DiggerBees: Depth First Search Leveraging Hierarchical Block-Level Stealing on GPUs

Yuyao Niu
Barcelona Supercomputing Center
Barcelona, Spain
Universitat Politècnica de Catalunya
Barcelona, Spain
yuyao.niu@bsc.es

Yuechen Lu
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
yuechen.lu@cup.edu.cn

Weifeng Liu
SSSLab, Dept. of CST
China University of Petroleum-Beijing
Beijing, China
weifeng.liu@cup.edu.cn

Marc Casas
Barcelona Supercomputing Center
Barcelona, Spain
Universitat Politècnica de Catalunya
Barcelona, Spain
marc.casas@bsc.es

Abstract

Depth First Search (DFS) is a fundamental graph traversal algorithm with broad applications. While existing work-stealing DFS approaches achieve strong performance on CPUs, mapping them to modern GPUs faces three major challenges: (1) limited shared memory cannot accommodate deep stacks, (2) frequent stack operations hinder efficient intra-block execution, and (3) irregular workloads complicate scalable inter-block execution.

In this paper, we propose DiggerBees, a GPU-optimized parallel DFS algorithm with hierarchical block-level stealing, consisting of three components. First, we introduce a two-level stack structure to mitigate shared memory limitations. Second, we employ warp-level DFS with intra-block work stealing to enable efficient execution within a block. Third, we implement inter-block work stealing to achieve scalable execution across blocks and sustain high parallelism. Experimental results on the latest NVIDIA GPUs show that DiggerBees outperforms existing DFS approaches, CKL-PDFS, ACR-PDFS, and NVG-DFS, achieving average speedups of 1.37×, 1.83×, and 30.18×, respectively. Moreover, DiggerBees even surpasses high-performance GPU BFS implementations on graphs with deep and narrow traversal paths, and scales efficiently across GPU generations.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Computing methodologies → Parallel algorithms**.

Keywords: Depth First Search, work stealing, GPU

ACM Reference Format:
Yuyao Niu, Yuechen Lu, Weifeng Liu, and Marc Casas. 2026. DiggerBees: Depth First Search Leveraging Hierarchical Block-Level Stealing on GPUs . In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '26), January 31 – February 4, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3774934.3786457

1 Introduction

Depth First Search (DFS) traverses a graph by exploring vertices along one branch as deeply as possible before backtracking, generating a valid DFS tree. As a fundamental algorithm in graph theory [45], DFS has a wide range of applications, including structural analysis (*e.g.*, strongly connected components [92]), ordering problems (*e.g.*, topological sorting [48]), and pattern recognition (*e.g.*, subgraph matching [98]).

Despite DFS and Breadth First Search (BFS) being equally essential as the two core graph traversal primitives, research on DFS in modern GPU platforms is far less extensive than that on BFS [35, 59, 61, 65, 70, 96, 99, 101]. One possible reason is that BFS naturally exposes parallelism through its level-synchronous exploration, where vertices at the same level can be processed simultaneously. In contrast, DFS's sequential, stack-based traversal creates dependencies between successive operations, making it difficult to parallelize. This inherent difficulty has led to a trend of "DFS-avoidance", where problems are reformulated to bypass DFS. While these approaches improve parallel scalability, they typically require more complex algorithmic designs [27]. An efficient parallel DFS primitive is therefore essential to reclaim the efficiency and structural insights of DFS-based designs.

Nevertheless, in most application scenarios, DFS does not require a strict lexicographic order of traversal, making unordered DFS a viable and valuable direction for parallelism. This insight has motivated extensive research efforts in parallel DFS. Theoretical analyses have shown that parallel unordered DFS can achieve close-to-linear work efficiency under ideal conditions [3, 37, 54]. Several practical implementations using work stealing techniques [10] have validated these findings, showing significant performance and scalability improvements on CPU-based multiprocessor systems [2, 19, 76]. Nonetheless, these advances largely focus on CPUs or distributed systems, leaving the challenges of efficient DFS on modern GPUs largely unsolved.

Implementing work-stealing DFS on GPUs faces three major challenges: (1) the deep, unpredictable recursion of DFS demands substantial stack space, which often exceeds the limited capacity of a GPU's fast on-chip shared memory, (2) frequent stack operations lead to inefficient execution within a thread block, as thread-private stacks cause warp divergence while shared stacks incur costly atomic operations and synchronization, and (3) irregular workloads from varying subgraph structures hinder scalability across multiple blocks, making it difficult to achieve efficient inter-block execution and full GPU utilization.

To resolve the above three challenges, we in this paper propose DiggerBees, a GPU-optimized DFS algorithm with hierarchical block-level stealing. First, to tackle the memory constraint, we introduce a two-level stack structure that combines a small, fast stack in shared memory with a large stack in global memory, ensuring both efficiency and capacity. Second, to achieve efficient intra-block execution, we employ warps as the basic execution unit, where each warp operates DFS independently, combined with intra-block work stealing that enables idle warps to acquire work from busy peers, ensuring all warps remain active. Third, to enable scalable inter-block execution, we develop inter-block stealing that allows idle blocks to acquire work from heavily loaded blocks, sustaining high and balanced GPU utilization.

We evaluate DiggerBees on 234 graphs from three widely used graph collections in the SuiteSparse Matrix Collection [22] using the latest NVIDIA A100 and H100 GPUs. The results show that DiggerBees significantly outperforms three existing CPU and GPU baselines: CKL-PDFS [19], ACR-PDFS [2], and NVG-DFS [69], achieving on average 1.37×, 1.83×, and 30.18× speedups (up to 6.24×, 12.44×, and 1841.68×), respectively. Compared with state-of-the-art GPU BFS implementations Gunrock [97] and BerryBees [70], DiggerBees delivers competitive or superior performance on representative graphs containing long and narrow traversal paths. Furthermore, our evaluation includes the performance breakdown of DiggerBees to confirm the substantial contributions of each proposed component to the overall speedups.

This work makes the following contributions:

- We identify and analyze three major challenges in implementing work-stealing DFS on GPUs.
- We design a two-level stack structure that maps DFS workloads onto the GPU memory hierarchy.
- We develop a hierarchical work-stealing mechanism tailored specifically for DFS traversal on GPUs.
- We achieve significant performance gains over existing approaches on the latest NVIDIA GPUs.

2 Background and Challenges

2.1 Serial DFS

DFS explores a graph by advancing as far as possible along one branch before backtracking [20]. Given a graph $G = (V, E)$ and a start vertex r, DFS discovers all vertices reachable from r and builds a DFS tree. An ordering can be derived that reflects the sequence in which vertices are visited.

DFS can be implemented using an explicit stack. Algorithm 1 presents a serial stack-based DFS on a graph stored in compressed sparse row (CSR) format. This algorithm outputs two arrays: `visited`, marking explored vertices, and `parent`, recording the DFS tree.

Algorithm 1 A pseudocode of the serial stack-based DFS

```
1: visited[$roort$] ← 1, parent[$root$] ← −1
2: $S$ ← empty stack of $\langle node, next\_idx \rangle$
3: $S$.push($\langle root$, row_ptr[$root$]$\rangle$)
4: while $S \neq \emptyset$ do
5:     $\langle u, i \rangle \leftarrow S$.top()
6:     if $i <$ row_ptr[$u$+1] then
7:         $v \leftarrow$ column_idx[$i$]
8:         $S$.updateTop($\langle u, i+1 \rangle$)
9:         if ¬ visited[$v$] then
10:            visited[$v$] ← true, parent[$v$] ← $u$
11:            $S$.push($\langle v$, row_ptr[$v$]$\rangle$)
12:        end if
13:    else
14:        $S$.pop()
15:    end if
16: end while
```

Serial DFS, with its strong dependencies and enforced lexicographic order, is P-complete [79], making it unlikely to admit parallel solutions. As shown in Figure 1, given input graph (a), it produces the unique lexicographically ordered DFS tree (b) with traversal $a \to b \to d \to e \to c \to f$.

2.2 Parallel DFS

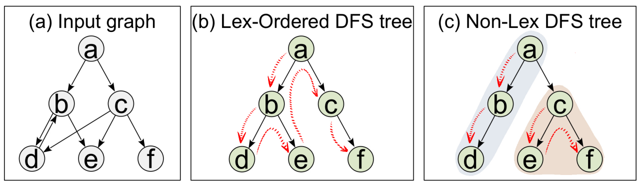

Figure 1. A DFS traversal example on graph (a). Serial DFS produces the lexicographically ordered DFS tree (b), while parallel DFS generates a valid but non-lexicographic tree (c).

Parallel DFS relaxes the constraints and constructs a valid DFS tree without enforcing lexicographic order [4].

Algorithm 2 A pseudocode of the parallel DFS

```
1: $S_i$ ← Local stack of processor $P_i$ ▷ issue #1: Limited SMEM for stacks
2: while not terminated do
3:     while $S_i \neq \emptyset$ do
4:         Execute DFS on $S_i$ ▷ issue #2: Inefficient intra-block execution
5:     end while
6:     Steal work from other processors
                    ▷ issue #3: Poor inter-block scalability and balance
7:     Termination Check
8: end while
```

A common strategy for parallel DFS is **work stealing**. Rao and Kumar [54, 76] were among the first to apply it to CPU-based shared memory multiprocessors. Algorithm 2 illustrates the high-level structure of their implementation.

In this approach, each processor maintains a local stack S_i (line 1 in Algorithm 2), and repeatedly checks: while S_i is not empty, it executes DFS on S_i (lines 3–5). Once S_i becomes empty, it attempts to steal work from neighboring processors (line 6). The algorithm continues until the termination is met, *i.e.*, all processors have empty stacks. Figure 1(c) shows an example of the result of such parallel DFS: one processor traverses $a \rightarrow b \rightarrow d$, while the other explores $c \rightarrow e \rightarrow f$.

2.3 Challenges of parallel DFS on GPUs

While work-stealing DFS achieves good parallelism on CPU-based multiprocessors [2, 19, 54, 76], mapping such a strategy to GPUs is non-trivial. There are three major issues (highlighted in lines 1, 4, and 6 of Algorithm 2, respectively).

The first issue (line 1) is that GPUs provide limited on-chip shared memory (typically tens to a few hundred KB per streaming multiprocessor (SM)). In contrast, DFS may require stacks as deep as the longest path in the graph. For example, road-network graphs contain paths with tens of thousands of vertices, demanding megabytes of stack space. Thus, it is hard to keep the whole stack in shared memory.

The second issue (line 4) is that even when stacks fit in the shared memory of a thread block, DFS performs frequent push and pop operations, which creates problematic control flow: thread-private stacks cause warp divergence as threads follow different execution paths, while stacks shared within a block require costly atomic operations and synchronization. Thus, it is hard to achieve efficient intra-block execution.

The third issue (line 6) is that the GPU's massive parallelism is not fully utilized. To saturate resources, execution must extend from single- to multi-block so that more SMs and blocks become active. However, this requires costly inter-block communication, and irregular DFS workloads further complicate balanced distribution. Thus, it is hard to achieve scalable inter-block execution while ensuring load balance.

## 3 DiggerBees

3.1 Overview

To resolve the above three challenges, we propose DiggerBees, a GPU-optimized DFS algorithm. DiggerBees addresses each challenge through three key components:

(i) Data structure (Section 3.2): To overcome shared memory limits (issue #1), DiggerBees introduces a two-level stack consisting of a fast *HotRing* in shared memory and a large *ColdSeg* in global memory. This structure leverages the GPU memory hierarchy to provide low-latency access for frequent operations and sufficient capacity for deep traversals.

