Rumor Has It: Optimizing the Belief Propagation Algorithm for Parallel Processing

Michael Trotter
The George Washington University
Washington, DC, USA
trotsky@gwu.edu

Timothy Wood
The George Washington University
Washington, DC, USA
timwood@gwu.edu

H. Howie Huang
The George Washington University
Washington, DC, USA
howie@gwu.edu

ABSTRACT

By modelling how the probability distributions of individuals' states evolve as new information flows through a network, belief propagation has broad applicability ranging from image correction to virus propagation to even social networks. Yet, its scant implementations confine themselves largely to the realm of small Bayesian networks. Applications of the algorithm to graphs of large scale are thus unfortunately out of reach.

To promote its broad acceptance, we enable belief propagation for both small and large scale graphs utilizing GPU processing. We therefore explore a host of optimizations including a new simple yet extensible input format enabling belief propagation to operate at massive scale, along with significant workload processing updates and meticulous memory management to enable our implementation to outperform prior works in terms of raw execution time and input size on a single machine. Utilizing a suite of parallelization technologies and techniques against a diverse set of graphs, we demonstrate that our implementations can efficiently process even massive networks, achieving up to nearly 121x speedups versus our control yet optimized single threaded implementations while supporting graphs of over ten million nodes in size in contrast to previous works' support for thousands of nodes using CPU-based multi-core and host solutions. To assist in choosing the optimal implementation for a given graph, we provide a promising method utilizing a random forest classifier and graph metadata with a nearly 95% F1-score from our initial benchmarking and is portable to different GPU architectures to achieve over an F1-score of over 72% accuracy and a speedup of nearly 183x versus our control running in this new environment.

KEYWORDS

GPGPU, Parallelization, graph processing, parallel processing, Bayesian Networks, Markov Random Fields

ACM Reference Format:
Michael Trotter, Timothy Wood, and H. Howie Huang. 2020. Rumor Has It: Optimizing the Belief Propagation Algorithm for Parallel Processing. In *49th International Conference on Parallel Processing - ICPP : Workshops (ICPP Workshops '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3409390.3409401

ICPP Workshops '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8868-9/20/08…$15.00
https://doi.org/10.1145/3409390.3409401

1 INTRODUCTION

Born from "attempts to devise a computational model for humans' inferential reasoning, namely, the mechanism by which people integrate data from multiple sources and generate a coherent interpretation of that data" [13, p. 241], *belief propagation* (BP) models broadly how a change to a node begins a chain reaction of updates which permeates throughout a graph [10]. Such a simple yet general model has thus wide utility in countless fields including physics, medical imaging, artificial intelligence, computer vision, language understanding, sociology and error correction [18]. However, the few public implementations of BP limit themselves to graphs of thousands of nodes in size in the inefficient Bayesian Interchange Format (BIF) [4]. Thus, any application beyond such a small scale is simply infeasible with what is available.

Moreover, BP's amenability to parallelization holds much promise for optimization research [13]. In the context of trees, updates flow forward from the root nodes level by level to the terminal nodes and then backwards from the terminal nodes to the source nodes and thus can execute concurrently for all nodes in a given level [10]. Furthermore, a variation of the algorithm called the loopy BP better suits parallelization and supports general graphs versus just trees [12]. In this case, all nodes in the graph emit their updates at once continuously until the states of each of the nodes converge individually within a given threshold [7]. This broadcasting has no dependencies aside from the previous state of the graph beforehand and thus runs simultaneously per iteration until the nodes converge [12]. Thus, loopy BP in particular is ripe for further refinement.

Alas, there are many obstacles which stymie this endeavor. Operators must take into account the numerous processing intricacies inherent to the algorithm. Without doing so, naive optimization attempts falter, as we demonstrate with our introductory analysis of the OpenMP and OpenACC-based parallelization efforts. Hence, we utilize a host of fine-grained improvements and techniques to realize the potential of the loopy BP algorithm.

Herein, we describe in depth Credo which can perform BP efficiently on both large and small graphs by leveraging a suite of refined solutions and a classification method to automatically match the best solution to a given graph. To that end, we first detail two valid approaches to parallelizing loopy BP. We then briefly discuss our initial parallelization efforts with OpenMP and OpenACC before exploring our CUDA work in depth. Moreover, we provide a method to automatically execute the best suited implementation for a given graph by its metadata a priori with portability in mind. Additionally, we provide memory and processing optimizations to maximize the performance of our custom built solution. Furthermore, to support processing graphs of a far larger scale compared to previous works, we alter an existing large scale graph file format called the

Matrix Market file format utilized already to store graphs of sizes beyond billions of nodes [16]. Finally, we evaluate all of our efforts with a diverse benchmark of graphs. We thus make the following contributions to realize the full utility of BP:

- A more flexible input file format to support massive sized graphs to represent Bayesian networks, Markov Random Fields or any similarly complex graph beyond the thousands of nodes scale of the existing standard
- Optimized methods for processing graphs by node or edge encompassing improvements such as memory footprint minimization and supporting work queues tuned to this task
- Support for these methods on both CPU and GPU platforms to enable our system to significantly outperform prior work
- A method for automatically matching the ideal implementation from our suite of approaches to an arbitrary graph ahead of time based solely on its metadata which is portable to different GPU environments

2 BACKGROUND

2.1 Belief Propagation

Belief propagation (BP) models how updates to one or more nodes' internal set of beliefs percolate throughout the whole network [10]. The original networks used for BP are the Bayesian network and Markov Random Field (MRF), which represent potentially complex cause and effect relationships [13][3]. In particular, BP allows for describing how these relationships change when some new information becomes available.

Bayesian networks detail how the probability distribution of an event being in one of several states, i.e. its belief about its current state, represented as the nodes in the graph depend on the distributions of their parent nodes in the graph [18]. Using the prior joint probability distributions of an event occurring given n causes in the form $p(x|x_0, x_1, ..., x_n)$ and treating the parents' states as prior distributions, Bayesian networks enable the calculation of posterior distributions of events occurring [13]. The posterior calculation is of value when there is new information about an event and how that change impacts the graph in a process called observation [3].

During observation, one now knows for certain if an event occurs and consequently statically sets the probability of that event occurring which in turn sets of a chain of updates to the posterior probabilities throughout the network in a process called belief propagation [13].

We provide a slightly modified version of a popular example of a Bayesian network: the *family-out* problem [3]. A family has a house with a dog. They leave the dog outside the house if they are out or the dog is being punished. Similarly, they may leave the lights on when they are out. Finally, the dog may bark if it is out. The prior probabilities and joint distributions are all given in Figure 1.

