Loop Perforation in OpenACC

Ahmad Lashgar*, Ehsan Atoofian†, and Amirali Baniasadi*
*Electrical and Computer Engineering Department, University of Victoria, Victoria, BC, Canada
†Electrical Engineering Department, Lakehead University, Thunder Bay, ON, Canada

***Abstract*—High-level programming models such as OpenMP and OpenACC are used to accelerate loop-parallelizable applications. In such applications, a very large number of loop iterations are launched as threads on the accelerator, where every iteration executes the same code sequence (loop body or kernel) but on different data. In such workloads, similarities in the input lead to wide similarities in the outputs. Motivated by this observation, we propose to run only a subset of loop iterations, accurately calculating some outputs and approximating the rest. To this end, we propose employing a new directive in OpenACC to trade off performance for accuracy by perforating loop iterations. The directive is only applicable to parallel loops with a perforation rate adjusted by the programmer. Moreover, we investigate the quality and runtime impact of this directive. In summary, first, we show that naïvely applying loop perforation to OpenACC, degrades performance significantly. This is because OpenACC parallel loops are often output-parallelized and every iteration calculates one single entry in the output. Consequently, dropping *k* iterations leads to *k* erroneous entries in the output. Second and in order to address this we propose an efficient low-overhead mechanism to recover the value of these missing output entries. Third, we show that due to the SIMD organization of accelerators, perforation does not always translate to runtime improvements. Our study shows that perforation can change the memory coalescing behavior and negatively impact runtime. In order to provide better insight we present workload characteristics that benefit from perforation the most. Our evaluations using a diverse set of benchmarks indicate that our proposed technique can improve performance up to 93%, while maintaining the quality loss at a rate below 10%.**

***Index Terms*—Approximate Computing, GPGPU, OpenACC, Loop perforation, CUDA.**

I. Introduction

OpenMP and OpenACC are high-level programming models for executing parallel loops on accelerators. Using these models, applications can run on accelerators by adding only few directives to the code. Output parallelism is often used in OpenACC applications where each iteration is assigned to calculate one (or many) entry(s) in the output independently. Our key observation is that consecutive iterations work on very similar data and their output are very similar too. This leaves an opportunity to use approximate computing and improve efficiency. In this work, our key idea is to run few samples of iterations and estimate the entire output based on these samples, instead of exhaustively running all iterations. A knob is given to an OpenACC programmer to determine the sampling rate. For example, for a loop with N parallel iterations and sampling rate of 50%, $N/2$ iterations are executed to exactly calculate half of the output, and this half is used to estimate the rest of the output. We call this technique *loop perforation*, named after a similar work on CPUs for serial applications [9].

TABLE I
Speedup and quality loss under naive loop perforation (perforation rate of 50%.)

Benchmark	Speedup	Quality loss
Backprop	16%	0.6%
BlackScholes	20%	34%
Fast Wal. Trsf.	27%	48%
Hotspot	-20%	29%
Matrix Mult.	60%	50%
Median Flt.	48%	37%
Pathfinder	-19%	98%
Sobel Flt.	97%	11%

While a speedup proportional to the sampling rate is expected from perforation (over the baseline without perforation), we found that application type (e.g. compute-bound or memory-bound) is also an important parameter in OpenACC. For compute-bound applications, we observe that speedup scales with perforation rate. However, for memory-bounded applications, speedup depends on the memory pattern. For example, if the memory pattern is irregular, we do observe superlinear speedup and if the memory pattern is regular, we observe insignificant sublinear speedup. As we throughly investigate this behavior later in this paper, this is explained by the organization of GPU-like accelerators.

While we generally see significant speedup from loop perforation, the challenge is to maintain accuracy high based on the output obtained from samples. In Table I we show naively applying loop perforation, by just storing the output from the samples at the corresponding entries and leaving the rest of entries unassigned/empty, significantly ruins the accuracy (for methodology refer to Section IV-A.) Our goal in this paper is to address this accuracy challenge, while keeping the speedup advantages high. We propose a very efficient solution that does not increase the number of kernel launches on accelerator and does not increase the number/size of memory transfers between host and accelerator. Instead, in the same kernel, a tiny *fix-up* code is attached to find unassigned entries and then approximate values for them. The *fix-up* code lookups neighbor entries for finding the unassigned entries and approximates the values based on the values obtained for the neighbors.

In summary, we make the following contributions:

- To the best of our knowledge, this is the first work investigating use of loop perforation in OpenACC with

GPGPU backend.

- We show that loop perforation speedup in OpenACC does not always scale well with perforation rate and highly depends on the application characteristics. We evaluate performance impacts of loop perforation under various workload types and memory patterns.
- We introduce a very efficient solution to fix quality loss caused by loop perforation, while maintaining the performance advantages still high.
- We investigate our method under eight different OpenACC benchmarks and show loop perforation can deliver up to 93% improvement in runtime, while the quality loss remains below 10%.

The rest of this paper is organized as follows. In Section II, we overview background. In Section III, we introduce our extensions for OpenACC to perform loop perforation and discuss techniques for maintaining the accuracy high under perforation. In Section IV, we evaluate loop perforation performance and accuracy. We discuss related issues in Section V. In Section VI, we overview related work. Finally, in Section VII we offer concluding remarks.