(ii) Intra-block execution (Sections 3.3 and 3.4): To achieve efficient intra-block execution (issue #2), DiggerBees combines two techniques: (1) warp-level workload, where each warp operates DFS independently, eliminating warp divergence and synchronization, and (2) intra-block stealing, which enables idle warps to acquire work from busy peers, ensuring all warps within a block participate actively.

(iii) Inter-block execution (Section 3.5): To achieve scalable multi-block execution while ensuring load balance (issue #3), DiggerBees implements inter-block work stealing. Idle blocks can steal work from heavily loaded blocks, enabling dynamic load redistribution across the GPU. This design sustains high parallelism by keeping more SMs and blocks active throughout execution.

Together, these components form a **hierarchical block-level stealing framework**, enabling DiggerBees to achieve efficient and scalable DFS on GPUs. Section 3.6 then presents a concrete example illustrating how these techniques are integrated in practice.

3.2 Two-Level Stack Data Structure

The two-level stack is composed of (1) a small, low-latency *HotRing* and (2) a large, high-capacity *ColdSeg*. Figure 2 illustrates the structure and its four core operations.

HotRing. The *HotRing* is a circular buffer in shared memory serving as the fast-access portion of the stack. Each stack entry is organized as a $\langle vertex|offset \rangle$ pair, where $offset$ points to the next neighbor to visit. We store them in two arrays, `hot_vertex` and `hot_offset`, each with size hot_size. Its state is tracked by two pointers: $head$ for the next free slot and $tail$ for the oldest unprocessed entry. The *HotRing* is empty when $head = tail$, and full when $(head + 1)\%hot\_size = tail$. Figure 2(a) illustrates this structure with a size-4 example. In our implementation, $hot\_size = 128$.

ColdSeg. The *ColdSeg* is a contiguous region in global memory that serves as the large-capacity portion of the stack. Each *ColdSeg* includes two arrays: `cold_vertex` and `cold_offset`, each with size $cold\_size = n_v/n_w$, where n_v is the number of vertices and n_w is the total number of warps. Its state is also tracked by two pointers: top and $bottom$, and it is empty when $top = bottom$. Figure 2(b) illustrates a *ColdSeg* example with six entries.

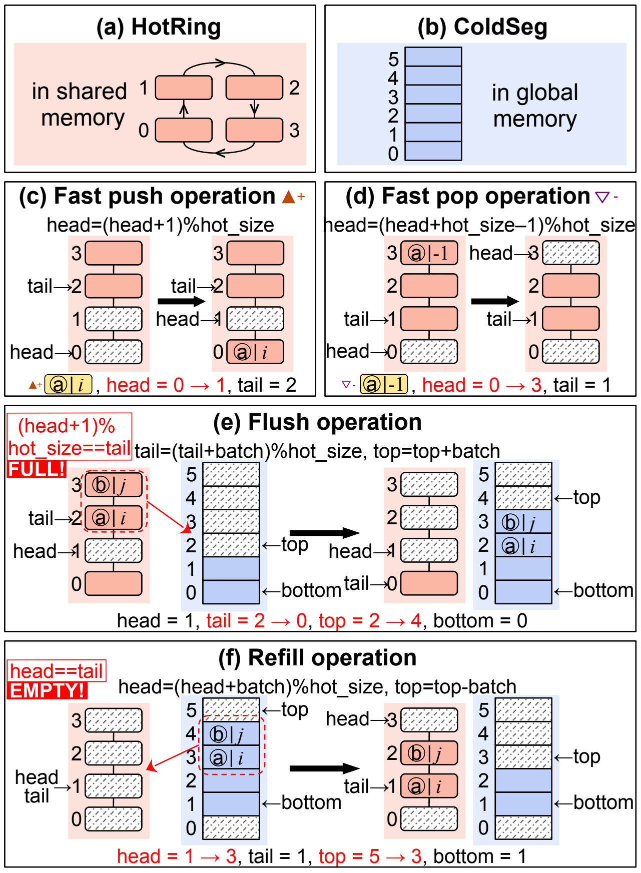

Figure 2. An example of the two-level stack structure and its four core operations. Subfigure (a) shows the circular *HotRing* in shared memory, while (b) shows the linear *ColdSeg* in global memory. Subfigures (c) and (d) show the fast push and pop operations in the *HotRing*. Subfigures (e) and (f) illustrate the interaction between *HotRing* and *ColdSeg*.

Core Operations. This two-level stack supports four core operations: (1) fast push and (2) fast pop in the *HotRing*, and (3) flush and (4) refill between the *HotRing* and the *ColdSeg*.

Figure 2(c) shows the **fast push** operation. A new entry is inserted into the *HotRing* at the *head* position, then *head* is updated as $(head + 1)\%hot\_size$. The example shows that $\langle a|i\rangle$ is pushed at $head = 0$, with *head* updated to $0 + 1 = 1$.

Figure 2(d) shows the **fast pop** operation. The top entry is retrieved by decrementing *head* to $(head + hot\_size - 1)\%hot\_size$. The popped entry has $offset = -1$, indicating the vertex has no unvisited neighbors. For example, entry $\langle a| - 1\rangle$ is popped and *head* is updated to $(0 + 4 - 1)\%4 = 3$.

Figure 2(e) shows the **flush** operation. When the *HotRing* is full, *i.e.*, $(head + 1)\%hot\_size = tail$, a batch of the oldest entries is moved to the *ColdSeg*. With $batch\_size = 2$, entries $\langle a|i\rangle$ and $\langle b|j\rangle$ starting from $tail = 2$ are moved to positions [2, 3] in *ColdSeg*. After the transfer, *tail* is updated to $(2 + 2)\%4 = 0$ and *top* to $2 + 2 = 4$.

Figure 2(f) shows the **refill** operation. When the *HotRing* is empty, *i.e.*, $head = tail$, a batch is refilled from the *ColdSeg*. In the example, entries $\langle a|i\rangle$ and $\langle b|j\rangle$ at [3, 4] in the *ColdSeg* are copied to the *HotRing* starting at $tail = 1$. After that, *head* is updated to $(1 + 2)\%4 = 3$ and *top* to $5 - 2 = 3$.

3.3 Warp-Level Workload

In DiggerBees, the warp is the fundamental unit of execution and stack ownership. Each warp performs DFS independently on its own *HotRing*, with all 32 threads in a warp following the same path, thereby eliminating warp divergence. The traversal procedure follows the standard stack-based DFS, requiring only lightweight warp-level synchronization. The only global synchronization required is vertex-access control using `atomicCAS` on the `visited` array to prevent multiple warps from processing the same vertex.

A key feature of our warp-level DFS is its management of the finite *HotRing* capacity. When a push operation finds the *HotRing* is full, a batch of entries is flushed to the *ColdSeg* to free space for the new entry. Conversely, when the *HotRing* becomes empty, a batch of entries is refilled from the *ColdSeg*. We optimize flush and refill using asynchronous copy. Contiguous batches in the *HotRing* are handled with specialized instructions: `cp_async_bulk` for flush operations and `cuda::memcpy_async` with Tensor Memory Accelerator (TMA) for refill operations. Our evaluation on the H100 GPU indicates this TMA-driven approach yields an approximately 5% performance improvement.

We choose to flush entries starting from the *tail* position for two reasons: (1) to preserve recently added entries near the *head*, thereby improving locality for ongoing DFS traversal, and (2) to prioritize older entries for flushing, as they typically correspond to larger unexplored branches and are better candidates for subsequent hierarchical work stealing.

3.4 Intra-Block Work Stealing

To ensure all warps within a block participate actively in DFS execution, we introduce intra-block work stealing, which operates in shared memory within a block and allows idle warps to acquire work from heavily loaded peers.

The intra-block work-stealing mechanism contains three steps: (1) victim selection, (2) work reservation, and (3) local transfer. Figure 3(a) illustrates an example of this process, and Algorithm 3 provides its pseudocode implementation.

In the first step, an idle warp, defined as having both *HotRing* and *ColdSeg* empty, acts as a "thief" and scans peers within the same thread block to identify a suitable "victim". The remaining tasks in each warp's *HotRing* are computed as $hot\_rest = (head - tail + hot\_size)\%hot\_size$. To avoid excessive fine-grained stealing, we introduce a threshold *hot_cutoff*. A warp is deemed a valid victim only if its *hot_rest* is the maximum among all warps and exceeds *hot_cutoff* (lines 4-10 in Algorithm 3). As illustrated in Figure 3(a), both Warp1 and Warp2 are idle. Warp0 has

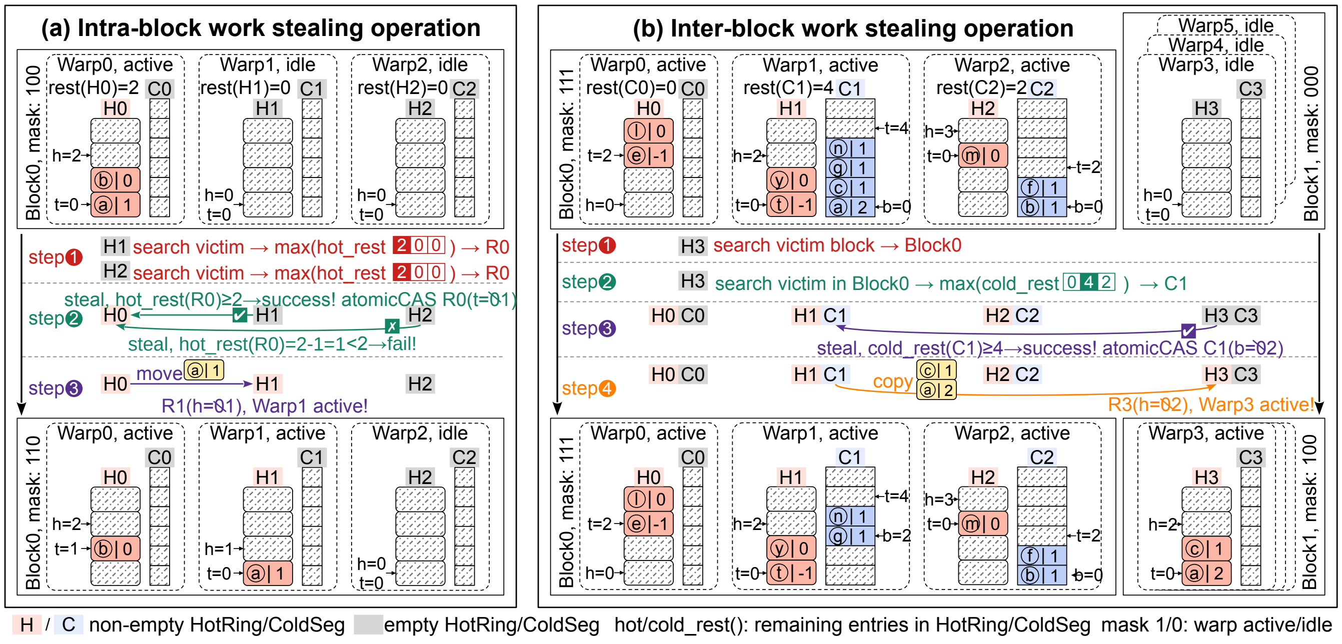
(a) Intra-block work stealing operation
(b) Inter-block work stealing operation
Block0, mask: 100
Warp0, active
Warp1, idle
Warp2, idle
rest(H0)=2
rest(H1)=0
rest(H2)=0
step❶ H1 search victim → max(hot_rest 2 0 0) → R0
H2 search victim → max(hot_rest 2 0 0) → R0
step❷ steal, hot_rest(R0)≥2→success! atomicCAS R0(t=0→1)
steal, hot_rest(R0)=2-1=1<2→fail!
step❸ move a|1
R1(h=0→1), Warp1 active!
Block0, mask: 110
Block0, mask: 111
Block1, mask: 000
Block1, mask: 100
Warp3, idle
Warp4, idle
Warp5, idle
rest(C0)=0
rest(C1)=4
rest(C2)=2
step❶ H3 search victim block → Block0
step❷ H3 search victim in Block0 → max(cold_rest 0 4 2) → C1
step❸ steal, cold_rest(C1)≥4→success! atomicCAS C1(b=0→2)
step❹ copy
R3(h=0→2), Warp3 active!
H / C non-empty HotRing/ColdSeg
empty HotRing/ColdSeg
hot/cold_rest(): remaining entries in HotRing/ColdSeg
mask 1/0: warp active/idle

Figure 3. An example of the intra-block and inter-block stealing operations. Subfigure (a) illustrates the three-step process of an intra-block stealing operation and the states of the warps before and after. Two idle warps (Warp1 and Warp2) attempt to steal work from the *HotRing* of the active warp (Warp0) within the same block, and only Warp1 succeeds. Subfigure (b) illustrates a four-step inter-block stealing process. The leader warp (Warp3) of idle Block1 selects Block0 as the victim block and steals work from the *ColdSeg* of Warp1 in Block0, successfully acquiring tasks and becoming active.

