POLITECNICO DI TORINO
Repository ISTITUZIONALE

Hydra-MCS: A Hybrid CPU-GPU Approach to Accelerate the Maximum Common Subgraph Computation

Original
Hydra-MCS: A Hybrid CPU-GPU Approach to Accelerate the Maximum Common Subgraph Computation / Cardone, L., Quer, S.. - In: IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. - ISSN 1558-2183. - ELETTRONICO. - 37:11(2026), pp. 2407-2422. [10.1109/TPDS.2026.3729765]

Availability:
This version is available at: 11583/3015877 since: 2026-09-24T12:30:15Z

Publisher:
IEEE Computer Society

Published
DOI:10.1109/TPDS.2026.3729765

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

16 October 2026

Hydra-MCS: A Hybrid CPU-GPU Approach to Accelerate the Maximum Common Subgraph Computation

Lorenzo Cardone, *Student IEEE*, Stefano Quer, *Senior IEEE*

***Abstract*—The Maximum Common Subgraph (MCS) problem is a fundamental challenge in graph theory. It generalizes the subgraph isomorphism problem and is known to be NP-complete and difficult to approximate. This inherent complexity underscores the need for efficient solutions. This work introduces Hydra-MCS, a novel hybrid CPU-GPU approach that significantly accelerates MCS computation. Our method uses a lightweight metric to accurately estimate the remaining computation (RC) for a given solution. Then it uses this metric to offload computations to the GPU, maintain load balancing, and minimize unnecessary task sharing between threads. We identify the essential information the CPU and GPU must share to optimize performance by carefully analyzing our hybrid implementation. Furthermore, we discuss the substantial advantages of offloading computation to the GPU. Our method establishes a new state of the art in performance. Under matched CPU resources, GPU integration provides approximately a $3\times$ additional speedup over Hydra-MCS CPU and approximately a $4\times$ speedup over parallel McSplit on the scaling workload; across long-running benchmark instances, speedups over McSplit typically reach 4-$5\times$, with peaks exceeding $12\times$. We also address scalability and robustness, as our algorithm's behavior significantly surpasses that of the original as the number of threads increases and the problems become harder. A comprehensive analysis comparing solved instances over time demonstrates that the hybrid implementation consistently solves 30-40% of the instances not yet solved by the original framework, peaking at 70% for the most complex graph pairs as they approach the timeout threshold.**

***Index Terms*—Algorithms, Algorithm design and analysis, Graph theory, Parallel computing**

I. Introduction

GRAPHS are versatile data structures that effectively abstract information and relationships into a machine-readable representation. In this framework, nodes represent data points, and edges symbolize their relationships. This powerful abstraction makes graphs applicable across a wide array of fields, including chemistry [1]–[4], social networks [5], web search [6], malware detection [7], hardware design simulation [8], and software component dependency detection [9], [10].

Since the 1970s, the Maximum Common Subgraph (MCS) problem has been a focal point of research, leading to numerous proposed solutions aimed at maximizing computational efficiency [11]–[16]. Among the most influential works is McGregor's backtracking algorithm [17], which systematically explores the solution search space. Researchers such as McCreesh et al. [18], and others introduced subsequent refinements. Vismara [19] transforms the MCS search into a maximum clique problem. Unfortunately, the transformation does not reduce the puzzle's inherent complexity and is ineffective for non-labeled graphs [20]. Given that the maximum clique problem is more amenable to parallelization, recent years have witnessed the emergence of GPU-based approaches to this converted problem [21]. Additionally, researchers developed heuristics and AI-based methods to reach optimal solutions in fewer iterations, particularly for large graph pairs where exhaustive search becomes computationally infeasible [22]–[29].

This paper focuses on the McSplit algorithm, one of the most recent and efficient approaches to solving the MCS problem, as proposed by McCreesh et al. [30]. This approach employs a branch-and-bound recursive procedure wherein each vertex in the two graphs is labeled based on its neighborhood in the current outcome. The procedure augments the current solution at each iteration by adding a new pair of vertices (one from each graph), with each vertex selected from those that share the same label. The process concurrently computes an upper bound for the current solutions at each iteration, effectively pruning the search space. This bound computation is exceptionally lightweight, considering only the size of each label class. While fast, this heuristic provides only an upper bound. Consequently, recent works have refined this aspect, aiming to guide the algorithm toward superior solutions with fewer recursions. Notable examples in this direction include McSplitRL [22], McSplitLL [23], McSplitDAL [24], McSplitSD [25], and RRSplit [26] Some works have also revisited the problem using GPUs, such as the work of Jayaraj et al. [21], Quer et al. [31], Jiang et al. [32], and Hu et al. [33], but with a precise target and limited advantages as analyzed in the related work section (Section II-C).

We propose Hydra-MCS, a hybrid CPU-GPU implementation of the McSplit algorithm that leverages the large number of GPU cores to offload part of the computational workload. During execution, the CPU creates new tasks and enqueues them for the GPU to process. Since task creation is generally less expensive than solving them, the queue can quickly accumulate pending tasks. To prevent bottlenecks caused by differences in processing speed, the CPU may process newly created tasks directly rather than continuously adding them to the GPU queue. This strategy enables both devices to work concurrently on partial solutions, optimizing resource utiliza-

L. Cardone and S. Quer are with the Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, Turin, Italy.

Authors contributed equally and are listed in alphabetical order.

tion. Unfortunately, this approach also introduces complexities in balancing CPU and GPU workloads, necessitating effective synchronization and intelligent information exchange. We present several parallelization schemes and discuss their respective trade-offs to elucidate the evolution of our current implementation and the rationale behind our design choices.

Hydra-MCS achieves greater speedup as the runtime of the original procedure increases, demonstrates greater scalability and robustness than traditional methods, and consistently outperforms the CPU on the most challenging instances. A comprehensive analysis shows that our hybrid method achieves peak speedups of over 12× and consistently solves 30-40% of the instances not yet solved by the original framework, with a peak improvement of 70% for the most complex graph pairs as they approach the timeout threshold.

We organize the paper as follows. Section II provides background on graph theory and introduces the McSplit algorithm. Section III-A presents improvements to the original McSplit parallelization scheme, including excessive and insufficient task sharing examples. Section III-B details our GPU implementation and the evolution of the parallelization strategy. Section IV reports our results, comparing the various algorithm versions and their scalability with increasing number of threads. Section V discusses the algorithm's limitations and suggests directions for future work.

Algorithm 1 Parallel McSplit.