II. Background

OpenACC is a directive-based API for parallelizing applications on accelerators (including GPUs, co-processors, FPGAs, etc.) *loop* directive, for example, allows programmers to explicitly specify parallelizable loops. Compiler reads this hint and launches a kernel on the accelerator to execute each iteration in parallel. For CUDA-like GPU accelerators, each loop iteration runs as a CUDA thread. During the runtime, threads are grouped into warps (32 threads) and executed in lock-step. In this case, every consecutive 32 loop iterations are executed by threads of one warp.

GPUs are highly optimized to execute warps. For example, upon memory loads, if all threads of a warp request words within a 128-byte memory line, accesses are coalesced and only one memory access is made. Otherwise, if there is an address divergence, requests are serialized and served one unique 128-byte at a time. Address divergence imposes a significant overhead since the load/store unit does not maintain a request queue and the load instruction should be replayed (decoded and issued again.) The load instruction is replayed several times until all sides of address divergence send their memory request (for worst case of 32 replays per warp instruction.) This replay occupies an issue slot from the instruction scheduler/dispatcher, preventing the core from issuing new instructions.

OpenACC applications have different memory access patterns. In some applications, consecutive iterations of a loop read/write consecutive words from an array. When a GPU runs such applications, threads of a warp read consecutive words from memory. This pattern is GPU-friendly and called regular pattern or well-coalesced. In some other applications, consecutive iterations of a loop read/write arbitrary words from an array. Running such applications on GPUs potentially causes address divergence. We refer to this as irregular memory access pattern.

III. Proposed Clause

In this Section we propose a clause, named *perforation*, to allow loop perforation in OpenACC. The clause is added to *loop* directive and applies to the corresponding parallel loop. Compiler's parser reads the clause and performs loop perforation on parallel loops.

A. Notation

perforation clause is defined by the following notation:

perforation(rate)

where *rate* specifies the percentage of the loop iterations that are perforated. For example, *perforation(0.2)* denotes 20% of the loop iterations are dropped and 80% run. The dropped iterations are picked uniformly from original range of the loop. For example, if perforation rate is 20%, from every five consecutive iterations one and only one should be dropped.

B. Examples

Listing 1 presents an example of *perforation* clause usage. Without using *perforation* clause, code adds *a* and *b* vectors and stores the results in *c*. Including *perforation* clause with parameter *0.3* hints the compiler to drop 30% of the loop iterations. In this case, 30% of the iterations do not run and, for this kernel, 30% of entries in *c* remain unassigned/unchanged. Next subsection explains a mechanism to approximate values of these entries.

Listing 1. Perforated vector-vector add in OpenACC.

```
#pragma acc parallel copy(a,b,c)
#pragma acc loop perforation (0.3)
for(i=0; i<length; ++i){
  c[i] = a[i] + b[i];
}
```

C. Fix-up Code

As explained earlier, loop perforation leaves portion of output entries unassigned. *Fix-up* is our solution to efficiently approximate values for these entries. Our solution does not launch an extra kernel nor impose extra memory transfers between host and device. Instead, within the same kernel, *fix-up* code first finds unassigned entries and then approximates their values. Below we explain each step.

Finding unassigned entries: Unassigned entries correspond to iterations that are perforated (not executed.) To find unassigned entries, it is enough to find the perforated iterations. Perforated iterations can be found simply by knowing that consecutive threads should have been mapped to consecutive loop iterations, unless one or more iteration is dropped in between. Based on this fact, *fix-up* generates a code for each thread to calculate the loop iteration that is assigned to the next thread. If the iteration (that is assigned to the next thread) is immediately after the iteration assigned to this thread, there is no perforated iteration between this thread

and the next thread. Otherwise, one or more iterations have been perforated. The thread that detects this takes over the task of the perforated loop iteration(s) and approximates the values for corresponding output entries. We refer to this thread as *take-over* thread.

Approximating the value: *Take-over* thread has already computed the loop iteration that is assigned to it (computed values are in the registers and cache) and is going to approximate the values for the perforated loop iteration(s). We evaluate several approximation methods for achieving the best accuracy. The thread can simply copy the value that has obtained for its entries to the perforated entries. As we present later in the results, this approach broadly recovers the quality loss. We refer to this method as *fix-copy*. Other methods are based on reducing the values obtained from this thread with the values obtained by neighbor threads (corresponding to the neighbor entries.) To lower down the communication overhead, we limit the thread to use the output from two neighbors only (the threads before and after this thread.[1]) Therefore, the thread has three values (one from itself and two from neighbors) to approximate the perforated entry. One method is to assign the average of three values to the perforated entry, referred to as *fix-avg*. Similar methods are assigning the minimum or maximum of three values to the perforated entries, referred to as *fix-min* and *fix-max*, respectively.

D. Implementation

OpenACC applications can be translated to CUDA or OpenCL to be able to run on accelerators. Here we discuss implementing *perforation* clause in CUDA backend of OpenACC. We implement *perforation* clause by modifying following two code blocks:

Kernel launch parameters: Number of threads are lowered down by the perforation rate. For instance, if originally there are N loop iterations and perforation rate is 50%, $N/2$ threads are launched on the accelerator.