Algorithm 3 Intra-block Work Stealing (Executed by each idle warp within a block)

```
1: Input: head, tail, HotRing arrays for all warps in the block
2: hot_cutoff ← threshold for HotRing depth
3: if warp w is idle then
4:     max_rest ← 0, v ← −1                                ▷ Step 1: Victim selection
5:     for each warp i in the block do
6:         hot_rest ←(head[i] - tail[i] + hot_size)%hot_size
7:         if hot_rest > max_rest then
8:             max_rest ← hot_rest, v ← i
9:         end if
10:    end for
11:    if v ≠ −1 and max_rest ≥ hot_cutoff then
12:        h_s ← hot_cutoff/2                                ▷ Step 2: Work reservation
13:        old_tail ← tail[v]
14:        new_tail ← (old_tail + h_s)%hot_size
15:        if atomicCAS( tail[v], old_tail, new_tail) == old_tail then
16:            threadfence_block()                           ▷ Step 3: Local transfer
17:            Copy h_s entry pairs from HotRing[v] to HotRing[w]
18:            head[w] ← (head[w] + h_s)%hot_size
19:            Mark warp w as active
20:        end if
21:    end if
22: end if
```

hot_rest = 2, while Warp1 and Warp2 have *hot_rest* = 0. Since Warp0 has the maximum *hot_rest* and satisfies the threshold condition *hot_rest* ≥ *hot_cutoff* = 2, both Warp1 and Warp2 select it as the victim. In our evaluation, we set *hot_cutoff* = 32.

In the second step, the thief warp must reserve the work from the victim. To avoid conflicts, the victim warp always operates at the *head* of its *HotRing*, while thieves target the *tail*. An atomicCAS operation on the victim's *tail* ensures that only one warp can successfully claim this specific work batch. If it succeeds, the thief reserves a batch of *hot_cutoff*/2 entries and updates the victim's *tail* (line 14). Otherwise, it aborts and may select a new victim. In Figure 3(a), both Warp1 and Warp2 attempt to reserve work from Warp0 with competing atomicCAS operations. Warp1 succeeds, reserving one entry and updating Warp0's *tail* to 1. Thus, Warp0's *rest* becomes 2−1 = 1. Warp2 observes that Warp0 no longer satisfies *rest* ≥ 2 and retries the victim selection process.

In the third step, after successfully reserving work, the thief warp issues threadfence_block() to ensure that the victim's updated *tail* is visible within the block. It then copies the reserved batch from the victim's *HotRing* into its own *HotRing* (line 17 in Algorithm 3). Next, the thief warp updates its *head* and becomes active (lines 18-19), allowing it to immediately resume DFS traversal. In the example, Warp1 copies an entry $\langle a|1\rangle$ from Warp0 into its own *HotRing*, updates its *head* to 0 + 1 = 1, and becomes active.

We coordinate state changes with a 32-bit mask in shared memory, where each bit indicates whether a warp is active (1) or idle (0). Active warps perform DFS while idle warps attempt to steal. As shown in Figure 3(a), when Warp1 steals work from Warp0, the mask is updated from '100' to '110'.

Through dynamically redistributing the workloads among warps, this intra-block work stealing keeps warps in a block as active as possible, achieving high productivity at the intra-block level.

3.5 Inter-Block Work Stealing

To fully leverage GPU's massive parallelism, DiggerBees scales from single-block to multi-block execution, activating more SMs and blocks across the GPU. However, extending DFS execution to multiple blocks introduces new challenges: (1) how to manage the global communication overhead compared to intra-block operations in shared memory, and (2) how to achieve balanced workload distribution across blocks when facing irregular DFS workloads.

Algorithm 4 A pseudocode of the inter-block stealing (Executed by the leader warp of each idle block)

```
1: Input: top, bottom, ColdSeg arrays for all warps of all blocks
2:        head, tail, HotRing arrays of idle block
3: $cold\_cutoff \leftarrow$ the threshold of ColdSeg length
4: $nB \leftarrow$ the total number of blocks
5: if the block $b$ is idle then
6:     $w \leftarrow$ the leader warp of block $b$ ▷ Step1: Victim block selection
7:     Randomly select two active blocks $v_1, v_2 \in [0, nB)$, $v_i \neq b$
8:     $vb \leftarrow$ block with higher cumulative workload among $\{v_1, v_2\}$
9:     max_cold $\leftarrow 0$, $vw \leftarrow -1$          ▷ Step2: Victim warp selection
10:    for each warp $i$ in block $vb$ do
11:        cold_rest $\leftarrow$ top[$i$] - bottom[$i$]
12:        if cold_rest > max_cold then
13:            max_cold $\leftarrow$ cold_rest, $vw \leftarrow i$
14:        end if
15:    end for
16:    if $vw \neq -1$ and max_cold $\geq$ $cold\_cutoff$ then
17:        $c\_s \leftarrow cold\_cutoff/2$               ▷ Step3: Work reservation
18:        $old\_bt \leftarrow$ bottom[$vw$]
19:        $new\_bt \leftarrow old\_bt + c\_s$
20:        if CAS(bottom[$vw$], $old\_bt$, $new\_bt$) succeeds then
21:            threadfence()                        ▷ Step4: Remote transfer
22:            Copy $c\_s$ entry pairs: $ColdSeg[vw] \rightarrow HotRing[w]$
23:            head[$w$] $\leftarrow$ (head[$w$] + $c\_s$)$\%hot\_size$
24:            Mark $w$, $b$ as active
25:        end if
26:    end if
27: end if
```

We develop an inter-block work-stealing mechanism that enables idle blocks to acquire work from loaded blocks, which proceeds in four steps: (1) victim block selection, (2) victim warp selection, (3) work reservation, and (4) remote transfer. To address challenge (1), we designate a single leader warp per block for all inter-block communications; for challenge (2), we adopt a two-choice load-aware victim selection strategy. Figure 3(b) illustrates this process, and Algorithm 4 provides the implementation details.

In the first step, the leader warp identifies a victim block. Instead of scanning all blocks, we adopt a power-of-two choices [68] with a load-aware selection strategy: the leader warp samples two active blocks randomly and selects the one with heavier workload as the victim (lines 6-8 in Algorithm 4). This approach balances discovery efficiency with load-balancing effectiveness. As shown in Figure 3(b), the leader warp Warp3 of idle Block1 selects Block0 as the victim.

In the second step, the leader warp selects the warp in the victim block with the maximum remaining entries in its *ColdSeg*, computed as $cold\_rest = top - bottom$ (lines 9-15 in Algorithm 4). A warp qualifies as the victim only if its $cold\_rest$ exceeds $cold\_cutoff$. In Figure 3(b), in Block0, Warp1 has $cold\_rest = 4$, while Warp2 has $cold\_rest = 2$. Thus, Warp3 selects Warp1 as the victim since it satisfies the condition $cold\_rest \geq cold\_cutoff = 4$ in this example. In our real evaluation, we set $cold\_cutoff = 64$.

In the third step, the leader warp must reserve work from the victim warp's *ColdSeg*. To handle contention when multiple blocks compete for the same victim, the leader warp attempts to reserve a batch of $cold\_cutoff/2$ entries by performing an `atomicCAS` operation on the victim's $bottom$ pointer (lines 17-19 in Algorithm 4). In the example, Warp3 updates Warp1's $bottom$ from 0 to 2, reserving two entries and reducing Warp1's $cold\_rest$ to $4 - 2 = 2$.

In the last step, the leader warp issues a `threadfence()` (line 21 in Algorithm 4) to ensure global memory consistency. It then copies the reserved entries from the victim's *ColdSeg* to its *HotRing* (line 22) using asynchronous copy (`cuda::memcpy_async`) for efficiency. As shown in Figure 3(b), Warp3 copies two entries $\langle a|2\rangle$ and $\langle c|1\rangle$ from Warp1's *ColdSeg* into its *HotRing* and updates its $head$ to $0 + 2 = 2$.

Together with intra-block execution, these two mechanisms form our hierarchical block-level stealing that enables DiggerBees to scale DFS across the GPU's parallelism.

3.6 An Execution Example

Figure 4 presents an execution example that shows the complete workflow of DiggerBees on a graph. The top subfigures show a 10-step DFS tree construction process. The example uses a two-block configuration with three warps per block (Warp0-Warp2 in Block0, Warp3-Warp5 in Block1). Different colored regions indicate subtrees explored by different warps. The detailed breakdown for Steps 6–termination on the bottom highlights the collaboration of our intra-block and inter-block execution.

The process begins with initialization, where the root vertex a is pushed into Warp0's *HotRing*. Then Warp0 starts the warp-level DFS. As the traversal progresses and Warp0's *HotRing* accumulates more entries, Warp1 and Warp2 begin stealing from Warp0 using the intra-block work stealing. This process expands parallelism from a single active warp to multiple cooperating warps within the block. As shown in Figure 4, by Step6, three warps in Block0 are actively working on different subtrees, while Block1 remains idle.

