GraphTuner: An Input Dependence Aware Loop Perforation Scheme for Efficient Execution of Approximated Graph Algorithms

Hamza Omar, Masab Ahmad and Omer Khan
University of Connecticut, Storrs, CT, USA

***Abstract*—Graph algorithms have gained popularity and are utilized in high performance and mobile computing paradigms. Input dependence due to input graph changes leads to performance variations in such algorithms. The impact of input dependence for graph algorithms is not well studied in the context of approximate computing. This paper conducts such analysis by applying loop perforation, which is a general approximation mechanism that transforms the program loops to drop a subset of their total iterations. The analysis identifies the need to adapt the inner and outer loop perforation as a function of input graph characteristics, such as the density or size of the graph. A predictive model is proposed to learn the near-optimal loop perforation rates using synthetic input graphs. When the input-aware loop perforation model is applied to real world graphs, the evaluated graph algorithms systematically degrade accuracy to achieve performance and power benefits. Results show ∼30% performance and ∼19% power utilization improvements on average at a program accuracy loss threshold of 10% for an NVidia® GPU. The analysis is also conducted for two concurrent Intel® CPU architectures, an 8-core Xeon™ and a 61-core Xeon Phi™ machine.**

I. Introduction

Graph algorithms working on structured and un-structured data have surpassed great computational complexities and memory requirements. Large-scale graph datasets, such as weather and transportation models require *sensor-to-decision* processing [1]. Due to the humongous available set of these diverse input graphs, sensitivity to data variations results in performance fluctuations. Therefor, the notion of reducing complexity falls on the target algorithms and data that is run for decision analytics, which is why approximate algorithms are utilized to improve performance.

Exact graph algorithms are used in various applications, such as the Bellman-Ford algorithm [2] that computes shortest paths, and PageRank [3] that ranks webpages. However, their exact nature exemplifies that a complete set of iterations and data is needed to be processed to get an optimal result. Undermining this exact nature results in approximations, where reduced complexity can be obtained whilst approximating output accuracy. When *input dependence* comes into play, approximation thresholds and other program level parameters change. Hence, performance constraints in such aspects becomes variable, and need further analysis for optimizations.

Many of today's graph applications must guarantee a response within a specified time constraint [4]. While designing such applications, the goal is to reduce the amount of computations for these time-critical systems, yet achieve an optimal quality of service (QoS). Loop perforation [5], [6] provides a general technique to trade accuracy for performance by transforming algorithmic loops to execute a subset of their iterations. It has the potential to achieve optimum performance, alongside a reasonable output accuracy for a given accuracy loss threshold. Prior work [6] shows that perforating appropriate loops within an application not only ensures reduction in computations, but also yields a managed decrease in output accuracy. This leads to an accuracy versus efficiency tradeoff.

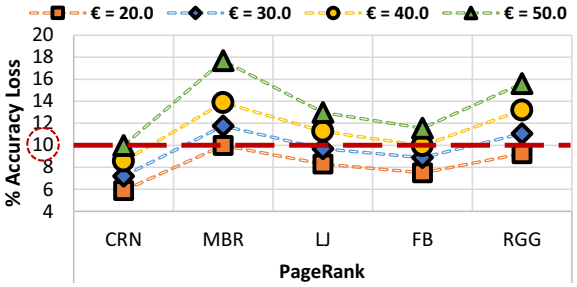

Fig. 1: Perforation rates vary the output accuracy across various input graphs for the PageRank graph algorithm.

In the context of graphs, perforations can be done on input graphs or graph algorithms. When perforating input graphs, many perforated graphs are stored and later looked up once a perforation rate is selected. However, in the case of perforating graph algorithms, additional computation steps are added to the code, while the graph itself is left unmodified. The algorithmic overheads are difficult to compensate with benefits acquired by loop perforation. We justify this observation in this paper, and observe that loop perforation is useful when the graphs are perforated. This leads to space overhead that is mitigated using a novel loop perforation predictor that can be used to store the additional perforated graph for future use. The evaluation justifies that applying loop perforation to the input graph leads to reductions in computations, and thus performance improvements and energy reductions. Furthermore, the proposed predictor is capable of choosing the right perforation rate that is input aware. This is fundamentally different from prior algorithmic works [7], where a single perforation rate is applied to the algorithm for all inputs.

Due to many available benchmark-input combinations for graph algorithms, statically assigned perforation rates show variations in output accuracy because of input dependence [8], [9]. This naive perforation approach is also shown in Figure 1, where different input graphs (CRN is a sparse CA road network, MBR is a dense Mouse brain graph, and LJ, FB, and

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.38

RGG are social networks) for the PageRank benchmark give different accuracy outcomes for a single perforation rate. If a user provides an accuracy loss constraint of 10%, then all the shown peroration rates pass the threshold constraint for CRN. However, for the remaining graphs, 50% perforation rate is not acceptable as it results in a higher accuracy loss. To satisfy the accuracy constraint for these input graphs, perforation rates lower than 50% are required. Consequqntly, for MBR and RGG, the perforation rate of 20% meets the accuracy threshold. For FB and LJ, perforation rates of 40% and 30% are used. This shows that for a given accuracy threshold there is a need to select the right perforation rate based on the input graph characteristics. If the selected perforation rate is lower than the optimal, it results in lost opportunity for efficiency gains. On the other hand, if the selected rate is higher, then the accuracy threshold is violated. Therefore, the perforation rate should be selected such that the algorithm operates close to the accuracy threshold for efficient execution, and not violate this constraint.