Mapping of CUDA threads to loop iterations: In original code, every loop iteration is mapped to a single CUDA thread. With perforation, every thread still runs single loop iteration but there are lower number of threads than loop iterations (this is where iterations are dropped.) The mapping of CUDA threads to loop iterations are modified to distribute the dropped iterations uniformly within the loop's range. For example, if the loop increment is $++$, $--$, $+=1$, or $-=1$, the loop iterations that are assigned to each thread are multiplied by $\frac{1}{1-PerforationRate}$ (referred to as *expansion rate*.) This distributes threads along the loop's range uniformly. Expansion rate formula depends on the loop increment and can be coded in the compiler statically for set of predefined operators (e.g. $*=$ and $\setminus=$.)

[1]Communications among neighbor threads is implemented efficiently using CUDA warp register exchange instructions which is as fast as register move.

TABLE II
SPECIFICATIONS OF THE BENCHMARKS.

Benchmark	Suite	# loops	# iterations	Quality
Backprop	Rodinia	3/5	1048576	Avg. Rel. Err
BlackScholes	PARSEC	1/1	10000128	Avg. Rel. Err
Fast Wal. Trsf.	SDK	1/1	2097152	Avg. Rel. Err
Hotspot	Rodinia	1/2	1048576	Avg. Rel. Err
Matrix Mult.	SDK	1/2	4194304	Avg. Rel. Err
Median Flt.	IPMACC	1/2	3145728	N. RMSE
Pathfinder	Rodinia	1/1	4194304	Avg. Rel. Err
Sobel Flt.	IPMACC	1/2	3145728	N. RMSE

IV. EXPERIMENTAL RESULTS

A. Methodology

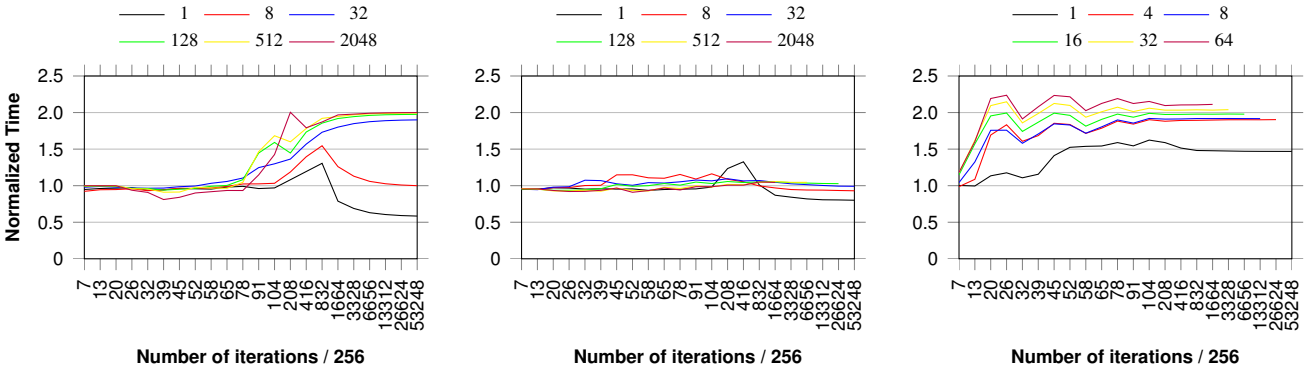
Compiler. We use our in-house framework for implementing *perforation* clause and compiling OpenACC applications. The framework is publicly available at [2]. Our framework translates OpenACC source codes to CUDA sources and then uses GNU GCC and NVIDIA *nvcc* compilers for generating host and device binaries. We use GNU GCC 4.9.3 and NVIDIA *nvcc* 8.0.

Benchmarks. We modified serial implementations of the benchmarks available in Rodinia Benchmark Suite, PARSEC, and NVIDIA GPU Computing SDK to form our benchmark set. We also included median and sobel filters from image processing domain. Initially, we modified the benchmarks to implement OpenACC version. We call this version *original OpenACC* in evaluations. This version runs on the GPU and produces the exact same result as the serial version. On top of the *original OpenACC* version, we added *perforation* clause to loop directives to create *perforated OpenACC* version of the benchmarks. In *perforated* version a portion of iterations are dropped, based on the perforation *rate* compile-time parameter. In evaluations, we compare *perforated OpenACC* version to *original OpenACC* and report execution time advantages and impacts on the quality of results, under various perforation rates. For execution time, we only report the kernel time, measured through NVIDIA *nvprof*. Table II lists the quality assessment scheme used for each benchmark. *Quality* column indicates the method for measuring the error: average relative error or normalized root-mean-square error. *loops* column reports the number of parallel loops perforated versus total parallel loops. In benchmarks with nested parallel loops (that includes all except BlackScholes, Fast Wal. Trsf., and Pathfinder), we only applied perforation to the most inner loop. *iterations* column reports the maximum number of parallel loop iterations.

Hardware. We run our evaluations on NVIDIA Tesla K20 GPU. The GPU has 13 Streaming Multiprocessors. Each multiprocessor has 192 CUDA cores and supports 2048 threads. Threads are scheduled for execution at warp (group of 32 threads) granularity.