Require: G_1, G_2, bds, sol, $best_sol$
1: **if** $sol.size() > best_sol.size()$ **then**
2: $\quad best_sol \leftarrow sol$
3: **end if**
4: $bound = compute_bound(bds, sol)$
5: **if** $bound \leq best_sol$ **then**
6: $\quad$ **return**
7: **end if**
8: $bd \leftarrow select_bd(bds)$
9: **if** $sol.size() < 5$ **then**
10: $\quad donate_work(G_1, G_2, bds, sol, best_sol)$
11: **end if**
12: $v = smallest_node(bd.left)$
13: $remove_vtx(v, bd.left)$
14: **for** $w \in bd.right$ **do**
15: $\quad remove_vtx(w, bd.right)$
16: $\quad sol.insert(v, w)$
17: $\quad new_bds = update_bds(G_1, G_2, bds, sol)$
18: $\quad McSplit(G_1, G_2, new_bds, sol, best_sol)$
19: $\quad insert_vtx(w, bd.right)$
20: **end for**
21: $McSplit(G_1, G_2, bds, sol, best_sol)$

II. Background and Related Works

A. Graphs and Notation

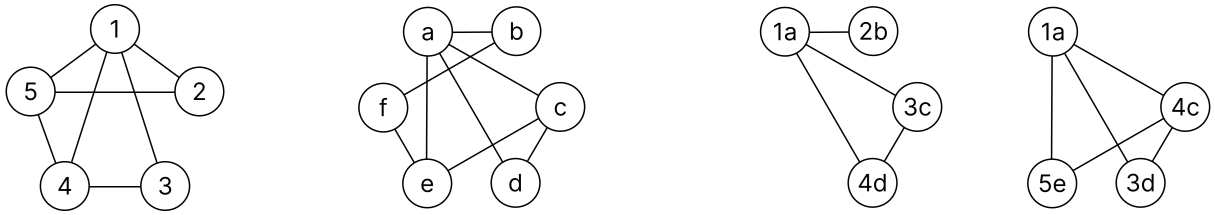
This work focuses on unweighted, directed, or undirected graphs. A graph is a set of vertices connected through edges that we define as $G = (V, E)$. V is the set of vertices of G, and E is a set of relationships between pairs of vertices. We indicate the number of elements in V with $|V|$, and the number of elements of E with $|E|$. When a graph is undirected, each edge $e \in E$ connecting two vertices $v, w \in V$ is an unordered relationship, i.e., it is identical to an edge $f \in E$ connecting the same two vertices in the opposite direction $w, v \in V$. Our approach does not consider multiple edges connecting the same two vertices, as it automatically collapses them into a single edge.

Although multiple definitions exist for the Maximum Common Subgraph, this work focuses on the variant with the maximum number of vertices. In our definition of the problem, we aim to maximize the number of vertices in the common subgraph, while disregarding the number of edges in the solution. We define the maximum common subgraph between two graphs G_1 and G_2 as $MCS(G_1, G_2)$. Figure 1 shows a graphical example of two graphs and some possible MCSs.

B. The McSplit Procedure

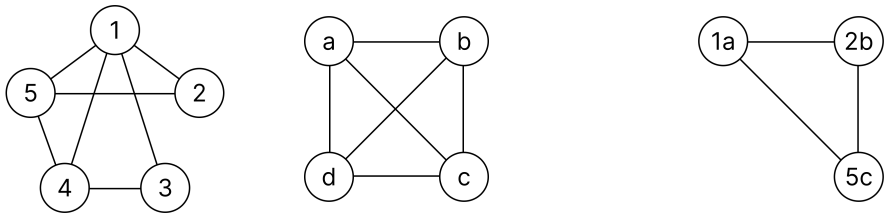
McSplit [30] is a state-of-the-art branch-and-bound procedure to search for the MCS of a graph pair using a depth-first search on a sorted and pruned version of the solution space.

Algorithm 1 shows a simplified version of the original parallel approach working on the graphs G_1 and G_2. The procedure starts by checking whether the current solution is larger than the best solution found so far (line 1). If this is the case, it updates the best solution in line 2. It then proceeds to compute an upper bound for the current solution (line 4) by summing together the size of each label class (as shown in Table I). The bidomain structure stores information about label classes, defining a class's size as the smaller of the left and right graph node counts. If the bound is not large enough to improve the current best solution, then there is no need to proceed with the computation, and the algorithm backtracks to find more promising pairings (line 6). Otherwise, the procedure selects one bidomain to proceed with the MCS search (line 8). We can employ various heuristics to make this choice. However, one of the most effective strategies is the min-max strategy, which identifies the bidomain with the fewest nodes in the graph that has the most nodes. For clarity, if we have a bidomain with one node on the first graph and five nodes on the second, and a second bidomain with four nodes on both graphs, we would select the latter. It then attempts to donate the current work to another thread (line 10) by sharing its execution status. To prevent duplicate work, a global atomic variable indicates which portion of the solution each thread is already processing. We select the node in the left graph with the lowest identifier (line 12); thus, we can speed up execution by sorting the graph vertices [34]. Once selected, a node is removed from the domain to avoid choosing it again (line 13). Finally, the function tries to pair the current node with each node of the second graph (line 14). It updates the bidomains with the new labels since the solution has changed (line 17) and then recurs. After the algorithm verifies all possible pairings for the node in the first graph, it proceeds recursively without reinserting the node into the domain; in this way, it will not match the same node to anything in the second graph (line 21).



(a) G_1 (b) G_2 (c) Two different $MCS(G_1, G_2)$

Fig. 1. Given the graphs G_1 and G_2 represented in Figures (a) and (b), respectively, Figure (c) reports two different admissible MCSs. Labels are shown only to help identify the vertices, as the graphs are not to be considered labeled.



(a) G_1 (b) G_2 (c) $MCS(G_1, G_2)$

Fig. 2. The figure shows two undirected, unlabeled graphs, G_1 and G_2, along with a common subgraph identified by the McSplit algorithm. Vertex identifiers, i.e., $\{1, 2, 3, 4, 5\}$ for G_1 and $\{a, b, c, d\}$ for G_2, are used solely for unique identification and do not represent vertex labels.

To help better explain the approach, we apply the McSplit algorithm to the two graphs, G_1 and G_2, represented in Figures 2a and 2b.

The McSplit algorithm starts from an initially empty mapping M between the vertex graphs G_1 and G_2. Then it incrementally extends M by adding one vertex pair at each recursion level. At the first level, the algorithm selects vertex $1 \in V(G_1)$ and vertex $a \in V(G_2)$, resulting in the partial mapping $M = \{(1, a)\}$. The algorithm then assigns labels to the unmatched vertices in both graphs. Each unmatched vertex in G_1 is labeled according to its adjacency to vertex 1, and each unmatched vertex in G_2 is labeled according to its adjacency to vertex a. A label of 1 indicates adjacency, whereas a label of 0 indicates non-adjacency. These labels guide the following node selection; we represent them in Table I(a).

TABLE I
LABELS (L) ON THE NON-MAPPED VERTICES (v) OF G_1 AND G_2.

G_1		G_2	
v	L	v	L
2	1	b	1
3	1	c	1
4	1	d	1
5	1		

(a) $M = \{1, a\}$

G_1		G_2	
v	L	v	L
3	10	c	11
4	10	d	11
5	11		

(b) $M = \{12, ab\}$

G_1		G_2	
v	L	v	L
3	100	d	111
4	101		

(c) $M = \{125, abc\}$

The algorithm extends the mapping M by adding a new pair of vertices with identical labels. McSplit adds the pair 2 and b during the second recursive step, resulting in $M = \{12, ab\}$. We summarize the associated label set in Table I(b).

In the third step, the mapping is extended with the pair 5 and c, since label class 11 is the only one common to both graphs. This process results in the mapping $M = \{125, abc\}$, with the corresponding label set presented in Table I(c).

Subsequently, we cannot add vertices 3, 4, and d to M due to label mismatches. As a result, the recursive procedure backtracks to explore alternative, potentially larger, mappings M.