This paper analyzes input dependence of graphs to formulate a prediction paradigm that selects loop perforation thresholds. As graph algorithms primarily consist of inner and outer loops (traversing edges and vertices respectively), perforation on these loops is done to improve efficiency. Perforating in a combined fashion can be captured by exhaustively exploring the perforation space, or by predicting perforation rates based on graph characteristics. A predictor is useful in real-time setups where applications exhibit timing constraints for processing graphs. We propose offline learning that reasons contextually for graph characteristics, and captures optimal perforation rates using a wide range of representative graphs. The learned predictor is evaluated online for real graphs to select the perforation rates for a given accuracy threshold. The objective of the predictor is to select accurate yet fast decision to choose the perforation rates that operate close to the accuracy threshold, and maximize the number of perforated edges in the graph.

This paper makes the following contributions:

1) We show that by perforating input graphs instead of graph algorithms, performance can be extracted from loop perforation in graph analytics.
2) We identify the challenge with input dependence in loop perforation for graphs. The proposed input-aware perforation predictor enables the graph algorithm to produce an output that satisfies program accuracy threshold, while maximizing the number of dropped edges.
3) The performance and energy gains due to loop perforation of graphs are evaluated for various CPU and GPU architectures.

II. Related Work

A significant amount of work on approximating graph algorithms has resulted in a wide range of heuristic algorithms. Examples include shortest path approximations [10] and streaming algorithms [11]. Δ-stepping [12] is a well-known implementation of the shortest path problem that classifies vertices as light or heavy based on their connectivity and edge weights. Based on this classification different vertices are relaxed iteratively using different iteration counts. Other works follow different flavors of approximations for light and heavy vertices [13]. However, such works do not analyze approximate graph algorithms across various real graph inputs, as well as diverse architectures. Such analysis is required in today's computational world where new compute paradigms are gaining momentum, such as multi-architecture compute nodes [14]. Analysis is done across accuracy, performance, and power for a diverse set of graph benchmark–input combinations.

Loop perforation enables a general approximation strategy by dropping a subset of total loop iterations. Prior works, such as [6], [15] use loop perforation to observe the performance and quality of service tradeoff space. These works focus on generic applications that do not include graph analytics. However, analyzing graph algorithms is the primary objective of this paper, and a novel input dependence aware loop perforation strategy is proposed.

III. Input Dependence Aware Loop Perforation

A. Loop Perforation in Graph Algorithms

In general, graph algorithms consists of two types of loops. The *outer loop* shown in Algorithm 1 Line 5, iterates over all vertices in the input graph network. For each vertex, the *inner loop* (shown in Algorithm 1 Line 6) traverses over all the neighbors to compute based on the connectivity and weights on the connected edges. Graph algorithms may consist of different phases and/or iterations of the inner and outer loops.

Algorithm 1 Generic Structure of Graph Algorithms

```
1: Total Vertices and Edges per Vertex: N, DEG
2: Set of Vertices: V = {v_1, v_2, ..., v_N}
3: Set of Edges: E_V = {e_1, e_2, ..., e_DEG}
4: \* May Iterate Multiple Times *\
5: for each v ϵ V do
6:     for each e ϵ E_v do
7:         \* Do Computations *\
```

Loop perforation can be applied to graphs at two different regions, namely *inner loop* and *outer loop*. Perforating the inner loop iterations implies perforating the edges of a vertex in the graph. This approach randomly drops the edges of any arbitrary vertex, as depicted in Figure 2. On the other hand, perforating outer loop iterations results in dropping the vertices from the input graph. Based on the perforation rate, vertices along with their connected edges are dropped in the input graph, as shown in Figure 2. Note that the proposed approach focuses on perforating the graph network based on graph parameters rather than algorithm-level perforations. Further justification of this perforation strategy is discussed in the next subsection.

The inner and outer loop perforation schemes exhibit tradeoffs in efficiency and accuracy. Inner loop perforation results

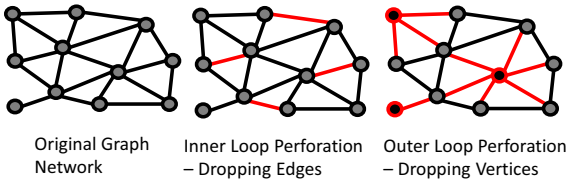

Fig. 2: An example of applying various perforation strategies on graphs.