B. Performance Analysis

To investigate the impact of application type on loop perforation's speedup, we evaluated the technique under three typ-



(a) compute-intensive
(lines represent ALU op(s)/iteration)

(b) memory-intensive regular-pattern
(lines represent load(s)/iteration)

(c) memory-intensive irregular-pattern
(lines represent load(s)/iteration)

Fig. 1. Perforation performance potential under various workload types.

ical application types: i) compute-bound (involves numerous ALU/FPU operations per iteration), ii) memory-bound regular-pattern (numerous memory reads performed per iteration, where consecutive iterations read words in the same cache line), iii) memory-bound irregular-pattern (numerous memory reads performed per iteration, where consecutive iterations read words from different cache lines.) Figure 1 reports the speedup under these application types for perforation rate of 50%, expecting 2X speedup. As reported in 1a, we start to observe speedup from perforation as the number of iterations and ALU/FPU ops increase. This is where ALU/FPU ops time starts to dominates the kernel launch time. For fairly large number of iterations and ALU/FPU ops per iterations, we observe roughly 2X speedup. As reported in 1b, speedup is not significant under memory-bound with regular memory access pattern. This is explained by the overhead of memory address divergence imposed by perforation. In the baseline, consecutive iterations access consecutive words in the same cache line. Under perforation, however, consecutive iterations are not necessarily reading words from the same line. This generates address divergence that increases instruction replay overhead [11] and the pressure on memory subsystem. This overhead ruins the advantages of loop perforation and lowers down the speedup significantly, caps at 5% when there are high number of iterations and loads per iteration. For the third application type, as reported in 1c, we observe superlinear speedup. In this type, every iteration loads one (up to 64) unique word(s) from one (up to 64) unique cache line(s), modeling irregular high-demanding memory pattern. Reducing the number of iterations through perforation yields significant speedup as this drops the memory subsystem's demand dramatically. Notice that here, in contrast to the case in memory-bound with regular pattern, address divergence will not be exacerbated by perforation. This is because the address divergence under irregular pattern is the worst case; 32 threads of the warp are fetching 32 different cache lines.

C. Benchmarks

Figure 2 to 5 present speedup versus accuracy trade-off under loop perforation, for various perforation rates and fix-up methods. We evaluated four different fix-up methods: fix-copy, fix-avg, fix-max, and fix-min[2]. Overall, all methods have very close speedup, but they differ in term of accuracy. For each benchmark, numbers are normalized to *original OpenACC* (perforation rate of 0.0). Speedup is significant for kernels with significant computations or irregular memory access patterns (e.g. Matrix Mult.) Otherwise, if the kernel has trivial amount of computations and memory accesses are regular and well-coalesced, perforation is not effective in improving performance (e.g. Hotspot and Pathfinder.) Below we explain each benchmark, separately.

1) Backprop.: This application implements backpropagation neural network training algorithm. There are three kernels and perforation is applied to all. The kernels' code is mostly composed of calculating the sum product of two arrays. Calculating the sum product is a serial iterative task. Perforation skips this serial calculation altogether and this is where the main speedup of perforation comes from. On the negative side, all memory accesses are regular and well-coalesced. Perforation ruins the regular access patterns and this mitigates the overall speedup. Perforation delivers 7% speedup under perforation rate of 0.5 (50%). Quality loss in this application is very insignificant under all fix-up methods.

2) BlackScholes.: The kernel code is composed of huge computations and eight memory accesses (seven loads and one store), where all memory accesses are well-coalesced. All memory accesses read elements of 4-byte floating-point data type. Without perforation, consecutive iterations read/write consecutive 4-byte elements. Assuming 128-byte memory transactions, every time a warp executes a load/store operation, warp's 32 threads access one consecutive 128 bytes (32×4 bytes.) Applying perforation deteriorates memory

[2]Note that quality loss and performance improvement from loop perforation without any fix-up code is reported earlier in Table I.

