Efficient GPU Implementation of Static and Incrementally Expanding DF-P PageRank for Dynamic Graphs

Subhajit Sahu
subhajit.sahu@research.iiit.ac.in
IIIT Hyderabad
Hyderabad, Telangana, India

ABSTRACT

PageRank is a widely used centrality measure that "ranks" vertices in a graph by considering the connections and their importance. In this report, we first introduce one of the most efficient GPU implementations of Static PageRank, which recomputes PageRank scores from scratch. It uses a synchronous pull-based atomics-free PageRank computation, with the low and high in-degree vertices being partitioned and processed by two separate kernels. Next, we present our GPU implementation of incrementally expanding (and contracting) Dynamic Frontier with Pruning (DF-P) PageRank, which processes only a subset of vertices likely to change ranks. It is based on Static PageRank, and uses an additional partitioning between low and high out-degree vertices for incremental expansion of the set of affected vertices with two additional kernels. On a server with an NVIDIA A100 GPU, our Static PageRank outperforms Hornet and Gunrock's PageRank implementations by 31× and 5.9× respectively. On top of the above, DF-P PageRank outperforms Static PageRank by 2.1× on real-world dynamic graphs, and by 3.1× on large static graphs with random batch updates.

KEYWORDS

Parallel GPU-based PageRank, Dynamic Frontier approach

1 INTRODUCTION

PageRank is an algorithm for assessing the significance of nodes within a network by assigning numerical scores based on link structures [43]. It operates on the principle that pages with more high-quality inbound links are of greater importance and thus should have higher ranks. While originally devised to rank web pages in search results, this metric finds applications in various domains, such as urban planning [62], video object tracking [26], traffic flow prediction [33], dynamic asset valuation [54], protein target identification [5], software system characterization [14], and identification of crucial species for environmental health [2].

With the rise of extensive interconnected data, interest in parallel PageRank computation has surged. This has led to a number of implementations of parallel PageRank for multicore CPUs [7, 12, 24, 28, 29, 36]. Unfortunately, multicore CPUs only offer a limited memory bandwidth. This makes them unsuitable for graph algorithms, such as PageRank, which have a low computation-to-communication ratio. GPUs, on the other hand, boast extremely high bandwidth memory, connected in close proximity to thousands of lightweight cores with user-managed caches. Further, the GPU hardware is designed to be able to switch between running threads at no cost in order to support memory access latency hiding. When graphs algorithms are suitably designed, they can significantly outperform a parallel CPU-based implementation.

In recent years, significant research effort has focused on developing efficient parallel implementations of PageRank for GPUs [8, 10, 11, 16, 17, 21, 23, 27, 28, 31, 40, 45, 57, 58, 60]. These implementations, referred to as Static PageRank, compute ranks from scratch for a given graph, assuming the graph remains static over time. In this paper, we present our GPU implementation of Static PageRank,[1] which utilizes a synchronous pull-based atomics-free computation method. It partitions and processes low and high in-degree vertices separately using two distinct kernels. Additionally, it avoids computation of global teleport rank contribution due to dead ends (vertices with no outgoing edges) — we ensure the input graph contains no dead ends. Our implementation represents, to the best of our knowledge, the most efficient implementation for parallel PageRank computation on the GPU. Our GPU implementation of Static PageRank is compared with other state-of-the-art implementations in Table 1. It includes both direct and indirect comparisons, with details given in Sections 5.2 and A.4 respectively.

Table 1: Speedup of our GPU implementation of Static PageRank compared to other state-of-the-art implementations. Direct comparisons are based on running the given implementation on our server, while indirect comparisons (denoted with a ∗) involve comparing results obtained by the given implementation relative to a common reference (Hornet/Gunrock).

PageRank implementation	Published	Our Speedup
Hornet [8]	2018	31×
Gunrock [58]	2016	5.9×
Galois with Gluon [17]	2018	448×∗
Tigr [40]	2018	40×∗
Grus [57]	2021	7.2×∗
Atos [10]	2022	7.6×∗
Multi-GPU Atos (1 GPU) [11]	2022	7.8×∗
Multi-GPU Atos (4 GPUs) [11]	2022	4.8×∗
GraphBLAST [60]	2022	11×∗
Meerkat [16]	2023	18×∗

Real-world graphs often exhibit dynamic characteristics, undergoing frequent edge updates [1, 6]. Recomputing PageRank scores for vertices upon each update, known as Static PageRank, can be resource-intensive. A strategy to reduce computation involves initiating PageRank computation from previous vertex ranks, thereby minimizing iterations required for convergence. We refer to this as the *Naive-dynamic (ND)* approach. To further optimize runtime,

---

[1] https://github.com/puzzlef/pagerank-cuda-dynamic

recalculating ranks solely for potentially affected vertices is crucial. One common approach, which we refer to as the *Dynamic Traversal (DT)* approach, entails identifying reachable vertices from updated graph regions and processing only those [18, 25, 32, 51]. However, marking all reachable vertices as affected, even for minor changes, may lead to unnecessary computation, particularly in dense graph regions. Our previous work [49] addressed these concerns by introducing incrementally expanding *Dynamic Frontier (DF)* and *Dynamic Frontier with Pruning (DF-P)* approaches. These approaches process only a subset of vertices likely to change ranks, and were implemented as parallel multicore algorithms [49]. Here, we present our GPU implementation of DF-P PageRank,[2] based on Static PageRank. It features partitioning between low and high out-degree vertices for incremental expansion of affected vertices using two additional kernels. Table 2 shows the performance comparison of DF-P PageRank with Static, ND, DT, and DF PageRank.

Table 2: Speedup of our GPU implementation of Dynamic Frontier with Pruning (DF-P) PageRank compared to other approaches of updating PageRank scores (on GPU), on real-world dynamic graphs (Table 3), and on large static graphs with random batch updates (Table 4), respectively.

PageRank approach	Speedup of DF-P
Static [43]	2.1×, 3.1×
Naive-dynamic (ND) [43, 63]	1.5×, 1.7×
Dynamic Traversal (DT) [18, 25, 32, 51]	1.8×, 13.1×
Dynamic Frontier (DF) [49]	2.1× 1.3×

2 RELATED WORK

2.1 Static PageRank

One of the earliest GPU implementations of PageRank is by Wu et al. [59]. They implement Static PageRank on AMD GPUs using OpenCL. For this, they develop a Sparse Matrix-Vector Multiplication (SpMV) routine as a primitive. Here, as a pre-processing step, they sort the rows of the PageRank sparse matrix based on the number of non-zero elements, while keeping track of the original row number. Subsequently, they employ three distinct kernels to process the rows: One thread per row (1T1R), one quarter wavefront per row (16T1R), and one wavefront per row (1W1R), depending on the number of non-zero elements in each row. Cevahir et al. [9] conduct PageRank computation on an NVIDIA GPU cluster (TSUBAME 1.2), with each node housing two Tesla GPUs. They partition the graph into chunks, assigning one chunk to each node. Each chunk is split into two portions, each of which can be loaded into the device memory of each GPU. They allocate one MPI process for each GPU, and utilize an SpMV kernel on the GPU for computation. Rungsawang and Manaskasemsak [48] improve upon the work of Cevahir et al. [9] by devising an algorithm that does not require large graphs to fit within the limited device memory. They achieve this by partitioning the graph between nodes and then further dividing each partition into chunks using the CPU at

each node. These chunks are sized to fit into the device memory of the Tesla GPU at each node, which then processes them one after the other. Duong et al. [20] introduce a push-based GPU implementation of Static PageRank, employing atomic operations. They utilize a single thread per vertex for rank computation and handle the global teleport rank contribution (dangling value) with atomic operations. Additionally, they propose a multi-GPU implementation where the input graph is partitioned among GPUs, allowing independent computation of PageRank scores on each GPU. After each iteration, the computed ranks are synchronized on the CPU, where they are combined and redistributed among the GPUs.

However, Wu et al. [59], Cevahir et al. [9], and Rungsawang and Manaskasemsak [48] must first build the PageRank matrix before performing PageRank computation using an SpMV kernel. Our GPU implementation of PageRank does not require the building of any sparse matrix — we directly utilize the adjacency list (stored in CSR format) of the graph for PageRank computation. Rungsawang and Manaskasemsak [48] simply assign a thread for processing each vertex in the graph, which fails to achieve good load balancing and suffers from thread divergence, while Wu et al. [59] sort the rows by the number of non-zero elements, for load balancing using three different kernels, which can be an expensive operation. In contrast, we partition the vertices in the graph into low and high degree vertex sets and process them with two kernels. Duong et al. [20] use a push-based PageRank computation, relying on atomic operations per edge, and also compute the global teleport contribution due to dead ends using atomic add operations. This can result in substantial memory contention among GPU threads. In contrast, our pull-based approach requires only one write per vertex, and by eliminating dead ends during graph loading we avoid the need for finding the global teleport contribution. Duong et al. [20] also do not partition vertices into low and high-degree sets.

We now discuss a number of graph processing frameworks, which include a GPU implementation of PageRank. Wang et al. [58] present the Gunrock multi-GPU CUDA library, which provides a high-level vertex/edge frontier-based bulk-synchronous / asynchronous API, and a number of high performance primitives. The PageRank implementation of Gunrock adopts a push-based approach, executing atomic adds per edge, employs a parallel for loop (using thrust) over the range of vertex IDs, utilizing either a thread- or a block-per-vertex approach. Further, its computes the global teleport contribution to each vertex attributable to dead ends. Busato et al. [8] introduce Hornet, a platform-independent data structure optimized for dynamic sparse graphs and matrices. Hornet accommodates large-scale data growth without necessitating any data reallocation or re-initialization throughout the dynamic evolution process. The GPU-based graph processing framework based on the Hornet data structure is known as cuHornet. In its PageRank implementation, Hornet follows a push-based approach, akin to Gunrock. It calculates the rank contribution of each vertex separately, stored in a distinct vector, and utilizes an additional kernel to compute ranks from these contributions. Similar to Gunrock, Hornet employs a parallel for loop (using a custom kernel) over all vertices, adopting a thread-per-vertex approach.

We now discuss issues with the PageRank implementation of Gunrock [58] and Hornet [8]. Both utilize push-based PageRank computation involving atomic add operations per edge, leading

---

[2]See Footnote 1

to significant memory contention among GPU threads. However, our approach is pull-based, and necessitates only a single write per vertex. Hornet computes vertex rank contributions separately and employs an additional kernel for rank computation, whereas we calculate rank contributions on the fly and utilize only a kernel pair for rank computation. Gunrock performs a parallel for (using thrust) over the range of vertex IDs, while Hornet performs a parallel for (using custom kernel) over all vertices using a thread per vertex. Unlike Gunrock and Hornet, we partition vertices into low and high in-degree sets and employ both thread- and block-per-vertex kernels. Gunrock uses a kernel to compute the global teleport contribution due to dead ends, while we preemptively eliminate dead ends during graph loading. Finally, Hornet uses a naive rank vector vector norm computation with atomic operations, while we implement a more efficient parallel reduce operation.

Wang et al. [57] introduce Grus, a GPU-based graph processing framework optimized for Unified Memory (UM) efficiency. Their focus lies in minimizing data migration, reducing page faults, and mitigating page migration overhead. Their PageRank implementation is push-based, and utilizes an adaptive UM policy, prioritizing frontier and rank data with high priority, Compressed Sparse Row (CSR) index array with medium priority, and CSR edges array with low priority. They use a bitmap-directed frontier, based on an 8-bit integer array alongside a queue, which eliminates the need for atomic operations. Load balancing is achieved with a warp-centric approach. Chen et al. [10] critique existing frameworks like Gunrock for launching each graph frontier as a separate GPU kernel in the Bulk Synchronous Parallel (BSP) model, leading to potential issues with parallelism, finish times, and kernel launch overhead, especially for small frontiers. To address this, they propose Atos, a persistent task scheduler designed to minimize kernel launch overhead and support asynchronous execution. For PageRank, they present a push-based asynchronous implementation, employing a queue-based frontier to track vertices for the next iteration, based on Adpative PageRank by Kamvar et al. [30]. In another study, Chen et al. [11] broaden their Atos dynamic scheduling framework to multi-node GPU systems, accommodating Partitioned Global Address Space (PGAS) style lightweight one-sided memory operations within and between nodes. Yang et al. [60] introduce GraphBLAST, a high-performance linear algebra-based graph framework on the GPU. They address the lack of efficient GraphBLAS implementations for GPUs, highlighting the performance gap compared to state-of-the-art GPU graph frameworks, like the GraphBLAS Template Library (GBTL). They focus on exploiting input and output sparsity to simplify algorithm development and improve performance. They employ edge-balanced load balancing approach (merge-based) with segmented scan for PageRank, alongside a heuristic to switch between push- and pull-based approaches, favoring an early switch akin to Ligra [55]. Concessao et al. [16] introduce Meerkat, a library-based framework for dynamic graph algorithms optimized for GPU architectures. Leveraging a GPU-tailored graph representation and the warp-cooperative execution model, Meerkat enhances performance by exploiting common iteration patterns such as iterating over vertex neighborhoods efficiently. The framework supports dynamic edge additions and deletions, including batched versions. In their implementation of PageRank, Meerkat first computes the rank contribution of each vertex before calculating the rank itself. It

employs a pull-based warp-per-vertex approach for parallel computation, where threads collaborate within a warp to compute vertex ranks, and achieves coalesced writes for rank updates.

We now note some concerns regarding GPU-based implementations of PageRank algorithms in existing frameworks. Grus [57] and Atos [10] adopt a push-based PageRank computation method. This involves atomic add operations per edge, and can introduce needless memory contention among GPU threads. Our approach is pull-based, and requires only a single write per vertex. Atos additionally employs a queue-based frontier for tracking vertices to be processed in the next iteration. This requires atomic operations, and is not conducive to parallelism. While Grus and Meerkat [16] utilize warp-centric load balancing, they do not partition vertices into low and high in-degree sets. We partition vertices into such sets for load balancing, and to minimize thread divergence. Finally, Meerkat computes the rank contribution of each vertex, followed by a separate kernel for computing the ranks. It the uses another kernel to compute the common teleport contribution with atomics. However, as mentioned earlier, we calculate rank contributions on the fly, and employ only a pair of kernels for rank computation.

2.2 Dynamic PageRank

Early work in dynamic graph algorithms in the sequential setting includes the sparsification method proposed by Eppstein et al. [22] and Ramalingam's bounded incremental computation approach [47]. Several approaches have been suggested for incremental computation of approximate PageRank values in a dynamic or evolving graph. Chien et al. [15] identify a small region near updated vertices in the graph and represent the rest of the graph as a single vertex in a smaller graph. PageRanks are computed for this reduced graph and then transferred back to the original graph. Chen et al. [13] propose various methods to estimate the PageRank score of a webpage using a small subgraph of the entire web, by expanding backwards from the target node along reverse hyperlinks. Bahmani et al. [4] analyze the efficiency of Monte Carlo methods for incremental PageRank computation. Zhan et al. [61] introduce a Monte Carlo-based algorithm for PageRank tracking on dynamic networks, maintaining R random walks starting from each node. Pashikanti et al. [44] also employ a similar Monte Carlo-based approach for updating PageRank scores upon vertex and edge insertions/deletions.

A few approaches have been devised to update exact PageRank scores on dynamic graphs. Zhang [63] uses a simple incremental PageRank computation system for dynamic graphs, which we refer to as the *Naive-dynamic (ND)* approach, on hybrid CPU and GPU platforms. Additionally, Ohsaka et al. [42] propose a method for locally updating PageRank using the Gauss-Southwell method, prioritizing the vertex with the greatest residual for initial updating; however, their algorithm is inherently sequential. A widely adopted approach for updating PageRank [18, 25, 32, 51] is based on the observation that changes in the out-degree of a node do not influence its PageRank score, adhering to the first-order Markov property. The portion of the graph undergoing updates, involving edge insertions or deletions, is used to identify the affected region of the graph in a preprocessing step. This is typically accomplished through Breadth-First Search (BFS) or Depth-First Search (DFS) traversal from vertices connected to the inserted or deleted edges.