in dropping less computational work, however it provides fine-grain quantization of accuracy loss, specially for relatively dense graphs. Taking sparse graphs as an example, which have limited edge connectivity, inner loop perforation needs a very high perforation rate to impact accuracy. However, in dense graphs there is more dependence on edges, and hence higher accuracy losses are expected. On the other hand, outer loop perforation results in more accuracy loss as compared to inner loop perforation. This strategy enables better control over accuracy loss for sparse graphs. However, as the density of the graphs increases, outer loop perforation results in higher accuracy loss. This is due to the fact that outer loop perforation skips more computations as vertices along with their edges are dropped. The goal of this paper is to find the optimal point where perforation rates are selected such that they result in an accuracy loss closest to the accuracy threshold, while maximizing the number of edges dropped. A systematic combinatorial search space of the outer (ϵ_o) and inner (ϵ_i) loop perforation rates can be conducted to find the near optimal accuracy loss. Loop perforation is done randomly based on the parameters of the input graph network. To conduct a rigorous analysis, various outer loop perforation rates (ϵ_o) are considered, and for each ϵ_o the input graph is subjected to various inner loop perforation rates (ϵ_i). The analysis is performed over all the benchmark–input combinations.

After random perforation of a graph there exists a scenario where a single vertex connecting two portions of an input graph gets perforated. This would result in a graph disconnect, which could cause algorithmic inaccuracies. To circumvent such a scenario, a *connection-check* step is added towards the end of the perforation framework. A connected components algorithm, with a complexity of $O(logn)$, is applied to the perforated graph which detects if a set of vertices are not connected to any part of the given graph. Multiple components in a given perforated graph will trigger a roll-back of the perforation step, which would then re-run another random perforation. This is done iteratively until a perforated graph is acquired with no multiple components.

B. Perforating Graphs versus Algorithms

Loop perforation can either be done at the algorithm level, or at the input graph level. Nevertheless, both perforation strategies have their pros and cons. In prior works for general applications [6], loop perforation is done at the algorithm level by dropping pre-calculated number of iterations from the program loops. Dropping first or last **x** number of iterations of the loop is a common practice when performing program level perforation. Moreover, loop iterations can also be dropped randomly within the algorithm. However, program level random perforation incurs large overheads as it requires random selection of loop iterations to be dropped. Random perforation can also be done at the input graph level by dropping vertices/edges rather than dropping loop traversals that iterate over all the vertices/edges. Table I shows the comparison between program level perforation and graph level perforation. In this table, **% Edges Dropped** refers to the computations dropped as a consequence of perforation. **Accuracy** and **Performance** quantify the accuracy loss and performance numbers, respectively, as a result of the applied perforation scheme. Following insights are observed from the comparison.

Program Level Perforation			
Strategy	**% Edges Dropped**	**Accuracy**	**Performance**
Dropping Random Iterations	49%	9.92	3× Loss
Dropping First Iterations	5.1%	9.89	20% Loss
Dropping Last Iterations	5.1%	9.89	13% Loss
Input Graph Level Perforation			
Strategy	**% Edges Dropped**	**Accuracy**	**Performance**
Dropping Random Vertices/Edges	49%	9.92	26% Gain

TABLE I: Comparison between optimal Program level perforation schemes and Graph level perforation for an accuracy threshold of 10%.

1) For analyzing perforation in graph algorithms, it is better to drop loop iterations randomly (49% dropped edges) because this scheme allows more computations to be dropped as compared to systematically dropping loop iterations at the beginning or end of a loop (5.1% dropped edges). However, introducing random functions to determine which loop iteration to drop adds significant computations to the graph algorithm. This results in an overall performance loss since loop perforation benefits do not compensate for the algorithm level overheads.
2) Within random perforation, it is beneficial in terms of performance to randomly drop vertices/edges from the input graph itself, rather than dropping respective loop iterations from the algorithm.

When the resultant graph obtained from dropping vertices/edges is provided to the graph algorithm, it achieves an average performance gain of 26% with respect to the original run of the algorithm. However, with program level random perforation, a huge performance loss of 3× is observed. This is due to the fact that *rand()* system call that is used for uniform distribution incurs high overheads. Hence, to avoid such performance hits during execution, we perforate the input graphs provided to the application. Graph level perforation

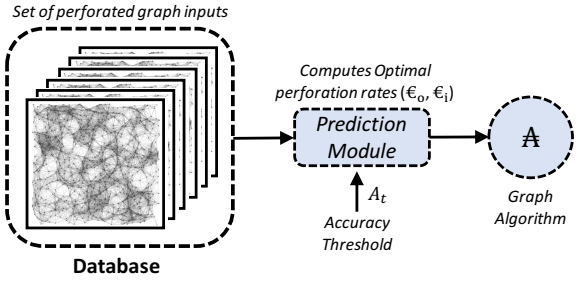

Fig. 3: Design flow to obtain close to optimal perforation rates for every benchmark-input combination.

not only provides better performance but it also preserves the exactness as the algorithm itself is never altered. Nevertheless, this random perforation strategy introduces a compute – space tradeoff, as introduced earlier in Section I. Perforating graphs requires a database to store the set of perforated graphs but provides higher performance gains. On the other hand, program level random perforation has no space limitations but incurs computational overheads.