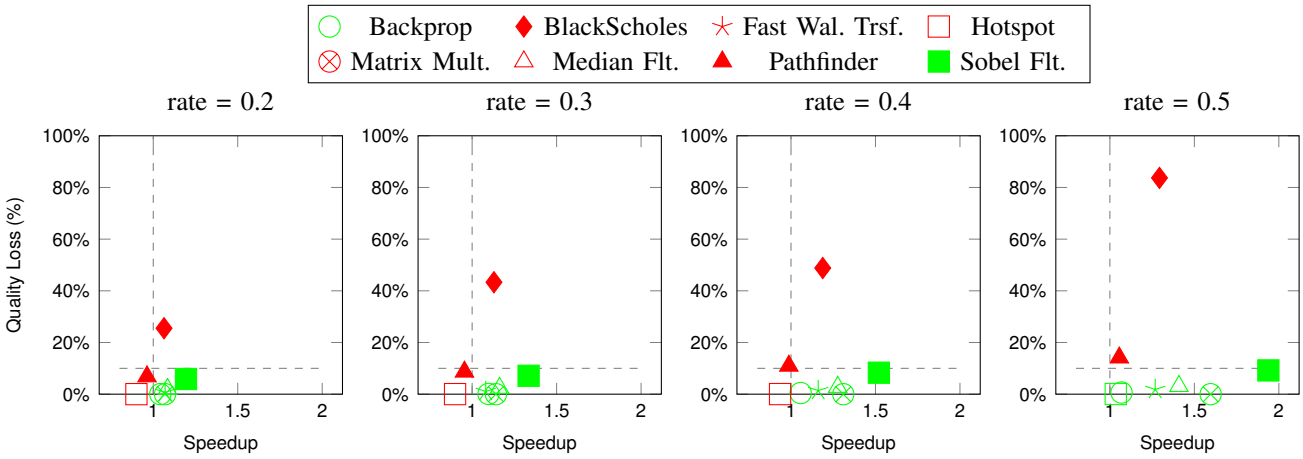


Fig. 2. Comparing speedup versus quality loss of perforated OpenACC over the baseline, under different perforation rates. fix-copy perforation fix-up is applied.

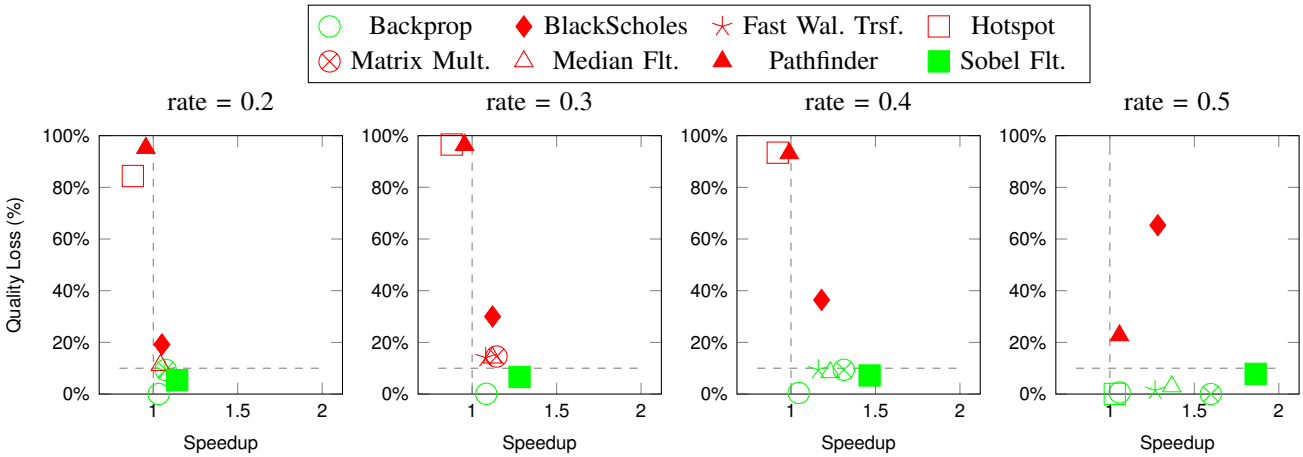


Fig. 3. Comparing speedup versus quality loss of perforated OpenACC over the baseline, under different perforation rates. fix-avg perforation fix-up is applied.

access coalescing and debilitates the speedup. For instance, perforation rate of 0.5 drops one iteration from every two iterations and doubles the memory bandwidth requirements of a warp (since two 128-byte transactions are made.) This causes address divergence within a warp and slowdowns the performance. Perforation's performance is boosted through reduction in computations of the kernel though. Overall, perforation boosts the performance through reduction in computations and slowdowns the performance by deteriorating the regular memory access pattern. Aggregating impacts on both sides, we observe tangible speedup for perforation rate of > 0.2, up to 29% under perforation rate of 0.5. Among different fix-up methods, fix-min is the most effective heuristic to recover the quality loss (Figure 5.) For perforation rate of 0.5, fix-min keeps the quality loss at 27%.

3) Fast Wal. Trsf.: The kernel code is composed of one XOR, one MAD, and three memory accesses (two loads and one store), where one of the loads is of irregular pattern and the rest are regular. Here the only opportunity for perforation to improve performance is to lower down the overhead of irregular memory accesses (no opportunity in tiny-scale computations nor regular memory accesses.) Perforation drops portion of iterations and avoids making irregular memory fetches and lowers down the pressure on memory subsystem. Speedup from perforation can be seen for perforation rate of > 0.1. For perforation rate below this range, perforation slowdowns. This can be explained by the negative impact of perforation on two regular memory accesses of this kernel (one load and one store.) These two accesses are well-coalesced and this pattern is ruined under perforation. Overall, 27% speedup is observed under perforation rate of 0.5. fix-max and fix-copy methods are very effective in recovering the quality loss in this benchmark, below 2% under perforation rate of < 0.5.