Within each label class, also referred to as a domain, McSplit prioritizes nodes with the highest degree. This strategy enhances search efficiency, as matching vertices with larger neighborhoods is more likely to preserve adjacency constraints and reduce the number of discarded candidates. McSplit computes a pruning bound to minimize computational overhead by eliminating search branches that cannot yield a better solution than the best solution already found. Specifically, during the traversal of a branch, McSplit evaluates the following bound:

$$bound = |M| + \sum_{l \in L} min(|\{v \in G_1 \backslash M : label(v) = l\}| \ , \\ |\{w \in G_2 \backslash M : label(w) = l\}|) \tag{1}$$

Here, $|M|$ denotes the cardinality of the current partial mapping, and L is the set of labels among the unmatched vertices in both graphs. We estimate the maximum capacity for extending M by summing the minimum number of available vertices per label across both graphs. If the computed bound is less than the size of the best mapping found thus far, the current search path is unpromising, as it cannot yield a mapping larger than the best one found thus far. The algorithm prunes the corresponding branch of the search tree and then evaluates the next path, significantly reducing computational effort.

C. McSplit Variants and GPU-based Approaches

Liu et al. [22] introduce McSplit+RL using a reinforcement-learning inspired branching heuristic. The method defines a

reward for reaching leaves early and uses it to guide branching, so that promising mappings are explored first, thereby improving pruning by selecting better incumbents.

Jianrong et al. [23] propose McSplit+LL, adding on top of McSplit two new operators to reduce the search tree and match multiple vertices at once. The Long-Short Memory operator assigns a score to branching nodes by combining short-term rewards (per vertex in the smaller graph) with long-term rewards (per vertex pair across both graphs). Consequently, the search "learns" which regions of the search space are more promising over time. The Leaf vertex Union Match operator matches multiple leaf vertices adjacent to already-matched vertices in one step, effectively expanding the partial isomorphism by several vertex pairs at once.

Yanli et al. [24] introduce McSplitDAL, a McSplit-based branch-and-bound algorithm that uses a reinforcement-learning-inspired value function and a hybrid vertex-selection strategy to guide matching decisions, resulting in significantly smaller search trees. The value function accounts for both upper-bound reduction and domain simplification, updating scores for vertices and vertex pairs to favor actions that reduce the future search space. The hybrid selection combines learned scores with heuristic rules to balance exploitation and exploration.

Trimble [25] introduces McSplitSD, a sparse-graph variant of McSplit that redesigns data structures and propagation to exploit sparsity, significantly accelerating induced subgraph and maximum common induced subgraph computation on large sparse graphs. It uses partitioning data structures to avoid domain duplication, enabling efficient constraint propagation and bound computation. Additional sparsity-oriented optimizations improve pruning effectiveness compared to earlier approaches.

Existing exact algorithms are either theoretically strong but slow in practice, or fast (like McSplit) but without worst-case guarantees. Yu et al. [26] introduce RRSplit, a backtracking algorithm that reduces redundancy by pruning branches that would generate isomorphic solutions. It also incorporates maximality-based reductions and improved upper bounds to eliminate unpromising branches. The authors show that RRSplit matches the best-known worst-case complexity for MCS search.

While prior learning-based McSplit variants mainly improve branching, Kothalawala et al. [35] focus on tightening the upper bound. They propose McSplit+DSB, which uses degree-sequence-based bounds on each bidomain to achieve stronger pruning, quantified via a "bound gap." Since this bound is more expensive to compute, a reinforcement learning agent is used to determine where to apply it in the search tree to maximize efficiency.

To our knowledge, few papers have proposed GPU-based strategies for computing the MCS.

Jayaraj et al. [21] reformulate the problem as a Maximum Clique Enumeration problem. Unfortunately, it does not disclose its implementation, making a direct comparison challenging.

Quer et al. [31] serves as the foundation for our implementation; however, their GPU approach did not yield significant improvements due to poor synchronization.

Jiang et al. [32] introduce VSGM. This GPU-accelerated subgraph matching framework uses a view-based execution model and pipelined CPU-GPU processing to handle large graphs that exceed GPU memory. By streaming subproblems as views, it overlaps computation and data transfer, reducing communication overhead and idle time. The framework supports multi-GPU execution and minimizes load imbalance and synchronization costs.

Hu et al. [33] introduce GAMMA, an out-of-core GPU framework for graph pattern mining that extends device memory with host memory to handle graphs exceeding GPU capacity. It supports tasks such as frequent pattern mining and subgraph matching, using adaptive data access to manage large intermediate results. GAMMA provides high-level primitives that abstract out-of-core data movement and low-level GPU optimizations.

Unlike previous GPU-based frameworks, Hydra-MCS employs CPU-GPU collaboration primarily as a load-balancing mechanism rather than as an out-of-core execution strategy. The rationale and implementation details of this design are discussed in Section III-B.

III. Methodology

We divide this section into two units to explain both the improvements to the base algorithm and the integration of the GPU into Hydra-MCS. The first part explains our improvements to the parallelization strategy, focusing on better workload distribution and creating new tasks only when necessary. The second part presents our GPU implementation, outlining the different versions we evaluated and analyzing which information is most effective to exchange between the CPU and GPU.

A. Improving the Original CPU-based Parallelization Strategy

The original implementation of the parallel version of McSplit uses an inefficient parallelization scheme. To share the workload, idle threads perform "work-stealing" to reduce imbalance[1]. However, in this process, preparing and sharing data often result in redundant copies, leading to significant time loss from data duplication and unnecessary synchronization. To minimize overhead, the original authors limit data sharing to small, expandable mappings (i.e., sets of compatible node pairs). Specifically, they set a threshold that allows sharing mappings with fewer than five node pairs. This value was chosen as a "rule of thumb" to ensure the mappings were large enough to generate multiple tasks and keep all the threads busy, yet small enough to require sufficient work to explore the instance, thereby justifying full sharing. For mappings larger than this threshold, the algorithm switches to a single-threaded solver that does not share information. Generally speaking, setting the work-stealing threshold too low leads to over-subscription and low efficiency, as many threads may

[1]We reserve the term "work-stealing" for the original McSplit mechanism in which idle CPU workers obtain work from active workers, whereas "work-sharing" denotes the more general task-distribution mechanism used in Hydra-MCS.

finish early due to a lack of tasks to process. On the contrary, setting it too high creates excessive overhead that outweighs the benefits of improved load balancing.

We improve the original parallelization scheme to achieve better load balancing while controlling the overhead of task sharing. To this end, we introduce a regulated work-sharing policy combined with a lightweight estimate of the remaining computation (RC). Let $|M|$ denote the size of the current mapping and $|G_{min}|$ the size of the smaller graph. Task sharing is governed by both the mapping size and the state of a bounded queue:

- If $|M| < \frac{1}{4}|G_{min}|$ and the queue is not full, the task is always enqueued.
- If $|M| \geq \frac{1}{4}|G_{min}|$, the task is enqueued only if the queue is not full and its estimated RC exceeds a given threshold.

This policy prioritizes sharing small, partial solutions, which are more likely to generate substantial work, while limiting the propagation of larger tasks to avoid unnecessary synchronization overhead. The bounded queue further constrains task proliferation. To selectively share larger tasks, we introduce a lightweight metric to estimate the remaining search effort. Unlike previous approaches based on factorial terms [36], which are computationally expensive and prone to overflow, we define:

$$RC = \sum_{l \in L} \left(\begin{array}{l} |v \in G_1 \backslash M : label(v) = l| \cdot \\ |w \in G_2 \backslash M : label(w) = l| \end{array} \right) \quad (2)$$

This formulation approximates the branching potential of the search space by assigning higher weights to balanced bidomains, which typically require greater computational effort. Thus, if we have three bidomains in G_1 and G_2 of size $\{\{2,3\},\{4,4\},\{1,5\}\}$, we compute the following value $RC = 2{\cdot}3{+}4{\cdot}4{+}1{\cdot}5 = 27$. The RC threshold determines when a task is worth sharing. Excessively high values reduce sharing and lead to imbalance, whereas low values incur significant overhead. Empirically, we found that the following threshold provides robust performance across our dataset:

$$RC = \frac{|G_1| \cdot |G_2|}{2}. \quad (3)$$

Overall, the proposed strategy combines size-based and workload-based criteria, enabling effective load balancing while limiting synchronization and data-sharing costs.