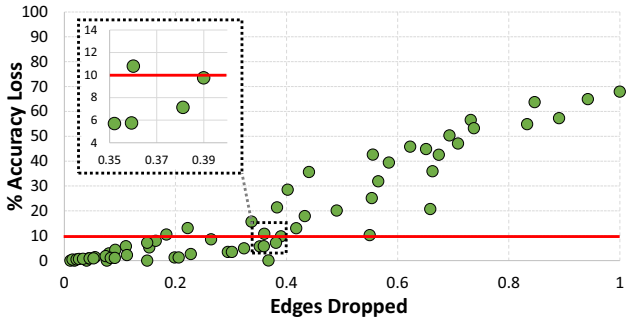

Fig. 4: Edges Dropped Vs. Output Accuracy tradeoffs for Triangle Counting - CRN.

C. Prediction Models for Selecting Graph Perforation

Due to sub-optimal accuracy and performance outputs acquired using static perforation (explained in Section I), it is imperative to use dynamic perforation rates for each input graph. A learning model framework is therefore developed (shown in Figure 3), that takes in a graph benchmark-input combination and an accuracy threshold from a user, and outputs the perforation rate for both inner and outer loops. Two different learning models are presented for learning on perforated graphs. Multiple Non-Linear Regression [16] exhibits low performance overhead, and tolerable output learning accuracy. The Multi-Layer Perceptron (MLP) learner [17] learns on the non-linear aspects of accuracy and performance tradeoffs [14] in perforated graphs. These non-linear aspects are exhibited in Figure 4, where for a given accuracy threshold of 10%, few points exist that can be potentially picked up by a predictor when selecting edges dropped by perforating the graph. The goal is to maximize the output accuracy loss close to 10%, but also maximize the performance by selecting the highest number of edges dropped via perforation. The zoomed

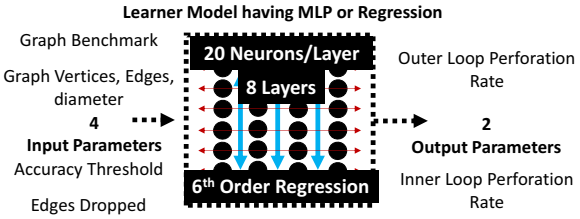

Fig. 5: Learner Paradigm with Inputs, Outputs, and the different applied learners.

version in Figure 4 shows these potential points. Prediction accuracy needs to be high enough to correspond to these two variables, and the point with highest performance must be selected. This classification accuracy is bounded to 5% within the optimal point, as in various prior works [5], [6].

For training the learners, uniform random graphs [18] are used as synthetic training inputs, as they precisely model representative real-world graphs. These input graphs correlate directly with perforation rates, and based on this insight the learner learns a wide range of input graph characteristics, such as graph size, degree, and diameter to name a few. Inner and outer perforations are amalgamated into a single parameter of edges dropped, which is then used for learning. The generated synthetic inputs are run with all target graph benchmarks, as well as with various inner and outer loop perforation rates. Accuracy results are computed as outputs for each benchmark-input combination, and used for the learning process to find a near-optimal perforation rate. Predicted results are required to be within 5% of the optimal accuracy point, as well as within 5% of the optimal point representing the number of dropped edges. This process is repeated for all given benchmark-input-accuracy threshold combinations. For a 90% classification accuracy, a 6th order equation is required for the non-linear regression model. For the MLP learner, a neural network with 160 neurons provides 90% classification accuracy. The high complexity of the learners pertain to the stringent 5% error requirement. A given learner model then outputs the perforation rate pair (ϵ_o, ϵ_i) for the user accuracy loss threshold, as shown in Figure 5. Variations of the regression and MLP model are evaluated in the evaluation section, along with their respective accuracies and overall average acquired performance.

IV. Methodology

A. Machine Settings

An *8-core Intel® i7, NVidia® Maxwell™ GPU [19]*, and a large-scale *Intel® Xeon Phi™ multicore* [20] are used for evaluation. Nvidia GTX-970 GPU is used in which the work group size, which specifies the number of worker-threads operating on a local memory chunk, can be varied from 1 to 1024, while the total thread count can be varied from 1 up to a million threads. A Xeon Phi 7120P is used as a large-scale target multicore. It has 61 Intel 4-way multi-threaded cores leading to a maximum exploitable concurrency of 244 threads.

Input Graphs	# Vertices	# Edges	Avg.Deg.
CA. Road Net. (CRN)	1,965,206	2,766,607	1.41
Mouse Retina 3 (MBR)	562	577,350	1027
Facebook (FB)	2,937,612	41,919,708	14.3
Livejournal (LJ)	4,847,571	85,702,475	17.6
rgg-n-2-24 (RGG)	16,777,216	387,553,689	23.1

TABLE II: Input Graph Networks [24]

Completion times are measured for all graph workloads at the selected pair of perforation rates (ϵ_o, ϵ_i). These are measured only for the kernel function (in the case of GPU and Xeon Phi), which is the time spent in the parallel regions in the multicore systems. All the completion times presented in this paper are normalized to the baseline performing computations over the original input graph network. The evaluation overhead of $\sim$3.3*ms* for the 160 neurons MLP prediction module is added to the overall completion time of each benchmark–input combination. The 6th order regression learner is more complex than the MLP, hence its overhead of $\sim$8.1*ms* is added for performance comparisons in Section V-D.