4) Hotspot.: The kernel code is composed of small number of ALU/FPU operations and 10 4-byte memory accesses (nine float loads and one float store), where all memory accesses are well-coalesced. There is no opportunity for perforation to improve performance in this kernel (tiny-scale computations and regular memory accesses.) Applying perforation deteriorates the regular memory access patterns of 10 memory accesses

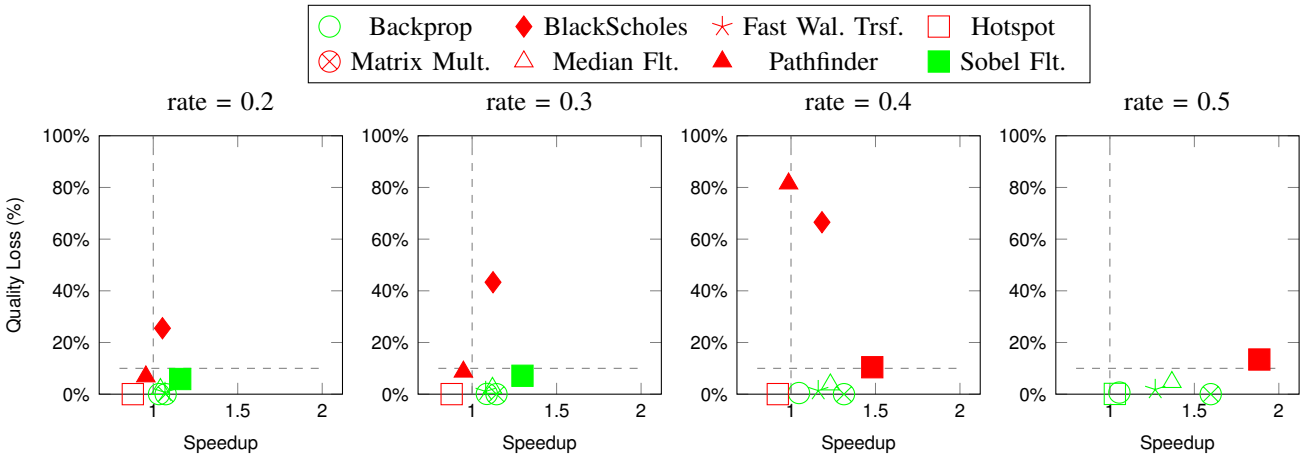


Fig. 4. Comparing speedup versus quality loss of perforated OpenACC over the baseline, under different perforation rates. fix-max perforation fix-up is applied.

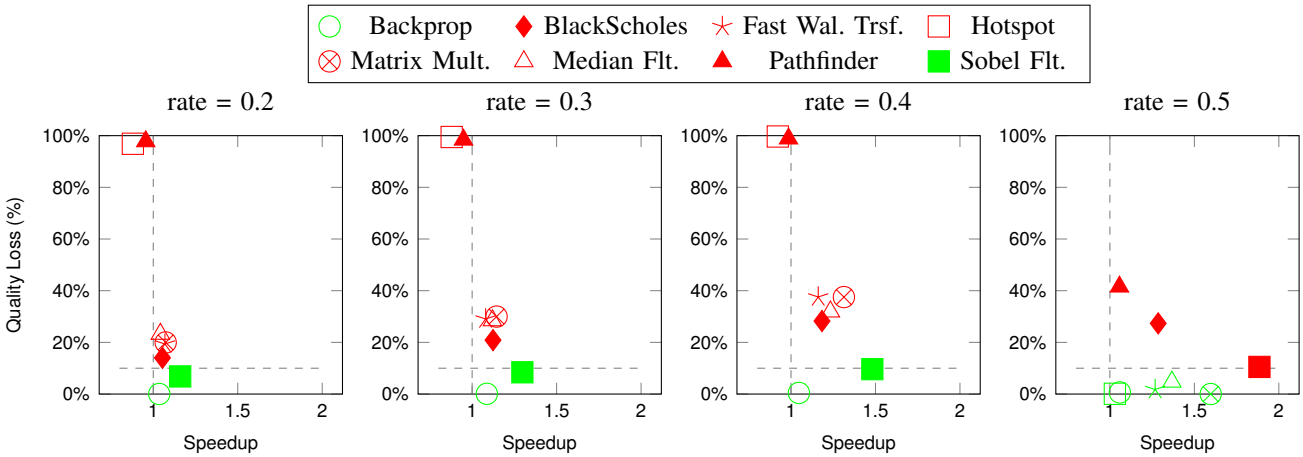


Fig. 5. Comparing speedup versus quality loss of perforated OpenACC over the baseline, under different perforation rates. fix-min perforation fix-up is applied.

and significantly lowers the performance. In this benchmark, the output is a 2D matrix representing the temperature of a processor chip. The kernel is called hundreds of time and each kernel calculates temperature variations in a time step. Intrinsically, neighbor elements of the matrix have very close values at the end (thermal conduction gradually equalizes the temperature of neighbors particles.) This explains why fix-copy recovers the quality loss dramatically keeping the quality loss below 1%. Also, the simulation starts at the point where elements have a low temperature and gradually grow to a large number. fix-max captures this trend and delivers very high accuracy (while fix-min opposes this trend and delivers a very low-quality approximation.)

5) Matrix Mult.: The kernel code is composed of a serial loop iterating through two arrays, one column-wise (regular pattern) and the other row-wise (irregular pattern), to calculate the sum-product of two vectors. The serial loop iterates for 2048 times, generating 1024 regular and 1024 irregular memory accesses. This, on the positive side, leaves a huge room for perforation to improve performance by minimizing irregular memory accesses. On the negative side, perforation deteriorates pattern of irregular memory accesses. Aggregating the impacts from two sides, perforation delivers speedup of 11% to 60% under perforation rates of 0.2 to 0.5. In Matrix Mult., the quality of the output highly depends on the input. Under every input, however, a row is multiplied by a column to find a single entry in the output. Two consecutive columns of a row work on the same row, but different columns, implying that 50% of the input is the same. This explains why fix-copy has a very high accuracy, below 1% quality loss under perforation rate of < 0.5.