Although the direct posterior calculations in the family-out are simple enough, they become unwieldy for more complex Bayesian networks and typically necessitate the Markov assumption, wherein an event node's state only depends upon the immediate parents' states and not on the other nodes in the dependency chain [3] However, this move necessitates using MRFs instead of Bayesian networks, as the latter does not allow for this assumption yet the former

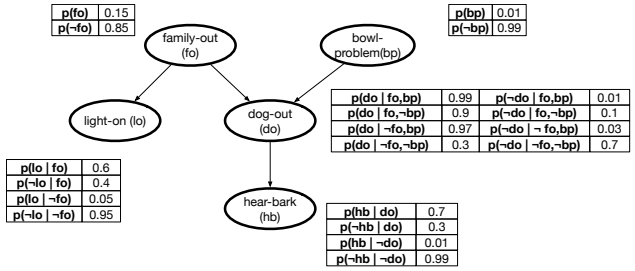

Figure 1: The Prior Probabilities of the Root Nodes and the Associated Joint Distributions Probabilities of the Children Nodes of the *family-out* Problem

does at the expense of only allowing for undirected pairwise relationships [18].

The calculation of the state of an individual node x_i with parents x_j and a joint probability distribution $p(x_i|x_j)$ linking the two thus is shown below.

$$p(x_i) = \prod_{(j,i)\in E}^{(i,j)} p(x_i|x_j) \tag{1}$$

Using the Markov assumption, each node in the dependency chain must compute its new state before broadcasting it down the chain. Due to the undirected nature of the MRF graph though, child events can now affect their parents' own states. Such an event occurs with multiple parents, and in such a case, the child node does not broadcast its update to the parent that initially prompted its update [7]. Thus, this calculation must occur in both directions except for the additional case of statically fixated observed nodes. Yet another complication to this computation is that an event needs not have a single binary state.

Indeed, an event may be in one of a wide range of discrete states beyond merely true or false. Therefore, the probability of an event being a given state is of particular value [18]. This marginal probability computation requires normalizing the final probabilities of the event's states [13]. Thus, the belief calculation of a node x_i with parent node x_j, child node x_k, functions $\phi(i,j) = p(x_j)p(x_i|x_j)$ and $\psi(i,k) = p(x_k)p(x_i|x_k)$ both subject to the aforementioned constraints and the marginalization factor Z is shown in the below equation:

$$p(x_i) = \frac{1}{Z} \prod_{(j,i)\in E}^{(i,j)} \phi(i,j) \prod_{(i,k)\in E}^{(i,k)} \psi(i,k) \tag{2}$$

To simplify processing, one can break up the BP into three phases. First, one emits the ϕ-based updates before emitting the ψ-based updates. Afterwards, one calculates the marginals. A major limitation of this method is that the updates must be ordered. The ϕ-value emissions must start from the root nodes and work their way down the tree. Likewise, the ψ-value emissions must start from the terminal nodes work their way up the tree to the roots.

An alternative to this processing is the loopy BP variant as shown in Algorithm 1 as described by Gonzalez et al. [5].

This form provides two major benefits. First, all nodes emit all ϕ-values simultaneously before likewise emitting all ψ-values. Second, these emissions relax the constraint that the graph needs to be acyclic.

Algorithm 1 Loopy Belief Propagation

```
1:  sum ← ∞
2:  while sum ≥ threshold do
3:      sum ← 0
4:      for all v ∈ V do
5:          beliefs_previous ← v_beliefs
6:          for all p ∈ V | (p, v) ∈ E do
7:              j ← joint_probability_matrix[p, v]
8:              update ← compute_update(p_beliefs, j)
9:              send(update, v)
10:         v_beliefs ← combine_updates(v)
11:         marginalize(v_beliefs)
12:         sum+ = diff(v_beliefs, beliefs_previous)
```

However, this method comes with the penalty that requires it to run until the nodes' beliefs converge rather than simply twice as before. Although we implement both versions of the algorithm, an initial, sequentially-processed evaluation reveals that the original BP approach has enormous overheads due to determining the levels of a graph and processing the graph by-level versus the by-node and by-edge methods of implementing loopy BP presented next.

2.1.1 Comparison of Belief Propagation Algorithms. Using the *10x40*, *100x400*, *1kx4k*, *10kx4k*, *100kx400k*, *200kx800k*, *400kx1600k*, *600kx2400k*, *800kx3200k*, *1Mx4M* and *2Mx8M* synthetic graphs alone presented in Table 1 in a single-threaded environment, the non-loopy BP implementation is 1032x slower than the by-edge version and 44x slower than the by-node *10kx40k* benchmark. This gap in performance widens to at most 11427x and 379x for the *2Mx8M* benchmark respectively. The traditional BP approach is on average circa 1014x and 300x slower than the by-edge and by-node versions. Given such a drastic performance difference between the two algorithms and loopy BP's better affinity for parallelization, we ultimately focus on it for the remainder of this paper.

2.2 Algorithmic Refinements for Large Graphs

We refine our broad description of the algorithm to reduce the memory footprint and to improve performance. Although loopy BP defines unique joint probabilities per edge, this requirement represents by far the largest amount of memory consumption for the graph. The joint probability matrix is a floating point matrix whose dimensions are that of the source and terminal nodes' belief arrays as shown in the Figure 1 as the probability tables below the *fo* and *bp* nodes. Loading and unloading a separate matrix per belief update computation also represents a significant performance and memory bottleneck. Thus, we come to a conclusion: this requirement is untenable for large graphs.

Indeed, the matrix stems from the statistics assembled to define the beliefs of a particular node given a specific neighbor's beliefs [3]. For large scale networks, assembling such statistics is unwieldy and necessitates using a single estimation for all nodes [3]. For instance, the operator assumes that the same error rate for any pixel applies to all others in an image or that a virus affects all people identically. Consequently, this consideration drastically reduces the size of the data and enables us to represent networks of millions of nodes.

To gauge the utility of a single joint probability matrix, we provide a simple demonstration. With this alteration in place and utilizing a micro-benchmark composed of a subset of just the graphs ranging from *10x40* to *800kx1200k* of the previously used synthetic graphs in Section 2.1.1, we observe a 2x speedup on average with both C and the CUDA Edge implementations. Given the high memory access cost on the GPU and the CUDA Node application's many more memory accesses compared to the CUDA Edge version, the impact of this change is far starker, yielding over 25x speedups for the larger graphs. Unfortunately, this optimization does break Credo's ability to support the original use case and forces the network to support nodes with beliefs of a constant size. For graphs not in that form, our initial version of Credo lacking this refinement would suffice for most cases, for they would also benefit from Credo's other processing optimizations.