The critical transition occurs when Block0 becomes heav-

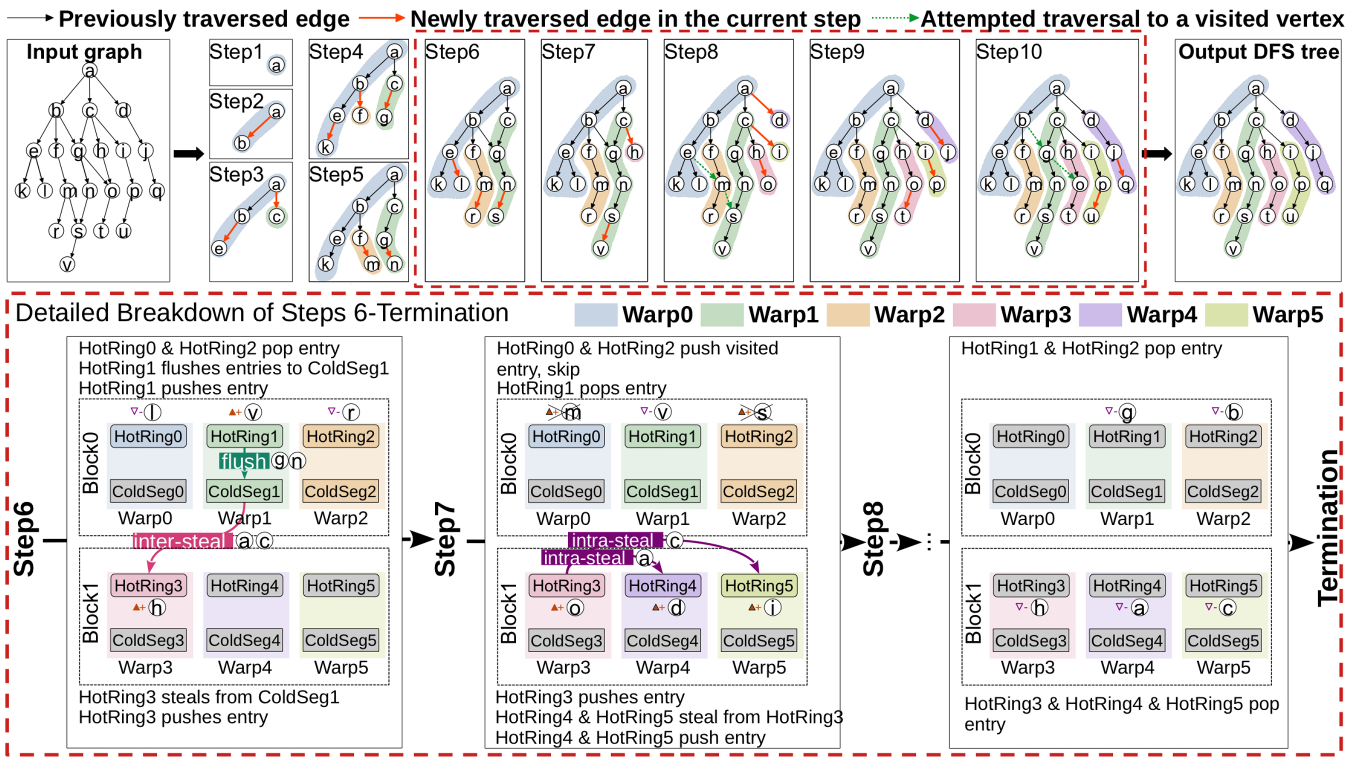
Previously traversed edge
Newly traversed edge in the current step
Attempted traversal to a visited vertex
Input graph
Step1
Step2
Step3
Step4
Step5
Step6
Step7
Step8
Step9
Step10
Output DFS tree
Detailed Breakdown of Steps 6-Termination
Warp0
Warp1
Warp2
Warp3
Warp4
Warp5
HotRing0 & HotRing2 pop entry
HotRing1 flushes entries to ColdSeg1
HotRing1 pushes entry
flush
inter-steal
HotRing3 steals from ColdSeg1
HotRing3 pushes entry
HotRing0 & HotRing2 push visited entry, skip
HotRing1 pops entry
intra-steal
HotRing3 pushes entry
HotRing4 & HotRing5 steal from HotRing3
HotRing4 & HotRing5 push entry
HotRing1 & HotRing2 pop entry
HotRing3 & HotRing4 & HotRing5 pop entry
Block0
Block1
Termination

Figure 4. An example of the complete execution flow of DiggerBees. The top part shows the 10-step DFS tree construction from root initialization (Step1) to the final output, where different colored regions indicate the subtrees explored by different warps. The figure omits the final pop-only steps, as they do not affect the DFS tree structure. The bottom shows a detailed breakdown of warp operations from Step6 to Termination. Each block maintains three warps with their two-level stack structure, illustrating how warps and blocks collaborate to sustain parallel execution.

ily loaded while Block1 remains idle. At this point, inter-block work stealing is triggered. As detailed in the bottom breakdown of Figure 4, after Warp1's *HotRing1* flushes two entries to *ColdSeg1*, Block1's leader warp Warp3 identifies Block0 as the victim block and targets *ColdSeg1* for work acquisition. In Step7, Warp3 successfully steals entries from *ColdSeg1* and transfers them to its own *HotRing3*, enabling Block1 to join the parallel DFS exploration.

Once Warp3 begins processing its work, the intra-block work stealing within Block1 is activated. Warp4 and Warp5 sequentially steal from Warp3's *HotRing3* and join the DFS exploration during Steps 7–8. By Step8, all six warps across both blocks are actively participating in the parallel traversal, achieving near-optimal GPU utilization.

The execution flow continues until all blocks become idle. At this point, the traversal reaches global termination. In the final state of Figure 4, the last active warps pop their remaining entries from *HotRings*, resulting in empty stacks and termination. The effectiveness of load balancing is evident from the final workload distribution: each warp processes a relatively balanced number of vertices (Warp0: 5 vertices, Warp1: 5 vertices, Warp2: 3 vertices in Block0; Warp3: 3 vertices, Warp4: 3 vertices, Warp5: 3 vertices in Block1).

Table 1. The three platforms and five evaluated methods.

Hardware	Method	Type
Intel Xeon Max 9462 CPU, 2×32 cores, 2×64GB HBM, B/W 1 TB/s	(1)CKL-PDFS [19] (2) ACR-PDFS [2]	DFS DFS
A100 (Ampere) PCIe GPU 108 SMs, 6912 CUDA cores 80 GB, B/W 1.94 TB/s	(3) NVG-DFS [69] (4) Gunrock [97] / BerryBees [70]	DFS BFS
H100 (Hopper) SXM5 GPU 132 SMs, 16896 CUDA cores 64 GB, B/W 2.02 TB/s	**(5) DiggerBees (this work)**	DFS

4 Evaluation

4.1 Experimental Setup

Our experimental platform includes one CPU and two GPUs. The CPU platform is equipped with a 64-core Intel Xeon Max 9462 processor. We use two NVIDIA GPUs: an A100 GPU (Ampere architecture) and an H100 GPU (Hopper architecture). All experiments are conducted under Ubuntu 22.04 with CUDA 12.8.

We compare DiggerBees against three DFS implemen-

Table 2. Output semantics of different traversal algorithms.

Method	visited	DFS Tree	Lex-Order	Level
CKL-PDFS	✓	N/A	N/A	N/A
ACR-PDFS	✓	N/A	N/A	N/A
NVG-DFS	✓	✓	**Ordered**	N/A
Gunrock/BerryBees	✓	N/A	N/A	✓
DiggerBees (this work)	✓	✓	**Unordered**	N/A

Table 3. Descriptions of three group collections.

Group	**Count**	**Description**
DIMACS10	151	Benchmark graphs from the 10th DIMACS Implementation Challenge, covering clustering, numerical simulation, and road networks.
SNAP [106]	68	Real-world networks from the Stanford Network Analysis Platform, including social, citation, and web graphs.
LAW [11, 12]	15	Large-scale web graphs from the Laboratory for Web Algorithmics, based on real web crawls and compressed via WebGraph.

tations: CKL-PDFS [19] and ACR-PDFS [2] on CPUs, and NVG-DFS [69] on GPUs. To the best of our knowledge, these methods represent all publicly available parallel DFS implementations with accessible or reproducible code[1]. Besides, we compare with two GPU BFS methods: Gunrock [97], a widely used graph processing framework and BerryBees [70], a recent high-performance BFS algorithm. Table 1 lists the specifications of our experimental setup.

It is worth noting that different algorithms produce different outputs. To ensure fair comparison, we evaluate each method using its native output semantics. Table 2 summarizes the output semantics of each method.

CKL-PDFS and ACR-PDFS: These two CPU implementations report only reachability information (the `visited` array) without constructing a DFS tree.

NVG-DFS: This method employs a three-phase BFS-style algorithm to construct a lexicographic DFS ordering. We evaluate only the tree construction phase.

Gunrock and BerryBees: These two BFS baselines output reachability and level information. For our purpose, we consider only the `visited` array.

DiggerBees (this work): Our method produces the standard parallel DFS result: the `visited` and `parent` arrays, representing a valid DFS tree.

As for the dataset, we evaluate all 234 graphs from three widely used graph collections, DIMACS10, SNAP [106], and

[1]We contacted the authors and confirmed that no official GPU implementation of NVG-DFS is publicly available. We reimplemented the path-based algorithm on GPU based on the paper's description and successfully reproduced the expected performance reported in the paper.

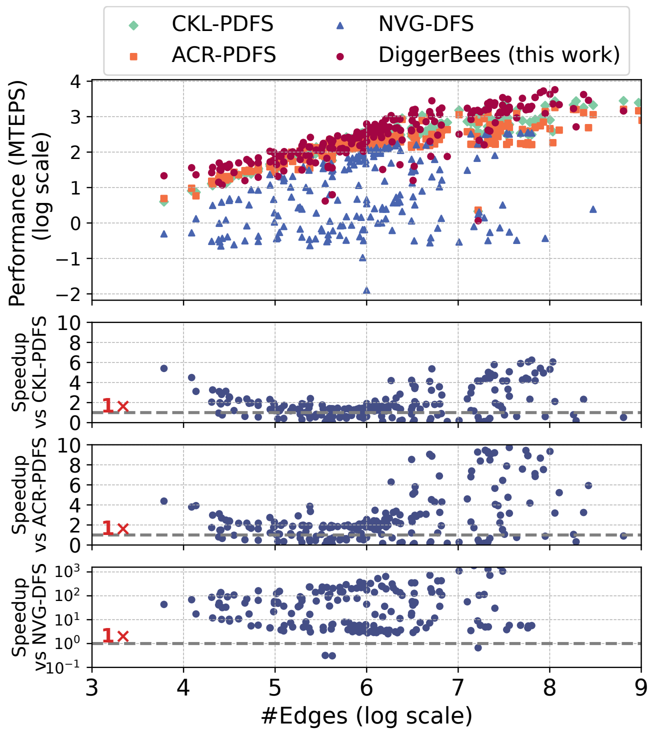

Figure 5. Performance comparison of DiggerBees with three state-of-the-art DFS methods on the H100 GPU, including two CPU implementations (CKL-PDFS and ACR-PDFS) and one GPU implementation (NVG-DFS). The top subplot shows the traversal performance, while the bottom three subplots report the speedup of DiggerBees over each baseline.

LAW [11, 12] available in the SuiteSparse Matrix Collection [22]. Specifically, our dataset includes 151 graphs from DIMACS10, 68 from SNAP, and 15 from LAW. The descriptions are summarized in Table 3. The graphs in our dataset require between 0.08 MB and 43.61 GB of GPU memory in CSR format. In addition, we select 12 representative graphs for detailed analysis, as listed in Table 4.

For fair comparison across all methods, we use 64 input vertices from the GAP benchmark suite [7] and report average performance as the ratio of traversed edges to runtime.

4.2 Comparison with Existing DFS Approaches

We evaluate the performance of DiggerBees against three existing DFS methods: CKL-PDFS and ACR-PDFS running on an Intel CPU, and NVG-DFS running on the H100 GPU across all 234 graphs of our dataset. Figure 5 presents the performance comparison of these four methods, measured in million traversed edges per second (MTEPS). The top subfigure shows the performance of each method, while the bottom three subfigures present the speedup of DiggerBees over each baseline.