6) Median Flt.: The kernel code is composed of huge computations and 11 memory accesses (7 loads and 4 stores), where all memory accesses are regular. Although the opportunity for increasing the performance through minimizing memory demand via perforation is not possible, but there is a significant opportunity in improving performance through lowering down the computations. Computations involve sort-

ing a nine-element array, 27 modulus operations, 27 divisions, and hundreds of additions, subtractions, and multiplies. Storing the array, modulus and divisions are expensive operations. Perforation improves performance by lowering down the computational demand. Perforation overhead for deteriorating memory access pattern is not significant since the memory requests are of *char* type (1 byte per element). For 1 byte data type, grouping well-coalesced accesses from 32 threads of a warp demands 32 bytes. Although each warp load demands 32 bytes, SM reads 128 bytes from global memory (since memory transaction size in this GPU (Kepler K20) is 128 bytes.) Therefore, as long as perforation rate is < 0.75, no extra address divergence will be exposed by perforation (0.75 means dropping 3 of every 4 iterations, quadrupling the memory demand of regular memory accesses.) Overall, perforation delivers 5% to 48% speedup under perforation rates of 0.1 to 0.5. For the input matrices, we tried different combinations of Hilbert, Lehmer, and Redheffer matrices and observed very similar results. Among the fix-up methods, fix-copy and fix-max deliver very high accuracy. In images, neighbor pixels generally have a very close intensity. This is why fix-copy works well for recovering the quality loss.

7) Pathfinder.: The kernel code is composed of six ALU operations and five memory accesses (4 loads and 1 store), where all memory accesses are regular. Applying perforation doubles memory address divergence and, since there is no performance advantages from lowering down the computation side of the kernel, all the address divergence translates to performance slowdown. Pathfinder is an iterative benchmarks running a single kernel hundreds of times. Numerous iterations of the kernel propagates the error across entire entries and, regardless of perforation rate, quality loss is significant. Among different fix-up methods, fix-copy performs the best in term of accuracy, keeping the quality loss below 14%.

8) Sobel Flt.: The kernel code is composed of a hundred of ALU and control-flow operations and seven memory accesses (5 loads and 1 store), where all memory accesses are of regular pattern. Perforation delivers significant speedup in this benchmark by lowering the computations of the application. While the memory accesses are well-coalesced, perforation does not increase the memory demand, since the data type is 1-byte (refer to the explanations of *Median Flt.* above for discussions on this.) All fix-up methods work very well in lowering down the quality loss. In Sobel Flt., most of the output entries are zero, which is equal to not assigning any values to the output. This is why even applying no fix-up method, as reported in Table I, delivers acceptable quality loss. Also this output is image, and images exhibit very high data similarity around neighbor pixels in general.

V. Discussion

Auto-tunning. Our proposal relies on the programmer to set the perforation rate to a number that produces reasonable results. We believe this is not challenging for OpenACC programmer since only one single parameter should be adjusted in our proposal and the optimal value of this parameter can be found in linear time. However, to offload this task from programmer, profiling approaches similar to [6] and [9] can be used.

Load balancing. Most accelerators (e.g. GPUs) run a group of threads (or loop iterations) together over the SIMD (referred to as warp). If there is a load imbalance among the thread group, one of the threads might run longer while others are done. This drops SIMD utilization and degrades performance. If loop iterations have different runtime (duration), loop perforation impacts load balancing. In the benchmarks that we investigated here, we did not observe load imbalance after loop perforation. We leave further investigation on this issue to future work.

VI. Related Work

Mittal [3] presents a survey of approximate computing techniques, programming frameworks for approximate computing, and techniques for using approximate computing in various computing platforms and memory technologies. Below we overview language extensions, loop perforation, directive-based, and GPU-related approximate computing techniques.

A. Language Extensions

Sampson et al. [7] propose ACCEPT compiler for approximate computing. ACCEPT compiler reads approximation hints (C/C++ type qualifiers) from the programmer and also includes compiler passes to identify approximable codes. In practice, the programmer hints in the code initiate approximate computing and compiler passes explore the applications dataflow afterwards and suggest more approximation opportunities. Then the programmer inspects compiler analysis and injects more hints. This feedback loop effectively allows spanning approximate computing across the code step by step. Also various relaxation methods from literature has been integrated in the framework. An autotuner is proposed to heuristically explore the space of possible relaxed programs to identify Pareto-optimal variants.

Park et al. [4] propose a small set of language extensions for practical approximate computing in Java. FlexJava compiler automatically infers approximable operations and data from the original programmer hints and selectively includes them in approximation. FlexJava promises more efficient and productive approximate computing extensions than previous work [8].