2.3 GPU Architecture

GPU architecture departs significantly from CPU architecture and being aware of its intricacies is vital to maximizing of the platform [15]. A CUDA-compatible GPU consists of a set of Streaming Multiprocessors (SMX) consisting of numerous CUDA cores on which a single GPU thread executes [11]. GPU code or *kernels* execute on these SMX units in parallel by marshalling a *grid* consisting of individual *thread blocks* running on a single SMX [1]. Within each thread block are individual *warps* of 32 threads which may execute concurrently as shown in Figure 2 [17]. Typically, each of these

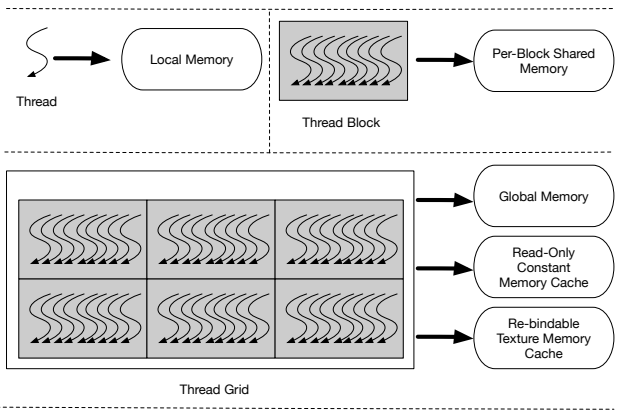

Figure 2: GPU SMX Architecture with the Associated Memory Hierarchy

threads perform the same instruction on different sections of data in parallel similar to SIMD architecture in Flynn's taxonomy [15].

Threads have several options for placing said data which have huge performance impacts. Each SMX has local memory used for stack allocated data [15]. Additionally, each thread block is able to access a small *shared* memory block for each thread block depending on the GPU generation capable of fast memory read and write access [11]. *Constant* and *texture* memory caches enable quick reads with the difference being that the former must be set before the kernel executes while the latter can be reset during kernel execution [15]. Finally, *global* memory, while slower than the others, is capable storing large data in the Video RAM (VRAM) of the GPU [1].

Figure 2 shows the relationships between threads and these different types of memory.

Given the benefits and drawbacks of each of these types of memory, the programmer must consequently take care in placing data onto the GPU.

2.4 OpenMP and OpenACC Parallelization

Given the difficulty writing such meticulous code with low-level APIs like CUDA, the programmer can instead use higher-level ones like OpenMP and OpenACC. They utilize *pragma* statements which the compiler then interprets at compile-time to generate parallelized code for each target platform. These *pragma* statements include information about data sharing, atomic operations, reduction operations, vectorization and in the case of OpenACC, placement on the GPU versus CPU. Nevertheless, they lack the fine grained control afforded by the likes of CUDA and come with additional overheads. We utilize OpenMP to distribute work across 8 CPU cores and OpenACC for the same across 1920 GPU cores as part of our preliminary attempts at parallelization with the all of the aforementioned optimizations enabled, save for the work queues which require finer grained control than what OpenACC offers to implement. We apply these statements directly on our fully-optimized, single-threaded C implementations' main loops which govern collecting the updates from the parent nodes, compute the updated states and send out the new information. The latter two operations also avail themselves of the vectorization statements to optimize the vector and matrix operations involved therein. Due to some aspects of the application having critical sections, we additionally utilize the atomic statements to ensure correctness. Moreover, the convergence check calculates via a reduction hint the sum of the differences between the previous and current iterations of the nodes' beliefs. Finally, we utilize OpenACC's data placement directives to finely manage data transfers between the CPU and GPU.

Unfortunately, our effort with OpenMP yields poor results. The performance actually decreases for 131 of the 132 benchmark graphs in Table 1 with the average performance penalty for running with 2 core case is circa 1.17x, with 4 cores is 1.65x and with all 8 cores is 4.03x. There is simply not enough work per thread to justify the overhead of spinning and shutting down threads in the aforementioned blocks of code as determined by *gprof*, taking on average less than 1ms to complete. Additionally, the tail distribution of the work described earlier which is a poor fit for the default scheduler as determined by *Intel vTune*, yet switching to the dynamic scheduler worsened the problem due to its additional overheads. Compounding the problem is the memory stalls and hyperthreading due to its usage of shared resources, yet disabling it only reduces the overhead to an average of 1.1x for 2 threads and 1.2x for 4 threads. Alas, our effort with OpenACC fares not much better.

At best, OpenACC offers a 1.25x increase in performance for the *K21* graph with the Edge paradigm. However, BP executes for far more iterations compared with our other implementations due to OpenACC's API failing to precisely compute the convergence check. Thus, they largely run for longer times than their C implementation counterparts by terminating much closer to the cap on iterations. However, the OpenACC execution times per iteration can be smaller, resulting in the slightly better performance of such benchmarks like *2Mx8M* and *LJ*. We are only able to achieve these results by keeping most of the data on the GPU after the initial load and only transfer the convergence check after predetermined number of batched iterations after overriding the default behavior of the OpenACC scheduler to try to schedule full transfers of the data between the CPU and GPU after every iteration. Although both OpenACC and OpenMP offer poor justification for parallelization, our efforts here act as a guide for our CUDA work which is uninhibited by such scheduler and platform overheads while providing the necessary finer grained control to achieve high performance.

3 CREDO DESIGN

3.1 Overview

In the following subsections, we describe the various components of Credo. The full system itself comprises of the optimized C and CUDA implementations derived from our dual processing techniques. Based on a given input graph and its metadata, Credo chooses the best from these implementations before executing BP with that method. To support this general functionality, Credo utilizes the components described in the below subsections.

3.2 Input Processing

In order to begin processing massive graphs, we first need to load them. There are two standards for this data: the Bayesian Interchange Format (BIF) and its XML-based sibling (XML-BIF) [4]. The former necessitates constructing a custom parser for its context-free grammar (CFG), while the latter requires an XML parser. To begin assembling the input graphs, both parsers must load the entire input file into memory first and then utilize hooks for each of the grammars' production rules for actions like defining a node and its metadata and constructing an edge. Compounding this problem is that both formats greatly exceed the size of the extracted graph. The simple *family-out* network has a 2KB BIF file size compared to handful of bytes used for its in-memory representation, and even a graph of 413 nodes and 602 edges occupying 5.3MB. Indeed, we could not hold graphs larger than 100,000 nodes in memory on a machine with 32GB of memory. Consequently, we would only be able to operate on 7 of the 34 binary belief benchmark graphs presented later on in Table 1. Moreover, the overhead of building the graphs is far larger than the actual BP execution time. Thus, we seek to obviate these issues by defining a new input format better suited for large scale graphs.

We propose a new format derived from the Matrix Market (MTX) format. Although MTX can support graphs of massive size, it simply lists out the edges of the graph by node ids after a header line defining the graph dimensions. Given that MRFs have many floating point numbers for the probabilities of each node's states, i.e. beliefs, and the edge's joint probability matrix, we break up the format in two: one for node data and the other for edge data. For both files, our structure is largely the same: two identifiers followed by the probabilities for the node's states or the edge's joint probability matrix. In preserving the original input format's basic structure of edges linked together by node ids, our node input format appears to be nothing but self-cycling nodes. However, this format is simple enough that it can be read line-by-line first by nodes and then edges without loading either fully into memory unlike BIF and BIF-XML. Additionally,

parsing it is trivial, requiring a handful of simple regular expressions rather than complex grammars. Moreover, it is general enough to support any network composed of the interaction of random variables and their state probabilities. Thus, Credo can support graphs of millions of nodes, compared with the thousands of nodes of previous works [12] [5] [6], with our dual processing methods combined with our other optimizations.