As shown in Figure 5, DiggerBees outperforms all other DFS implementations on the majority of graphs. Compared

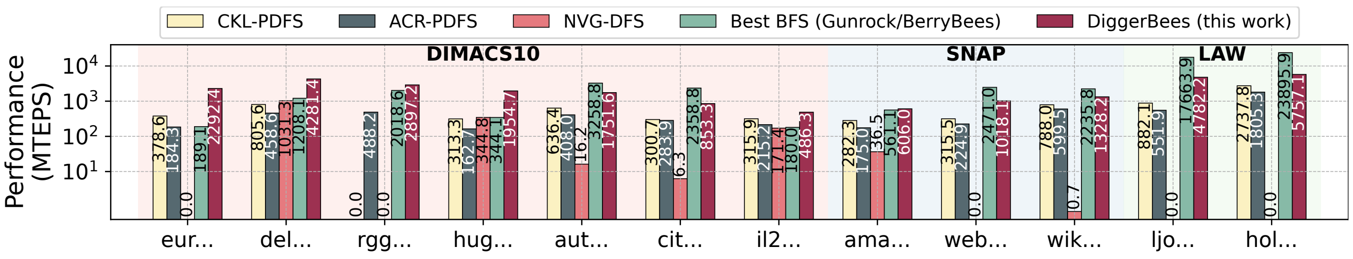

Figure 6. Performance comparison of four DFS methods and the best BFS baseline (the better-performing result between Gunrock and BerryBees) across 12 representative graphs from three groups on the H100 GPU.

with the two CPU implementations, DiggerBees achieves an average speedup (geometric mean) of 1.37× and 1.83× over CKL-PDFS and ACR-PDFS, respectively. It is important to note that this gain is achieved even though DiggerBees performs more work by constructing a full DFS tree (`visited` + `parent`), whereas the CPU baselines only output reachability (`visited`). The highest speedups are observed on 'hugebubbles' (vs. CKL-PDFS) and 'euro_osm' (vs. ACR-PDFS), where DiggerBees outperforms the CPU baselines by 6.24× and 12.44×, respectively. This performance advantage primarily comes from the high parallelism offered by modern GPUs, enabling thousands of concurrent DFS execution compared to the limited parallelism of the 64-core CPU implementations, enabling DiggerBees to achieve favorable performance on large-scale graphs.

DiggerBees achieves significantly better performance than the GPU-based NVG-DFS, with an average speedup of 30.18× and over 1000× on graphs such as 'higgs-twitter' (1841.68×) and 'soc-Pokec' (1075.21×). This performance gap stems from output semantics: NVG-DFS enforces strict lexicographic DFS ordering, whereas DiggerBees generates a valid DFS tree without such constraints. In practice, many graph applications require only the tree structure (*e.g.*, cycle detection or topological sorting), so strict ordering offers little benefit. Moreover, NVG-DFS incurs high memory overhead due to its path-tracking design, failing on 44 out of 234 graphs. In contrast, DiggerBees successfully processes all graphs, demonstrating robustness for large-scale workloads.

4.3 Comparison with GPU BFS Approaches

To provide an exhaustive evaluation, we compare DiggerBees with high-performance GPU BFS algorithms. We select two representative BFS methods: Gunrock and BerryBees. Although BFS and DFS solve different problems, comparing their reachability performance provides useful insights into the effectiveness of our approach. Figure 6 shows the results on 12 representative graphs. For each graph, we report the performance of the three DFS baselines, the better-performing BFS method ("Best BFS"), and DiggerBees.

Surprisingly, our DiggerBees outperforms the BFS implementations on several graphs. This is particularly noteworthy given that BFS is typically more GPU-friendly due to its level-parallel nature. Our advantage is especially evident on road network graphs (*e.g.*, 'euro_osm') and certain mesh-like graphs (*e.g.*, 'hugebubbles' and 'delaunay'). These graphs contain long and narrow traversal paths that require tens of thousands of levels in BFS (*e.g.*, 'euro_osm' requires 17,346 levels). In contrast, DiggerBees leverages hierarchical block-level work stealing to distribute these deep paths across warps, achieving high efficiency. On 'euro_osm', for instance, DiggerBees achieves a 12.12× speedup over the Best BFS.

On the other hand, on some social network graphs like 'ljournal' from LAW, BFS completes in only 10 levels, allowing it to process a large number of vertices in parallel at each level. DFS traversal on such graphs, however, involves many short paths with frequent backtracking and limited parallel expansion, resulting in low warp occupancy. On 'ljournal', DiggerBees is 3.70× slower than BFS.

Nonetheless, our results demonstrate that DFS should no longer be considered a weak competitor on GPUs. With careful hierarchical task distribution, DiggerBees not only narrows the traditional performance gap but, on graphs containing long and narrow paths, even surpasses state-of-the-art BFS implementations.

Table 4. Detailed information of 12 representative graphs.

Group	Graph	$\|V\|$	$\|E\|$	Graph	$\|V\|$	$\|E\|$
DIMACS10	euro_osm	50.9M	108.1M	delaunay	16.8M	100.7M
	rgg	16.8M	265.1M	hugebubble	21.2M	63.6M
	auto	0.4M	6.6M	citation	0.3M	2.3M
	il2010	0.5M	2.2M			
SNAP	amazon	0.3M	1.2M	google	0.9M	5.1M
	wiki	1.8M	28.6M			
LAW	ljournal	5.4M	79.0M	hollywood	1.1M	113.9M

4.4 Scalability Comparison with GPU DFS

To evaluate the scalability of our approach, we compare the performance of DiggerBees and NVG-DFS on both A100 and H100 GPUs across 234 graphs. Figure 7 shows this comparison. The top subfigure shows the performance, with fitted trend lines to visualize overall performance growth from A100 to H100. The bottom two subplots report scalability as

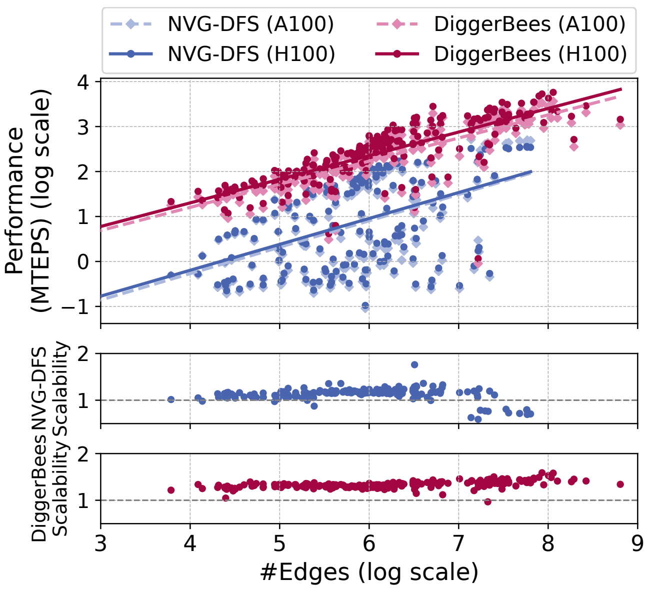

Figure 7. Scalability comparison of DiggerBees and NVG-DFS on A100 and H100 GPUs. The top subplot shows the performance with fitted trends. The bottom two subplots report scalability as the performance ratio (H100/A100).

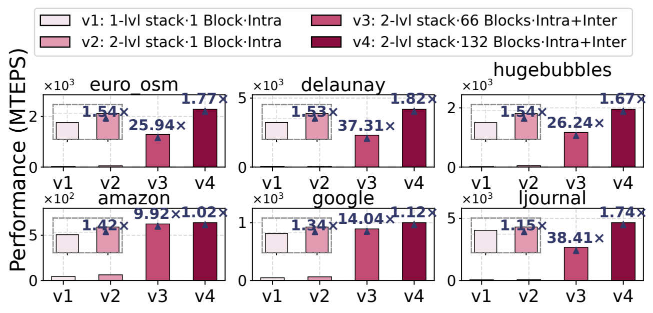

Figure 8. Performance breakdown of four versions of DiggerBees across six representative graphs on the H100 GPU.

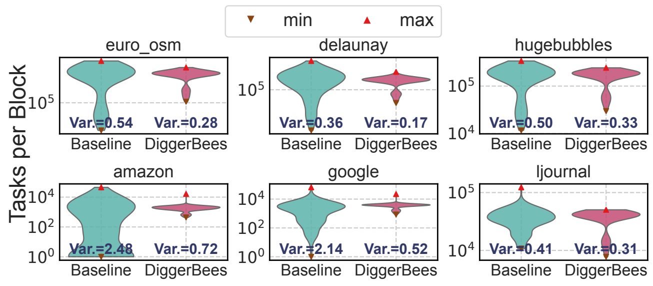

Figure 9. Block-level workload distribution for six representative graphs, comparing Baseline (left) and DiggerBees (right). Markers show minimum, median, and maximum workloads. *Var.* denotes the coefficient of variation.

the performance ratio (H100/A100) for each method.

As shown in Figure 7, DiggerBees consistently outperforms NVG-DFS on both A100 and H100 GPUs. In addition, the performance gain from A100 to H100 is more pronounced for DiggerBees. Specifically, the geometric mean of the H100-to-A100 speedup is 1.33× for DiggerBees, compared to only 1.18× for NVG-DFS. This scalability advantage stems from DiggerBees' ability to effectively utilize the increased compute resources of the H100 GPU. The H100 GPU provides 132 SMs compared to 108 SMs in the A100 GPU (a 22.2% increase), and DiggerBees achieves a performance improvement that closely matches this hardware scaling. This demonstrates that DiggerBees scales naturally with increased SM count, fully exploiting the enhanced parallelism of modern GPU generations.

4.5 Performance Breakdown

To better understand the contributions of our design, we conduct a breakdown analysis with four progressive versions of DiggerBees: (v1) a one-level stack and intra-block execution, (v2) a two-level stack and intra-block execution, (v3) a two-level stack with 66 blocks and both intra- and inter-block stealing, and (v4) the full implementation with 132 blocks (one block per SM on H100). Figure 8 reports the results on six representative graphs.

The transition from v1 to v2 demonstrates the effectiveness of our two-level stack design. By leveraging the GPU memory hierarchy with hot entries in shared memory, DiggerBees achieves low-latency stack access. As a result, v2 achieves approximately 45% higher throughput on average, validating the benefits of this hierarchical data structure.

The transition from v2 to v3 shows the benefit of inter-block work stealing. By enabling multiple blocks to work collaboratively, DiggerBees achieves dramatic improvements. For instance, v3 achieves 25.94× speedup on 'euro_osm' and 37.31× speedup on 'delaunay'. These results demonstrate that inter-block work stealing is essential to scale DFS across SMs and fully utilize GPU parallelism.

Finally, the step from v3 to v4 illustrates the effect of increasing the block count to match all available SMs. Most graphs show an additional 67–82% improvement, while certain small graphs, such as 'amazon' and 'google', see limited gains (2–12%), as their smaller workloads have been well distributed within fewer blocks.

4.6 Block-Level Load Balance Analysis

To evaluate the effectiveness of our hierarchical block-level work stealing in balancing workloads, we measure the distribution of tasks per block. Figure 9 shows the results across six representative graphs, comparing the baseline strategy (random victim block selection) with DiggerBees (load-aware two-choice strategy). The reported *Var.* denotes the coefficient of variation (lower is better).

As shown in Figure 9, the baseline exhibits highly uneven task distribution: some blocks process substantially more tasks while others receive very few, resulting in high variance (*e.g.*, 2.48 for 'amazon' and 2.14 for 'google'). In contrast, DiggerBees narrows the spread of task counts across blocks and consistently reduces variance by more than half. For example, the variance on 'amazon' drops to 0.72, a 3.44× improvement over the baseline. These results confirm that by leveraging load-aware two-choice victim selection, our hierarchical work stealing effectively balances workloads among blocks, thereby enhancing both scalability and performance.