Power numbers are recorded at (ϵ_o, ϵ_i) for all the workloads. The power utilization results are computed for all architectures to observe the variance across different systems. Power is reported for the core package as well as the DRAM accesses made by parallel execution regions of the application. Moreover, all these results are normalized to the baseline simulations to observe improvements.

B. Benchmarks and Inputs

Graph benchmarks are acquired from the CRONO [21], Rodinia [22], and Pannotia [23] benchmark suites. These consist of Single Source Shortest Path (SSSP), PageRank, Triangle Counting (TRI-CNT), Community Detection (COMM), and Connected Components (CONN-COMP). In certain cases a GPU version of the workload is rewritten to allow interfaces to different input graphs. However, the CPU version is application for the Xeon Phi simulations as well. All benchmarks use compressed sparse row (CSR) representations for input graphs. To emulate the runtime environment, program analysis is evaluated and performed using real world graphs [24], [25], as shown in Table II.

Training graphs consist of various synthetically generated graphs with variable vertex and edge counts (degree). 32 input graphs are used in this regard, with vertices varying between 128 and 16 million, and edge counts per vertex varying between 2 and 4096. Based on the accuracy results obtained from inner and outer loop perforations, the prediction module computes close to optimal pair of perforation rates (ϵ_o, ϵ_i) out of different combinations (256 pairs for each benchmark-input combination = 16 outer loop perforation rates * 16 inner loop perforation rates) based on a given accuracy threshold to maximize the efficiency. This means for each outer loop perforation rate, 16 inner loop perforation rates are applied. For training, this results in $40,961$ accuracy points, that are acquired via 32 synthetic graphs and 5 benchmarks.

C. Accuracy Analysis

Program output accuracy is quantified for all benchmark-input combinations using the combined inner–outer loop perforation scheme. The evaluation is presented for an accuracy threshold of 10%. However, performance improvements vary with accuracy threshold constraints. Average performance numbers are also reported for different accuracy thresholds.

The outputs of algorithms with perforated graphs are compared with the outputs of un-perforated graphs. For example, in the case of SSSP and other algorithms with output arrays, the output solution arrays are compared, and their percentage differences are analyzed. Other algorithms have single outputs such as the total triangle count in triangle counting, which are compared for accuracy metric. In order for the learner to select a near optimal perforation rate pair (ϵ_o, ϵ_i), the learning space would become large if all the combinations are taken into account. Therefore, to reduce the space, we analyze proposed perforation scheme at various inner and outer perforation rates combinations ranging from 1% to 90% with increments of 5% (making 256 combinations in total) to observe the effects of perforation on accuracy of algorithms. Accuracy is evaluated for all the benchmarks by generating perforated graph networks. Perforated versions of all input graphs (shown in Table II) are generated by randomly selecting vertices or edges. Multiple simulations (1000 in our analysis) are run to determine a final perforated graph for a specific perforation method at a certain perforation rate. Accuracy quantification varies from one benchmark to another. For example, for TRI-CNT, accuracy is quantified by comparing the triangle counts obtained from the golden run with the triangle counts evaluated by the simulations running perforated graphs. On the other hand, the accuracy for PageRank is quantified by comparing the golden *rank* values for each vertex with the *ranks* computed using the perforated graphs.

V. Evaluation

The evaluation analyzes results from varying the inner and outer perforation rates, and shows the impact on GPU performance, power, and accuracy of each benchmark-input combination. Further analysis compares various perforation strategies, with and without the input-aware loop perforation learner (utilizing the neural network learner), for various parallel machines.

A. Accuracy Results

The combinations of outer and inner loop perforation rates (ϵ_o, ϵ_i) are used for quantifying accuracy losses at a given accuracy loss threshold. However, because of the perforation rate space being too large due to a large number of (ϵ_o, ϵ_i) combinations, accuracy results for this exhaustive space are not shown in this paper. Figure 6 shows the accuracy loss of algorithms for different input graphs when subjected only to outer loop perforation. Various accuracy results are acquired on the Xeon CPU by running graph algorithms at different perforation rates. The accuracy losses for each benchmark in Figure 6 are reported with the increasing order of the density

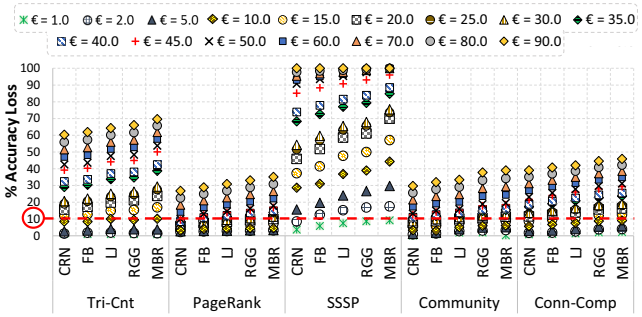

Fig. 6: % Accuracy loss for benchmarks with *outer loop* perforation for different inputs at various perforation rates.