3.2.1 Comparison of Input Processors. We conduct a simple set of benchmarks to demonstrate the value of this format. The simple *family-out* graph presented in Figure 1 takes 162μs and 638μs to process with the BIF and BIF-XML parsers respectively. The largest BIF graph we have is circa of 1000 nodes and 2000 edges in size and takes 21ms to parse, while a similar 1000 node and 2000 XML-BIF file takes 83ms to parse. However, our custom MTX parser takes a mere 2ms to parse an equivalent file and produce the same graph logically. The largest BIF-XML file of 100,000 nodes and 200,000 edges we can parse without exceeding the 32GB of memory for our test system takes 8.4s. In comparison, our MTX-based parser can parse a similar 100,000 and 400,000 graph in .28s. Our BP methods take between 0.05 and 4.7s to process that graph. Indeed, our parser significantly reduces total execution time while being capable of parsing a graph of over 250 million edges.

3.3 Per-Node and Per-Edge Processing

Credo supports two possible ways of processing a graph using BP: by node and edge. In the former, each edge pulls the current state of the parent node and combines it with the joint probability matrix along the edge and the child node's state to produce the new state of the child node. In contrast, per-node processing pulls the states of all the parent nodes of a given node, combines them with the joint probability matrix for the edges linking the parents with the child before combining the updates with the child node's state to produce its new state as shown in Figure 3. Thus, when treating the undirected edges of an MRF as containing two separate edges to account for observed nodes being statically set, these two approaches enable Credo to perform the ϕ and ψ calculations described in Equation 2 and implement lines 6-10 of Algorithm 1. All other BP operations such as marginalization and the convergence check are the same across these designs. However, there are some trade-offs between the two which impact the performance of these approaches.

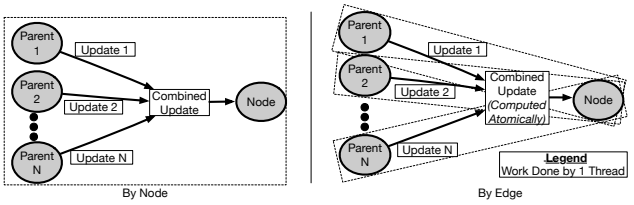

Figure 3: Processing by Node and Edge

In particular, there is an issue of the overhead of extra atomic operations versus memory lookups when moving from a single threaded environment to a multi-threaded one as shown in Figure 3. With the edge approach, a child node may have many parents and thus must combine each edge's contribution to its new state atomically to avoid race conditions. In contrast, the per-node approach does not require the use of extra atomic operations but performs more memory lookups by querying the state of each parent node for a given node. Additionally, these lookups occur in random order, hampering effective caching. Afterwards, the node approach must combine all of these states in memory with the joint probability matrix for the combined update message sent to the recipient, while the edge approach merely only has one state to combine for its message. Given there are far more edges than nodes in a graph, the latter-based approach requires more iterations to converge than the former. Yet, as our evaluation demonstrates later on, deciding which of these two different approaches for a given graph is not immediately obvious and Credo provides a mechanism to help choose that approach which includes both CPU and GPU-based implementations.

3.4 Data Structures

For all of our implementations, we make numerous data structure optimizations to minimize the size of the graph in memory, to minimize costly memory lookups and to maximize performance. We therefore only store the minimum necessary information about the graph which is the nodes' names and beliefs and indices for the edges between nodes. Before our processing refinement, we also store the joint probability matrices per edge. To minimize the overhead of examining each of the edges of the graph during BP computation, Credo indexes the edges' nodes and utilize compressed adjacency lists to represent the edges. Thus, Credo keeps itself largely to these indices and only touches the actual edge and node values when performing the actual mathematics involved in BP. However, we consider two avenues to go about implementing these structures.

In particular, we have options of a *struct* of arrays (SoA) and an array of *structs* (AoS) for storing the belief and joint probability data, which are simply sequences of single precision floating point numbers and their dimensions. During the initial development of Credo, we implement both before the other optimizations below and perform a limited analysis with the synthetic benchmark graphs up to and including 100,000 nodes in size (*10x40* to *100kx400k*) used for the algorithm comparisons in Section 2.1.1 and profile the code using *valgrind*'s *cachegrind* utility. With the SoA design, we have large, flattened, parallel-indexed arrays consisting for the probabilities and dimensions, while for the AoS paradigm, we have arrays holding *structs* consisting of a statically allocated float array and unsigned integers for the dimensions. Regardless of the graph and processing approach, we see that the AoS approach has circa 56% fewer data cache reads and writes. Thus, we opt to only use the AoS design with Credo.

3.5 Work Queues

From profiling, we observe that most nodes converge quickly after a few iterations and that graph convergence becomes dependent on a few nodes. To only process these nodes, we utilize work queues for both approaches. Instead of operating on a full list of node or edge indices depending on the approach, the queues merely consist of the indices of unconverged nodes or edges. However, after every iteration, the queue clears itself and populates atomically with the indices of elements which have yet to converge to a given threshold. As a result, this computation can drastically reduce the processing time overall at the cost of additional overhead in managing the queue.

Thus, we enable them for our C, OpenMP and CUDA implementations. Although other graph frameworks [17] [1] also make this optimization, their efforts cannot support BP.

3.6 CUDA Parallelization

Building upon our OpenACC and OpenMP work, our CUDA-based node and edge approaches parallelize the same loops with the same atomicity restrictions and aim to minimize CPU-GPU transfers utilizing batching. However, they incorporate performance optimizations only available with CUDA. In particular, the reductive sum makes use of the shared memory per block in Figure 2 to speed up the computation. Additionally, we make use of the global constant memory cache in Figure 2 to store the static joint probability matrix versus storing it in global memory due to its frequent reference. Nevertheless, there are significant data transfer costs with the CUDA approaches that limit them to smaller graphs (e.g. 100,000 for 2 beliefs and 1,000 for 32 beliefs with similar complexity). Below these thresholds, the C versions are faster. Thus, the Edge and Node implementations in both sequential C and parallel CUDA form the core of Credo.

3.7 Classification

From our initial evaluation, we could quickly discern a rule to use the CUDA implementations for when the graph has 100,000 nodes or more and the C versions for 1,000 nodes or fewer. Yet, this rule does not account for the middle ground. To alleviate this dilemma, we construct classifiers from the graph metadata obtained during input parsing to classify graphs as best suited for either the Node or Edge approach based on the nodes' in-degrees and out-degrees along with the number of nodes and edges. However, given the inter-dependencies among these variables, we ultimately derive the following features as inputs to our classifiers after some manual feature engineering.