4.7 Sensitivity Analysis of Cutoff Selection

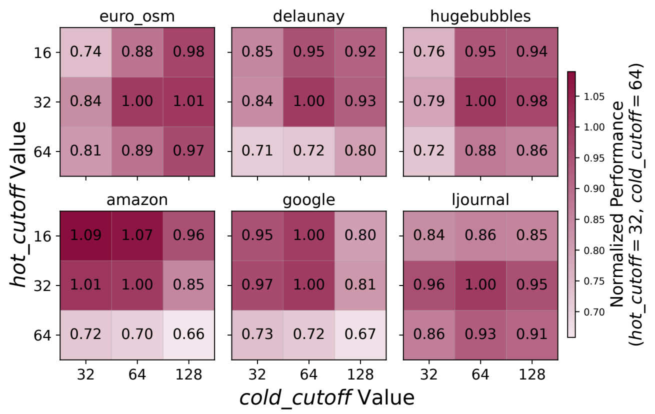

Figure 10. Sensitivity of DiggerBees to the *hot_cutoff* and *cold_cutoff* parameters on six representative graphs. Performance is normalized to the baseline configuration (*hot_cutoff* = 32, *cold_cutoff* = 64), with darker colors indicating higher performance.

To evaluate the impact of work stealing granularity, we conduct a sensitivity analysis by varying *hot_cutoff*={16,32,64} and *cold_cutoff*={32,64,128}. Figure 10 presents the performance heatmaps, where all results are normalized to the default configuration (*hot_cutoff* = 32, *cold_cutoff* = 64).

As shown in Figure 10, our default setting consistently achieves near-optimal performance. When the cutoff values are set too small, idle warps attempt to steal work more frequently, which increases contention on atomic operations (*e.g.*, `atomicCAS`) and leads to higher synchronization overhead. In contrast, excessively large cutoff values raise the stealing threshold, making it harder for idle warps or blocks to obtain new work, which reduces the reactivity of the load-balancing mechanism and results in underutilization.

Moreover, our analysis reveals that performance is generally more sensitive to *cold_cutoff* than to *hot_cutoff*. For example, on the graph 'google', setting *cold_cutoff* = 128 while keeping *hot_cutoff* = 32 leads to a performance degradation of about 20%. This behavior can be attributed to the fact that inter-block stealing requires transferring work from global memory into shared memory. Large *cold_cutoff* values delay such transfers and increase the cost of remote data movement, resulting in higher overhead and latency.

5 Related Work

DFS has long been recognized as *P*-complete [79] and hard to parallelize [39, 93]. However, the practical importance of DFS has motivated extensive research into the **parallel unordered DFS algorithms and theory**, with early efforts on distributed systems [28, 34, 46, 47, 75, 81, 84]. Kumar established an analytical framework for DFS, and a series of works with Rao and others explored parallelization across multiprocessors [52–56, 76–78]. Other studies targeted graphs such as DAGs [23, 38], planar graphs [41, 43, 49, 85, 88], and others [4, 50]. To improve performance on multi-core CPUs, Cong et al. [19] and Acar et al. [2] introduced dynamic strategies to redistribute DFS workloads among threads. Moreover, recent work focuses on memory efficiency. [6], variants of Tarjan's algorithm [60], and nearly work-efficient solutions [37]. However, most of these works remain theoretical or tied to outdated hardware, offering limited guidance for modern parallel platforms such as GPUs.

Extensive research has explored **graph traversal on GPUs**, but the majority focuses on BFS [35, 59, 61, 65, 70, 96, 99, 101]. Modern graph processing frameworks likewise provide optimized BFS implementations, including Ligra [86, 87], GBBS [25] and EGACS [104] on CPUs, as well as Gunrock [97], CuSha [51], Cagra [103], Tigr [72], SEP-Graph [95], Groute [8] and Graphie [42] on GPUs. Beyond single-node systems, BFS has been scaled to extreme levels in distributed environments [16, 58]. In contrast, DFS remains largely unexplored on GPUs due to its inherently sequential nature. Naumov et al. [69] approximated DFS ordering through BFS-style traversal. Spampinato et al. [89] proposed a linear algebra formulation of DFS, but their approach remains theoretical without practical implementation guidelines. Our work bridges this gap by demonstrating that efficient parallel DFS on GPUs is achievable, providing a practical solution for applications requiring DFS semantics.

To efficiently balance irregular workloads like DFS, **work stealing** has emerged as a fundamental scheduling strategy [10]. Research has evolved to tackle various architectural complexities, such as optimizations of cache locality [1, 40] and cost-aware scheduling for irregular loops [67] on multicore CPUs, hierarchical designs [66, 74] and distributed protocols [26] on clusters, and affinity-aware load balancing on heterogeneous platforms [5, 9, 31, 36]. Cong et al. [19] and Acar et al. [2] applied dynamic workload redistribution to DFS, while D'Antonio et al. [21] recently utilized work stealing for single-source shortest path. Besides, frameworks such as Julienne [24] and Gemini [105] extend dynamic scheduling to graph algorithms. In multi-GPU environments, mechanisms for remote stealing have been proposed for large-scale graph analytics [57, 64]. Complementary to scheduling, an-other line focuses on concurrent data structures, such as non-blocking trees [29, 30], balanced search structures [13–15], and relaxed balancing techniques [32, 33]. In comparison, DiggerBees introduces a fine-grained work-stealing DFS specifically optimized for the GPU memory hierarchy.

The wide applicability of DFS has motivated **application-specific DFS optimizations**, such as IDA* [80], branch-and-bound [63] and puzzle-solving [62]. Recently, graph mining and subgraph matching using DFS on GPUs have become active [17, 18, 73, 102]. Wei and Jiang [98] introduced stack-based DFS loops, Qiu et al. [73] designed batch-dynamic updates for dynamic matching, and Yuan et al. [102] accelerated DFS-style matching with pruned expansion. Sun and Luo [90, 91] parallelized recursive backtracking. DFS also appears in distributed querying (aDFS [94]), dynamic algorithms for directed graphs [100], and tree traversals [44, 82, 83]. These efforts remain domain-specific and do not address general DFS traversal. In contrast, our work introduces DiggerBees, a general-purpose DFS method on GPUs.

6 Conclusion

In this paper, we have proposed DiggerBees, a new parallel DFS algorithm optimized for GPUs. Our method addresses three key challenges via a two-level stack structure and hierarchical block-level stealing. Experimental results on NVIDIA GPUs show that DiggerBees significantly outperforms state-of-the-art CPU and GPU DFS implementations, surpasses GPU BFS on specific graph types, and exhibits robust scalability across modern GPU architectures.

Data Availability Statement

The artifact for this paper is publicly available on Zenodo [71].

Acknowledgments

We deeply appreciate the invaluable comments from all the reviewers. We thank Weichen Cao for helpful discussions during the rebuttal phases, and Yida Li, Haocheng Lian, and Hemeng Wang for their assistance with setting up the experimental environment. This work has been partially supported by the European HiPEAC Network of Excellence and has also received funding from 'Future of Computing, a Barcelona Supercomputing Center and IBM initiative' (2023). It has also been partially supported by the project PID2023-146511NB-I00 funded by the Spanish Ministry of Science, Innovation and Universities MCIU /AEI /10.13039/501100011033 and EU ERDF, and by the National Natural Science Foundation of China (No. U23A20301 and No. 62372467). Yuyao Niu is supported by Grant PRE2022-104335, funded by MICIU/AEI/10.13039/501100011033, and by the "ESF+" Grant CEX2021-001148-S-20-2, funded by MICIU/AEI/10.13039/501100011033. Marc Casas is the corresponding author.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *SPAA '00*. 1–12. doi:10.1145/341800.341801
[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work-efficient algorithm for parallel unordered depth-first search. In *SC '15*. 1–12. doi:10.1145/2807591.2807651
[3] Alok Aggarwal and Richard Anderson. 1987. A random NC algorithm for depth first search. In *STOC '87*. 325–334. doi:10.1145/28395.28430
[4] A. Aggarwal, R. J. Anderson, and M.-Y. Kao. 1989. Parallel depth-first search in general directed graphs. In *STOC '89*. 297–308. doi:10.1145/73007.73035
[5] Matthew Agostini, Francis O'Brien, and Tarek Abdelrahman. 2020. Balancing Graph Processing Workloads Using Work Stealing on Heterogeneous CPU-FPGA Systems. In *ICPP '20*. Article 50, 12 pages. doi:10.1145/3404397.3404433
[6] Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. 2014. Depth-First Search Using Bits. In *ISAAC '14*. 553–564. doi:10.1007/978-3-319-13075-0_44
[7] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
[8] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *PPoPP '17*. 235–248. doi:10.1145/3155284.3018756
[9] A Tarun Beri, B Sorav Bansal, and C Subodh Kumar. 2015. Locality aware work-stealing based scheduling in hybrid CPU-GPU clusters. In *PDPTA '15*. 48.
[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46 (Sept. 1999), 720–748. doi:10.1145/324133.324234
[11] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW '11*. 587–596. doi:10.1145/1963405.1963488
[12] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW '04*. 595–601. doi:10.1145/988672.988752
[13] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *PPoPP '14*. 329–342. doi:10.1145/2555243.2555267
[14] Trevor Brown and Joanna Helga. 2011. Non-blocking k-ary search trees. In *OPODIS'11*. 207–221. doi:10.1007/978-3-642-25873-2_15
[15] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-blocking interpolation search trees with doubly-logarithmic running time. In *PPoPP '20*. 276–291. doi:10.1145/3332466.3374542
[16] Huanqi Cao, Yuanwei Wang, Haojie Wang, Heng Lin, Zixuan Ma, Wanwang Yin, and Wenguang Chen. 2022. Scaling graph traversal to 281 trillion edges with 40 million cores. In *PPoPP '22*. 234–245. doi:10.1145/3503221.3508403
[17] Weichen Cao, Ke Meng, Zhiheng Lin, and Guangming Tan. 2025. GLumin: Fast Connectivity Check Based on LUTs For Efficient Graph Pattern Mining. In *PPoPP '25*. 455–468. doi:10.1145/3710848.3710889
[18] Xuhao Chen and Arvind. 2022. Efficient and scalable graph pattern mining on {GPUs}. In *OSDI '22*. 857–877. doi:10.1109/PADSW.2018.8644869
[19] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. 2008. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *ICPP '08*. 536–545. doi:10.1109/ICPP.2008.88
[20] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*.
[21] Marco D'Antonio, Thai Son Mai, Philippas Tsigas, and Hans Vandierendonck. 2025. Wasp: Efficient Asynchronous Single-Source

Shortest Path on Multicore Systems via Work Stealing. In *SC '25*. 2109–2125. doi:10.1145/3712285.3759872

[22] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011). doi:10.1145/2049662.2049663

[23] Pilar Delatorre and Clyde P. Kruskal. 1995. Fast Parallel Algorithms for All-Sources Lexicographic Search and Path-Algebra Problems. *Journal of Algorithms* 19, 1 (1995), 1–24. doi:10.1006/jagm.1995.1025

[24] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing. In *SPAA '17*. 293–304. doi:10.1145/3087556.3087580

[25] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (2021), 70 pages. doi:10.1145/3434393

[26] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. 2009. Scalable work stealing. In *SC '09*. 11 pages. doi:10.1145/1654059.1654113

[27] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2023. Provably Fast and Space-Efficient Parallel Biconnectivity. In *PPoPP '23*. 52–65. doi:10.1145/3572848.3577483

[28] Ossama Ibrahim El-Dessouki and Wing H. Huen. 1980. Distributed enumeration on between computers. *IEEE Trans. Comput.* 29, 09 (1980), 818–825. doi:10.1109/TC.1980.1675681

[29] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. 2014. The amortized complexity of non-blocking binary search trees. In *PODC '14*. 332–340. doi:10.1145/2611462.2611486

[30] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *PODC '10*. 131–140. doi:10.1145/1835698.1835736

[31] Naila Farooqui, Rajkishore Barik, Brian T. Lewis, Tatiana Shpeisman, and Karsten Schwan. 2016. Affinity-aware work-stealing for integrated CPU-GPU processors. In *PPoPP '16*. Article 30, 2 pages. doi:10.1145/2851141.2851194

[32] Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. 2019. Persistent Non-Blocking Binary Search Trees Supporting Wait-Free Range Queries. In *SPAA '19*. 275–286. doi:10.1145/3323165.3323197

[33] Panagiota Fatourou and Eric Ruppert. 2025. Lock-Free Augmented Trees (Abstract). In *HOPC '25*. 1–3. doi:10.1145/3746238.3746251

[34] Raphael Finkel and Udi Manber. 1987. DIB—a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.* 9, 2 (1987), 235–256. doi:10.1145/22719.24067

[35] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *HPDC '19*. 121–131. doi:10.1145/3307681.3326606

[36] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. 2013. Locality-aware work stealing on multi-CPU and multi-GPU architectures. In *MULTIPROG '13*.