B. Loop Perforation

Our work is different from the loop perforation technique proposed by [9] in two ways. Firstly, while the technique proposed in [9] does not compensate for perforated loop iterations, our approach runs a tiny post-kernel fix-up code to compensate for these iterations. Secondly, the technique proposed in [9] considers all loops as candidates for perforation. This will unmanageably grows the size of optimization space and requires auto-tunning to consider interaction between perforated loops and finding the optimal configuration. Our work, however, is applied to a subset of loops (parallel loops). This

keeps the interaction between perforated loops manageable and a programmer can tune the parameters through directives, with an insignificant development effort. In summary, our work is tailored to output-parallelized directive-based parallel APIs like OpenACC.

C. Directive-based Techniques

Rahimi et al. [5] propose two new OpenMP directive, referred to as the *accurate* and *approximate* directives. Directives allow programmers to annotate a code block for accurate or approximate computing. The difference between accurate and approximate code blocks is that the floating point instructions in an approximate code block are executed on energy-efficient less-precise floating point unit (FPU) in the hardware. The proposed directive trades accuracy for energy-efficiency and requires compiler and hardware support to achieve this. To *bound* the quality loss, they provide a knob to programmer to specify the floating point precision requirements. This hint is passed to the compiler through a clause in the *approximate* directive.

Vassiliadis et al. [10] propose a directive-based extension for OpenMP task model to trade accuracy for energy-efficiency. In practice, the programmer submits numerous tasks and configures a *significance* parameter for each task. Significance ranges from 0.0 to 1.0, where 1.0 means the task should be executed accurately and 0.0 means the task can tolerate approximation. Based on the the significance recorded per task, runtime system schedules the tasks in the goal of delivering best energy-efficiency through approximate computing.

D. GPU-based Techniques

Figurnov et al. [1] propose a novel approach to address computational complexity of convolutional neural networks. They propose to trade accuracy for performance and speedup convolutional layers by skipping their evaluations when there is a strong spacial locality. They evaluate their proposal under both CPU and GPU and show that this approach provides 2-4.2X speedup, at the cost of up to 9.9% of error.

Yazdanbakhsh et al. [12] investigate opportunity to replace segments of GPU kernels with neural approximation. First, they show that on average, 58% of the runtime is spent in approximable segments. Based on this strong potential for neural approximation of GPU kernels, they introduce modifications to GPU architecture to implement an efficient neural accelerator using GPU cores. Their evaluations show that GPU+DNA (GPU plus neural accelerator) architecture achieves 1.9X speedup over the baseline GPU architecture in executing GPU kernels.

Our work is different from Paraprox [6] in two ways. Firstly, Paraprox replaces the original kernel with an approximate kernel, running the same number of threads. This will impact all results since the running kernel is different. Our work, however, runs the exact same kernel but on lower number of threads. Our work accurately calculates a subset of outputs by lower number of threads and approximates the remaining outputs from the calculated set. Secondly, Paraprox requires a profiling phase to configure the parameters of approximated kernel while our proposal relies on the rate parameter adjusted by the programmer.

VII. Conclusion

In this work we evaluated a directive-base technique in OpenACC for approximate computing. The technique shrinks the complexity of the task by an adjustable rate to obtain samples of the output. From these samples, a large portion of output is approximated with very high accuracy. In our technique, a programmer uses the adjustable rate as a knob to trade output quality for performance. We also investigated the type of applications that can benefit from perforation. Compute-intensive applications as well as memory-intensive applications with irregular-pattern can take advantage of perforation and boost performance. On the other side, perforation has sublinear speedup impact on applications with regular memory pattern. Investigating synthetic and real benchmarks, we found the technique very effective in improving performance, at a reasonable quality loss.

References

[1] M. Figurnov, A. Ibraimova, D. Vetrov, and P. Kohli, "PerforatedCNNs: Acceleration through Elimination of Redundant Convolutions", 30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

[2] IPMACC OpenACC Framework. Available: https://www.github.com/lashgar/ipmacc

[3] S. Mittal. "A Survey of Techniques for Approximate Computing", ACM Comput. Surv. 48, 4, Article 62 (March 2016), 33 pages. DOI: https://doi.org/10.1145/2893356

[4] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. "FlexJava: language support for safe and modular approximate programming", In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 745-757.

[5] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters", In Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '13). IEEE Press, Piscataway, NJ, USA, , Article 35 , 10 pages.

[6] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. 2014. Paraprox: pattern-based approximation for data parallel applications. SIGPLAN Not. 49, 4 (February 2014), 35-50.

[7] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, M. Oskin, "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing", UW Tech Report (2015).

[8] A. Sampson et al. "EnerJ: approximate data types for safe and general low-power computation", In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11). ACM, New York, NY, USA, 164-174.

[9] S. Sidiroglou-Douskos et al. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE '11). ACM, New York, NY, USA, 124-134.

[10] V. Vassiliadis et el. "A programming model and runtime system for significance-aware energy-efficient computing", In 1st Workshop On Approximate Computing (WAPCO'15).

[11] D. Wodniok et al. 2013. Analysis of cache behavior and performance of different BVH memory layouts for tracing incoherent rays. In Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '13). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 57-64.

[12] A. Yazdanbakhsh et al. "Neural acceleration for GPU throughput processors", In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48). ACM, New York, NY, USA, 482-493.