Our feature vector consists of the *number of nodes*, the *nodes to edges* ratio, the *number of beliefs*, the *degree imbalance* (the ratio of the max in-degree to the max out-degree) and the *skew* (the ratio of average in-degree to max in-degree). We then simply assign a label of *Node* for when the a Node implementation is best for that benchmark and a label of *Edge* otherwise. Figure 4 shows the covariances among these features and the labels.

Although the skew is the only one which has some relation to the other features, dropping it actually reduces the quality of the resulting classifiers. Using a tuned random forest classifier from *scikit-learn* with a max-depth of 6 and 14 estimators is able to attain over 94% accuracy in F1-scoring with the following contributions of each feature as shown in Figure 5.

Exploring the benchmarks in finer detail yields significant insights into why the classifier is able to perform as well as it does. The classifier is able to expand upon our rule to only use the C Edge implementation for graphs smaller than 1,000 nodes and the CUDA Node version for graphs with 100,000 nodes or more only accounts for 80% of the benchmark graphs with the remainder being dominated by either C Node or CUDA Edge versions. Adding the nodes to edges ratio boosts the quality of the classifier noticeably. With these two features alone, a decision tree of max-depth of 2 is

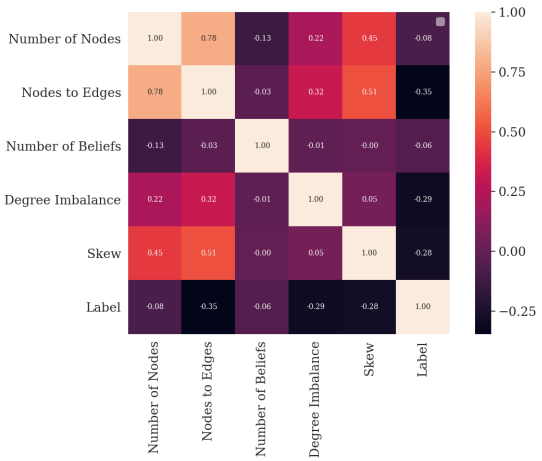

Figure 4: Covariances among the Feature Vectors and the Labels

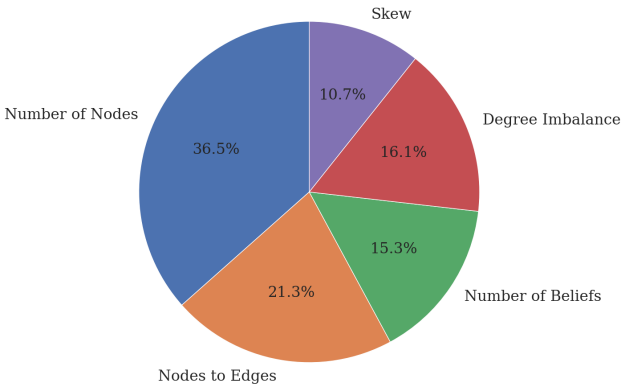

Figure 5: Percent Contributions to Random Forest Classification

able to achieve over 89% F1-score accuracy with the below structure with normalized feature values as shown in Figure 6.

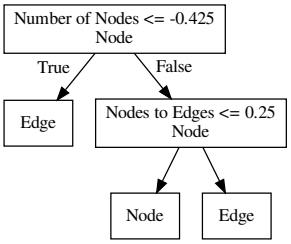

Figure 6: Structure of a Sample Decision Tree of Max-Depth of 2

However, to better account for graphs in the middle ground, the classifier must take into account the remaining features which describe the complexity of the graph in terms of data storage requirements and connectivity. Thus, they account for the penalties for overheads such as data transfer and expensive operations specific to each processing paradigm.

As a result of the importance of each feature to the quality of results, running primary component analysis (PCA) preprocessing on these features results in worse F1-score metrics for our classifiers. The combination of these features and their associated labels thus form the inputs for the classifiers discussed in the evaluation section. By querying any of them after parsing the input graph after applying the aforementioned rule for determining when to use the C versus CUDA implementations, Credo can choose the best of its assorted implementations to process any graph. With all of the optimizations discussed herein enabled, these implementations enable us to run more efficiently and outperform previous efforts as demonstrated in the next Evaluation section.

4 EVALUATION

For all of the experiments, we utilize the same machine running Ubuntu 18.04 LTS with *gcc* 7.4.0, *nvcc* 10.1.105, *pgcc* 18.10 and the 418.56 CUDA driver. It holds 32GB of memory, an Intel Core i7-7700HQ with 4 physical and 4 logical cores, and an nVidia GTX 1070 with 15 SMX processors, a total of 1920 CUDA cores and 8GB of VRAM. We utilize a block size of 1024 CUDA threads for all benchmarks. Table 1 shows our benchmark belief networks, composed of a diverse set of synthetic and real graphs obtained from the *networkrepo* [16], to support our three use cases. The first use case represents a simple binary true/false belief network. The second one models virus propagation with three states wherein people can be uninfected, infected or recovered. The final one mimics image correction with the beliefs in each bit's value in a 32-bit image's pixels.

We have permutations of these graphs adjusted for each use case and randomly encode generated beliefs into the input files for each graph for a total of 132 graphs. Due to the sheer size of the benchmarks, we only render figures for a subset of them with only the first binary belief use case, yet our analysis applies to all of them. We denote the subset's graphs in bold in Table 1. To ensure fairness, we also include the averages of all benchmarks as the *AVG* grouping in Figures 7 and 9. We execute each of the benchmarks until they achieve a convergence within 0.001 before cutting off at a maximum of 200 iterations.

4.1 Initial Benchmarking

In our first experiment, we benchmark the C and CUDA versions of the Node and Edge techniques with the work queues on as shown with Figure 7. We focus our analysis by comparing our control single-threaded C implementations against the others.

4.1.1 CUDA Results. The CUDA implementations exhibit noticeable performance gains for graphs with 100,000 nodes or more while running within 10 iterations of the sequential versions. Below this threshold, the various overheads involved with GPGPU execution including GPU memory allocation and transfers, kernel launch, synchronization and memory stalls prohibit the CUDA implementations' performance from overtaking the equivalent C's performance [15]. Indeed, for our smallest benchmark, the GPU memory management overhead alone accounts for 99.8% of the CUDA execution time which reduces to an average of 71% for the graphs at or above 100,000 nodes. Although the Edge version has at best roughly 3.4x performance improvement with the *2Mx8M* benchmark with three