Subsequently, PageRanks are computed solely for this region. Desikan et al. [18] originally proposed this, which we term as the *Dynamic Traversal (DT)* approach in this report. Kim and Choi [32] apply this approach with an asynchronous PageRank implementation, while Giri et al. [25] utilize it with collaborative executions on multi-core CPUs and massively parallel GPUs. Sahu et al. [51] employ this strategy on a Strongly Connected Component (SCC)-based graph decomposition to limit computation to reachable SCCs from updated vertices, on multi-core CPUs and GPUs. Sallinen et al. [52] recently proposed an event-based approach for asynchronously updating PageRank scores of vertices in dynamic graphs. However, their per-thread event processing and termination detection methods are relatively complicated.

In our prior research [49], we introduced two parallel approaches, Dynamic Frontier (DF) and Dynamic Frontier with Pruning (DF-P), for updating PageRank on dynamic graphs while using the power-iteration method [43]. In the report, we recommended DF-P PageRank for real-world dynamic graphs, suggesting a switch to DF PageRank if higher error rates were observed. For large graphs with random batch updates, we recommended DF-P PageRank, except for web graphs, where we suggested choosing DF PageRank.

3 PRELIMINARIES

3.1 PageRank algorithm

The PageRank, represented as $R[v]$, of a vertex $v \in V$ in the graph $G(V, E)$, measures its importance based on incoming links. Equation 1 describes the PageRank computation for vertex v in graph G, where V denotes the set of vertices, E represents the set of edges, $G.in(v)$ and $G.out(v)$ denote incoming and outgoing neighbors of v, respectively, and α is the damping factor. Initially, each vertex has a PageRank of $1/|V|$. The *power-iteration* method iteratively updates these values until they converge within a specified iteration tolerance τ. This is often measured using the L_1-norm [42], though L_2 and L_∞-norm are also occasionally used.

Core to the PageRank algorithm is the random surfer model. It conceives a surfer navigating the web by following links on each page. The damping factor α, defaulting to 0.85, indicates the likelihood of the surfer continuing along a link rather than jumping randomly. PageRank for each page reflects the long-term probability of the surfer visiting that page, originating from a random page and following links. PageRank values essentially constitute the eigenvector of a transition matrix, encoding probabilities of page transitions in a Markov Chain.

Dead ends, also referred to as dangling vertices, present a significant challenge in PageRank computation due to their lack of out-links, which compels the surfer to transition to a random web page. As a consequence, dead ends evenly distribute their rank across all vertices in the graph, necessitating computation in each iteration, thereby incurring overhead. We tackle this problem by introducing self-loops to all vertices in the graph [3, 34, 37], a strategy observed to be particularly effective in streaming environments and spam-link applications [34].

$$R[v] = \alpha \times \sum_{u \in G.in(v)} \frac{R[u]}{|G.out(u)|} + \frac{1 - \alpha}{|V|} \qquad (1)$$

3.2 Fundamentals of a GPU

The fundamental building block of NVIDIA GPUs is the Streaming Multiprocessor (SM). Each SM consists of multiple CUDA cores, which are responsible for executing parallel threads. SMs also include shared memory, registers, and special function units. The number of SMs varies across different GPU models, and each SM operates independently, executing multiple threads in parallel [39, 53].

The memory hierarchy of NVIDIA GPUs includes global memory, shared memory, and local memory. Global memory is the largest and slowest memory type, providing high-capacity storage for data. Shared memory is a fast, low-latency memory that is shared among threads within an SM, facilitating efficient data sharing and communication. It is used to store data that is frequently accessed by multiple threads, which can improve performance by reducing the number of memory accesses. Local memory is per-thread private memory used to store variables that do not fit in registers [39, 53].

Threads on a GPU are structured differently from those on a multicore CPU, organized into a hierarchy of warps, thread blocks, and grids. A warp comprises threads executing instructions in synchronous manner, i.e., in lockstep. NVIDIA GPUs typically have a warp size of 32 threads. Thread blocks, on the other hand, are collections of threads executing on the same SM. Within a thread block, warps execute in lockstep, with the SM scheduling alternate warps if any threads stall, such as during memory requests. All threads within a block can communicate through a user-managed cache, known as shared memory, which is private to each SM. Lastly, a grid encompasses multiple thread blocks, each functioning independently. Grids offer a higher-level structure for managing parallelism, optimizing GPU resource utilization. Thread blocks within a grid communicate solely through global memory.

3.3 Dynamic Graphs

A dynamic graph can be construed as a succession of graphs, where $G^t(V^t, E^t)$ denotes the graph at time step t. The transitions between successive time steps $t - 1$ and t, from $G^{t-1}(V^{t-1}, E^{t-1})$ to $G^t(V^t, E^t)$, can be denoted as a batch update Δ^t at time step t. This update encompasses a collection of edge deletions Δ^{t-}, defined as $\{(u, v) \mid u, v \in V\} = E^{t-1} \setminus E^t$, and a set of edge insertions Δ^{t+}, defined as $\{(u, v) \mid u, v \in V\} = E^t \setminus E^{t-1}$.

3.4 Existing approaches for updating PageRank on Dynamic Graphs

3.4.1 Naive-dynamic (ND) approach. This approach simply updates vertex ranks in dynamic networks by initializing them with ranks from the previous graph snapshot and executing the PageRank algorithm on all vertices. Rankings obtained with this method are at least as accurate as those from the static algorithm.

3.4.2 Dynamic Traversal (DT) approach. The Dynamic Traversal (DT) approach was initially proposed by Desikan et al. [18]. It entails skipping the processing of vertices unaffected by the given batch update. For each edge deletion or insertion (u, v) in the batch update, all the vertices reachable from vertex u in either graph G^{t-1} or G^t are flagged as affected, typically using DFS or BFS.

4 APPROACH

4.1 Our GPU-based Static PageRank

We first discuss our GPU implementation of Static PageRank. Its psuedocode is given in Algorithm 1. The explanation of the psuedocode is given in Section 4.2.

Copying data to the device: We begin by copying the CSR representation of the transpose of the current graph $G^{t'}$ to the GPU.

Partitioning vertices into low-degree and high-degree sets: We employ a pair of kernels, namely a thread-per-vertex kernel and a block-per-vertex kernel, for rank computation on the GPU. For this, we partition the vertex IDs for the two kernels by their in-degree. The thread-per-vertex kernel handles vertices with low in-degrees, while the block-per-vertex kernel computes ranks of high in-degree vertices. Using two different kernels allows us to improve processing performance for high-degree vertices, while minimizing thread divergence for low-degree vertices. The psuedocode for partitioning is given in Algorithm 4, and is explained in Section A.2.

Rank computation of each vertex: We find that, unlike on multicore CPUs, employing a synchronous implementation of the PageRank algorithm, utilizing two separate rank vectors, yields superior performance. Consequently, we opt for a *synchronous* implementation of the PageRank algorithm. During the rank computation phase in each iteration, we employ two distinct kernels. The *thread-per-vertex* kernel manages vertices with low in-degree, assigning one thread per vertex to independently update each vertex's rank in a sequential manner. Conversely, the *block-per-vertex* kernel allocates a thread block per vertex. Within each thread block, threads compute the rank contribution of in-neighbors to the vertex in a strided manner. These partial rank contributions are stored in shared memory, and a block reduction (summation) operation is performed to obtain the net rank contribution. Subsequently, the first thread in the thread block updates the rank of the vertex.

Convergence detection: To determine if the ranks of vertices have converged, we calculate the L_∞-norm of the difference between the current and previous ranks. If this value is below the specified iteration tolerance τ, the algorithm terminates, indicating convergence. If not, we swap the current and previous ranks and proceed to the next iteration. The calculation of the L_∞-norm involves two kernels. The first kernel computes the L_∞-norm of the rank differences for each thread block in a grid and stores the results in a temporary buffer. The second kernel computes the net L_∞-norm of the results in the buffer, which is then transferred to the CPU.

4.2 Our Static PageRank implementation

Algorithm 1 outlines the psuedocode of our GPU-based Static PageRank. It takes as input the transpose of the current graph snapshot $G^{t'}$, and returns the computed rank vector R.

The algorithm begins by initializing the rank vectors R and R_{new}, where each vertex's initial rank is set to $1/|V^{t'}|$ (lines 3-5). We then partition the vertex IDs in P' based on their in-degree (line 7). This partitioning step is crucial for efficient computation on the GPU. Next, in lines 9-12, PageRank iterations are performed. Within each iteration, the ranks are updated using the updateRanks() function (line 10), which computes the updated PageRank scores R_{new} for

each vertex based on the previous iteration's ranks R (synchronous). When invoking updateRanks(), we disable the use of vertex and neighbor affected flags (δ_V and δ_N) in order the utilize the same function for Static PageRank (updateRanks() is also used with DF-P PageRank). After updating the ranks, we calculate the L_∞-norm difference ΔR between the current R_{new} and previous ranks R to check for convergence (line 11). If the change in ranks falls below the specified tolerance τ, the algorithm terminates (line 12). Finally, the algorithm returns the converged rank vector R (line 13).

Algorithm 1 uses a pull-based approach for PageRank computation, where each vertex's rank is updated through a single write by a thread. This is in contrast to a push-based approach, where each thread calculates and sums the outgoing PageRank contribution of its vertex to its neighbors, necessitating atomic updates [56]. We find this to be more efficient and employ it for all implementations [49]. We also employ an synchronous implementation of Static PageRank, using two separate rank vectors, which we observe to perform better than an asynchronous implementation.

Algorithm 1 Our GPU-based Static PageRank.

▷ $G^{t'}(V^{t'}, E^{t'})$: Transpose of current input graph
▷ R, R_{new}: Rank vector in the previous, current iteration
☐ P': Partitioned vertex IDs, low in-degree first
☐ N'_P: Number of vertices with low in-degree
☐ ΔR: $L\infty$-norm between previous and current ranks
☐ τ: Iteration tolerance

1: **function** STATIC($G^{t'}$)
2:     ▷ Initialize ranks
3:     $R \leftarrow R_{new} \leftarrow \{\}$
4:     **for all** $v \in V^{t'}$ **in parallel do**
5:         $R[v] \leftarrow R_{new}[v] \leftarrow 1/|V^{t'}|$
6:     ▷ Partition vertex IDs by in-degree
7:     $\{P', N'_P\} \leftarrow partition(G^{t'})$ ▷ Alg. 4
8:     ▷ Perform PageRank iterations
9:     **for all** $i \in [0..MAX\_ITERATIONS)$ **do**
10:         $updateRanks(\cdots, \cdots, R_{new}, R, G^{t'}, P', N'_P)$ ▷ Alg. 3
11:         $\Delta R \leftarrow l_\infty NormDelta(R_{new}, R) ; swap(R_{new}, R)$
12:         **if** $\Delta R \leq \tau$ **then break**
13:     **return** R

4.3 Our GPU-based Dynamic Frontier with Pruning (DF-P) PageRank

When dealing with a batch update comprising edge deletions $(u, v) \in \Delta^{t-}$ and insertions $(u, v) \in \Delta^{t+}$, if the total update size $|\Delta^{t-} \cup \Delta^{t+}|$ is relatively small compared to the total edge count $|E|$, only a small fraction of vertices are expected to undergo rank changes. To handle this, our earlier work proposed **Dynamic Frontier (DF)** and **Dynamic Frontier with Pruning (DF-P)** approaches, which employ an incremental process to identify affected vertices and update their ranks. Our research showed that DF/DF-P PageRank outperformed Static and Dynamic Traversal (DT) PageRank by 5.2×/15.2× and 1.3×/3.5× on real-world dynamic graphs, and by 7.2×/9.6× and 4.0×/5.6× on large graphs with random batch updates [49].

Unfortunately, multicore CPUs have limited memory bandwidth and parallelism, making them unsuitable for graph algorithms like PageRank, which have a low computation-to-communication ratio. In contrast, GPUs offer high-bandwidth memory, connected in close proximity to thousands of lightweight cores with user-managed caches. Moreover, GPU hardware is designed to switch running threads at no cost to support memory access latency hiding. When appropriately designed, graph algorithms can significantly outperform CPU-based implementations on GPUs. This makes a GPU implementation of DF and DF-P PageRank attractive. In this section, we describe the design of DF and DF-P PageRank for GPUs.

Copying data to the device: We copy the CSR representation of the current graph G^t, its transpose $G^{t'}$, the previous ranks of each vertex R^{t-1}, and the batch update consisting of edge deletions Δ^{t-} (both source and target vertex IDs in separate arrays) and edge insertions Δ^{t+} (source vertex IDs only) to the global memory of the GPU. The CSR of $G^{t'}$ is used for computing PageRank scores, while that of G^t is used for the initial and incremental marking of affected vertices. Note that we only need the source vertex IDs of edge insertions $(u, v) \in \Delta^{t+}$ as we only need to mark the outgoing neighbors of u as affected. On the other hand, for edge deletions $(u, v) \in \Delta^{t-}$, we need both the source and target vertex IDs as we need to mark both the outgoing neighbors of u and the target vertices of the edge deletions, i.e., v, as affected.

Partitioning vertices into low-degree and high-degree sets: The work to be performed by each thread is proportional to the in-degree of each vertex during the rank computation, and to the out-degree of each vertex during incremental marking of affected vertices. Thus, similar to Static PageRank, we use a pair of kernels, i.e., a *thread-per-vertex* kernel and a *block-per-vertex* kernel, for both the rank computation and the incremental marking phases, by partitioning the vertex IDs for the two kernels by their in-degree for the rank computation phase, and by their out-degree for the incremental marking phase. This also helps minimize thread divergence with the thread-per-vertex kernel. The psuedocode for partitioning the vertices (by in- or out-degree of vertices) is given in Algorithm 4, with its explanation given in Section A.2.

Marking the initial set of affected vertices: Upon each edge deletion $(u, v) \in \Delta^{t-}$ and insertion $(u, v) \in \Delta^{t+}$ in the batch update, with the DF and DF-P approaches, we need to mark the outgoing neighbors of vertex u in both the previous G^{t-1} and current graph G^t as affected. This is equivalent to marking the outgoing neighbors of the source vertex, u, for each edge deletion and insertion $(u, v) \in \Delta^{t-} \cup \Delta^{t+}$, and marking the target vertex, v, of each edge deletion $(u, v) \in \Delta^{t-}$ as affected. We use one kernel to mark the target vertex, v, for each edge deletion $(u, v) \in \Delta^{t-}$ as affected, and use a temporary array to indicate that the outgoing neighbors of the source vertex, u, for each edge deletion and insertion $(u, v) \in \Delta^{t-} \cup \Delta^{t+}$ need to be marked as affected later. A thread-per-vertex kernel and block-per-vertex kernel are then used to actually mark the outgoing neighbors of such vertices as affected. The vertex IDs between the two kernels are partitioned by out-degree, as mentioned above.

Rank computation and incremental expansion / contraction of the set of affected vertices: Similar to our Static PageRank, we adopt a synchronous implementation of the PageRank algorithm, employing a kernel pair during each iteration's rank computation phase. In addition, with the DF and DF-P approaches, if the relative change in the rank of a vertex u is greater than the frontier tolerance τ_f, we need to incrementally mark the outgoing neighbors $v \in G^t.out(u)$ of u as affected [49]. However, performing this incremental marking during the rank computation phase may introduce significant thread divergence (the work is proportional to the out-degree of u, and not its in-degree). To avoid this, we use a temporary array to indicate that the neighbors of u need to be incrementally marked as affected. This can later be used with a pair of kernels to actually mark the outgoing neighbors of such vertices as affected.

With the DF-P approach, if the relative change in the rank of a vertex u falls within the prune tolerance τ_p, the vertex u needs to be marked as not affected [49]. This is achieved by directly unflagging it in the set of affected vertices, which entails $O(1)$ work and does not introduce significant thread divergence. Additionally, considering the potential pruning of vertices and the incorporation of a self-loop for each vertex in the graph (as discussed in Sections 5.1.3 and 5.1.4), we employ a closed-loop formula for computing the rank of each vertex (Equation 2). This formula is specifically designed to accommodate the presence of the self-loop, thereby mitigating the need for recursive rank calculations stemming from it [49]. We refer the reader to our prior work [49] for an example of the DF and DF-P approaches.