[37] Mohsen Ghaffari, Christoph Grunau, and Jiahao Qu. 2023. Nearly Work-Efficient Parallel DFS in Undirected Graphs. In *SPAA '23*. 273–283. doi:10.1145/3558481.3591094

[38] Ratan K. Ghosh and G. P. Bhattacharjee. 1984. A parallel search algorithm for directed acyclic graphs. *BIT Numerical Mathematics* 24, 2 (1984), 133–150. doi:10.1007/BF01937481

[39] Raymond Greenlaw. 1992. A model classifying algorithms as inherently sequential with applications to graph searching. *Information and Computation* 97, 2 (1992), 133–149. doi:10.1016/0890-5401(92)90033-C

[40] Yan Gu, Zachary Napier, and Yihan Sun. 2022. *Analysis of Work-Stealing and Parallel Cache Complexity.* 46–60. doi:10.1137/1.9781611977059.4

[41] Torben Hagerup. 1990. Planar Depth-First Search in $O(\log n)$ Parallel Time. *SIAM J. Comput.* 19, 4 (1990), 678–704. doi:10.1137/0219047

[42] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *PACT '17*. 233–245. doi:10.1109/PACT.2017.41

[43] Xin He and Yaacov Yesha. 1988. A Nearly Optimal Parallel Algorithm for Constructing Depth First Spanning Trees in Planar Graphs. *SIAM J. Comput.* 17, 3 (1988), 486–491. doi:10.1137/0217028

[44] Nikhil Hegde, Jianqiao Liu, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. Treelogy: A benchmark suite for tree traversals. In *ISPASS '17*. 227–238. doi:10.1109/ISPASS.2017.7975294

[45] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378. doi:10.1145/362248.362272

[46] Masaharu Imai, Yuuji Yoshida, and Teruo Fukumura. 1979. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. In *IJCAI'79*. 416–418.

[47] Virendra K Janakiram, Dharma P Agrawal, and Ravi Mehrotra. 1987. Randomized Parallel Algorithms for Prolog Programs and Backtracking Applications.. In *ICPP '87*. 278–281.

[48] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562. doi:10.1145/368996.369025

[49] Ming-Yang Kao. 1988. All graphs have cycle separators and planar directed depth-first search is in DNC. In *Aegean Workshop on Computing*. 53–63. doi:10.1007/BFb0040373

[50] George Karypis and Vipin Kumar. 1994. Unstructured tree search on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems* 5, 10 (1994), 1057–1072. doi:10.1109/71.313122

[51] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *HPDC '14*. 239–252. doi:10.1145/2600212.2600227

[52] Vipin Kumar. 1987. Depth-first search. *Encyclopaedia of Artificial Intelligence* 2 (1987), 1004–1005.

[53] Vipin Kumar, Ananth Y. Grama, and V. Nageshwara Rao. 1994. Scalable Load Balancing Techniques for Parallel Computers. *J. Parallel and Distrib. Comput.* 22, 1 (1994), 60–79. doi:10.1006/jpdc.1994.1070

[54] Vipin Kumar and V. Nageshwara Rao. 1987. Parallel depth first search. part ii. analysis. *International Journal of Parallel Programming* 16, 6 (1987), 501–519. doi:10.1007/BF01389001

[55] Vipin Kumar and V. Nageshwara Rao. 1990. *Scalable parallel formulations of depth-first search.* 1–41. doi:10.1007/978-1-4612-3390-9_1

[56] Vipin Kumar, V. Nageshwara Rao, and K. Ramesh. 1988. *Parallel Depth First Search on the Ring Architecture.* Technical Report.

[57] João V.F. Lima, Thierry Gautier, Nicolas Maillard, and Vincent Danjean. 2012. Exploiting Concurrent GPU Operations for Efficient Work Stealing on Multi-GPUs. In *SBAC-PAD '12*. 75–82. doi:10.1109/SBAC-PAD.2012.28

[58] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. 2018. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *SC '18*. 706–716. doi:10.1109/SC.2018.00059

[59] Hang Liu and H. Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC '15*. 1–12. doi:10.1145/2807591.2807594

[60] Gavin Lowe. 2016. Concurrent depth-first search algorithms based on Tarjan's Algorithm. *Int. J. Softw. Tools Technol. Transf.* 18, 2 (2016), 129–147. doi:10.1007/s10009-015-0382-1

[61] Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *DAC '10*. 52–55. doi:10.1145/1837274.1837289

[62] Basel A. Mahafzah. 2014. Performance evaluation of parallel multithreaded A* heuristic search algorithm. *Journal of Information Science* 40, 3 (2014), 363–375. doi:10.1177/0165551513519212

[63] Nihar R. Mahapatra and Shantanu Dutt. 1999. Sequential and parallel branch-and-bound search under limited-memory constraints. *Institute for Mathematics and Its Applications* 106 (1999), 139. doi:10.1007/978-1-4612-1492-2_6

[64] Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, and Jingren

Zhou. 2023. Efficient Multi-GPU Graph Processing with Remote Work Stealing. In *ICDE '23*. 191–204. doi:10.1109/ICDE55515.2023.00022

[65] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *PPoPP '12*. 117–128. doi:10.1145/2370036.2145832

[66] Seung-Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical work stealing on manycore clusters. In *PGAS11 '11*, Vol. 625.

[67] Prasoon Mishra and V. Krishna Nandivada. 2024. COWS for High Performance: Cost Aware Work Stealing for Irregular Parallel Loop. *ACM Trans. Archit. Code Optim.*, Article 12 (2024), 26 pages. doi:10.1145/3633331

[68] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104. doi:10.1109/71.963420

[69] Maxim Naumov, Alysson Vrielink, and Michael Garland. 2017. Parallel Depth-First Search for Directed Acyclic Graphs. In *IA3'17*. Article 4, 8 pages. doi:10.1145/3149704.3149764

[70] Yuyao Niu and Marc Casas. 2025. BerryBees: Breadth First Search by Bit-Tensor-Cores. In *PPoPP '25*. 339–354. doi:10.1145/3710848.3710859

[71] Yuyao Niu, Yuechen Lu, Weifeng Liu, and Marc Casas. 2025. *DiggerBees Artifact*. doi:10.5281/zenodo.18072817

[72] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *ASPLOS '18*. 622–636. doi:10.1145/3296957.3173180

[73] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *ICDE '24'*. 3204–3216. doi:10.1109/ICDE60146.2024.00248

[74] Jean-Noël Quintin and Frédéric Wagner. 2010. Hierarchical work-stealing. In *Euro-Par '10*. 217–229. doi:10.1007/978-3-642-15277-1_21

[75] Stefan Radtke, Jens Bargfrede, and Walter Anheier. 1995. Distributed automatic test pattern generation with a parallel FAN algorithm. In *ICCD '95*. 698–702. doi:10.1109/ICCD.1995.528944

[76] V. Nageshwara Rao and Vipin Kumar. 1987. Parallel depth first search. part i. implementation. *International Journal of Parallel Programming* 16, 6 (1987), 479–499. doi:10.1007/BF01389000

[77] V. Nageshwara Rao and Vipin Kumar. 1988. Superlinear speedup in parallel state-space search. In *Foundations of Software Technology and Theoretical Computer Science*. 161–174. doi:10.1007/3-540-50517-2_79

[78] V. Nageshwara Rao and Vipin Kumar. 1993. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems* 4, 4 (1993), 427–437. doi:10.1109/71.219757

[79] John H. Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229–234. doi:10.1016/0020-0190(85)90024-9

[80] Alexander Reinefeld and Volker Schnecke. 1994. AIDA*-Asynchronous Parallel IDA*. In *Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence*. 295–302.

[81] A. Reinefeld and V. Schnecke. 1994. Work-load balancing in highly parallel depth-first search. In *SHPCC '94*. 773–780. doi:10.1109/SHPCC.1994.296719

[82] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: a framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.*, Article 76 (2017), 30 pages. doi:10.1145/3133900

[83] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, fine-grained traversal fusion for heterogeneous trees. In *PLDI 2019*. 830–844. doi:10.1145/3314221.3314626

[84] Vikram A. Saletore and L. V. Kalé. 1990. Consistent linear speedups to a first solution in parallel state-space search. In *AAAI'90*. 227–233.

[85] Gregory E. Shannon. 1988. A linear-processor algorithm for depth-first search in planar graphs. *Inform. Process. Lett.* 29, 3 (1988), 119–123. doi:10.1016/0020-0190(88)90048-8

[86] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP '13*. 135–146. doi:10.1145/2442516.2442530

[87] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *DCC '15*. 403–412. doi:10.1109/DCC.2015.8

[88] Justin R. Smith. 1986. Parallel Algorithms for Depth-First Searches I. Planar Graphs. *SIAM J. Comput.* 15, 3 (1986), 814–830. doi:10.1137/0215058

[89] Daniele G. Spampinato, Upasana Sridhar, and Tze Meng Low. 2019. Linear algebraic depth-first search. In *ARRAY '19*. 93–104. doi:10.1145/3315454.3329962

[90] Shixuan Sun, Yulin Che, Lipeng Wang, and Qiong Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *ICDE '19*. 232–243. doi:10.1109/ICDE.2019.00029

[91] Shixuan Sun and Qiong Luo. 2018. Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine. In *ICPADS '18*. 1–9. doi:10.1109/PADSW.2018.8644869

[92] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160. doi:10.1137/0201010

[93] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[94] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. 2021. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *USENIX ATC '21*. 273–287.

