 102–112, 2011.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009.

[10] EMC Corporation. Extracting value from chaos, 2011. www.emc.com /collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, 2012.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 449–460, 2012.

[13] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2013.

[14] C. A. Martinez, J. C. S. Adrian, and M. Cortes. Dynamic tolerance region computing for multimedia. *IEEE Transactions on Computers*, pages 650–665, 2012.

[15] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, 2012.

[16] P. D. Michailidis and K. G. Margaritis. Accelerating kerne stimation on the GPU using the CUDA framework. *Journal of Applied Mathematical Science*, 7(30):1447–1476, 2013.

[17] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Proc. of the 18th Static Analysis Symposium*, pages 316–333, 2011.

[18] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *Proc. of the 32nd ACM/IEEE conference on Software Engineering*, pages 25 –34, 2010.

[19] NVIDIA. *CUDA C Programming Guide*, Oct. 2012.

[20] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012. www.nvidia.com/content/PDF/NVIDIA_Kepler_GK110 _Architecture_Whitepaper.pdf.

[21] K. Ohishi, H. Okamura, and T. Dohi. Gompertz software reliability model: Estimation algorithm and empirical validation. *Journal of Systems and Software*, pages 535 – 543, 2009.

[22] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. Programming with relaxed synchronization. In *Proc. of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, pages 41–50, 2012.

[23] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proc. of the 2006 International Conference on Supercomputing*, pages 324–334, 2006.

[24] M. Rinard. Parallel synchronization-free approximate data structure construction. In *Proc. of the 5th USENIX Workshop on Hot Topics in Parallelism*, pages 1–8, 2013.

[25] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *Proc. of the 22nd annual ACM SIGPLAN conference on Object-Oriented Systems and applications*, pages 369–386, 2007.

[26] D. Roger, U. Assarsson, and N. Holzschuch. Efficient stream reduction on the GPU. In *Proc. of the 1st Workshop on General Purpose Processing on Graphics Processing Units*, pages 1–4, 2007.

[27] 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.

[28] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: approximate data types for safe and general low-power computation. *Proc. of the '11 Conference on Programming Language Design and Implementation*, 46(6):164–174, June 2011.

[29] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 25–36, 2013.

[30] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *IEEE Transactions on on Multimedia*, pages 427–428, 2012.

[31] H. Sheikh, M. Sabir, and A. Bovik. A statistical evaluation of recent full reference image quality assessment algorithms. *IEEE Transactions on Image Processing*, 15(11):3440–3451, 2006.

[32] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. OptiML: an implicitly parallel domain specific language for machine learning. In *Proc. of the 28th International Conference on Machine learning*, pages 609–616, 2011.

[33] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 356–367, 2012.

[34] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 1–12, 2013.

[35] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of the 2010 IEEE Symposium on Performance Analysis of Systems and Software*, pages 235–246, 2010.

[36] X.-L. Wu, N. Obeid, and W.-M. Hwu. Exploiting more parallelism from applications having generalized reductions on GPU architectures. In *Proc. of the 2010 10th International Conference on Computers and Information Technology*, pages 1175–1180, 2010.