Convergence detection: This is similar to that of Static PageRank.

$$R[v] = \frac{1}{1 - \alpha/|G.out(v)|}\left(\alpha K + \frac{1-\alpha}{|V|}\right) \quad (2)$$

$$\text{where, } K = \left(\sum_{u \in G.in(v)} \frac{R[u]}{|G.out(u)|}\right) - \frac{R[v]}{|G.out(v)|} \quad (3)$$

4.4 Determining suitable Partitioning approach

In order to optimize the performance of DF and DF-P PageRank for the GPU, we attempt three different partitioning techniques for work distribution between the thread-per-vertex and block-per-vertex kernels — for updating ranks of vertices in the graph and incrementally marking affected vertices.

With the first technique, which we refer to as *Don't Partition*, we do not partition the graph, and instead selectively execute the thread/block-per-vertex kernels on each vertex, depending on the in/out-degree of the vertex — for both the rank computation phase and the incremental marking of affected vertices. With the second technique, which we refer to as *Partition G'* (G' stands for transpose of G, the current graph), we partition the graph into low in-degree and high in-degree vertices, and run the kernels on respective partitions for updating ranks — the incremental marking of affected vertices is still done selectively. With the third technique, which we refer to as *Partition G, G'*, we partition the graph by both in-degrees and out-degrees, and run the kernels of respective partitions (i.e., thread-per-vertex kernel on low degree vertices, and block-per-vertex kernel on high degree vertices) for both rank computation and incremental marking of affected vertices.

Figure 1 illustrates the relative runtime with each partitioning technique. Here, the measured runtime includes the overall runtime of the DF/DF-P PageRank, and not just the time needed for partitioning the vertices, or performing rank computation. Results indicate that the *Partition G, G'* technique performs the best, as shown in Figure 1. Note that partitioning the vertex IDs by out-degree, i.e., *Partition G*, is useful for incremental marking of affected vertices, but comes with added runtime cost — hence the small improvement in performance when moving from *Partition G'* to *Partition G, G'*.
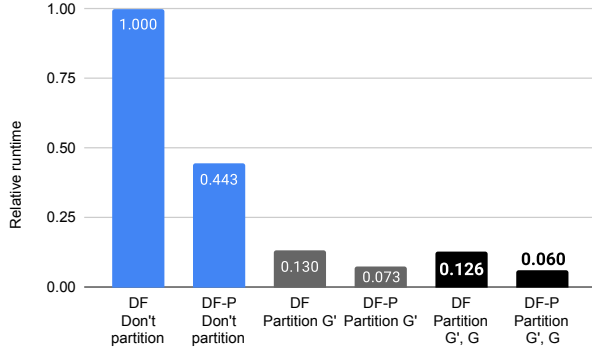
Figure 1: Mean relative runtime with our *Dynamic Frontier (DF)* and *Dynamic Frontier with Pruning (DF-P)* approaches across three different levels of work-partitioning for GPU computation. Here, *Partition G* denotes partitioning the vertices of the current graph G by their out-degree, while *Partition G'* signifies partitioning the vertices by their in-degree. Note that G' stands for the transpose of the current graph G.

4.5 Our DF* PageRank implementation

Algorithm 2 presents the psuedocode of GPU-based DF and DF-P PageRank, which aims to efficiently compute PageRank on large-scale graphs with dynamic updates. The algorithm takes several inputs, including the current graph snapshot G^t and its transpose $G^{t'}$, edge deletions Δ^{t-} and insertions Δ^{t+}, and the previous rank vector R^{t-1}. It returns the updated rank vector R.

We start by initializing the rank vectors R and R_{new} with the previous rank vector R^{t-1} (line 3). Next, we partition the vertex IDs based on their out- and in-degree, aiming for efficient computation on the GPU (lines 5-6). Then, we mark the initial set of affected vertices using the provided edge deletions and insertions, and expand the affected set to include relevant neighbor vertices (lines 7-9). Subsequently, we start with PageRank iterations (lines 11-16), continuing until either the maximum number of iterations $MAX_ITERATIONS$ is reached or the change in ranks falls below the specified tolerance τ (line 15). Within each iteration, we update the rank vector R_{new} based on the affected vertices (line 13), while marking the vertices whose neighbors must be incrementally marked as affected (with relative change in rank greater than the frontier tolerance τ_f) or contracting the set of affected vertices if the change in rank of a vertex is small (with relative change in rank below the prune tolerance τ_p, only with DF-P PageRank). The

L_∞-norm difference between the current R_{new} and previous ranks R is then computed to check for convergence, and the rank vectors are swapped for the next iteration (line 14). In line 15, we perform a convergence check. If convergence has not yet been achieved, we incrementally expand the set of affected vertices (line 16) from the vertices identified during rank computation (line 13). Finally, we return the updated rank vector R (line 17). The details for updateRanks(), partition(), and initialAffected()/expandAffected() functions is given in Sections A.1, A.2, and A.3, respectively.

Algorithm 2 also uses a pull-based synchronous implementation of PageRank, similar to Algorithm 1. This is in contrast to our multicore CPU implementation of DF and DF-P PageRank, where we observe that an asynchronous implementation offers better performance [49, 50]. We also utilize synchronous implementations for Naive-dynamic (ND) and Dynamic Traversal (DT) PageRank.

Algorithm 2 Our GPU-based Dynamic Frontier (DF*) PageRank.

▷ $G^t(V^t, E^t), G^{t'}$: Current input graph, and its transpose
▷ Δ^{t-}, Δ^{t+}: Edge deletions and insertions (input)
▷ R^{t-1}: Previous rank vector
▷ R, R_{new}: Rank vector in the previous, current iteration
□ δ_V, δ_N: Is a vertex, or neighbors of a vertex affected
□ P, P': Partitioned vertex IDs — low out-, in-degree first
□ N_P, N_P': Number of vertices with low out-, in-degree
□ ΔR: $L\infty$-norm between previous and current ranks
□ τ: Iteration tolerance

1: **function** DYNAMICFRONTIER($G^t, G^{t'}, \Delta^{t-}, \Delta^{t+}, R^{t-1}$)
2:    ▷ Initialize ranks
3:    $R \leftarrow R_{new} \leftarrow R^{t-1}$
4:    ▷ Partition vertex IDs by out- and in-degree
5:    $\{P, N_P\} \leftarrow partition(G^t)$ ▷ Alg. 4
6:    $\{P', N_P'\} \leftarrow partition(G^{t'})$ ▷ Alg. 4
7:    ▷ Mark initial set of affected vertices
8:    $\{\delta_V, \delta_N\} \leftarrow initialAffected(G^t, \Delta^{t-}, \Delta^{t+})$ ▷ Alg. 5
9:    $expandAffected(\delta_V, \delta_N, G^t, P, N_P)$ ▷ Alg. 5
10:   ▷ Perform PageRank iterations
11:   **for all** $i \in [0..MAX\_ITERATIONS)$ **do**
12:       $\delta_N \leftarrow \{\}$
13:       $updateRanks(\delta_V, \delta_N, R_{new}, R, G^t, P', N_P')$ ▷ Alg. 3
14:       $\Delta R \leftarrow l_\infty NormDelta(R_{new}, R)$; $swap(R_{new}, R)$
15:       **if** $\Delta R \leq \tau$ **then break**
16:       $expandAffected(\delta_V, \delta_N, G^t, P, N_P)$ ▷ Alg. 5
17:   **return** R

---

5 EVALUATION

5.1 Experimental Setup

5.1.1 System used. Experiments are performed on a system featuring an NVIDIA A100 GPU with 108 SMs and 64 CUDA cores per SM. The GPU has 80 GB of global memory, with a bandwidth of 1935 GB/s. Each SM has a shared memory capacity of 164 KB. The system also has an AMD EPYC-7742 processor with 64 cores, running at a frequency of 2.25 GHz. The server is set up with 512 GB of DDR4 system memory and runs Ubuntu 20.04.

5.1.2 Configuration. We utilize 32-bit integers to represent vertex IDs and 64-bit floating-point numbers for vertex ranks. Affected vertices are denoted by an 8-bit integer vector. Our configuration sets the damping factor to $\alpha = 0.85$ [37], the frontier tolerance τ_f to 10^{-6} [49], the prune tolerance τ_p to 10^{-6} [49], and the iteration tolerance τ to 10^{-10} using the L_∞-norm [19, 46]. We cap the maximum number of iterations *MAX_ITERATIONS* at 500 [41]. Compilation is performed using NVCC, a part of CUDA toolkit 11.4, and OpenMP 5.0 (for CPU code). OpenMP's *dynamic schedule* with a chunk size of 2048 is used for CPU-based computations to facilitate dynamic workload balancing among threads.

5.1.3 Dataset. To experiment with real-world dynamic graphs, we employ five temporal networks sourced from the Stanford Large Network Dataset Collection [38]. Details of these graphs are summarized in Table 3. Here, the number of vertices range from 24.8 thousand to 2.60 million, temporal edges from 507 thousand to 63.4 million, and static edges from 240 thousand to 36.2 million. For investigations involving large static graphs with random batch updates, we utilize 12 graphs as listed in Table 3, obtained from the SuiteSparse Matrix Collection [35]. Here, the number of vertices range from 3.07 to 214 million, and edges from 25.4 million to 3.80 billion. To mitigate dead ends (vertices without out-links) and the associated overhead of global teleport rank computation in each iteration, we augment all vertices with self-loops [3, 34, 37].

Table 3: List of 5 **real-world dynamic graphs, sourced from the Stanford Large Network Dataset Collection [38]. Here,** $|V|$ **denotes the number of vertices,** $|E_T|$ **represents the count of temporal edges (inclusive of duplicates), and** $|E|$ **indicates the number of static edges (without duplicates).**

Graph	$|V|$	$|E_T|$	$|E|$
sx-mathoverflow	24.8K	507K	240K
sx-askubuntu	159K	964K	597K
sx-superuser	194K	1.44M	925K
wiki-talk-temporal	1.14M	7.83M	3.31M
sx-stackoverflow	2.60M	63.4M	36.2M

5.1.4 Batch Generation. For experiments involving real-world dynamic graphs, we first load 90% of each graph from Table 3, and add self-loops to all vertices. Subsequently, we load B edges in 100 batch updates. Here, B denotes the desired batch size, specified as a fraction of the total number of temporal edges $|E_T|$ in the graph. For experiments on large graphs with random batch updates, we select each base (static) graph from Table 4 and generate a random batch update comprising an 80% : 20% mix of edge insertions and deletions to emulate realistic batch updates. To prepare the set of edges for insertion, we select vertex pairs with equal probability. For edge deletions, we uniformly delete each existing edge. To simplify, we ensure no new vertices are added or removed from the graph. The batch size is measured as a fraction of edges in the original graph, ranging from 10^{-7} to 0.1 (i.e., $10^{-7}|E|$ to $0.1|E|$), with multiple batches generated for each size for averaging. Self-loops are added to all vertices alongside each batch update, as earlier.

Table 4: List of 12 **graphs obtained from the SuiteSparse Matrix Collection [35], with directed graphs marked by** ∗. **Here,** $|V|$ **denotes the number of vertices,** $|E|$ **denotes the total number of edges (including self-loops), and** D_{avg} **denotes the average degree of a vertex in each graph.**

Graph	$|V|$	$|E|$	D_{avg}
Web Graphs (LAW)			
indochina-2004*	7.41M	199M	26.8
arabic-2005*	22.7M	654M	28.8
uk-2005*	39.5M	961M	24.3
webbase-2001*	118M	1.11B	9.4
it-2004*	41.3M	1.18B	28.5
sk-2005*	50.6M	1.98B	39.1
Social Networks (SNAP)			
com-LiveJournal	4.00M	73.4M	18.3
com-Orkut	3.07M	237M	77.3
Road Networks (DIMACS10)			
asia_osm	12.0M	37.4M	3.1
europe_osm	50.9M	159M	3.1
Protein k-mer Graphs (GenBank)			
kmer_A2a	171M	531M	3.1
kmer_V1r	214M	679M	3.2

5.1.5 Measurement. We assess the runtime of each approach on the entire updated graph, including partitioning, initial marking of affected vertices, and convergence detection time. However, we exclude the time taken for memory transfers (to and from the GPU) as well as allocation/deallocation. The mean time and error for a specific method at a given batch size are computed as the geometric mean across input graphs. Additionally, we evaluate the error/accuracy of each approach by measuring the $L1$-norm [42] of the returned ranks compared to ranks obtained from a reference Static PageRank run on the updated graph with an extremely low iteration tolerance of $\tau = 10^{-100}$, limited to 500 iterations.

5.2 Performance of Our Static PageRank

We now evaluate the performance of our GPU implementation of Static PageRank, and compare it with the performance of Static PageRank in the Hornet [8] and Gunrock [58] graph processing frameworks on large (static) graphs from Table 4. For Hornet, we use a CUDA C++ program to read each input graph with `GraphStd::read()`, perform `HornetInit`, create a `HornetGraph`, and set up `StaticPageRank` with a damping factor α of 0.85, and iteration tolerance τ of 10^{-10}, and limit the maximum number of iterations to 500. We also define a new PageRank operator called `Max`, to compute L_∞-norm of the absolute difference between the previous and current rank vectors (since we use L_∞-norm for convergence detection), and use this for convergence detection (with `forAllnumV()`) instead of $L1$-norm (used by default). To perform the PageRank computation, we then use `StaticPageRank::run()`, and measure its runtime with `Timer<DEVICE>`. For Gunrock, we use a CUDA C++ program to read each input graph in Table 4 with
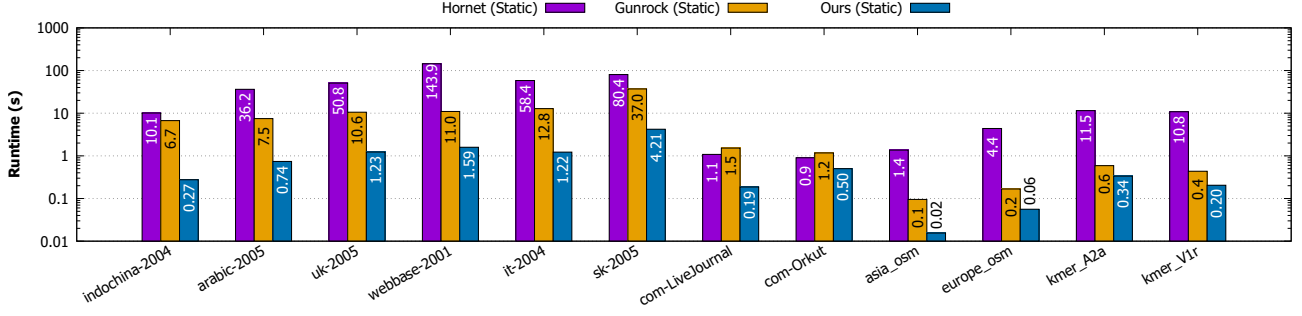
(a) Runtime in seconds (logarithmic scale) with *Hornet*, *Gunrock*, *Our* Static PageRank.
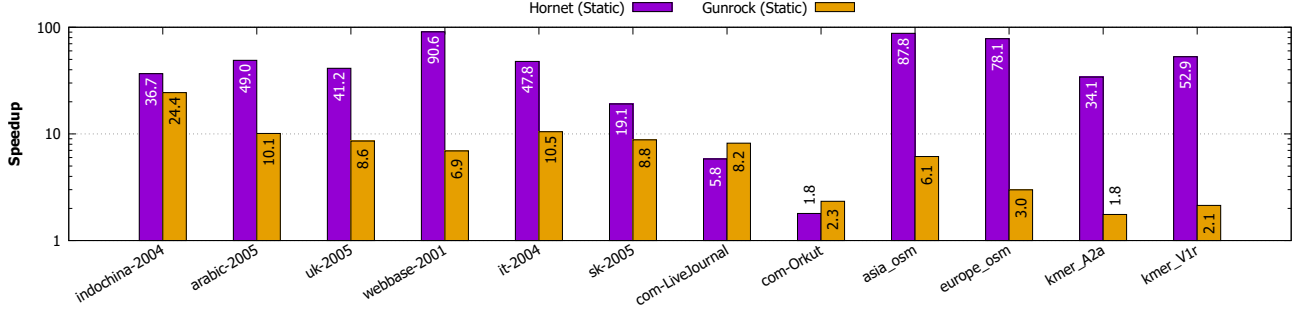
(b) Speedup of *Our* Static PageRank (logarithmic scale) with respect to *Hornet* and *Gunrock*.