Figure 3 intentionally isolates the CPU behavior to analyze the effect of the RC threshold independently from GPU offloading. In the hybrid implementation, GPU offloading can reduce CPU-side task imbalance and therefore makes overall performance less sensitive to the RC threshold, although GPU-side branch imbalance remains. The plots relate the RC threshold to CPU utilization and, indirectly, to runtime via its effects on load balancing and synchronization overhead. To magnify the impact of RC, we increased its value and removed the work-sharing threshold, thereby increasing the variance in thread termination times. However, we observe the same behavior even when the work-sharing threshold is present, though less pronouncedly. In Figure 3a, we select a too significant value for RC, namely 75% of $|G_1| \cdot |G_2|$. This choice causes several threads to finish after 4-8 seconds of execution, leaving only one thread still running after 10 seconds. Consequently, splitting the remaining task would improve occupancy and reduce runtime by allowing more parallel execution. In Figure 3c, we assign to RC a small value, namely 25% of $|G_1| \cdot |G_2|$. This decision causes even trivial tasks to be shared; thus, all cores spend excessive time managing the synchronization queue (running at 50-80% of their maximum capacity) rather than performing practical computation. This behavior is similar to the case in which there is no limit on the number of shared tasks. In Figure 3b, we set the value of RC to the value indicated in Equation 3. This selection enables effective task sharing. Not all threads finish exactly simultaneously, but more work is shared, keeping all cores active much longer. Moreover, threads do not share small tasks; thus, little time is wasted on unnecessary sharing.

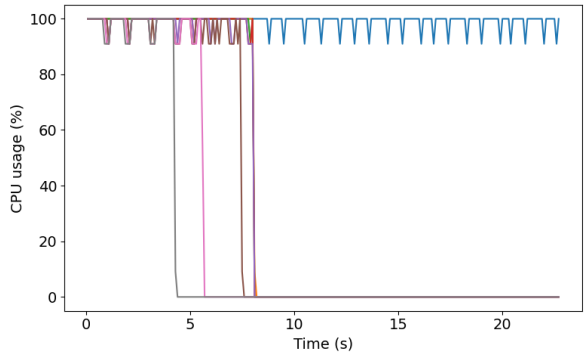
To summarize, our main contributions to the original CPU algorithm are the following:

- We limit the size of the shared task queue so that no more expandable matchings are shared when a substantial amount of work is already waiting. This recommendation saves a significant amount of time that we would otherwise spend on synchronization and enqueuing.
- We increase the upper limit on the matching size eligible for sharing. Combined with the previous strategy, this action yields a more balanced algorithm and a notable speedup.
- We propose a lightweight metric RC to approximate the remaining computational effort associated with the current matching. This insight enables the algorithm to share tasks it would otherwise not share, helping to address outliers that still cause imbalance despite the increased sharing limit.

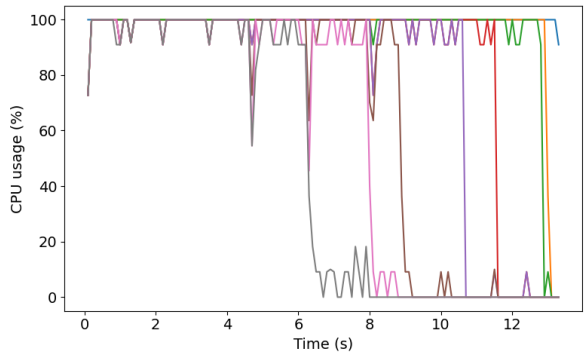
B. GPU implementation

Solving the MCS problem primarily on the GPU is challenging; a GPU-oriented implementation requires frequent synchronizations, and the resulting tasks are often unbalanced, leading to poor device utilization. In the earlier GPU-dominant implementation [31], the CPU generated partial matchings but delegated their solution entirely to the GPU, resulting in substantially lower performance than the hybrid method proposed here. To address these issues, we propose Hydra-MCS, a hybrid CPU-GPU solver that adopts a multi-kernel scheme and shares pruning information between the CPU and GPU. In our scheme, the GPU serves as a secondary device for processing groups of partial solutions in parallel. Our GPU algorithm is a non-recursive implementation of McSplit. Because the implementation targets medium-sized graphs, the per-thread search state can be represented within the preallocated GPU device-memory buffers. Only a small set of globally visible variables is required for CPU-GPU coordination. To further reduce the overhead of global memory, the slower communication channel, we never share subtasks between GPU threads.

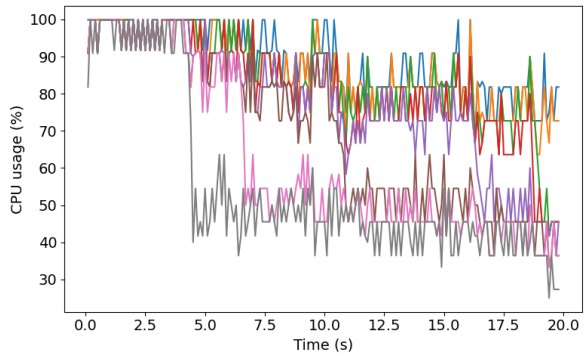
Implementing this procedure on the GPU requires strategic memory management. First, because the input graph descriptions are static, we can upload them once at initialization



(a) The RC threshold assumes a too high value.



(b) The RC threshold assumes a correct (balanced) value.



(c) The RC threshold assumes a too low value.

Fig. 3. CPU occupancy (with 8 threads) using different values of the RC threshold. Selecting RC in different ways implies different workload balancing strategies and code efficiency. Consequently, a good selection can improve the overall efficiency.

and never update them. Second, regarding partial matchings, there is a trade-off between memory usage and computational speed. Transmitting only the list of matched nodes minimizes memory overhead but forces the GPU to recompute bidomains. Conversely, including bidomain information increases memory consumption but significantly reduces redundant processing. Therefore, whenever sufficient memory is available, transferring both the matched nodes and bidomain data is preferable as it yields the fastest execution times.

The five configurations described below are successive designs developed by us during the construction of Hydra-MCS; they should not be interpreted as five schemes proposed in prior literature. The non-recursive GPU implementation of the underlying McSplit search builds on [31], whereas the CPU-GPU scheduling and synchronization mechanisms described below constitute the design evolution that this work investigates.