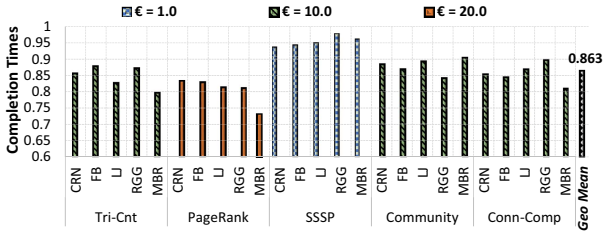

Fig. 7: GPU performance improvements for the naive loop perforation where outer loop perforation rates are statically chosen on a per-benchmark granularity.

of the input graph. As shown in Table II, it is evident that the density (D) of the input graphs increases in the order $D_{CRN} < D_{FB} < D_{LJ} < D_{RGG} < D_{MBR}$. The following observations are summarized based on the collected data:

1) As perforation rates increase, the accuracy losses are observed to rise for all benchmark-input combinations.
2) The density and connectivity of the input graph plays an important role in output accuracy. With the increase in graph density, more computations are dropped as a consequence of perforation for a given accuracy loss threshold.

As expected, the effects of perforating edges or vertices are different for each benchmark-input combination. This variation shows the importance of capturing *input dependence* for graph algorithms. Similar trends are observed when all the benchmark-input combinations are subjected to inner loop perforation (results not shown). However, the accuracy losses are low as compared to outer loop perforation. This is due to the fact that inner loop perforation drops less computational work compared to outer loop perforation. We observe that on average ∼36.89% more computational work is dropped via outer loop perforation in comparison to inner loop perforation. Similar trends are observed when both the outer loop and inner loop perforation spaces are considered together.

B. Statically Choosing Perforation Rates

Figure 7 shows the performance improvements for the case when outer loop perforation rates are statically chosen for all

Fig. 8: Normalized GPU completion times, power, and accuracy loss for each benchmark-input combination at the perforation rate pairs (ϵ_o, ϵ_i) provided by the learner.

input graphs, and applied for each graph benchmark. Statically dropping outer loop iterations (or vertices) results in minimum accuracy loss for a given benchmark-input combination. The programmer selects an outer loop perforation rate that satisfies the accuracy threshold for all inputs of a graph benchmark. However, this does not guarantee optimal performance as some benchmark-input combinations may satisfy the accuracy threshold at a higher perforation rate. Thus, it is better to use different perforation rates for different benchmark-input combinations. A static outer loop perforation rate of 10% is the best candidate for the Tri-Cnt, Community, and Conn-Comp benchmarks satisfying the 10% accuracy loss threshold. Outer loop perforation rates of 20% and 1% are chosen for PageRank and SSSP respectively. The reason for applying a lower perforation rate for SSSP is because it has propagative dependencies across outer loop iterations. Loop perforation results in propagation of accuracy losses. Due to its structure, SSSP requires lower perforation rate to meet the accuracy loss threshold, leaving no margin for significant performance improvements. However, the remaining benchmarks tolerate higher perforation rates that satisfy the accuracy loss threshold. Overall, the static loop perforation scheme delivers ∼13% performance improvement.

C. Proposed Input-Dependence Aware Perforation

Figure 8 shows the normalized GPU completion times, power utilization, and the accuracy loss at the selected perforation rate pairs for each benchmark-input combination. Based on the accuracy threshold, the proposed learning model selects close to optimal outer and inner loop perforation rate (ϵ_o, ϵ_i) pairs. The MLP learner provides an accuracy of **90.8%** when applied to real-world graph inputs, bringing the accuracy close to near optimal efficiency. To operate near the accuracy threshold, the learner provides (ϵ_o, ϵ_i) pairs such that all the benchmarks show accuracy losses in the region of threshold of 10%– geometric mean accuracy loss value is 9.92%.

The performance results are normalized to the completion

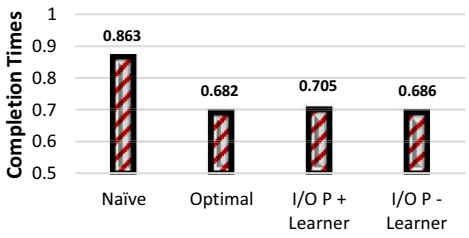

Fig. 9: Comparisons of the proposed input-aware loop perforation learner (with & without overhead) to the naive and optimal perforation approaches.

times of the algorithms running graph inputs with no perforation. Graph algorithms operating at the selected perforation rates provide an average performance improvement of ∼29.5% on the GPU. One critical observation in `Community-RGG` combination is that the completion time is much lower as compared to other benchmark-input combinations. This is mainly because of the fact that the learner has selected a better perforation rate compared to the naive approach. This also happens because Community is structured in a way that most of the computations are done within the outer loop among different phases. As discussed in Section III-C, the proposed perforation scheme maximizes the outer loop perforation rate, which maximizes the drop in edge computations. The performance is further optimized because perforating a large graph (RGG) allows it to be tiled properly in the smaller GPU cache. For SSSP, a lower perforation rate is applied because higher rates increase accuracy losses due to the propagative aggravation of shortest path costs, and hence acquired performance improvements are minimal. Similar trends are observed for the power utilization of the GPU. An average power utilization improvement of ∼19.0% for the GTX-970 GPU is observed.