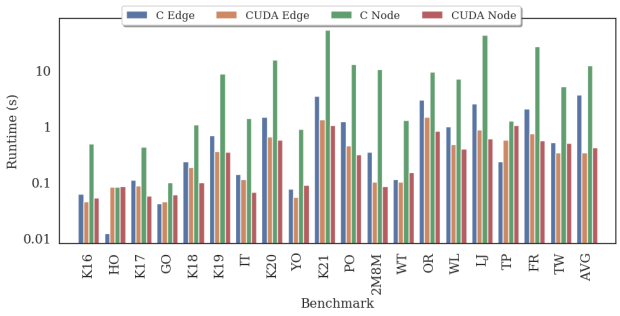

Figure 7: Runtimes of the C and CUDA Implementations

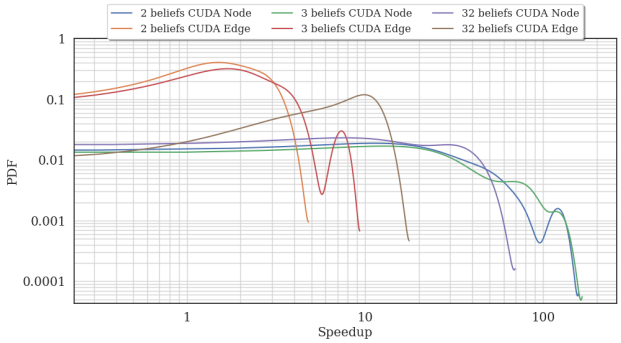

Figure 8: Probability Distribution Function (PDF) of Benchmark Speedups by Beliefs

beliefs, the Node version fares even better, attaining a 120x improvement with the same graph and over 40x improvements with benchmarks including the *K21*, *LJ* and *PO* graphs all with three beliefs. However, the speedup for the Node paradigm decreases beyond that three beliefs. Yet for Edges, it consistently increases with the number of beliefs as shown in Figure 8.

Indeed, for the same *K21*, *LJ* and *PO* graphs with 32 beliefs, the Node technique only musters a relatively consistent average 29x performance gain for each of those graphs, while the Edge achieves a similarly consistent average speedup of about 10x. This trend applies to all of the benchmark graphs as shown in Figure 8. The overhead of the additional atomic operations of the Edge implementation on the GPU becomes less stark against the additional memory loads necessitated by the Node implementation as the number of beliefs increases. In contrast, the single-threaded environment of the C implementations do not have the overhead of atomic operations but are still subject to the memory load penalties. Thus, in the C implementation tend to be dominated by the Edge paradigm. Generally, these memory loads do not outweigh the cost of the extra atomic operations except for graphs with high amounts of data complexity and connectivity as shown in the previous feature analysis. To gauge the full impact of atomic operations on the GPU, we nevertheless move on to explore the impact of the other source for their usage: the work queues.

4.2 Impact of the Work Queues

With 32 beliefs and the same graphs excluding the *TW* and *OR* which exceed the GPU's VRAM, we compare the impact of the

Table 1: Benchmark Graphs

Name	Abbrev.	Description	# Nodes	# Edges
10_nodes_40_edges	10x40	Synthetic 10x40 graph	10	40
1000_nodes_4000_edges	1k4k	Synthetic 1000x4000 graph	1000	4000
kron-g500-logn16	**K16**	Kronecker generator	55,321	2,456,398
100000_nodes_400000_edges	100kx400k	Synthetic 100,000x400,000 graph	100,000	400,000
loc-gowalla	**GO**	Gowalla location-based social network	196,591	1,900,654
soc-google-plus	GP	Google+ social network	211,187	1,506,896
web-Stanford	ST	Web graph of *stanford.edu*	281,903	2,312,497
kron-g500-logn19	**K19**	Kronecker generator	409,175	21,781,478
web-it-2004	**IT**	IT network graph	509,338	71,784,13
600000_nodes_1200000_edges	600kx1200k	Synthetic 600,000x1,200,000 graph	600,000	1,200,000
800000_nodes_3200000_edges	800kx3200k	Synthetic 800,000x3,200,000 graph	800,000	3,200,000
com-youtube	**YO**	Friendship network on YouTube	1,134,890	2,987,624
soc-pokec-relationships	**PO**	Pokec social network graph	1,632,803	30,622,564
2000000_nodes_8000000_edges	**2Mx8M**	Synthetic 2,00,000x8,000,000 graph	2,000,000	8,000,000
soc-orkut	**OR**	Orkut social network	2,997,166	106,349,209
soc-LiveJournal1	**LJ**	LiveJournal social network	4,846,609	68,475,391
friendster	**FR**	Friendster social network	8,658,744	55,170,227
soc-twitter-2010	**TW**	Twitter social network	21,297,772	265,025,809
100_nodes_400_edges	100x400	Synthetic 100x400 graph	100	400
10000_nodes_40000_edges	10kx40k	Synthetic 10,000x40,000 graph	10,000	40,000
hollywood-2009	**HO**	Hollywood actor network	83,832	549,038
kron-g500-logn17	**K17**	Kronecker generator	131,071	5,114,375
200000_nodes_800000_edges	200kx800k	Synthetic 200,000x800,000 graph	200,000	800,000
kron-g500-logn18	**K18**	Kronecker generator	262,144	10,583,222
400000_nodes_1600000_edges	400kx1600k	Synthetic 400,000x1,600,000 graph	400,000	1,600,000
soc-twitter-follows-mun	TF	Twitter followers graph	465,017	835,423
soc-delicious	DE	Delicious social network	536,108	1,365,961
kron-g500-logn20	**K20**	Kronecker generator	795,241	44,620,272
1000000_nodes_4000000_edges	1Mx4M	Synthetic 1,00,000x4,000,000 graph	1,000,000	4,000,000
kron-g500-logn21	**K21**	Kronecker generator	1,544,087	91,042,010
web-wiki-ch-internal	**WW**	Web graph of Chinese Wikipedia Articles	1,930,275	9,359,108
wiki-Talk	**WT**	Communication network of English Wikipedia	2,394,385	5,021,410
wikipedia-link-en	**WL**	Wikipedia English internal links	3,371,716	31,956,268
tech-p2p	**TP**	eDonkey p2p network	5,792,297	8,105,822
average	**AVG**	Average across all benchmark graphs. **Note: merely a derived statistic**	N/A	N/A

work queues across the C and CUDA implementations for the Edge and Node paradigms. Although there is a slight loss in performance with this optimization enabled for C Edge implementation with an average reduction of about two percent in performance versus without the work queue, the CUDA equivalent exhibits an average 1.3x improvement as shown in Figure 9. The latter benefits from this optimization due to the batching used, although the Edge versions tend to converge in only a few iterations. Indeed, the Node versions run for tens of iterations and therefore benefit even more from this optimization as shown in Figure 9.

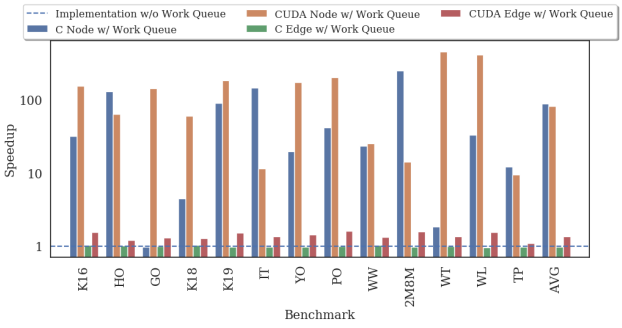

Figure 9: Speedups of Work Queues by Implementation

Under the Node processing paradigm, the C version achieves an approximate average 87x compared to the CUDA implementation's average of just over 82x. The CUDA version benefits less from this optimization due to its faster execution time and the greater impact of atomic operations on GPU performance from the many more threads running concurrently versus on the CPU. With the successes of the aforementioned optimizations, we thus turn our attention to our final contribution.