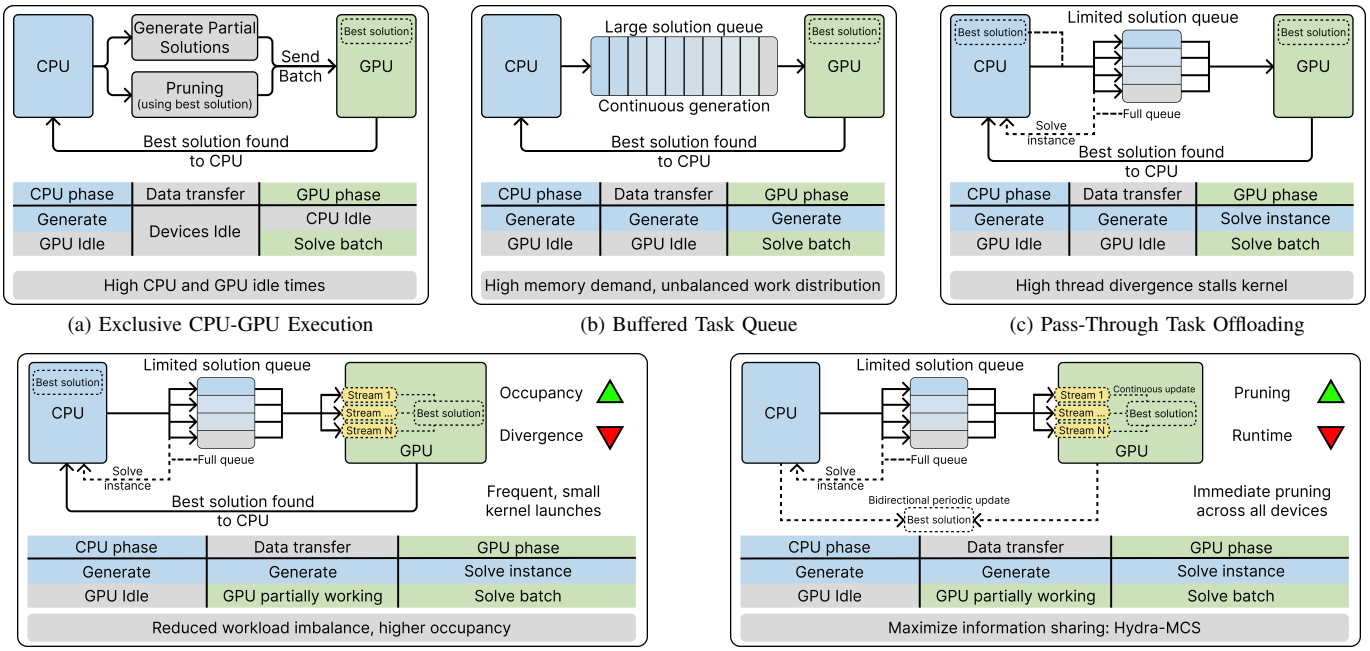
1) Exclusive CPU-GPU Executions: In our first approach, the CPU, after generating the block of partial solutions to offload to the GPU, waits for the GPU to solve the assigned tasks. This approach ensured that the largest solution found after each GPU kernel launch was always included in the final solution. Consequently, the algorithm efficiently prunes the search space, avoiding the transfer of unfit partial solutions to the GPU. However, the resulting CPU idle time constitutes a significant overhead that ultimately hinders the algorithm's overall performance.

2) Buffered Solutions: This methodology prioritizes continuously generating partial solutions, aiming to saturate the GPU with tasks without pausing to wait for previous execution cycles to finish. While this aggressive strategy successfully mitigates CPU idle time, it unfortunately introduces two substantial complications. The first is a resource constraint: the need to buffer such a vast queue of pending solutions imposes an exorbitant demand on system memory. The second issue is a significant synchronization imbalance: the CPU completes its generation tasks far more rapidly than the GPU can consume them, resulting in a backlog of work that persists long after the host processor has finished.

3) Buffer Solutions Pass-Through: In our third approach, the CPU prioritizes populating the GPU buffer before dedicating itself to solving the remaining search branches. This strategy ensures that both the host and device remain active concurrently, often resulting in significant runtime reductions. We also incorporated a mechanism to share the largest common subgraph found between the CPU and GPU. However, despite its promise, this implementation faces a fundamental architectural constraint: A GPU kernel cannot terminate until its final active thread completes. Because accurately predicting the computational depth of specific branches is infeasible, achieving optimal load balancing is nearly impossible. Consequently, this leads to high thread divergence, where the slowest threads stall the kernel.

4) Multiple GPU Kernels: To mitigate the impact of workload imbalance, we reduced the granularity of individual GPU kernel calls. This strategy ensures that fewer threads remain idle while waiting for the longest-running computations to complete. To compensate for the reduced parallelism within each specific kernel, we implemented a multi-stream approach, launching multiple kernels in parallel to process distinct batches of partial solutions. Although this architecture increases the frequency of memory operations between devices, the trade-off proved highly effective. Indeed, the slight performance penalty incurred by communication overhead is far outweighed by the substantial gains in hardware occupancy and overall execution speed.

5) Maximize Information Sharing: While propagating the size of the best-known solution is not strictly required to compute the MCS, it is a critical optimization for efficient pruning of the search space. To enable this optimization,



(a) Exclusive CPU-GPU Execution (b) Buffered Task Queue (c) Pass-Through Task Offloading

(d) Multi-Kernel GPU Execution (e) Final Hydra-MCS Architecture with Shared Incumbent Solution

Fig. 4. Successive Hydra-MCS design variants, i.e., logical evolution of CPU-GPU interaction strategies leading to the final architecture of Hydra-MCS. The diagrams reflect the sequential descriptions provided in the text, with accompanying notes at the bottom highlighting the salient characteristics of each strategy.

we utilize global memory rather than relying solely on on-chip shared memory. Shared memory is faster but limited to communication within a single thread block. In contrast, global memory is accessible to all GPU threads and the CPU, enabling universal synchronization. The latency overhead of global memory is a necessary trade-off: improving the lower bound by even a single node can reduce the algorithm's runtime by orders of magnitude, far outweighing the cost of memory access. Additionally, this approach ensures cache coherence between the host and the device, enabling bidirectional updates. Finally, the performance impact is further mitigated by CUDA's caching hierarchy, which effectively serves many of these read operations without requiring access to DRAM.

6) Section Summary: The design of Hydra-MCS was primarily driven by the need to mitigate GPU workload imbalance and synchronization overhead. Unlike in regular data-parallel applications, the computational effort required for each partial solution can vary by several orders of magnitude. Consequently, assigning a single partial solution per GPU thread naturally led to thread divergence, with some threads terminating quickly while others continued exploring substantially larger search subtrees.

This behavior is particularly evident in the early versions of our implementation. In the pass-through strategy, GPU kernels frequently remained active while only a small subset of threads continued executing. Since a kernel cannot terminate until all active threads complete, the longest-running threads effectively determine the kernel's completion time. This phenomenon motivated the introduction of multiple smaller kernels, each executed through an independent CUDA stream.

Another important consideration is the synchronization overhead between the CPU and the GPU. Although sharing the incumbent solution is not strictly required for correctness, it significantly improves pruning efficiency. The resulting reduction in search-space exploration largely outweighs the latency associated with accesses to globally visible memory. Consequently, the final implementation favors frequent propagation of incumbent updates between the host and the device.

As for memory bandwidth, we observed an effective host-to-device traffic rate of approximately 400 MB/s, well below the interconnect's theoretical peak bandwidth. This value reflects the demand generated by our application rather than a hardware bandwidth limitation. This value is given by the fact that graph structures are transferred only once during initialization. Conversely, runtime communication is limited to compact descriptions of partial solutions and incumbent updates. As discussed in Section IV-B, the average amount of transferred data per kernel is relatively small compared to the computational effort required to solve the associated search tasks. Therefore, execution efficiency is primarily constrained by workload imbalance and synchronization rather than by raw memory bandwidth.