Figure 2: Runtime in seconds and speedup (log-scale) with *Hornet, Gunrock, Our* Static PageRank for each graph in the dataset.

io::matrix_market_t::load(), convert it to a CSR representation with format::csr_t::from_coo(), and build a graph with graph::build::from_csr(). We then perform PageRank computation with gunrock::pr::run() upon the loaded graph with a damping factor α of 0.85, an iteration tolerance of 10^{-10}, and limit the number of iterations to 500 by modifying the gunrock::pr::en actor_t::is_ converged() function (note that Gunrock uses L_∞-norm for convergence detection by default), and record the runtime reported by gunrock::pr::run().

Figure 2(a) illustrates the runtime of Hornet, Gunrock, and our Static PageRank on the GPU, for each graph in the dataset. On the *sk-2005* graph, our Static PageRank computes the ranks of vertices with an iteration tolerance τ of 10^{-10} in 4.2 seconds, achieving a processing rate of 471 million edges/s. Figure 2(b) shows the speedup of Our Static PageRank with respect to Hornet and Gunrock. Our Static PageRank is on average 31× faster than Hornet, and 5.9× times faster than Gunrock. This speedup is particularly high on the *webbase-2001* graph and road networks with Hornet, and on the *indochina-2004* graph with Gunrock. Further, our GPU implementation of Static PageRank is on average 24× times faster than our parallel multicore implementation of Static PageRank.

5.3 Performance of Our DF-P PageRank

5.3.1 Results on real-world dynamic graphs. We now compare the performance of our GPU implementation of Dynamic Frontier (DF) and Dynamic Frontier with Pruning (DF-P) PageRank with Static, Naive-dynamic (ND), and Dynamic Traversal (DT) PageRank on

real-world dynamic graphs from Table 3. This is done on batch updates of size $10^{-5}|E_T|$ to $10^{-3}|E_T|$ in multiples of 10. For each batch size, as mentioned in Section 5.1.4, we load 90% of the graph, add self-loops to all vertices in the graph, and then load B edges (where B is the batch size) consecutively in 100 batch updates. Figure 3(a) displays the overall runtime of each approach across all graphs for each batch size, while Figure 3(b) illustrates the overall rank error compared to a reference Static PageRank run (as described in Section 5.1.5). Additionally, Figures 3(c) and 3(d) present the mean runtime and rank error of the approaches on each dynamic graph in the dataset. Figure 6 presents a comparison of the overall runtime and error between the GPU implementation of each approach and its respective CPU counterpart. Finally, Figures 9, 10, 11, 12, and 13 show the runtime and rank error of the approaches on each dynamic graph in Table 3, upon each consecutive batch update.

Figure 3(a) shows that DF PageRank is, on average, 1.4× faster than Static PageRank for batch updates of size $10^{-5}|E_T|$. In contrast, DF-P PageRank demonstrates average speedups of 3.6×, 2.0×, and 1.3× over Static PageRank for batch update sizes of $10^{-5}|E_T|$, $10^{-4}|E_T|$, and $10^{-3}|E_T|$, respectively. Furthermore, DF-P PageRank achieves average speedups of 4.2×, 2.8×, and 3.6× compared to DT PageRank for the same batch updates. This speedup is particularly pronounced on the *sx-mathoverflow* graph, as indicated by Figure 3(c). Regarding rank error, Figures 3(b) and 3(d) illustrate that DF and DF-P PageRank generally exhibit higher error on average compared to ND and DT PageRank but lower error than Static PageRank. This makes the ranks obtained with DF and
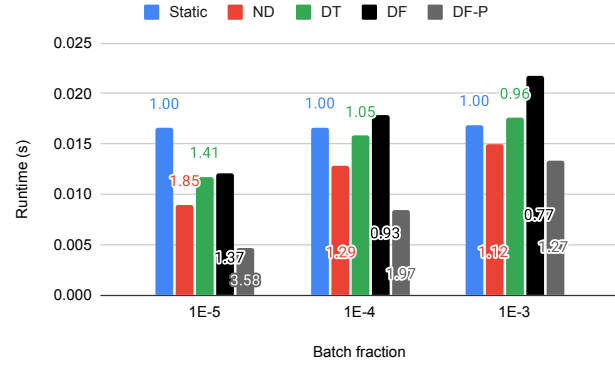
(a) Overall Runtime
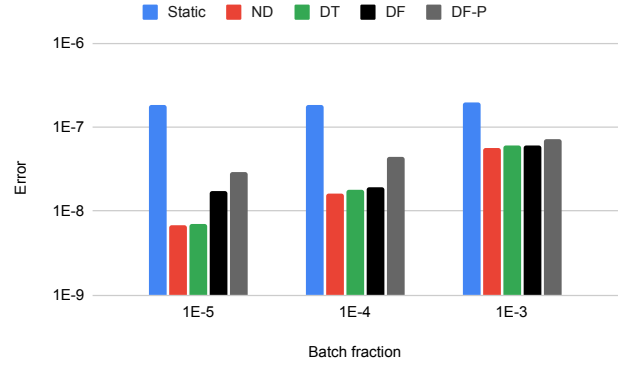
(b) Overall Error in ranks obtained
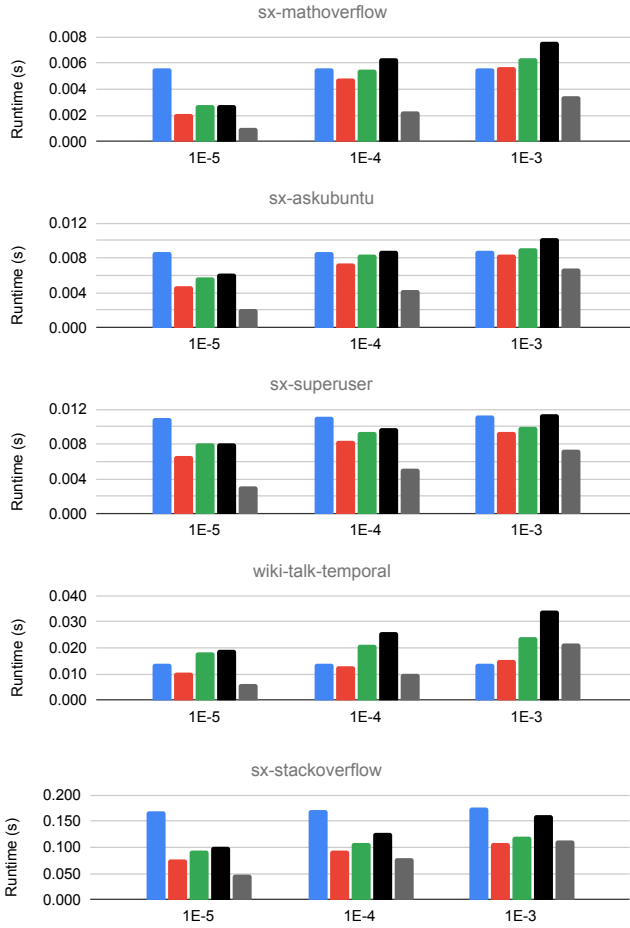
(c) Runtime on each dynamic graph
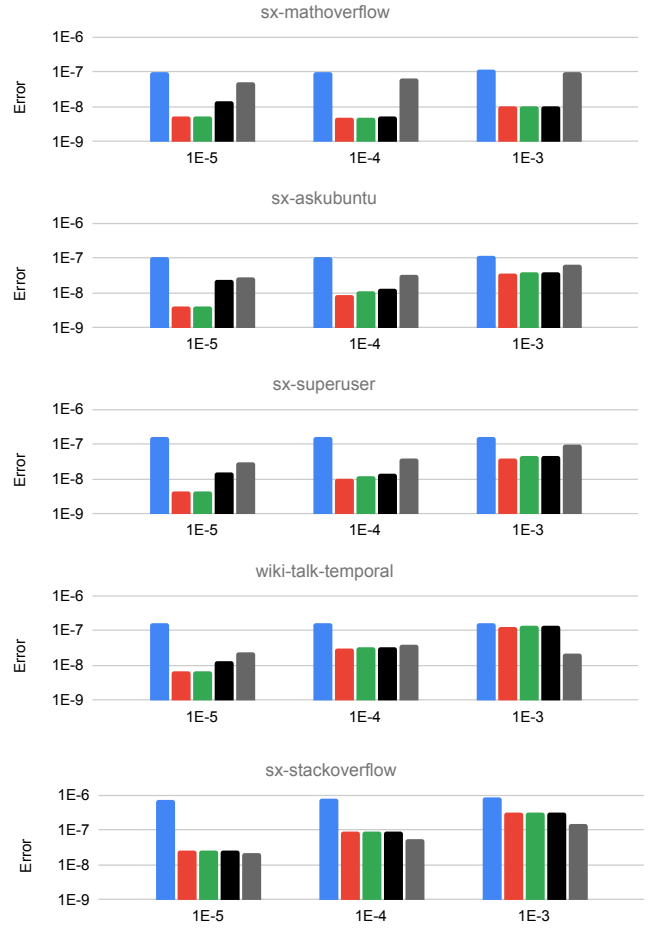
(d) Error in ranks obtained on each dynamic graph

Figure 3: Mean Runtime and Error in ranks obtained with our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on real-world dynamic graphs, with batch updates of size $10^{-5}|E_T|$ to $10^{-3}|E_T|$. Here, (a) and (b) show the overall runtime and error across all temporal graphs, while (c) and (d) show the runtime and rank error for each approach (relative to reference Static PageRank, see Section 5.1.5). In (a), the speedup of each approach with respect to Static PageRank is labeled.
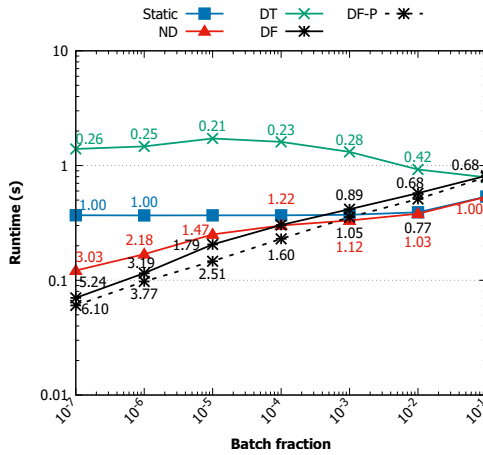
(a) Overall result
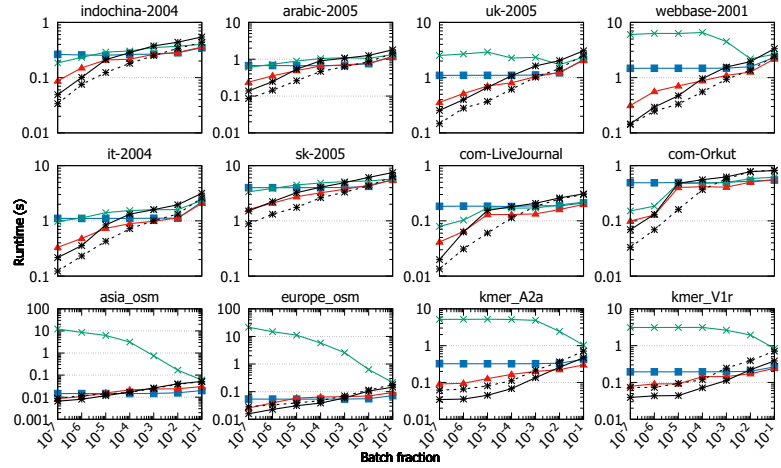
(b) Results on each graph

Figure 4: Runtime (logarithmic scale) of GPU implementation for *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF)*, and *Dynamic Frontier with Pruning (DF-P)* PageRank on large (static) graphs with generated random batch updates. Batch updates range in size from $10^{-7}|E|$ to $0.1|E|$ in multiples of 10. These updates consist of 80% edge insertions and 20% edge deletions, mimicking realistic changes in a dynamic graph scenario. The right subfigure illustrates the runtime of each approach for individual graphs in the dataset, while the left subfigure presents overall runtimes (using geometric mean for consistent scaling across graphs). Additionally, the speedup of each approach relative to Static PageRank is labeled.
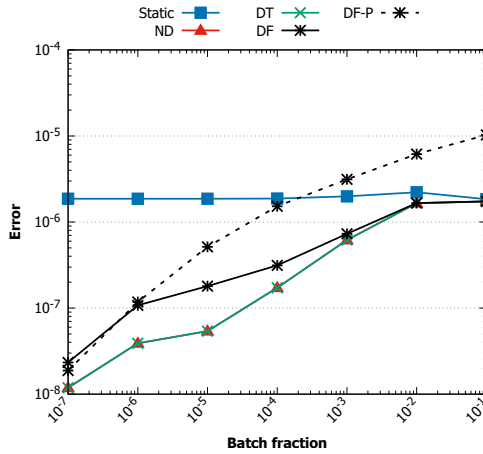
(a) Overall result
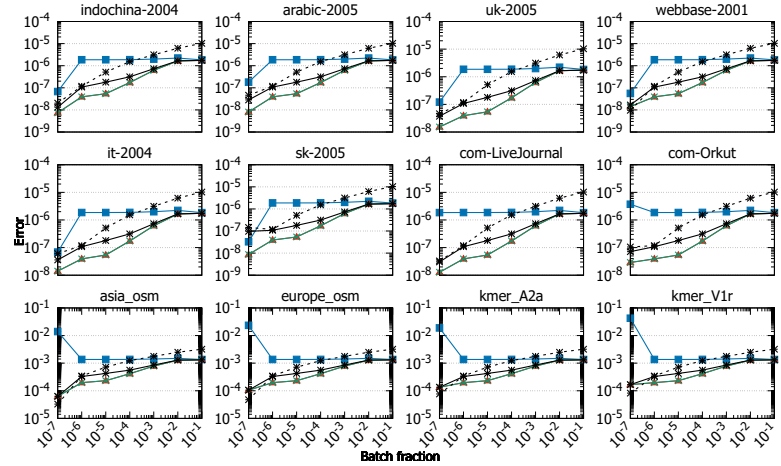
(b) Results on each graph

Figure 5: Error comparison of our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF)*, and *Dynamic Frontier with Pruning (DF-P)* PageRank on large (static) graphs with generated random batch updates, relative to a Reference Static PageRank (see Section 5.1.5), using $L1$-norm. The size of batch updates range from $10^{-7}|E|$ to $0.1|E|$ in multiples of 10 (logarithmic scale), consisting of 80% edge insertions and 20% edge deletions to simulate realistic dynamic graph updates. The right subfigure depicts the error for each approach in relation to each graph, while the left subfigure showcases overall errors using geometric mean for consistent scaling across graphs.

DF-P PageRank acceptable. DF-P PageRank can thus be the default choice for updating PageRank scores on dynamic graphs. However, if elevated error levels are observed (during intermediate empirical tests), transitioning to ND PageRank is advisable.

5.3.2 Results on large graphs with random updates. We also assess the performance of our GPU implementation of DF and DF-P PageRank alongside Static, ND, and DT PageRank on large (static) graphs listed in Table 4, with randomly generated batch updates.