4.3 Classification

We first begin our experiments with a tuned decision tree with a max depth of 2 levels to take into account the small size of our dataset. With a set of 95 graphs variations of the 34 aforementioned graphs that can fit into our GPU's VRAM and for which we consequently have a full dataset, we train a simple decision tree with a train-test split of 60-40 to attain an 89.5% F1-score. To improve our classifications, we refine this approach using an ensemble method: random forests. With a likewise tuned random forest consisting of a max-depth of 6 levels and 14 trees, we boost the F1-score to 94.7%. Consequently, it shows promise to reliably predict best suited Credo implementation for a given graph and its metadata.

To validate that assertion, we randomly shuffle the input dataset and draw well-balanced samples from it before applying the same train-test split as before. Moreover, we compare our tree-based classifiers with other classifiers available from *scikit-learn* [14]. Figure 10 shows the impact of the dataset size on these classifiers' scores along with the standard deviation of a three-fold cross validation as the error bars.

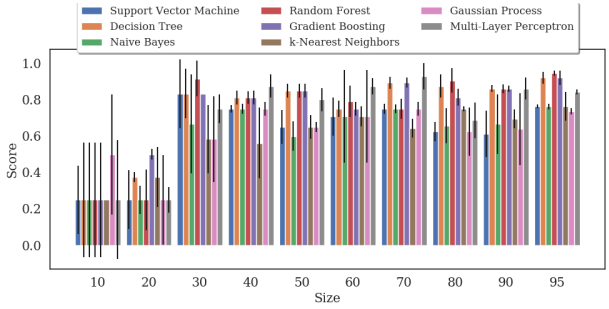

Figure 10: Classifier F1-Scores with Varying Training Data Sizes

Although additional training data would yield improved results, the current dataset is sufficient. Remarkably, the tree-based classifiers need only a dataset of about 40 elements divided in 24 training samples and 16 testing samples before achieving an F1 score of at least 80% accuracy with low error especially with 90 or more data points. Before 40 elements, all classifiers simply overfit, as the F1-scores of the training set which are nearly perfect. Compared with the other classifiers, decision trees and random forests are well suited for this classification compared to others as described by *scikit-learn* [14]. Due to the majority of the features being ratios between zero and one, they are fairly bounded. This heavy normalization limits the

utility of the remapping that the Support Vector Machine classifier does, which PCA hints at when it failed to improve upon the original features. Furthermore, they have largely nonlinear relationships to one another as shown in Figure 4. Yet, that analysis also reveals they have some interrelation, which violates the assumptions of the Gaussian Process and Naive Bayes classifiers. Both classifiers assume a normal distribution of the features and a lack of covariances among them. This interplay between features also inhibits the k-Nearest Neighbors classifier which only excels when the features can yield entirely separable clusters before classification. Finally, the Gradient Boosting and Multi-Layer Perception classifiers are poorly suited for this use case despite their decent performances, as both classifiers needs hundreds of thousands of training data to be useful. Such a requirement is simply untenable with the current resources available. Thus, given these limitations, we nevertheless demonstrate that the decision tree-based classifiers in particular can indeed predict the most useful implementation for a particular graph ahead of time by using its metadata alone.

To demonstrate the impact of our classification, we present our penultimate experiment. As a control, we use a naive assumption of always choosing the C Edge implementation against our Credo classifier with all execution overheads included in the analysis. Figure 11 shows the results of this experiment.

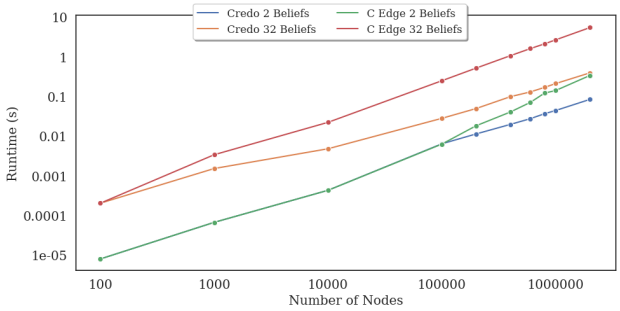

Figure 11: Execution Time of Credo vs. C Edge

For very small graphs, Credo offers little improvement over the C Edge implementation. However, at around 1,000 nodes, the classifier enters the middle ground area where the Node paradigm may offer improvements over the Edge paradigm. At 100,000 nodes, the CUDA aspects of Credo consistently offer noticeably greater performance than their C counterparts as described by our observed rule. Beforehand, the number of beliefs determines where exactly in this middle ground that this change occurs as shown in Figure 11.

4.4 Portability of Classification

For our last experiment, we examine the portability of the classifiers to a new nVIDIA GPU architecture: Volta. With such a small dataset, transfer learning is infeasible due to the amount of new data necessary, while retraining from scratch is antithetical to portability. We procure a *p3.2xlarge* in AWS which comes equipped with 61GB of memory, an Intel(R) Xeon(R) CPU E5-2686 v4 with 8 cores and an nVIDIA Volta V100 SXM2 16GB GPU with 5120 CUDA cores. We proceed to run our same 97 benchmarks within this new environment and compare the ability of Credo to classify graphs within this environment. Thus, our random forest classifier trained for the GTX1070 achieves an F1-score of 72.2% using this setup.

There are several factors which inhibit the portability. Chief among them is that Volta significantly alters the thread synchronization by introducing independent thread scheduling, forcing us to move our invocations of the *__syncthreads* function to different code locations to adhere to this new scheduler [15]. Consequently, the overhead for the atomic operations is lower on this architecture. Additionally, Volta introduces a considerably 1.5x higher memory bandwidth over Pascal, resulting in improved performance reading from global memory. Accounting for these factors, the CUDA Edge implementation surpasses the CUDA Node implementation in 8.3% more test cases from our benchmark suite. However, the difference between the two versions is seldom significant with the CUDA Node running on average 0.27 seconds and the CUDA Edge running in 0.30 seconds.

Indeed, Credo demonstrates similar behavior as before against always choosing the C Edge implementation as shown in Figure 12.