The architecture we ultimately adopted, and diagrammatically illustrated in Figure 4, integrates the most effective optimizations discussed in the previous subsections. We discarded early prototypes that isolated the GPU for solving entire partial solutions, as they proved inefficient. Instead, the combination of the pass-through strategy and multiple parallel GPU kernels yielded substantial performance gains. Furthermore, maximizing information exchange between the CPU and GPU proved critical, significantly enhancing the pruning capabilities of both devices. Consequently, the final

version utilized in our experiments represents a synthesis of these high-impact contributions.

Table II provides a qualitative comparison of the five GPU parallelization strategies. The purpose of this comparison is to highlight the design trade-offs and the rationale that guided the evolution toward the final Hydra-MCS architecture. Since these strategies represent intermediate design stages rather than standalone configurations used in the final experimental campaign, the table focuses on their architectural characteristics, advantages, and limitations, rather than reporting detailed quantitative benchmarks for each variant.

Scheme	CPU-GPU Activation	Solution Sharing	Multi stream	Speedup (relative)
Exclusive CPU-GPU	Sequential	Yes	No	$\sim 1\times$
Buffered Solutions	Unbalanced	Delayed	No	1-1.5$\times$
Pass-Through	Parallel	Delayed	No	$> 2\times$
Multiple GPU Kernels	Parallel	Delayed	Yes	$> 2\times$
Final Hydra-MCS	Parallel	Yes	Yes	$\sim$4-5$\times$

TABLE II
THE TABLE ILLUSTRATES SUCCESSIVE HYDRA-MCS DESIGN VARIANTS THAT DIFFER IN THEIR DESIGN CHOICES REGARDING WORKLOAD BALANCE, SYNCHRONIZATION OVERHEAD, AND OVERALL EXECUTION EFFICIENCY COMPARED TO THE BASE CPU SCHEME.

IV. EXPERIMENTAL RESULTS

We organize this section into four parts. First, we describe our experimental setting, including the hardware and the benchmark used. After that, we compare the performance of our CPU and hybrid implementations with the original McSplit algorithm, evaluating both memory usage and time efficiency. We present data in different formats to highlight the strengths and weaknesses of our approach. Finally, we analyze the scalability of our method across four hardware configurations, varying both CPU and GPU, to assess the impact of each computing unit on the overall runtime.

A. Experimental Setting

We conduct our tests on a workstation with an Intel Core i9-10900KF (with 10 cores, 20 threads), 64 GB DDR4 RAM, and an NVIDIA GeForce RTX 4070 Super GPU. All code is written in C++ and CUDA (for the GPU implementation), and compiled with GCC 9.4.0 and NVCC 11.6, utilizing all available optimization options.

We run our experiments on the public ARG benchmark repository by Foggia et al. [37] and De Santo et al. [38]. The dataset includes graphs with varying numbers of nodes, edge densities, and topologies; moreover, these benchmarks are publicly available online[2]. We uniformly sampled all graph types described in De Santo's paper, i.e., randomly connected graphs, regular meshes (2D, 3D, and 4D), irregular meshes, bounded-valence graphs, and irregular bounded-valence graphs. The reader should refer to the original paper for more in-depth information on how they have been generated.

All experiments were conducted with a strict timeout of $3,600$ seconds (one hour). To ensure meaningful analysis, we restricted our selection to graphs with 40 to 50 vertices; instances outside this range tended to be either trivial (solving in under one second) or intractable within the time limit. Based on these constraints, we generated a uniform random sample of 272 graph pairs across all dataset categories. Finally, to mitigate variability caused by operating system scheduling, each experiment was repeated ten times, and we report the mean runtimes across these runs. While recorded for the analysis, run-to-run variability was negligible in percentage terms relative to the corresponding mean runtimes; thus, we omit detailed variance metrics.

All code is available on GitHub[3] under an open-source license to encourage further development and reproducibility.

B. Memory and Structural Considerations

A primary bottleneck in hybrid CPU-GPU computing is the latency of data transfer between devices. To mitigate this problem, our approach strictly decouples communication overhead from the memory required for the algorithm's internal state. We minimize host-to-device communication, capping peak VRAM usage for data transfers at 24MB across all evaluations. Furthermore, we rely heavily on preallocated buffers (reusing memory across successive kernel launches) to eliminate allocation overhead.

Conversely, the MCS computation primarily uses "bidomains" to store algorithmic information. These structures are significantly more memory-intensive and require up to 3GB of maintenance memory for the largest and most computationally intensive graph pairs.

Across our experiments, the average measured effective host-to-device transfer rate is approximately 400*MB/s*. This value characterizes the communication demand generated by Hydra-MCS and should not be interpreted as the peak bandwidth of the interconnect. Our strategy does not aim to saturate the bus performance, but rather to keep the GPU fully utilized by moving the required data on demand. This rate is strictly dependent on kernel runtime and can vary with problem complexity. On average, a single kernel requires a single transfer of ~ 300*KB*, or ~ 140*B* per thread. Although we could preallocate a large number of solutions to tackle only as the previous ones are solved, reducing the already small data transfer overhead, this could increase the runtime. If a search branch containing the optimal solution gets stuck in a task queue, the remaining threads could spend a long time on suboptimal solutions because the bound requirements are less strict.

To maximize GPU utilization, we also implemented multiple parallel streams. We achieved optimal performance by launching 64 concurrent streams, each executing a 64-block grid with 32 threads per block. This configuration deliberately oversubscribes the hardware, allowing the hardware scheduler to maximize throughput by interleaving warp execution and hiding memory latency.

[2]https://mivia.unisa.it/datasets/graph-database/arg-database/

[3]https://github.com/Lorenzo-Cardone/Hydra-MCS

C. Time Results

This section presents a comparative runtime analysis between our algorithm and the original McSplit procedure.

We focus our baseline comparison on the original McSplit algorithm for two primary reasons. First of all, it is the fastest publicly available implementation among the methods we evaluated on our benchmark under the reported configuration. Secondly, the lack of publicly available multi-threaded implementations for its recent heuristic (enhanced variants) such as McSplitRL [22], McSplitLL [23], McSplitDAL [24], McSplitSD [25], and RRSplit [26]. While we could have implemented parallel versions of these variants ourselves, doing so would introduce potential optimization biases, rendering the baseline comparison methodologically questionable. Furthermore, our preliminary testing revealed that while these heuristic variants successfully reduce search tree sizes, their heavy per-node computational overhead actually results in longer execution times than the original McSplit, especially on smaller graphs. Benchmarking against these slower variants would artificially lower the performance standard, granting our approach an unjustified advantage.

However, to ensure a comprehensive evaluation against the state of the art, we compare our tool with McSplitDAL and RRSplit[4], which represent the most recent sequential advancements for the MCS problem. While McSplitDAL's performance is modest (averaging roughly the same number of traversed branches as the baseline), RRSplit demonstrates exceptional search-space pruning, reducing the number of evaluated pairs by over $3\times$. However, this enhanced pruning comes at the expense of reduced iteration speed due to increased computational complexity. Consequently, McSplitDAL fails to compete with the original approach, suffering a $\sim 3\times$ slowdown. Conversely, RRSplit outpaces single-threaded McSplit by a marginal $1.05\times$ on 40-node graphs (underscoring that high iteration throughput remains vital), though its advantage widens to $\sim 1.2\times$ on 50-node instances. Ultimately, neither sequential approach can match the sheer throughput of our parallel 20-thread configuration, which yields increasingly pronounced speedups as instance sizes scale.