[95] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *PPoPP '19*. 38–52. doi:10.1145/3293883.3295733

[96] Letong Wang, Guy Blelloch, Yan Gu, and Yihan Sun. 2025. Parallel Cluster-BFS and Applications to Shortest Paths. In *ALENEX '25*. 42–55. doi:10.1137/1.9781611978339.4

[97] 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 *PPoPP '16*. 1–12. doi:10.1145/2851141.2851145

[98] Yihua Wei and Peng Jiang. 2022. STMatch: Accelerating Graph Pattern Matching on GPU with Stack-Based Loop Optimizations. In *SC '22*. 1–13. doi:10.1109/SC41404.2022.00058

[99] Hao Wen and Wei Zhang. 2019. Improving Parallelism of Breadth First Search (BFS) Algorithm for Accelerated Performance on GPUs. In *HPEC '19*. 1–7. doi:10.1109/HPEC.2019.8916551

[100] Bohua Yang, Dong Wen, Lu Qin, Ying Zhang, Xubo Wang, and Xuemin Lin. 2019. Fully dynamic depth-first search in directed graphs. *Proc. VLDB Endow.* 13, 2 (2019), 142–154. doi:10.14778/3364324.3364329

[101] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2022. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *ASPLOS '23*. 78–92. doi:10.1145/3567955.3567963

[102] Lyuheng Yuan, Da Yan, Jiao Han, Akhlaque Ahmad, Yang Zhou, and Zhe Jiang. 2024. Faster Depth-First Subgraph Matching on GPUs. In *ICDE '24'*. 3151–3163. doi:10.1109/ICDE60146.2024.00244

[103] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *Big Data '17*. 293–302. doi:10.1109/BigData.2017.8257937

[104] Ruohuang Zheng and Sreepathi Pai. 2021. Efficient Execution of Graph Algorithms on CPU with SIMD Extensions. In *CGO '21*. 262–276. doi:10.1109/CGO51591.2021.9370326

[105] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI '16*. 301–316.

[106] Marinka Zitnik, Rok Sosič, Sagar Maheshwari, and Jure Leskovec. 2018. BioSNAP Datasets: Stanford Biomedical Network Dataset Collection. http://snap.stanford.edu/biodata.

A Artifact Description

A.1 Artifact DOI

https://doi.org/10.5281/zenodo.17709254.

A.2 Prerequisites

- **Operating System**: Any Linux system that supports CUDA v12.8 or above and GCC v11.3 or above.
- **Libraries**: This artifact includes:
  – DiggerBees (this work)
  – NVG-DFS
  – CKL-PDFS and ACR-PDFS (https://github.com/deepsea-inria/sc15-pdfs)
  – Gunrock (https://github.com/gunrock/gunrock)
  – BerryBees (https://doi.org/10.5281/zenodo.14222153)
- **Program**: CUDA and C/C++ OpenMP code.
- **Run-time environment**: Ubuntu 22.04 with CUDA v12.8 and GPU driver version 535.86.10 (as tested).
- **Hardware Requirements**:
  – Any CUDA-enabled GPU with compute capability 8.0 or above (an NVIDIA A100 (Ampere architecture) and an H100 (Hopper architecture) as tested).
  – Any Intel CPU (Intel Xeon Max 9462 as tested).
  – Disk Space: at least 600 GB (for full benchmark dataset).
- **Software Requirements**:
  – To evaluate DiggerBees: NVIDIA nvcc and GNU GCC (v12.8 and v11.4.0, as tested, respectively).
  – To evaluate other baselines: CMake (v3.30.5, as tested).
  – To reproduce the figures: Python v3.7 or above with the following libraries: numpy, pandas, seaborn, and matplotlib.
- **Input Data**:
  – six representative graphs in Table 4 of the paper (`rep_graphs.csv` in `DiggerBees_Artifact/dataset/` for quick-start testing in 30 minutes).
  – 234 graphs (`benchmark_list.csv` in `DiggerBees_Artifact/dataset/`) from the SuiteSparse Matrix Collection for in-depth evaluation.
- **Note**: Docker is provided as a convenient alternative. A prebuilt Docker container is available, including all dependencies, required software, six representative graphs, and libraries.

A.3 Quick test using the Docker container

- **Download Docker (if needed)**: Ensure the Docker and the NVIDIA Container Toolkit are installed on your system. For Ubuntu users, Docker can be installed following the instructions from https://docs.docker.com/engine/install/ubuntu/.
- **Pull the prebuilt Docker image**: Pull the prebuilt Docker image from Docker Hub using the command:

```
$ docker pull yuyaoniu/diggerbees:latest
```

or load the Docker image from Zenodo using the command:

```
$ docker load < \
diggerbees_docker_image_latest.tar.gz
```

- **Run the Docker container**: Start a Docker container using the pulled image with the following command:

```
$ docker run -it --rm \
  --gpus all yuyaoniu/diggerbees:latest
```

- **Quick start in about 40 minutes**: Perform the following steps to run a quick test:
  – Inside the container, navigate to the artifact directory:

```
$ cd /workspace/DiggerBees_Artifact/
```

The GPU architecture is configured via a centralized configuration file:

```
$ vim config.mk
```

Set the CUDA architecture according to your GPU:

```
CUDA_ARCH=80    # NVIDIA A100
CUDA_ARCH=90    # NVIDIA H100
```

Save the file after selecting the appropriate architecture.

  – Run the quick test.

```
$ cd scripts/
$ bash run_all.sh quick_start
```

This script automatically compiles all required components and runs experiments on six representative graphs.

A.4 Expected output

Upon completing the quick start test, the evaluator should observe the following expected outputs:

- **Performance results**:
  (a) Verify the summary CSV files (merged results) using the command:

```
$ ls ../data
```

  – `merged_dfs_perf.csv`: performance results of four DFS methods (CKL-PDFS, ACR-PDFS, NVG-DFS, and DiggerBees v4).
  – `merged_bfs_perf.csv`: Performance results of two BFS baselines and selection of best-performing BFS for each graph.

  - `merged_perf_rep.csv`: Final summary table of all methods (DFS + BFS) on representative graphs.

  (b) Verify detailed outputs of DiggerBees using the command:

  ```
  $ ls ../DiggerBees_master/results/
  ```

  - `DiggerBees_v{version}_{xx}_perf.csv`: Performance results of DiggerBees v1–v4 on tested representative graphs.
  - `balance_baseline/balance_{graph}.csv`: Load-balance logs (per-block task counts) for baselines.
  - `balance_diggerbees/balance_{graph}.csv`: Load-balance logs (per-block task counts) for DiggerBees v4.

  These files are automatically named according to GPU type (A100, H100, etc.), graph name, and variant version.

- **Generated Figures**:
  Verify figures using the command:

  ```
  $ ls ../figures/
  ```

  - `fig_5.pdf`: DFS performance comparison.
  - `fig_6.pdf`: Performance on representative graphs.
  - `fig_8.pdf`: Performance breakdown of DiggerBees.
  - `fig_9.pdf`: Load-balance visualization.

  These figures should match the trends (not exact values) shown in the paper, verifying performance advantage and improved load balancing.

Note: The quick start test does *not* generate Figure 7 automatically. Figure 7 (Scalability comparison) requires performance data collected from two different GPU systems (e.g., A100 and H100). Therefore, it is only produced in the full evaluation workflow described in Section A.5.

A.5 Step-by-Step Instructions

A.5.1 Overview of Evaluation Goals. The purpose of the full evaluation is to reproduce all key experimental results presented in Section 4 of the paper, based on the full benchmark dataset. Specifically, the evaluator can expect to reproduce the following results:

(1) DFS Performance Comparison (Figure 5 in Section 4.2): Scatter plot comparing the performance of four DFS methods over the full dataset.

(2) Representative Graph Performance (Figure 6 in Section 4.3: Bar chart showing the performance of four DFS methods and the best BFS baseline on the 12 representative graphs.

(3) Scalability Comparison (Figure 7 in Section 4.4): Scatter plot comparing the performance of two GPU DFS methods on A100 and H100 GPUs.

(4) Performance Breakdown (Figure 8 in Section 4.5): Breakdown of four versions of DiggerBees across six graphs.

(5) Load Balance Analysis (Figure 9 in Section 4.6): Violin plots visualizing per-block task distribution for DiggerBees compared to baselines.

A.5.2 Detailed Steps for running the full benchmark. The command below automatically executes the entire evaluation workflow, including the following steps:

```
$ bash run_all.sh run_bench
```

This script contains the entire experimental workflow and includes the following steps:

- **Dataset Preparation (Approx. 30+ hours)**: The script automatically downloads all required datasets using the following command:

  ```
  $ python3 download_graphs.py run_bench
  ```

  Our matrix parser supports input files in the Matrix Market format (*.mtx). All graphs are publicly available from the SuiteSparse Matrix Collection, which can be accessed at https://sparse.tamu.edu/. This process will take approximately 30 hours or more (assuming a download speed of 6 MB/s, with a total dataset size estimated to be 600 GB). Matrices will be stored in the directory:
  `DiggerBees_Artifact/dataset/MM/`
- **Run Experiments (Approx. 10 hours)**: This step performs the full execution of all DFS and BFS methods. The process is managed by:

  ```
  $ bash run_experiments.sh run_bench
  ```

  This workflow completes the following tasks:
  - Compile and run GPU-based DFS methods: DiggerBees and NVG-DFS.
  - Compile and run CPU-based DFS baselines: ACR-PDFS and CKL-PDFS.
    Note: Modify the `-proc` parameter according to the number of CPU cores available on your system (lines 29-30 in `DiggerBees_Artifact/baseline/DFS/SC15_unordered_dfs/run_dfs_cpu.sh`).
  - Compile and run GPU BFS methods: Gunrock and BerryBees.

  Each method produces individual CSV files and is organized under its respective `results/` folder. This process takes approximately 12 hours to complete, depending on the GPU and CPU configuration.
- **Data Collection (Approx. 1 minute)**: This step automatically merges all `results/` folders across methods, extracts performance metrics, and generates the merged summary files in `DiggerBees_Artifact/data/`. These summary files will be used in later steps to generate Figures 5 and 6.
- **Figures Plotting (Approx. 15 minutes)**: After the performance data has been successfully collected, all

figures in the paper can be plotted using the command:

```
$ python3 plot/plot_fig_{x}.py
```

– Figures 5 and 6: These two figures are generated from `DiggerBees_Artifact/data/merged_dfs_perf.csv` and `DiggerBees_Artifact/data/merged_perf_rep.csv`.
– Figure 7 (Required: Two GPUs): This figure uses data collected from two different GPU architectures (e.g., A100 and H100 in the paper). It will not be generated in single-GPU mode.
– Figure 8: This figure is generated after running all four DiggerBees versions (v1–v4) under the `run_subset` mode, which collects detailed execution time breakdown for six graphs.
– Figure 9 (Balance Logging Required): This figure is generated after recording the balance logs using the command:

```
cd ../DiggerBees_master
make balance
bash run_diggerbees.sh run_balance
make balance BALANCE_POLICY=0
bash run_diggerbees.sh run_balance
```

All generated figures will be saved to `DiggerBees_Artifact/figures/`.

A.6 Customization & Extensibility

Users can evaluate their own graph datasets by following the steps below.

- Place the graph files (in '.mtx' format) into the 'dataset/MM/' directory.
- Create a CSV file containing the metadata of the added graphs. The CSV format should follow the same structure as the provided 'dataset/benchmark_list.csv'.
- Place the new CSV file under the 'dataset/' directory (e.g., 'dataset/user_bench.csv').
- Register the new benchmark list in 'scripts/run_all.sh'.

Example: Consider a user who wishes to evaluate a custom benchmark defined in 'dataset/user_bench.csv'.

First, add a new execution mode in 'scripts/run_all.sh':

```
elif [ "$MODE" == "run_user_bench" ]; then
input="../dataset/user_bench.csv"
echo "[INFO] Using CSV: $input"
python3 download_graphs.py "$input"
```

Then run the evaluation using:

```
$ bash run_all.sh run_user_bench
```

Received 2025-09-01; accepted 2025-11-10