D. Input-aware Learner vs. Naive and Optimal Approaches

Figure 9 shows the comparisons of static perforation versus an all-optimal implementation and the proposed learner (with and without learning overheads). The average completion time results are normalized to no perforation results for the GPU. The optimal version uses all-optimal hand-tuned perforation rates for each benchmark-input combination, providing an accuracy loss equivalent to the threshold value and the best performing data point. The figure also shows the performance numbers for the *combined inner-outer loop perforation* approach (**I/O P ± Learner**), with and without the MLP learner overheads added to each benchmark-input combination. The MLP learner introduces an overhead of about ∼1% to the proposed perforation scheme, as observed in Figure 9. Overall, the proposed perforation approach performs close to the optimal version with a modest overhead of around 1.2% when averaged across various configurations. The proposed perforation scheme shows an average performance improvement of ∼16.9% (with the learner overhead) over the the naive approach.

Learner	{Accuracy (%), Normalized Performance}							
Regression	**4th**		**5th**		**6th**		**7th**	
	77	0.88	84	0.84	91	0.76	94	0.78
MLP	**64**		**128**		**160**		**256**	
	70	0.91	82	0.82	90	0.70	93	0.73

TABLE III: Comparisons between learning schemes. Normalized performance taken from a GPU setting (lower is better).

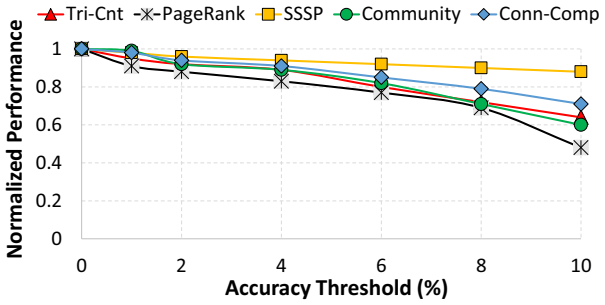

Fig. 10: Performance Vs. Accuracy Threshold comparisons for various benchmarks (average is taken for all input graphs).

Table III shows the accuracy and performance for various parameters of the target learners. Regression based learner is shown to have higher overheads but similar accuracy to the MLP learner. This makes the case to deploy the MLP learner for perforated graph learning. Moreover, the MLP learner is less complex as compared to the regression based learner. Therefore, it leads to a lower latency implementation as discussed in Section IV-A. Given a 90% average accuracy for various inputs and accuracy thresholds give best performance, it is selected as the default learner. A 6th order regression equation and a 160 neuron multi-layer perceptron based learner enables this accuracy.

E. Impact of Accuracy Threshold on Performance

Figure 10 shows the impact of various accuracy thresholds on performance. Each point in the figure shows the average performance improvement over all the input graphs for every benchmark. As expected, higher accuracy thresholds provide higher performance improvements due to more potential perforation opportunities. SSSP shows the lowest performance benefits throughout the analysis. This happens because the computations done in each iteration of the algorithm propagate to future iterations, making it sensitive to accuracy losses. The learning framework also provides lower classification accuracy for lower accuracy thresholds, which happens because there are more variables and non-linearities in such regions. Performance and accuracy points spread out at higher thresholds as more work is perforated, exhibiting benchmark-input differences. Benchmarks with more parallelism and compute, such as PageRank and Community tolerate higher perforation rated, while other benchmarks, such as SSSP with more control code do not tolerate higher perforation rates. An

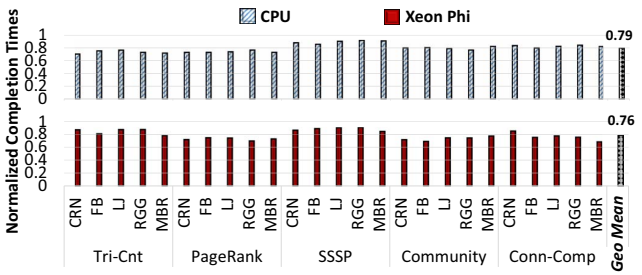

Fig. 11: Normalized CPU and Xeon Phi completion times at selected perforation rate pairs (ϵ_o, ϵ_i) for 10% accuracy threshold.

average classification of 90% is seen for all analyzed accuracy loss thresholds.