Finally, while the algorithmic enhancements in the McSplit variants (RL, LL, DAL, SD) and RRSplit are highly sophisticated, their execution profiles introduce high memory overhead and complex control flow. Although these methods can, in principle, benefit from coarse-grained parallel search, integrating their more stateful and computationally expensive branching and pruning mechanisms into the fine-grained GPU execution model considered here would require additional synchronization and per-thread state. We therefore leave such algorithm-specific hybrid implementations to future work.

Table III outlines the average runtime and CPU utilization under varying work-sharing parameters. We specifically consider the work-sharing threshold (as a function of the minimum graph size) and the task queue size (as a function of the thread count). Proper parameter tuning critically affects CPU efficiency, yielding up to a 20% speedup under optimal settings but causing an execution slowdown of up to $10\times$ when poorly configured. For the CPU-only implementation, peak performance is reached with a queue size limit equal to $5\times$ the number of threads and a work-sharing threshold set at 25% of the smallest graph's size. While the 25% threshold remains optimal for the hybrid variant, its performance is largely insensitive to changes in the task queue size. The hybrid version demonstrates greater robustness to parameter variation because the GPU effectively absorbs a substantial volume of generated tasks, thereby accelerating overall execution and stabilizing performance.

We also investigate the influence of the Remaining Computation (RC) threshold, a critical trigger for work-sharing and load balancing. Specifically, we evaluated our tool across four RC values: 10%, 25%, 50%, and 75% of the product $(|V_{G1}| \cdot |V_{G2}|)$. At the aggregate level, varying this parameter has a relatively limited impact on average runtime, with the 50% threshold outperforming the other configurations by approximately 5-6%.

The per-instance results, however, show that the sensitivity to the RC threshold is not uniform across the benchmark. While the choice of threshold has little effect on most instances, some graph pairs exhibit considerably larger runtime differences, reaching approximately 30-40% between configurations. Moreover, the threshold associated with the best performance is not uniform across all instances: some graph pairs are adversely affected by thresholds below 50%, whereas others perform worse at higher thresholds. This observation indicates that the effectiveness of the RC threshold is instance-dependent; however, the available results do not provide sufficient evidence to associate this behavior with a specific structural property of the input graphs.

Among the values considered, the 50% threshold provides the most robust overall trade-off. Although it is not the optimal choice for every individual graph pair, its average penalty when another threshold performs better is approximately 5%, and does not exceed 10% in our experiments. In contrast, alternative configurations can incur slowdowns of approximately 30-40% on some instances. We therefore select 50% as a conservative default that provides stable performance across the considered benchmark without requiring instance-specific parameter tuning.

Figure 5 shows the speedup of our approaches (CPU-only and CPU-GPU) compared to McSplit. We sort all instances by increasing baseline runtime. We configure the plot so that the baseline algorithm, i.e., McSplit, has a speedup of 1, whereas our approaches show their speedups relative to each instance.

Our hybrid CPU-GPU approach is not competitive for instances that require too little time, as the overhead of memory allocation and data transfer to the GPU outweighs any potential benefit. However, as the solution time increases, our algorithm achieves substantial speedups over the original, averaging 3-5$\times$ in the most challenging cases and reaching $12\times$ or more. The magnitude of the speedup depends strongly on the structure of the input graphs. The dashed trend line in the plot shows that our hybrid approach becomes increasingly advantageous as execution time increases, outperforming all other implementations on the most challenging instances.

[4]The source code for both approaches is available at the authors' repository: https://github.com/KaiqiangYu/SIGMOD25-MCSS (accessed April 1, 2026).

	Performances with varying work-sharing thresholds (% of smaller-graph size) [CPU]							
	5%		5 nodes		25%		50%	
Task Queue Size	Sol time (%)	CPU usage (%)	Sol time (%)	CPU usage (%)	Sol time (%)	CPU usage (%)	Sol time (%)	CPU usage (%)
1×	228	26	146	43	91	66	176	54
5×	179	35	105	61	81	78	404	46
10×	173	34	104	62	91	78	697	52
∞	176	34	100	61	94	77	1110	61
	Performances with varying work-sharing thresholds (% of smaller-graph size) [CPU+GPU]							
	5%		5 nodes		25%		50%	
Task Queue Size	Sol time (%)	CPU usage (%)	Sol time (%)	CPU usage (%)	Sol time (%)	CPU usage (%)	Sol time (%)	CPU usage (%)
1×	177	48	135	61	99	77	128	71
5×	158	51	114	67	89	85	159	66
10×	164	52	115	67	88	85	191	64
∞	156	51	100	67	86	84	245	65

TABLE III

AVERAGE RUNTIME AND CPU UTILIZATION ACROSS VARYING WORK-SHARING CONFIGURATIONS. THRESHOLDS ARE EVALUATED AS A FUNCTION OF THE MINIMUM GRAPH SIZE, AND THE NUMBER OF THREADS SCALES THE SIZES OF THE TASK QUEUES. PERFORMANCE IMPACTS ARE SHOWN FOR BOTH CPU-ONLY (TOP TABLE) AND HYBRID CPU-GPU (BOTTOM TABLE) IMPLEMENTATIONS.

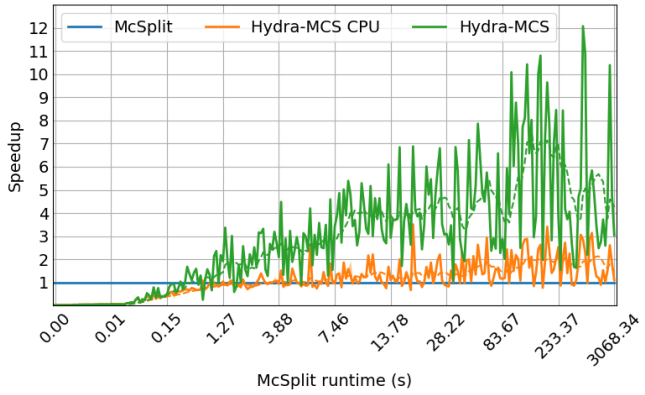


Fig. 5. Speedup of our CPU-only and CPU-GPU approaches compared to the original McSplit procedure (the fastest algorithm we tested for finding maximal solutions to the problems).

For experiments running under approximately 0.5 seconds, our CPU implementation is, on average, faster than the GPU implementation, but the speedup is negligible. In these cases, avoiding GPU offloading is preferable, as the synchronization and transfer times exceed any computational savings. Even so, the CPU-only version may still lag behind the original McSplit due to the extra logic required for task sharing. This approach provides no benefit when the RC is minimal.

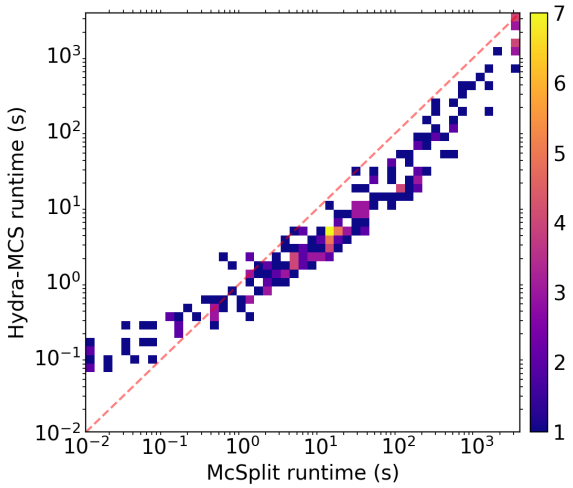
For more complex instances, our CPU-only implementation achieves an average speedup of about 2× over McSplit, peaking at 3.49×, making it a strong alternative when GPU resources are unavailable. While not as fast as the GPU implementation, it still outperforms the original, making it a better choice for difficult instances even without a CUDA-enabled GPU.