As elaborated in Section 5.1.4, the batch updates vary in size from $10^{-7}|E|$ to $0.1|E|$ (in multiples of 10), comprising 80% edge insertions and 20% edge deletions to mimic realistic scenarios. Self-loops are added to all vertices with each batch update. Figure 4 illustrates the runtime of Static, ND, DT, DF, and DF-P PageRank, while Figure 5 displays the error in ranks obtained with each approach. Figures 7 and 8 depict a comparison of the overall runtime and error between the GPU and CPU implementation of each approach.

Figure 4(a) illustrates that for batch updates ranging from $10^{-7}|E|$ to $10^{-4}|E|$, DF PageRank achieves an average speedup of 2.5×, 1.3×, and 10.4× compared to Static, ND, and DT PageRank, respectively. Further, DF-P PageRank achieves average speedups of 3.1×, 1.7×, and 13.1× over Static, ND, and DT PageRank, respectively. This acceleration is particularly pronounced on road networks and protein k-mer graphs, characterized by a low average degree (as depicted in Figure 4(b)). It may be noted that DT PageRank exhibits slower performance than ND PageRank on large graphs with uniformly random batch updates. This is attributed to DT PageRank marking a significant number of vertices as affected, as updates are scattered randomly across the graph, rendering most of the graph reachable from the updated regions [50]. Particularly on road networks and protein k-mer graphs, characterized by a low average degree and a large diameter, DT PageRank's performance is further hindered due to limited parallelism exploitable by the GPU. Figures 5(a) and 5(b) indicate that DF-P PageRank generally exhibits higher error compared to ND, DT, and DF PageRank, but lower error than Static PageRank (up to a batch size of $10^{-4}|E|$). Hence, we recommend utilizing DF-P PageRank for large random batch updates up to a batch size of $10^{-4}|E|$. For larger batch updates, we advise the reader to switch to ND PageRank instead.

6 CONCLUSION

In conclusionl, in this report, we introduced one of the most efficient GPU implementations of Static PageRank, which involves recomputing PageRank scores from scratch. This implementation employs a synchronous pull-based approach for PageRank computation, devoid of atomics. It partitions and processes low and high in-degree vertices using two separate kernels. Furthermore, we presented our GPU implementation of Dynamic Frontier with Pruning (DF-P) PageRank, which incrementally expands (and contracts) the set of vertices likely to undergo rank changes. It is based on Static PageRank, and introduces additional partitioning between low and high out-degree vertices for the incremental expansion of the affected vertices set, implemented through two additional kernels. On a server equipped with an NVIDIA A100 GPU, we showed that our Static PageRank implementation outperforms Hornet and Gunrock's PageRank implementations by 31× and 5.9×, respectively. Additionally, it achieves a speedup of 24× compared to our multi-core Static PageRank. Furthermore, DF-P PageRank demonstrates superior performance compared to Static PageRank, achieving a speedup of 2.1× on real-world dynamic graphs and 3.1× on large static graphs with random batch updates.

REFERENCES

[1] Manoj K Agarwal, Krithi Ramamritham, and Manish Bhide. 2012. Real time discovery of dense clusters in highly dynamic graphs: identifying real world events in highly dynamic environments. *arXiv preprint arXiv:1207.0138* (2012).

[2] Stefano Allesina and Mercedes Pascual. 2009. Googling food webs: can an eigenvector measure species' importance for coextinctions? *PLoS computational biology* 5, 9 (2009), e1000494.

[3] R. Andersen, F. Chung, and K. Lang. 2007. Local partitioning for directed graphs using pagerank. In *in Proc. WAW*. 166–178.

[4] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. 2010. Fast incremental and personalized pagerank. *arXiv preprint arXiv:1006.2880* (2010).

[5] Dániel Bánky, Gábor Iván, and Vince Grolmusz. 2013. Equal opportunity for low-degree network nodes: a PageRank-based method for protein target identification in metabolic graphs. *PLoS One* 8, 1 (2013), e54204.

[6] Claudio DT Barros, Matheus RF Mendonça, Alex B Vieira, and Artur Ziviani. 2021. A survey on embedding dynamic graphs. *ACM Computing Surveys (CSUR)* 55, 1 (2021), 1–37.

[7] S. Beamer, K. Asanovic, and D. Patterson. 2017. Reducing pagerank communication via propagation blocking. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.

[8] Federico Busato, Oded Green, Nicola Bombieri, and David A Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[9] Ali Cevahir, Cevdet Aykanat, Ata Turk, B Barla Cambazoglu, Akira Nukada, and Satoshi Matsuoka. 2010. Efficient PageRank on GPU clusters. *IPSJ SIG Notes* 21 (2010), 1–6.

[10] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluc, Katherine Yelick, and John Owens. 2022. Atos: A task-parallel GPU scheduler for graph analytics. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[11] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluç, Katherine Yelick, and John D Owens. 2022. Scalable irregular parallelism with GPUs: getting CPUs out of the way. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[12] YuAng Chen and Yeh-ching Chung. 2021. HiPa: Hierarchical partitioning for fast PageRank on NUMA multicore systems. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–10.

[13] Yen-Yu Chen, Qingqing Gan, and Torsten Suel. 2004. Local methods for estimating pagerank values. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 381–389.

[14] Alexei D Chepelianskii. 2010. Towards physical laws for software architecture. *arXiv preprint arXiv:1003.5455* (2010).

[15] S. Chien, C. Dwork, R. Kumar, and D. Sivakumar. 2001. Towards Exploiting Link Evolution.

[16] Kevin Jude Concessao, Unnikrishnan Cheramangalath, MJ Dev, and Rupesh Nasre. 2023. Meerkat: A framework for Dynamic Graph Algorithms on GPUs. *arXiv preprint arXiv:2305.17813* (2023).

[17] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*. 752–768.

[18] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. 2005. Incremental Page Rank Computation on Evolving Graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (Chiba, Japan) (WWW '05)*. Association for Computing Machinery, New York, NY, USA, 1094–1095. https://doi.org/10.1145/1062745.1062885

[19] H. Dubey and N. Khare. 2022. Fast parallel computation of PageRank scores with improved convergence time. *IJDMMM* 14, 1 (2022), 63–88.

[20] N. Duong, Q. Nguyen, A. Nguyen, and H. Nguyen. 2012. Parallel pagerank computation using GPUs. In *In Proceedings of the Third Symposium on Information and Communication Technology*. 223–230.

[21] Nhat Tan Duong, Quang Anh Pham Nguyen, Anh Tu Nguyen, and Huu-Duc Nguyen. 2012. Parallel pagerank computation using gpus. In *Proceedings of the 3rd Symposium on Information and Communication Technology*. 223–230.

[22] D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig. 1997. Sparsification — A technique for speeding up dynamic graph algorithms. *J. ACM* 44, 5 (September 1997), 669–696. http://doi.acm.org/10.1145/265910.265914

[23] A. Fender, N. Thejaswi, and B. Rees. [n. d.]. rapidsai/nvgraph. https://github.com/rapidsai/nvgraph/blob/main/cpp/src/pagerank.cu#L149

[24] P. Garg and K. Kothapalli. 2016. STIC-D: Algorithmic Techniques For Efficient Parallel Pagerank Computation on Real-World Graphs. In *Proceedings of the 17th International Conference on Distributed Computing and Networking - ICDCN '16*. ACM Press, 1–10.

[25] H. Giri, M. Haque, and D. Banerjee. 2020. HyPR: Hybrid Page Ranking on Evolving Graphs. In *Proc. IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 62–71.

[26] Chen Gong, Keren Fu, Artur Loza, Qiang Wu, Jia Liu, and Jie Yang. 2013. Pagerank tracker: From ranking to tracking. *IEEE Transactions on Cybernetics* 44, 6 (2013), 882–893.

[27] Thomas Grützmacher, Hartwig Anzt, Florian Scheidegger, and Enrique S Quintana-Orti. 2018. High-performance GPU implementation of PageRank with reduced precision based on mantissa segmentation. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 61–68.

[28] Thomas Grützmacher, Terry Cojean, Goran Flegar, Hartwig Anzt, and Enrique S Quintana-Ortí. 2020. Acceleration of PageRank with customized precision based on mantissa segmentation. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–19.

[29] Baofu Huang, Zhidan Liu, and Kaishun Wu. 2020. Accelerating PageRank in shared-memory for efficient social network graph analytics. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 238–247.

[30] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. 2004. Adaptive methods for the computation of PageRank. *Linear Algebra Appl.* 386 (2004), 51–65.

[31] Seunghwa Kang, Alex Fender, Joe Eaton, and Brad Rees. 2020. Computing pagerank scores of web crawl data using dgx a100 clusters. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–4.

[32] Kyung Soo Kim and Yong Suk Choi. 2015. Incremental iteration method for fast pagerank computation. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*. 1–5.

[33] Y. Kim, H. Kim, C. Shin, K. Lee, C. Choi, and W. Cho. 2015. Analysis on the transportation point in cheongju city using pagerank algorithm. In *Proceedings of the International Conference on Big Data Applications and Services - BigDAS '15*, C. Leung and A. Nasridinov (Eds.). ACM Press, New York, New York, USA, 165–169.

[34] Tamara G Kolda and Michael J Procopio. 2009. Generalized badrank with graduated trust. *Sandia National Laboratories, California* (2009).

[35] S. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. Davis, M. Henderson, Y. Hu, and R. Sandstrom. 2019. The SuiteSparse matrix collection website interface. *The Journal of Open Source Software* 4, 35 (Mar 2019), 1244.

[36] K. Lakhotia, R. Kannan, and V. Prasanna. 2018. Accelerating PageRank using Partition-Centric Processing. In *USENIX Annual Technical Conference (USENIX ATC '18)*. USENIX Association, Boston, MA, 427–440.

[37] A.N. Langville and C.D. Meyer. 2006. A reordering for the PageRank problem. *SIAM SISC* 27, 6 (2006), 2112–2120.

[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[39] John Nickolls and William J Dally. 2010. The GPU computing era. *IEEE micro* 30, 2 (2010), 56–69.

[40] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[41] NVIDIA Corporation. 2019. nvGRAPH Library User's Guide. https://docs.nvidia.com/cuda/archive/10.1/pdf/nvGRAPH_Library.pdf

[42] Naoto Ohsaka, Takanori Maehara, and Ken-ichi Kawarabayashi. 2015. Efficient pagerank tracking in evolving networks. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 875–884.

[43] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[44] R.P. Pashikanti and S. Kundu. 2022. FPPR: fast pessimistic (dynamic) PageRank to update PageRank in evolving directed graphs on network changes. *SNAM* 12, 1 (2022), 141.

[45] Diego Piccinotti, Edoardo Ramalli, Alberto Parravicini, Rolando Brondolin, and Marco Santambrogio. 2019. Solving write conflicts in GPU-accelerated graph computation: A PageRank case-study. In *2019 IEEE 5th International forum on Research and Technology for Society and Industry (RTSI)*. IEEE, 144–148.

[46] S.J. Plimpton and K.D. Devine. 2011. MapReduce in MPI for large-scale graph algorithms. *Parallel Comput.* 37, 9 (2011), 610–632.

[47] G. Ramalingam. 1996. Bounded Incremental Computation. *Lecture Notes in Computer Science* 1089 (1996), 101–129.

[48] A. Rungsawang and B. Manaskasemsak. 2012. Fast pagerank computation on a GPU cluster. In *20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 450–456.

[49] Subhajit Sahu. 2024. DF* PageRank: Improved Incrementally Expanding Approaches for Updating PageRank on Dynamic Graphs. *arXiv preprint arXiv:2401.15870* (2024).

[50] Subhajit Sahu. 2024. An Incrementally Expanding Approach for Updating PageRank on Dynamic Graphs. *arXiv preprint arXiv:2401.03256* (2024).

[51] Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. 2022. Dynamic Batch Parallel Algorithms for Updating PageRank. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1129–1138.

[52] Scott Sallinen, Juntong Luo, and Matei Ripeanu. 2023. Real-Time PageRank on Dynamic Graphs. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 239–251.

[53] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional.

[54] Reginald Sawilla. 2006. Abstracting PageRank to dynamic asset valuation. *DRDC Ottawa TM* 243 (2006).

[55] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[56] Merijn Verstraaten, Ana Lucia Varbanescu, and Cees de Laat. 2015. Quantifying the performance impact of graph structure on neighbour iteration strategies for pagerank. In *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers 21*. Springer, 528–540.

[57] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 2 (2021), 1–25.

[58] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[59] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. 2010. Efficient pagerank and spmv computation on AMD gpus. In *39th International Conference on Parallel Processing*. IEEE, 81–89.

[60] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.

[61] Zexing Zhan, Ruimin Hu, Xiyue Gao, and Nian Huai. 2019. Fast incremental pagerank on dynamic networks. In *International Conference on Web Engineering*. Springer, 154–168.

[62] Q. Zhang and T. Yuan. 2018. Analysis of China's Urban Network Structure from the Perspective of "Streaming". In *26th International Conference on Geoinformatics*. IEEE, 1–7.

[63] T. Zhang. 2017. Efficient incremental pagerank of evolving graphs on GPU. In *IEEE ICCSEC*. 1232–1236.

A APPENDIX

A.1 Updating ranks of vertices in parallel

Algorithm 3 provides a psuedocode for updating the ranks of vertices in parallel. Here, the function updateRanks() takes as input the set of affected vertices δ_V, affected neighbor vertices δ_N, the previous and current rank vectors R and R_{new}, respectively, the current input graph G^t, partitioned vertex IDs P' (low in-degree vertices come first), and the number of vertices with low in-degree N'_P. It also requires parameters such as the damping factor α, frontier tolerance τ_f, and prune tolerance τ_p.

The algorithm operates in two phases: a thread-per-vertex kernel (for low degree vertices) and a block-per-vertex kernel (for high degree vertices). In the *thread-per-vertex kernel* (lines 3-22), we use each thread to process each low degree vertex in parallel, iterating over the partitioned vertex IDs (P'). For each vertex v, if it's not marked as affected, we skip it (line 5). Otherwise, we compute the new rank r based on the incoming edges and ranks of neighboring vertices (lines 6-13). Depending on whether DF or DF-P PageRank is selected, we compute ranks using either Equation 1 or 2, respectively. Next, we compute the change in rank Δr of the current vertex v from its previous iteration (line 14). If the relative change in rank of v, i.e., $\Delta r / \max(r, R[v])$, is within the prune tolerance τ_p, we perform pruning by marking v and no longer affected. However, if the relative change in rank of v is above the frontier tolerance τ_f, we mark the vertices whose neighbors must be incrementally marked as affected (the incremental marking of affected vertices is performed at a later point of time, using the expandAffected() function (Algorithm 5). Finally, we update the rank of vertex v in the R vector. In the *block-per-vertex kernel* (lines 25-26), we use each thread block to process each high degree vertex in parallel, utilizing block-level parallelism. It involves similar operations as the thread-per-vertex kernel, but uses block-reduce operations and shared memory.

A.2 Parallel vertex partitioning by degree

Algorithm 4 outlines the psuedocode for parallel vertex partitioning by degree. It aims to split the vertices of the input graph $G(V, E)$ into two groups based on their degree: low-degree vertices and high-degree vertices. The algorithm provides partitioned vertex IDs P with low-degree vertices being listed first, along with the number of vertices with low degree N_P, as its output.

In the function partition(), we start by initializing an empty buffer B_k and the set of partitioned vertex IDs P (line 2). We then proceed to populate B_k with boolean values indicating whether each vertex has a degree less than or equal to a specified threshold D_P (lines 4-5). Afterward, we perform an exclusive prefix sum operation on B_k to determine the number of low-degree vertices N_P (lines 6-7). In the subsequent loop, we assign low-degree vertices to the appropriate positions in the partitioned vertex IDs array P (lines 8-9). We then repeat a similar process for high-degree vertices. We populate B_k with boolean values indicating whether each vertex has a degree greater than D_P (lines 11-12), perform another exclusive prefix sum operation on B_k (line 13), and assign high-degree vertices to the appropriate positions in P (lines 14-15). Finally, we return the partitioned vertex IDs P along with the number of low-degree vertices N_P (line 16).

Algorithm 3 Updating ranks of vertices in parallel.