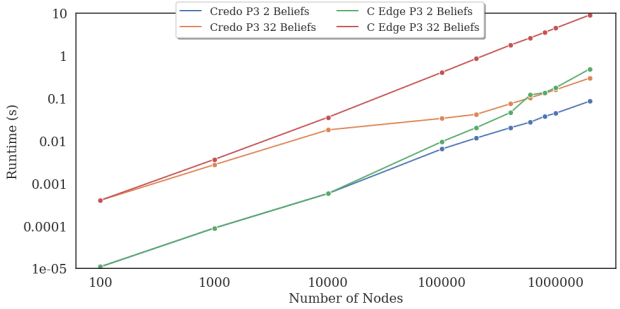

Figure 12: Execution Time of Credo vs. C Edge on a p3.2xlarge

Initially, Credo always chooses the C Edge implementation before the pivot point determined by the number of beliefs. Afterwards, it switches to the relevant CUDA implementation before always choosing the CUDA Node implementation. As a result of the performance improvements between the GPU architectures, we observe faster runtimes with the switch to the CUDA implementations. Indeed, on average the Edge and Node implementations are 3.2x and 3.8x respectively faster compared with the Pascal architecture. Indeed, our CUDA Node implementation improves its speedup to almost 183x faster C Node implementation on the *p3.2xlarge*.

5 RELATED WORK

5.1 Belief Propagation Parallelization

While deploying BP onto the GPU is the subject of some research, these efforts are limited in scope and operate on small graphs. Bistaffa et al. [2] recompile the graph into a optimized form called a "junction tree" for GPU computation but limit themselves to only belief networks consisting of a few thousand nodes. While they obtain nearly a 20x speedup in some cases, the actual execution times are in the range of milliseconds for all of their graphs regardless of implementation. The transfer time for data onto the GPU is likewise on this scale which defeats the purpose of using their solution [15]. Grauer-Gray and Palaniappan [8] implement a version of BP optimized for stereo image processing and motion tracking for a paltry 5x speed gain versus the CPU for a total of 15s on the GPU. However, they operate on the pixels' intensities rather than values and limit themselves to images of 512x512 resolution. Moreover,

they break the images down significantly to 256x256 blocks for the actual computations which does introduce additional errors into their results from not fully processing the graph. Thus, there is still much opportunity for efficiently parallelizing BP for large graphs without making the sacrifices to input size or actual processing.

Indeed, efforts utilizing parallelization techniques ranging from dynamically scheduling to MapReduce and MPI have found great success. Nevertheless, they too limit themselves to small graphs. Ma et al. [12] implement a custom scheduler using pthreads to efficiently schedule updates on a 40-core CPU to process graphs of roughly 4,000 nodes in 4s, while we can process a similar graph in about 1ms. Gonzalez et al. [5] utilize MapReduce to parallelize traditional BP by performing the updates at a given level of a tree in parallel to process a 460,000 node graph in about 12s, while in another effort, Gonzalez et al.[6] take 6.4s for a 58,000 edge graph using 40 servers using pthreads and OpenMPI. In contrast, Credo can process graphs of comparable size in 0.7s and 0.06s respectively. Kang et al. [9] successfully employ BP to process massive-scale graphs consisting of billions of edges using the message passing interface (MPI) library, although this effort necessitates reprocessing the graph into a form amenable to this distributed environment. Additionally, due to network latencies from the frequent message passing inherent to BP, their solution takes hours to process our benchmark graphs as previewed during our analysis. However, Credo can process similar graphs in 2-3s. In contrast to these previous works, our GPU design for BP does not suffer from the overhead of CPU-based pthreads, does not limit itself to trees in terms of input graph structure and can process considerably larger graphs of at least an order of magnitude greater size in seconds rather than hours.

5.2 Graph GPU-based Frameworks

Several GPU-based frameworks enable application developers to process massive graphs using common algorithms such as single-source shortest path (SSSP) and PageRank [17] such as Gunrock, nvGRAPH and Groute [1] [15] [17]. Although they do not implement BP and are heavily reliant on the CSR format, there are several optimizations they utilize of note. nvGRAPH [15] borrows the concept of semi-rings from linear algebra to genericize common graph operations and provides a custom scheduler optimized for semi-rings. Gunrock abstracts all graph operations as a series of advance, filter and computation steps operating either on nodes or edges utilizing optimizations such as kernel fusion, push-pull traversal, idempotent traversal and priority queues. Groute [1] asynchronous execution using a custom scheduler for multi-GPU support using nVIDIA's NCCL library. However, all of these optimizations are useless to complex graph algorithms like BP which do not adhere directly to the CSR format and its assumption of one floating point number or integer per node. Consequently, these frameworks cannot perform complex graph processing on the level of BP, despite their impressive results. Meanwhile, our solution can while also profiting from several of their optimizations.

6 CONCLUSIONS

Through the course of our research, we successfully enable belief propagation to run for small and large scale graphs utilizing Credo. To support this endeavor, we present a simple yet flexible input format to represent those graphs. We provide a host of designs utilizing significant workload, memory and threading management optimizations to handle a plethora of benchmark graphs. We even attain speedups over 120x in some cases versus our control single threaded implementations on our initial evaluation system and over 184x speedup in other environments. Finally, we describe a viable, portable method for selecting a priori the best implementation for a given graph and process the graph using that method automatically with Credo.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grants 1763548, 1618706 and 1717774. We also would like to thank Huang Liu for his input in the early parts of the project.

REFERENCES

[1] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 235–248.

[2] Filippo Bistaffa, Alessandro Farinelli, and Nicola Bombieri. 2014. Optimising memory management for belief propagation in junction trees using GPGPUs. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 526–533.

[3] Eugene Charniak. 1991. Bayesian networks without tears. *AI magazine* 12, 4 (1991), 50–50.

[4] Gal Elidan. 1998. Bayesian Network Repository. http://www.cs.huji.ac.il/~galel/Repository/.

[5] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. 2009. Residual splash for optimally parallelizing belief propagation. In *Artificial Intelligence and Statistics*. 177–184.

[6] Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. 2010. *Parallel splash belief propagation*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA OFFICE OF SPONSORED RESEARCH.

[7] Joseph E Gonzalez, Yucheng Low, Carlos Guestrin, and David O'Hallaron. 2009. Distributed parallel inference on large factor graphs. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*. AUAI Press, 203–212.

[8] Scott Grauer-Gray, Chandra Kambhamettu, and Kannappan Palaniappan. 2008. GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction. In *2008 IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS 2008)*. IEEE, 1–4.

[9] U Kang, Duen Horng Chau, and Christos Faloutsos. 2011. Mining large graphs: Algorithms, inference, and discoveries. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 243–254.

[10] Frank R Kschischang, Brendan J Frey, Hans-Andrea Loeliger, et al. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory* 47, 2 (2001), 498–519.

[11] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[12] Nam Ma, Yinglong Xia, and Viktor K Prasanna. 2012. Task parallel implementation of belief propagation in factor graphs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1944–1953.

[13] Judea Pearl. 1986. Fusion, propagation, and structuring in belief networks. *Artificial intelligence* 29, 3 (1986), 241–288.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).

[15] nVidia Corporation. 2019. *CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/

[16] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[17] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, et al. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*.

[18] Jonathan S Yedidia, William T Freeman, and Yair Weiss. 2003. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium* 8 (2003), 236–239.