F. Performance & Power Results for two CPU Machines

Figure 11 shows the normalized power and completion times for benchmark-input combinations running on a 8-core Xeon and a 61-core Xeon Phi machine at perforation rates selected by the learner. We observe that GPU outperforms both CPUs in terms of performance. The CPUs show an average performance improvement of $\sim$21% and $\sim$24%, which is lower as compared to the GPU ($\sim$29.5%). This is due to the fact that GPUs have more computational units, potentially allowing more proportional performance gains from dropping edges. Moreover, GPUs have smaller caches, thus perforating a large graph reduces off-chip misses and consequently improves performance. The Xeon Phi outperforms Xeon because of the increased thread count of Xeon Phi, potentially resulting in more perforation performance. Similar results are observed for power utilization. The trend of power utilization of different architectures is $Power_{GPU} < Power_{XeonPhi} < Power_{CPU}$. GPU provides a power utilization improvement of $\sim$19% (c.f. Figure 8), while Xeon Phi and Xeon CPUs result in $\sim$14% and $\sim$12% power reductions respectively.

VI. Conclusion

This paper proposes novel insights about input dependence in graph algorithms in the context of approximate computing. To exploit the efficiency versus accuracy tradeoffs, an input-aware loop perforation scheme is proposed for graph algorithms. The perforations are selectively applied on graph input data rather than on the graph algorithm itself. This is done so as to mitigate the algorithmic code overheads of perforating loops. Moreover, a machine learning framework is developed to predict perforation rates for a user-defined output accuracy threshold. This input dependence aware loop perforation scheme allows a perforated graph execution at near-optimal accuracy threshold, while delivering performance gains and energy reductions. Results show an improvement of $\sim$30% in performance, and $\sim$19% in power utilization on an Nvidia GPU machine. Similar results are also observed for two multicore CPU machines.

VII. Acknowledgments

This research was partially supported by the National Science Foundation under Grant No. CCF-1550470. This work was also supported in part by the US Government under a grant by the Naval Research Laboratory.

References

[1] "Volvo announces plans for deathproof cars by the year 2020," in *http://inhabitat.com/volvo-developing-accident-avoiding-self-driving-cars-for-the-year-2020/*, 2016.
[2] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler gpu architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2222–2233, Aug 2016.
[3] J. Berkhout, "Google's pagerank algorithm for ranking nodes in general networks," in *2016 13th International Workshop on Discrete Event Systems (WODES)*, May 2016, pp. 153–158.
[4] Y. Li, J. Niu, X. Long, and M. Qiu, "Energy efficient scheduling with probability and task migration considerations for soft real-time systems," in *2014 IEEE Computers, Communications and IT Applications Conf.*
[5] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, May 2010, pp. 25–34.
[6] S. S. Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *13th European Conf. on Foundations of Software Engg*.
[7] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *17th Int. Conf. on Architectural Support for Prog. Languages and Operating Systems*.
[8] J. Ansel and et. al., "Petabricks: A language and compiler for algorithmic choice," in *Proc. of the 30th ACM SIGPLAN Conf. on Prog. Language Design and Implementation*, ser. PLDI '09. NY, USA: ACM, 2009.
[9] M. Ahmad and O. Khan, "Gpu concurrency choices in graph analytics," in *2016 IEEE International Symposium on Workload Characterization*.
[10] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum, "Fast and accurate estimation of shortest paths in large graphs," in *Proc. of the 19th ACM Int. Conf. on Information and Knowledge Management*.
[11] A. McGregor, "Graph stream algorithms: A survey," *SIGMOD Rec.*, vol. 43, no. 1, pp. 9–20, May 2014.
[12] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114 – 152, 2003, 1998 European Symposium on Algorithms.
[13] T. Eden and et. al., "Approximately counting triangles in sublinear time," in *IEEE 56th Annual Symp on Found. of Comp. Sc.*, 2015, pp. 614–633.
[14] M. Ahmad, C. J. Michael, and O. Khan, "Efficient situational scheduling of graph workloads on single-chip multicores and gpus," *IEEE Micro*.
[15] S. S. Douskos and et. al., "Using code perforation to improve performance, reduce energy consumption, and respond to failures," MIT-Tech-Reports, Cambridge, USA.
[16] H. J. Motulsky and L. A. Ransnas, "Fitting curves to data using nonlinear regression: a practical and nonmathematical review." *The FASEB Journal*.
[17] R. Collobert and et. al., "Links Between Perceptrons, MLPs and SVMs," in *Proceedings of the 21st Int'l Conference on Machine Learning*, 2004.
[18] Bader and Madduri, "Gtgraph: A synthetic graph generator suite."
[19] Nvidia, "https://developer.nvidia.com/maxwell-compute-architecture," 2014.
[20] Intel, "http://ark.intel.com/products/75799/intel-xeon-phi-coprocessor-7120p-16gb-1_238-ghz-61-core," 2014.
[21] M. Ahmad and et. al., "Crono : A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *Proc. of IEEE Int. Symposium on Workload Characterization*, 2015.
[22] S. Che and et. al., "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IEEE International Symposium on*, 2009.
[23] K. Skadron and et. al., "Pannotia: Understanding irregular gpgpu graph applications," in *IEEE Int. Symp. on Workload Characterization*, 2013.
[24] J. Leskovec and et. al., "SNAP Datasets: Stanford large network dataset collection," 2014.
[25] J. W. Lichtman and et. al., "The big data challenges of connectomics," in *Nature Neuroscience 17*, 2014.