▷ $G^t(V^t, E^t)$: Current input graph
▷ R, R_{new}: Rank vector in the previous, current iteration
▷ δ_V, δ_N: Is a vertex, or neighbors of a vertex affected
▷ P': Partitioned vertex IDs — low in-degree first
▷ N'_P: Number of vertices with low in-degree
□ Δr: Change in rank of a vertex
□ τ_f, τ_p: Frontier, prune tolerance
□ α: Damping factor

1: **function** UPDATERANKS($\delta_V, \delta_N, R_{new}, R, G^t, P', N'_P$)
2:     ▷ A thread-per-vertex kernel
3:     **for all** $i \in [0, N'_P)$ **in parallel**$_{thread/V}$ **do**
4:         $v \leftarrow P'[i]$
5:         **if not** $\delta_V[v]$ **then continue**
6:         $c \leftarrow 0 \; ; d \leftarrow |G^t.out(v)|$
7:         $C_0 \leftarrow (1 - \alpha)/|V^t|$
8:         **for all** $u \in G^t.in(v)$ **do**
9:             $c \leftarrow c + R[u]/|G^t.out(u)|$
10:       **if is** *DF-P* **then**
11:          $r \leftarrow 1/(1 - \alpha/d) * (C_0 + \alpha * (c - R[v]/d))$
12:       **else**
13:          $r \leftarrow C_0 + \alpha * c$
14:       $\Delta r \leftarrow |r - R[v]|$
15:       ▷ Prune v if its relative rank change is small
16:       **if is** *DF-P* **and** $\Delta r / \max(r, R[v]) \leq \tau_p$ **then**
17:          $\delta_V[v] \leftarrow 0$
18:       ▷ Expand frontier if relative rank change is large
19:       **if** $\Delta r / \max(r, R[v]) > \tau_f$ **then**
20:          $\delta_N[v] \leftarrow 0$
21:       ▷ Update rank of v
22:       $R[v] \leftarrow r$
23:     ▷ Similarly with block-per-vertex kernel
24:     ▷ (using block-reduce, requires shared memory)
25:     **for all** $i \in [N'_P, |V^t|)$ **in parallel**$_{block/V}$ **do**
26:         $\cdots$

A.3 Parallel marking of affected vertices

Algorithm 5 presents the psuedocode for parallel marking of affected vertices, consisting of two functions: initialAffected() and expandAffected().

The initialAffected() function performs an initial marking step of DF and DF-P PageRank. It takes as input the current graph snapshot G^t and the sets of edge deletions Δ^{t-} and insertions Δ^{t+}. Here, we first initialize two arrays, δ_V and δ_N, which represent whether each vertex and its neighbors are affected, respectively (line 2). Next, for each edge deletion in Δ^{t-}, we mark both the source and target vertices as affected (lines 4-8). Similarly, for each edge insertion in Δ^{t+}, we mark the source vertex as affected (lines 10-12). Finally, we return δ_V and δ_N (line 13).

The expandAffected() function propagates the affected status to neighboring vertices. It takes as input flags indicating whether each vertex is affected δ_V or its neighbors are affected δ_N, the current graph snapshot G^t, partitioned vertex IDs P with low degree

Algorithm 4 Parallel vertex partitioning by degree.

▷ $G(V, E)$: Input graph
□ D_P: Maximum degree of a low-degree vertex
□ B_k: Temporary buffer used for partitioning
□ P: Partitioned vertex IDs — low degree vertices first
□ N_P: Number of vertices with low degree

1: **function** PARTITION(G)
2:     $P \leftarrow B_k \leftarrow \{\}$
3:     ▷ Populate vertex IDs with degree $\leq D_P$
4:     **for all** $v \in V$ **in parallel do**
5:         $B_k[v] \leftarrow |G.out(v)| \leq D_P$
6:     $B_k[|V|] \leftarrow 0$
7:     $exclusiveScan(B_k)$; $N_P \leftarrow B_k[|V|]$
8:     **for all** $v \in V$ **in parallel do**
9:         **if** $|G.out(v)| \leq D_P$ **then** $P[B_k[v]] \leftarrow v$
10:     ▷ Populate vertex IDs with degree $> D_P$
11:     **for all** $v \in V$ **in parallel do**
12:         $B_k[v] \leftarrow |G.out(v)| > D_P$
13:     $exclusiveScan(B_k)$
14:     **for all** $v \in V$ **in parallel do**
15:         **if** $|G.out(v)| > D_P$ **then** $P[N_P + B_k[v]] \leftarrow v$
16:     **return** $\{P, N_P\}$

Algorithm 5 Parallel marking of affected vertices.

▷ $G^t(V^t, E^t)$: Current input graph
▷ Δ^{t-}, Δ^{t+}: Edge deletions and insertions (input)
▷ δ_V, δ_N: Is a vertex, or neighbors of a vertex affected
▷ P: Partitioned vertex IDs (low-degree vertices first)
▷ N_P: Number of vertices with low degree

1: **function** INITIALAFFECTED($G^t, \Delta^{t-}, \Delta^{t+}$)
2:     $N_D \leftarrow |\Delta^{t-}|$; $N_I \leftarrow |\Delta^{t+}|$
3:     ▷ For edge deletions
4:     **for all** $i \in [0, N_D)$ **in parallel do**
5:         $u \leftarrow \Delta^{t-}[i].source$
6:         $v \leftarrow \Delta^{t-}[i].target$
7:         $\delta_N[u] \leftarrow 1$
8:         $\delta_V[v] \leftarrow 1$
9:     ▷ For edge insertions
10:     **for all** $i \in [0, N_I)$ **in parallel do**
11:         $u \leftarrow \Delta^{t+}[i].source$
12:         $\delta_N[u] \leftarrow 1$
13:     **return** $\{\delta_V, \delta_N\}$

14: **function** EXPANDAFFECTED($\delta_V, \delta_N, G^t, P, N_P$)
15:     ▷ A thread-per-vertex kernel
16:     **for all** $i \in [0, N_P)$ **in parallel**$_{thread/V}$ **do**
17:         $u \leftarrow P[i]$
18:         **if** $\delta_N[u]$ **then** $markNeighbors(\delta_V, G^t, u)$
19:     ▷ Similarly with block-per-vertex kernel
20:     **for all** $i \in [N_P, |V^t|)$ **in parallel**$_{block/V}$ **do**
21:         $\cdots$

vertices placed first, and the number of vertices with low degree N_P. This algorithm also operates in two phases: a thread-per-vertex kernel (for low degree vertices) and a block-per-vertex kernel (for high degree vertices). In the *thread-per-vertex kernel* (lines 16-18), we use each thread to process each low degree vertex in parallel by iterating through the partitioned vertex IDs array P. For each vertex u marked as affected in δ_N, we invoke the markNeighbors() function to mark its neighbors as affected in δ_V (line 18). In the *block-per-vertex kernel* (lines 20-21), we use each thread block to process each vertex in parallel, utilizing block-level parallelism. It involves similar operations as the thread-per-vertex kernel.

A.4 Indirect Comparison with State-of-the-art PageRank Implementations (Static)

We now indirectly compare the performance of our GPU implementation of Static PageRank with other similar state-of-the-art implementations, listed in Table 1. Wang et al. [57] present Grus, a unified memory efficient high-performance graph processing framework for GPUs. They achieve a 1.2× speedup over Gunrock [58] on the *uk-2005* graph (refer to Table 4 in their paper [57]). On the same graph, Wang et al. observe that Gunrock achieves a speedup of 4.6× over Tigr [40]. However, our implementation achieves an 8.6× speedup over Gunrock on the same graph (Figure 2(b)). Chen et al. [10] introduce Atos, a task-parallel GPU scheduler for graph analytics. They achieve a 3.2× speedup over Gunrock on the *indochina-2004* graph (refer to Table 1 in their paper [10]). However, our implementation achieves a 24.4× speedup over Gunrock on the same graph (Figure 2(b)). In a subsequent work, Chen et al. [11] extend their Atos dynamic scheduling framework to multi-node GPU systems supporting Partitioned Global Address Space

(PGAS) style lightweight one-sided memory operations within and between nodes. Even with 4 GPUs, they fail to surpass our speedup with respect to Gunrock on the *indochina-2004* graph (refer to Table 4 in their paper [11]). On the same graph, Chen et al. observe that they achieve a speedup of 57.5× over Galois with Gluon [17] on a single GPU (see Table 5 in their paper [11]). Yang et al. [60] introduce GraphBLAST, a high-performance linear algebra-based graph framework for GPUs. On the *indochina-2004* graph, they achieve a 2.2×/1.2× speedup over Gunrock (see Table 12/13 in their paper [60]). Nonetheless, our implementation achieves a 24.4× speedup over Gunrock on the same graph (Figure 2(b)). Concessao et al. [16] propose Meerkat, a library-based framework for dynamic graph algorithms that utilizes a GPU-tailored graph representation and exploits the warp-cooperative execution model. The PageRank implementation of Meerkat performs, on average, 1.7× faster than Hornet [8]. However, our Static PageRank outperforms Hornet by an average speedup of 31× (Figure 2(b)).
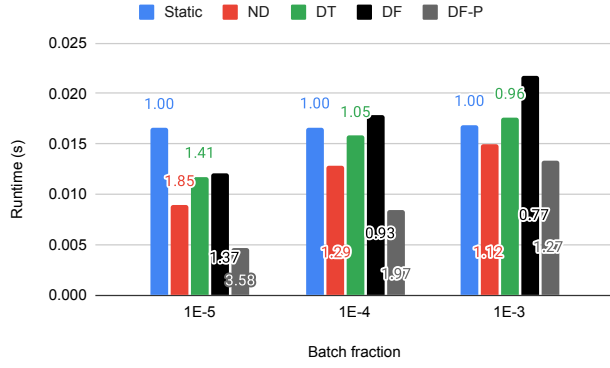
(a) Overall Runtime **(GPU)**
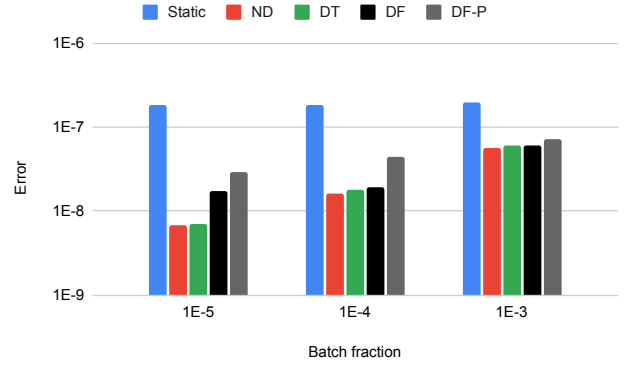
(b) Overall Error in ranks obtained **(GPU)**
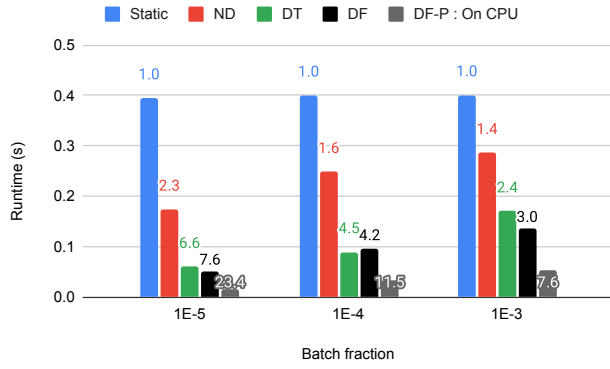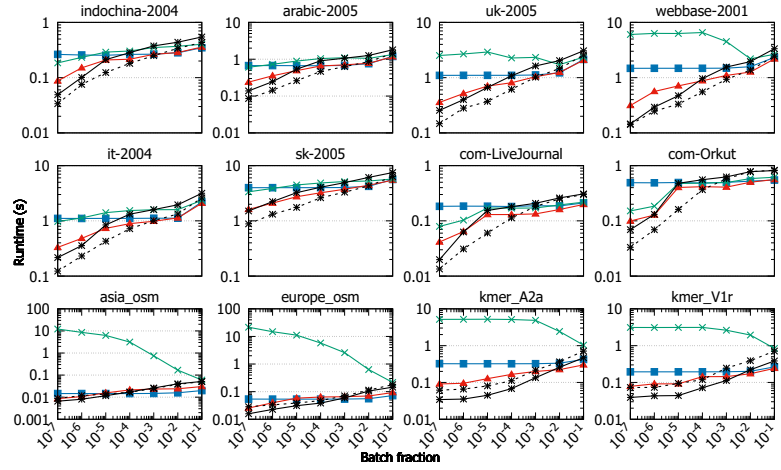

(c) Overall Runtime **(CPU)**
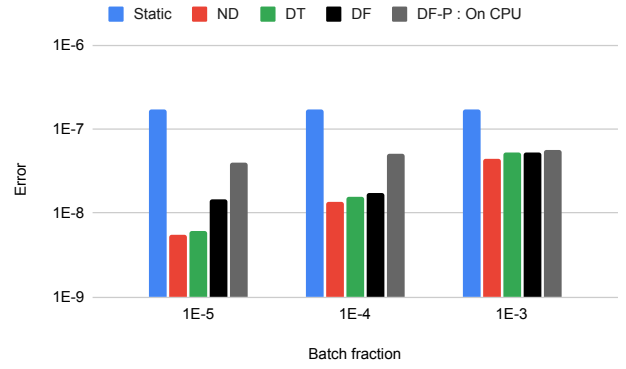
(d) Overall Error in ranks obtained **(CPU)**
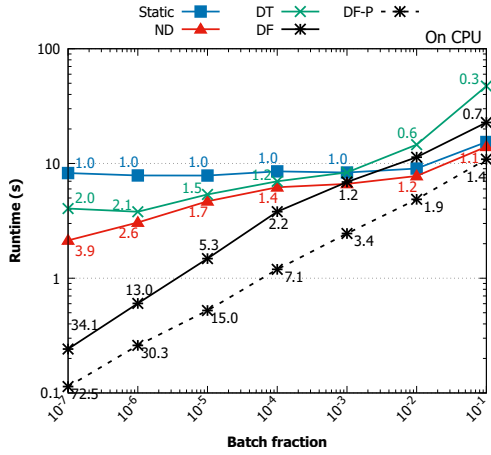
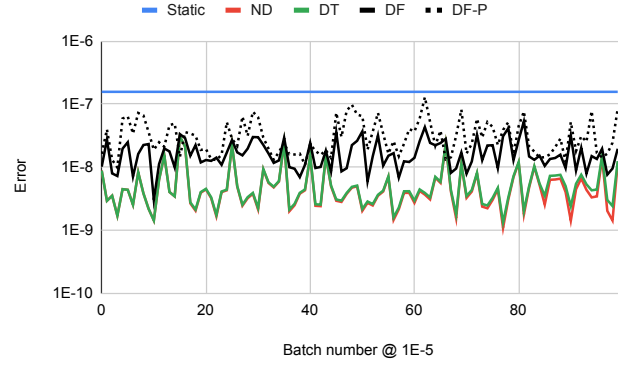**Figure 6: Mean Runtime and Error in ranks obtained with our GPU / multicore CPU implementation [49] of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on real-world dynamic graphs, with batch updates of size $10^{-5}|E_T|$ to $10^{-3}|E_T|$. Here, (a) and (b) show the overall runtime and error (relative to reference Static PageRank, see Section 5.1.5) of the approaches on the GPU, while (c) and (d) show the overall runtime and error of the approaches on the CPU, across all temporal graphs. In (a) and (c), the speedup of each approach with respect to Static PageRank is labeled.**
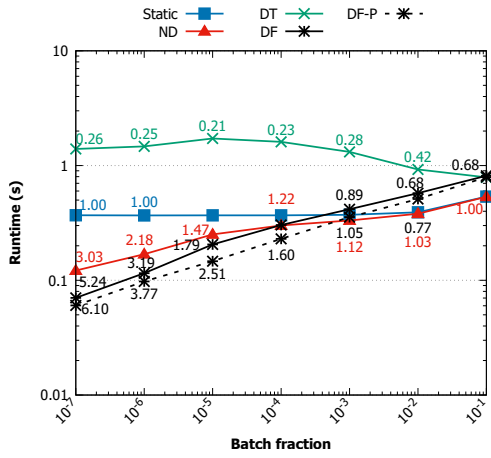
(a) Overall result **(GPU)**

(b) Results on each graph **(GPU)**

(c) Overall result **(CPU)**
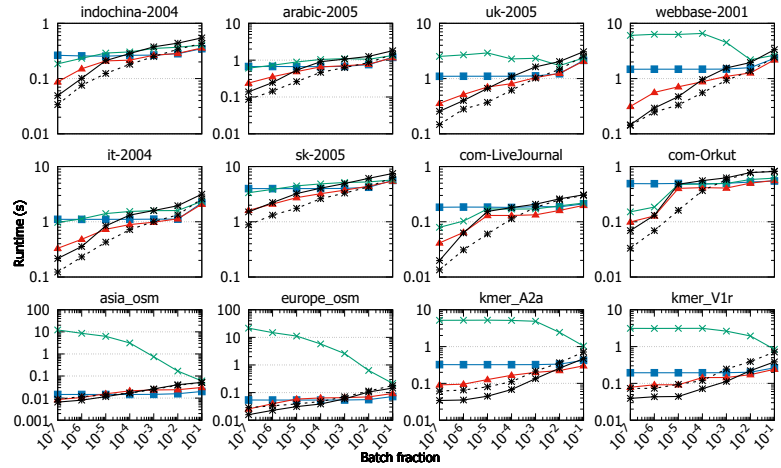
(d) Results on each graph **(CPU)**

Figure 7: Runtime (logarithmic scale) of our GPU implementation / multicore CPU implementation [49] of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF)*, and *Dynamic Frontier with Pruning (DF-P)* PageRank on large (static) graphs with generated random batch updates. Batch updates range in size from $10^{-7}|E|$ to $0.1|E|$ in multiples of 10. These updates consist of 80% edge insertions and 20% edge deletions, mimicking realistic changes in a dynamic graph scenario. The right subfigures illustrate the runtime of each approach for individual graphs in the dataset, while the left subfigures present overall runtimes (using geometric mean for consistent scaling across graphs). Additionally, the speedup of each approach relative to Static PageRank is labeled on respective lines.
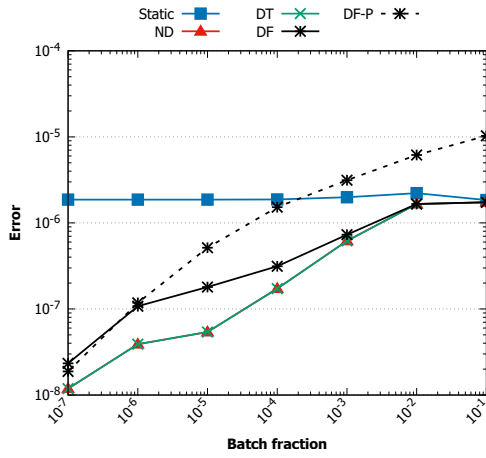
(a) Overall result **(GPU)**
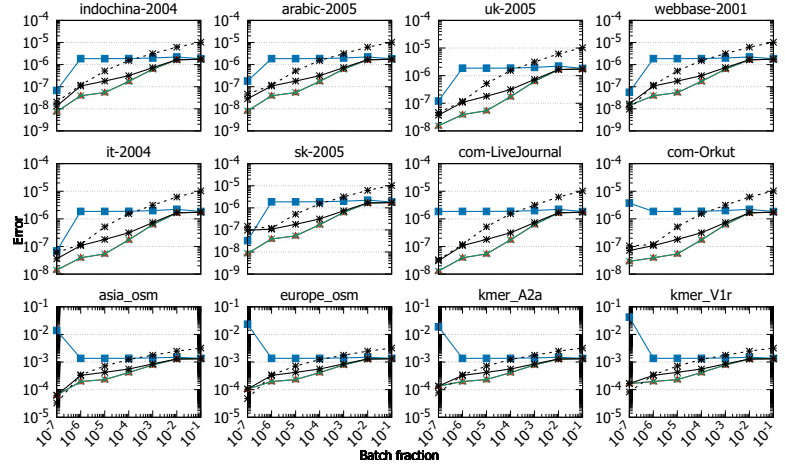
(b) Results on each graph **(GPU)**
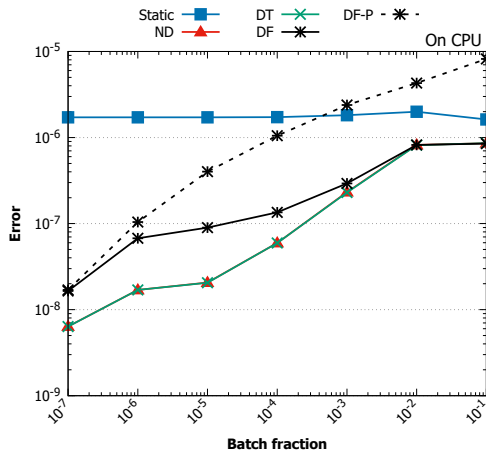
(c) Overall result **(CPU)**
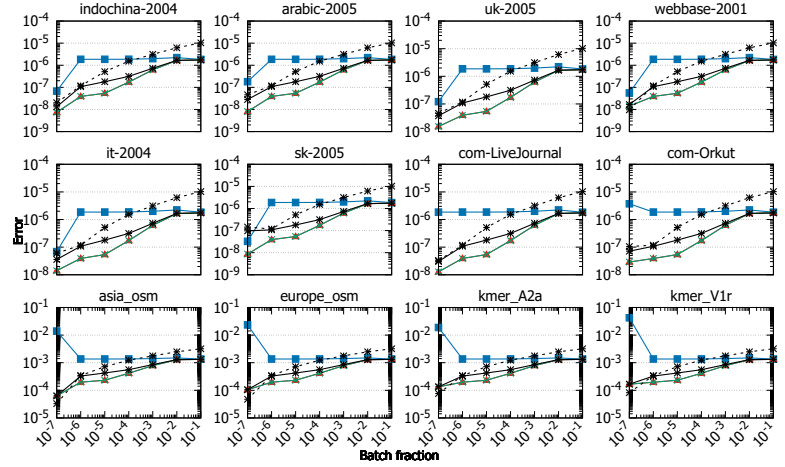
(d) Results on each graph **(CPU)**

Figure 8: Error comparison our GPU implementation / multicore CPU implementation [49] of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on large (static) graphs with generated random batch updates, relative to a Reference Static PageRank (see Section 5.1.5), using $L1$-norm. The size of batch updates range from $10^{-7}|E|$ to $0.1|E|$ in multiples of 10 (logarithmic scale), consisting of 80% edge insertions and 20% edge deletions to simulate realistic dynamic graph updates. The right subfigures depict the error for each approach in relation to each graph, while the left subfigures showcase overall errors using geometric mean for consistent scaling across graphs.
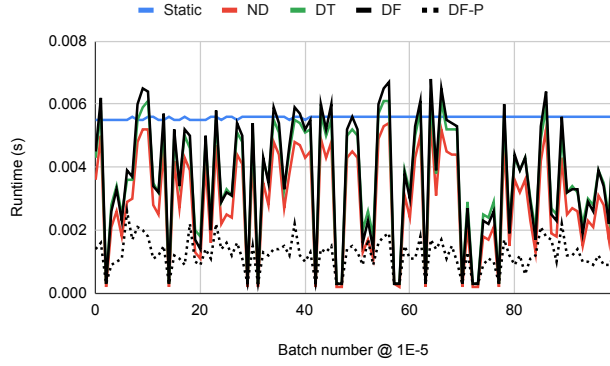
(a) Runtime on consecutive batch updates of size $10^{-5}|E_T|$
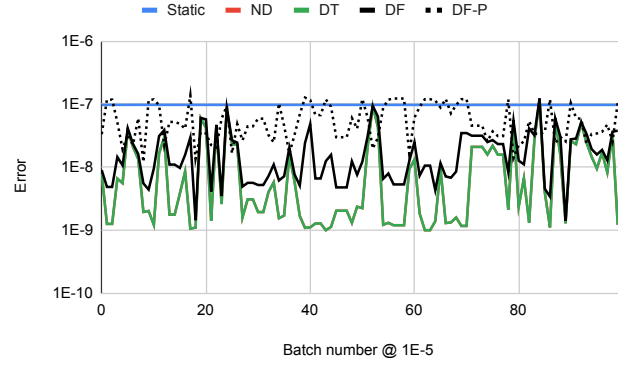
(b) Error in ranks obtained on consecutive batch updates of size $10^{-5}|E_T|$
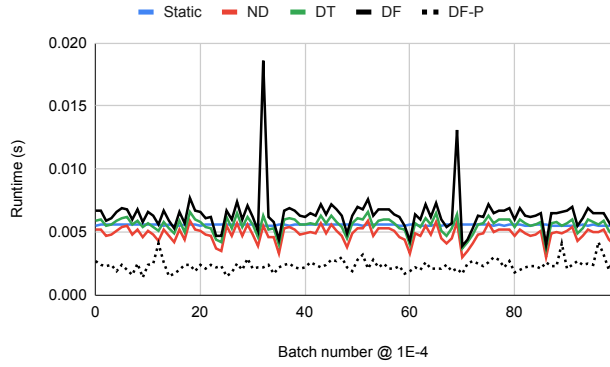
(c) Runtime on consecutive batch updates of size $10^{-4}|E_T|$
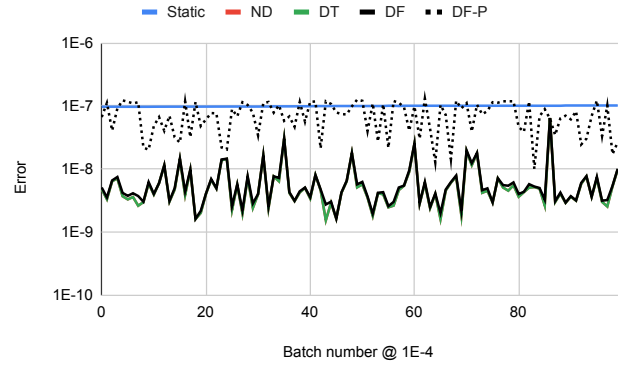
(d) Error in ranks obtained on consecutive batch updates of size $10^{-4}|E_T|$
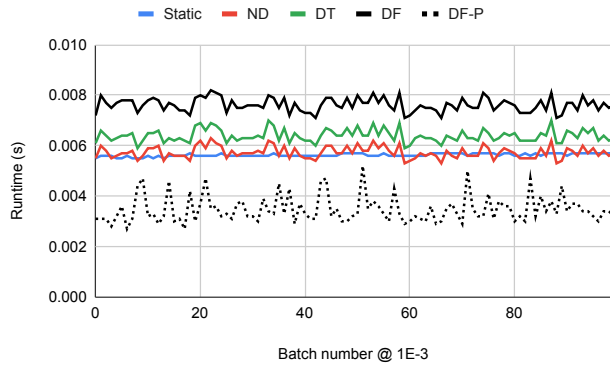
(e) Runtime on consecutive batch updates of size $10^{-3}|E_T|$
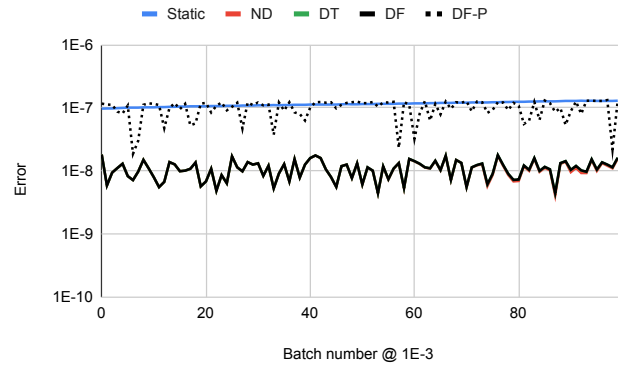
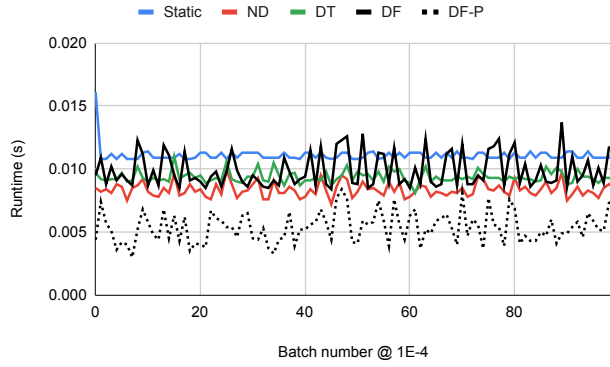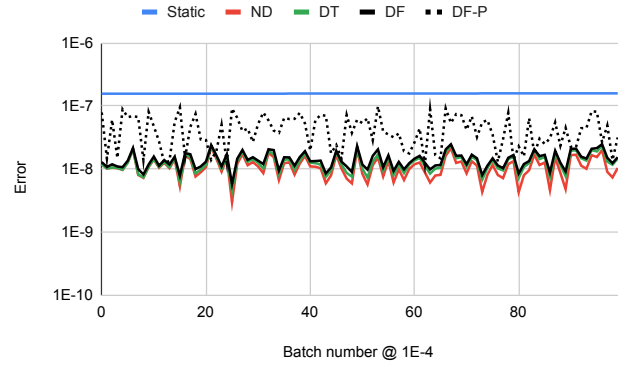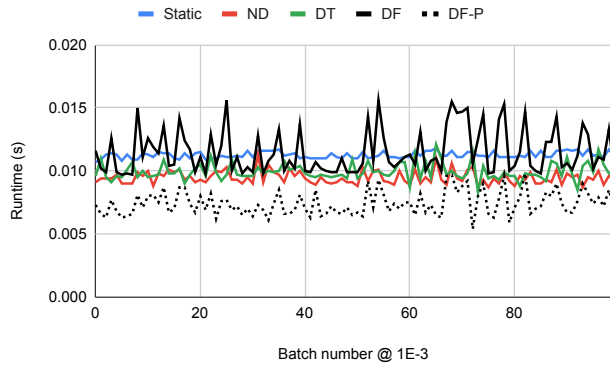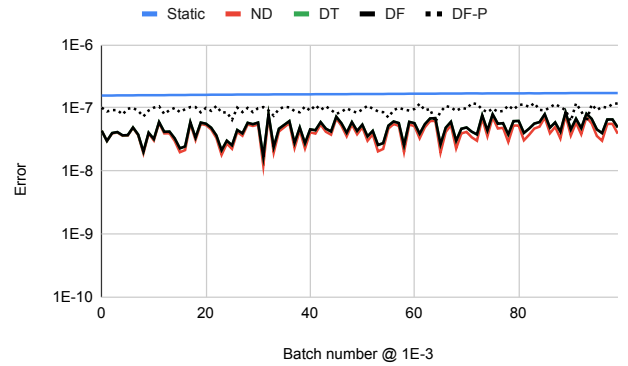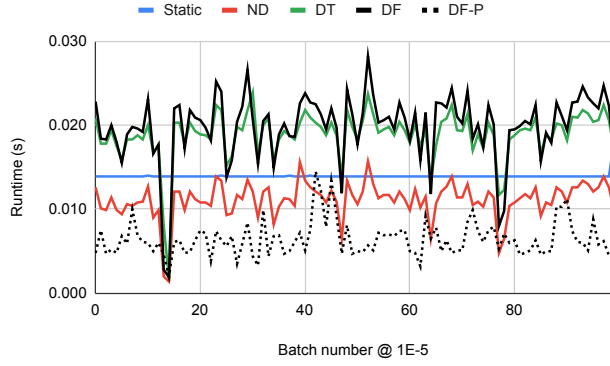(f) Error in ranks obtained on consecutive batch updates of size $10^{-3}|E_T|$

Figure 9: Runtime and Error in ranks obtained with our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on the *sx-mathoverflow* dynamic graph. The size of batch updates range from $10^{-5}|E_T|$ to $10^{-3}|E_T|$. The rank error with each approach is measured relative to ranks obtained with a reference Static PageRank run, as detailed in Section 5.1.5.
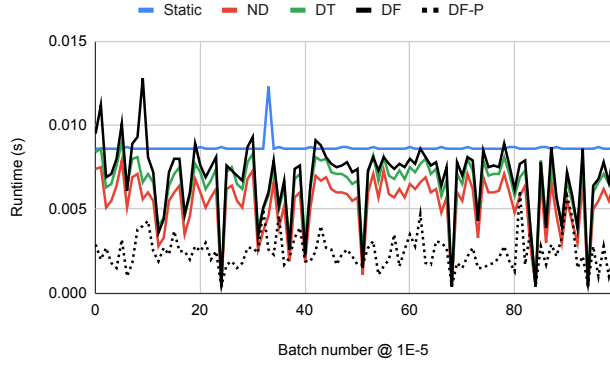
(a) Runtime on consecutive batch updates of size $10^{-5}|E_T|$
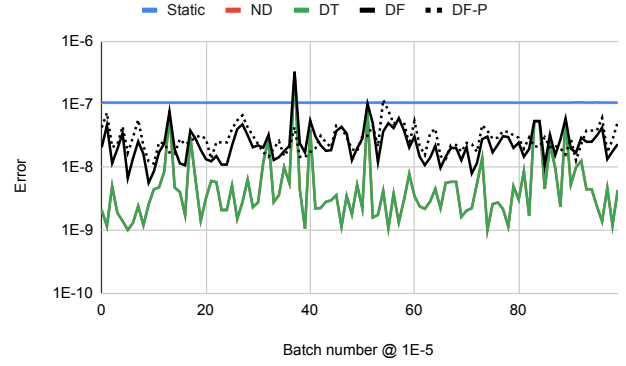
(b) Error in ranks obtained on consecutive batch updates of size $10^{-5}|E_T|$
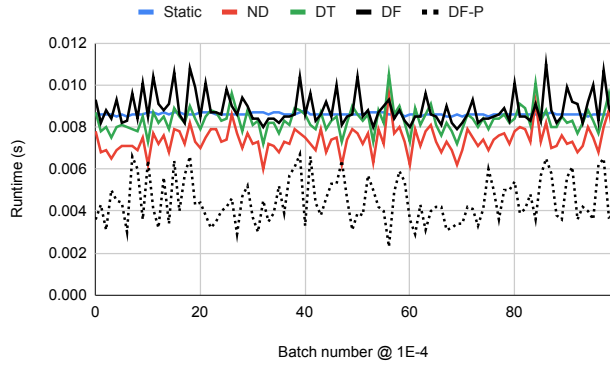
(c) Runtime on consecutive batch updates of size $10^{-4}|E_T|$

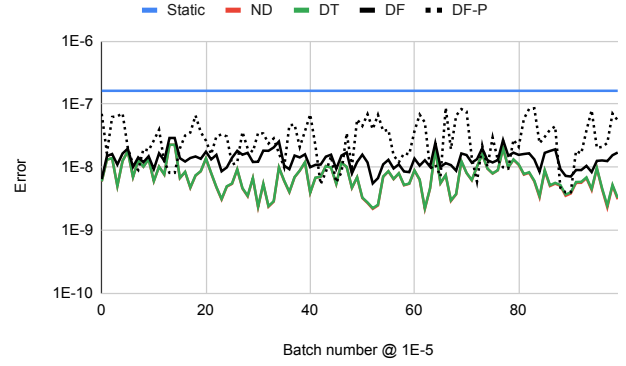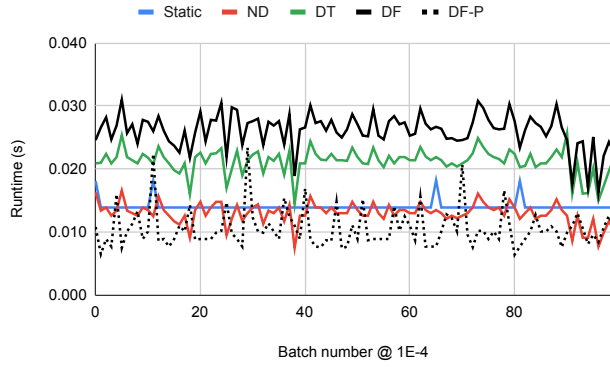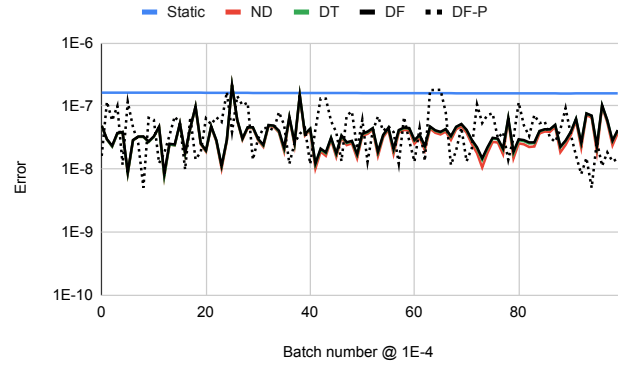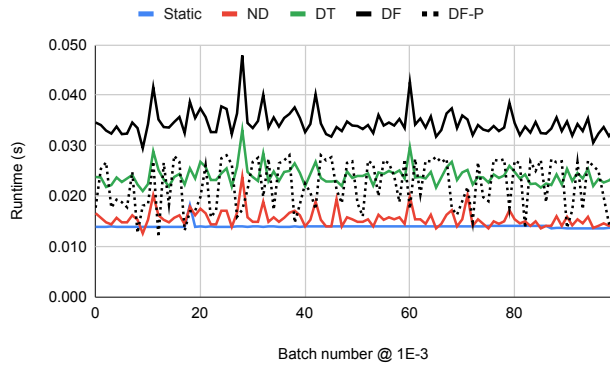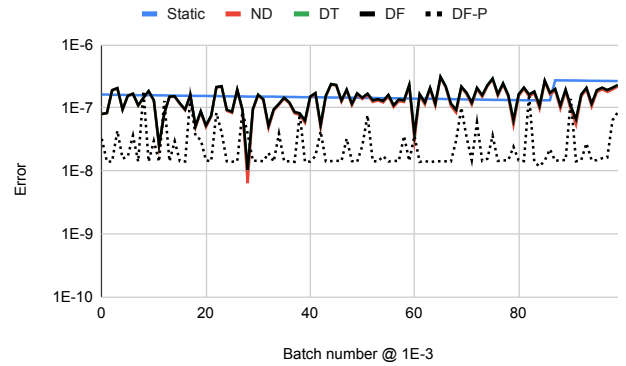(d) Error in ranks obtained on consecutive batch updates of size $10^{-4}|E_T|$

(e) Runtime on consecutive batch updates of size $10^{-3}|E_T|$
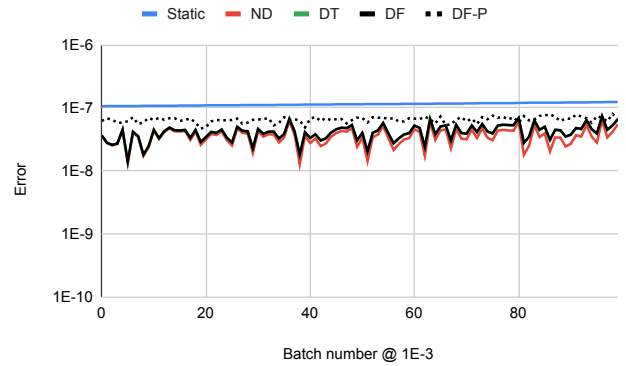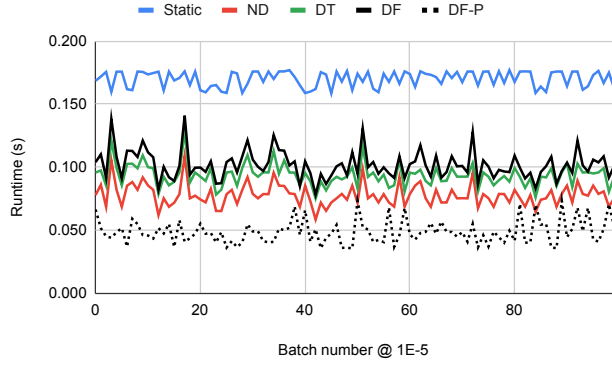
(f) Error in ranks obtained on consecutive batch updates of size $10^{-3}|E_T|$

Figure 10: Runtime and Error in ranks obtained with our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on the *sx-askubuntu* dynamic graph. The size of batch updates range from $10^{-5}|E_T|$ to $10^{-3}|E_T|$. The rank error with each approach is measured relative to ranks obtained with a reference Static PageRank run, as detailed in Section 5.1.5.
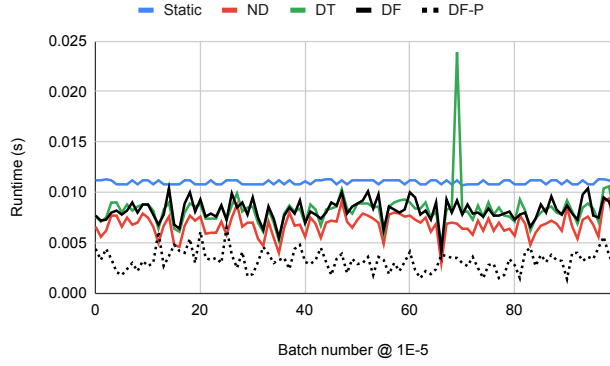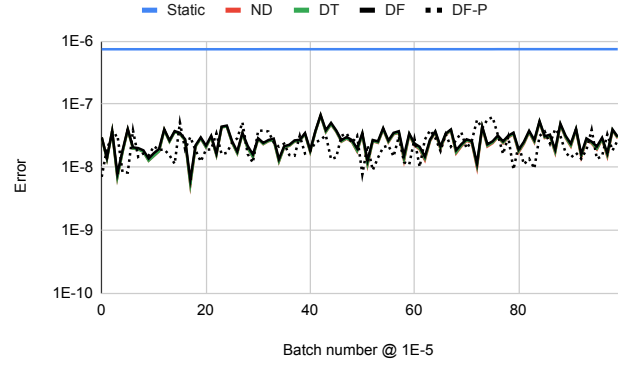
(a) Runtime on consecutive batch updates of size $10^{-5}|E_T|$

(b) Error in ranks obtained on consecutive batch updates of size $10^{-5}|E_T|$

(c) Runtime on consecutive batch updates of size $10^{-4}|E_T|$

(d) Error in ranks obtained on consecutive batch updates of size $10^{-4}|E_T|$

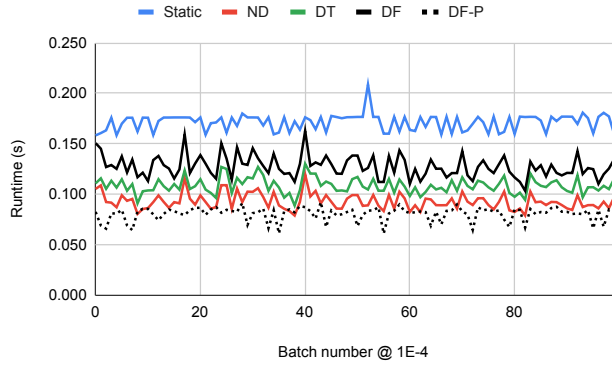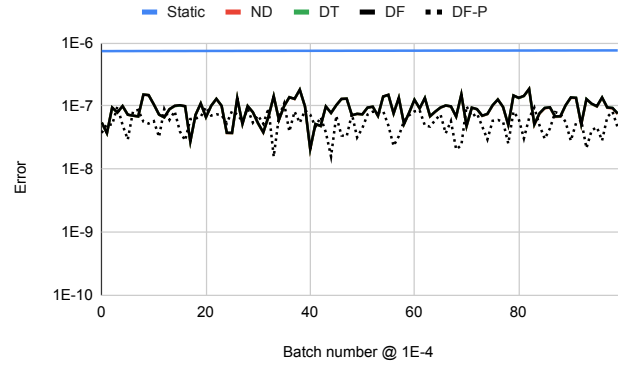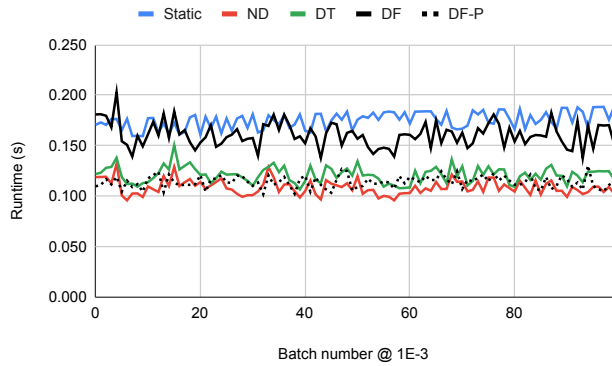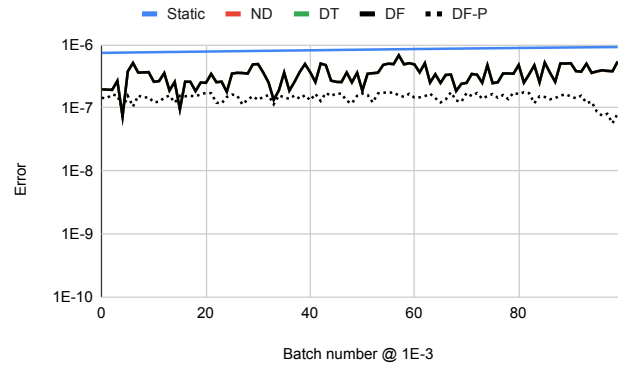(e) Runtime on consecutive batch updates of size $10^{-3}|E_T|$

(f) Error in ranks obtained on consecutive batch updates of size $10^{-3}|E_T|$

Figure 11: Runtime and Error in ranks obtained with our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on the *sx-superuser* dynamic graph. The size of batch updates range from $10^{-5}|E_T|$ to $10^{-3}|E_T|$. The rank error with each approach is measured relative to ranks obtained with a reference Static PageRank run, as detailed in Section 5.1.5.

(a) Runtime on consecutive batch updates of size $10^{-5}|E_T|$

(b) Error in ranks obtained on consecutive batch updates of size $10^{-5}|E_T|$

(c) Runtime on consecutive batch updates of size $10^{-4}|E_T|$

(d) Error in ranks obtained on consecutive batch updates of size $10^{-4}|E_T|$

(e) Runtime on consecutive batch updates of size $10^{-3}|E_T|$

(f) Error in ranks obtained on consecutive batch updates of size $10^{-3}|E_T|$

Figure 12: Runtime and Error in ranks obtained with our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on the *wiki-talk-temporal* dynamic graph. The size of batch updates range from $10^{-5}|E_T|$ to $10^{-3}|E_T|$. The rank error with each approach is measured relative to ranks obtained with a reference Static PageRank run, as detailed in Section 5.1.5.

(a) Runtime on consecutive batch updates of size $10^{-5}|E_T|$

(b) Error in ranks obtained on consecutive batch updates of size $10^{-5}|E_T|$

(c) Runtime on consecutive batch updates of size $10^{-4}|E_T|$

(d) Error in ranks obtained on consecutive batch updates of size $10^{-4}|E_T|$

(e) Runtime on consecutive batch updates of size $10^{-3}|E_T|$

(f) Error in ranks obtained on consecutive batch updates of size $10^{-3}|E_T|$

Figure 13: Runtime and Error in ranks obtained with our GPU implementation of *Static, Naive-dynamic (ND), Dynamic Traversal (DT), Dynamic Frontier (DF),* and *Dynamic Frontier with Pruning (DF-P)* PageRank on the *sx-stackoverflow* dynamic graph. The size of batch updates range from $10^{-5}|E_T|$ to $10^{-3}|E_T|$. The rank error with each approach is measured relative to ranks obtained with a reference Static PageRank run, as detailed in Section 5.1.5.