Figure 6 shows a scatterplot comparing the runtime of the original McSplit implementation and our hybrid one. Points above the dotted red diagonal line represent instances where McSplit is faster, while those below indicate cases where our hybrid approach is superior. As shown in Figure 6a, most instances that McSplit solves under one second appear above the red line, indicating that the hybrid implementation needs a longer time to complete the search. Consequently, if we focus on the instances requiring more than one second, we obtain the plot of Figure 6b. This graphic shows that as the required runtime increases, the hybrid implementation solves more instances faster, causing the scatterplot to shift below the red line.

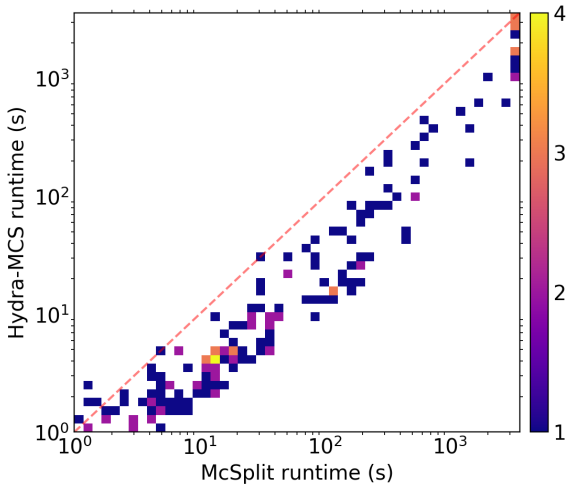
Based on our experiments, Hydra-MCS is preferable when the expected runtime of the original McSplit exceeds approximately one second. This one-second value is an empirical crossover observed in the comparison and is not an internal offloading threshold of Hydra-MCS. Below this threshold, CPU-GPU synchronization and task-offloading overheads may outweigh the benefits of GPU acceleration. For more computationally demanding instances, Hydra-MCS generally delivers superior performance and increasingly large speedups as problem difficulty increases.

Figure 7 provides a cumulative performance profile for the tested algorithms, mapping the elapsed time against the total count of successfully solved instances within the established timeout. To offer a granular analysis, we present the data in two distinct views. Figure 7a depicts the aggregate results for the entire dataset, including all graph pairs regardless of difficulty. Conversely, Figure 7b isolates the more computationally demanding cases; it filters the dataset to exclude trivial instances (specifically, those solved by the original baseline algorithm in under one second), thereby highlighting the performance differences on complex workloads.

The vertical distance between the curves represents the difference in the number of instances solved within a given timeout. In contrast, the horizontal distance indicates the additional time required by the other algorithm to solve the same number of instances (not necessarily the same ones). Consequently, both figures show that the gap between the two algorithms widens over time. Toward the end of the graph, the vertical gap narrows, but this is due to the scarcity of instances with similar solution times at such high runtimes. Because the problem's complexity increases rapidly, finding instances with comparable solution times beyond a certain point becomes difficult. This behavior is the same as what Figure 6 shows. After about 100 seconds, the number of solved

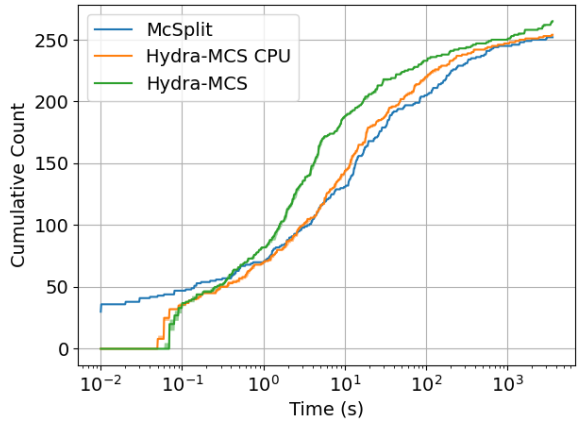


(a) All instances.

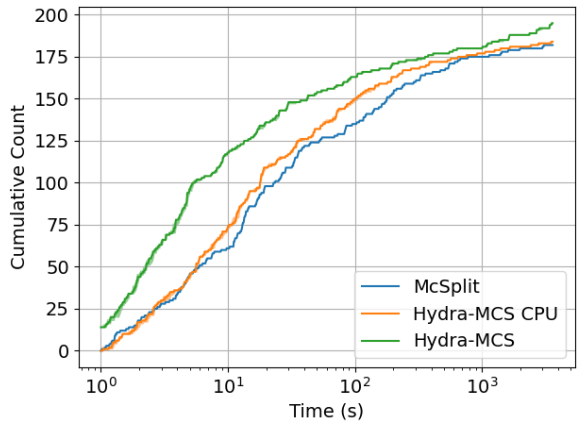


(b) Instances that McSplit solves in more than one second.

Fig. 6. Runtime distribution and acceleration profile. The top plot maps execution times across all evaluated instances. The bottom plot isolates complex instances (whose runtimes exceed one second for for McSplit), stripping away trivial cases to clearly expose the substantial performance gains unlocked by our approach.



(a) All instances.



(b) Instances that McSplit solves in more than one second.

Fig. 7. Cumulative number of solved instances.

graphs decreases significantly, making the plot sparser, and this is even more evident after about 1000 seconds. Crucially, the empirical evidence from both plots demonstrates that the proposed method maintains strict performance dominance on all significant instances requiring more than one second resolution time.

Throughout the cumulative performance profile, the original McSplit implementation never overtakes the proposed approaches. The persistent separation between the curves indicates that the observed speedups are not limited to a small subset of instances, but represent a systematic improvement across the benchmark as a whole. Indeed, at the 3600-second timeout, the gap remains: Out of 272 graph pairs, the original McSplit algorithm solved 252, while our hybrid approach solved 267 (including all those solved by McSplit plus 15 additional cases). Additionally, in one of the instances that neither algorithm solved (five overall), our implementation found a larger solution than McSplit.

Similarly, the CPU-only version of our algorithm also demonstrates improvements over the McSplit implementation. As with the hybrid approach, the curve for the solved instances shifts left, indicating that the same results are achieved in less time. In two cases, the new implementation solves instances that the original algorithm could not, given the timeout (note that the hybrid CPU-GPU approach also solves these instances within the time limit). Among the unsolved cases, our implementation once found a solution three nodes larger than the original, and once a solution one node smaller.

Figure 8 illustrates the proportion of instances that the new implementations successfully solve over time, specifically among those the original approach failed to complete. The hybrid version consistently solves 30-40% of the instances not yet solved by the original, peaking at 70% for the last few graph pairs as it approaches the timeout. Our CPU implementation also yields promising results, typically solving, on average, 10-20% of instances not yet solved by the original algorithm, demonstrating improved performance.

From a practical perspective, Hydra-MCS is most beneficial for medium- and difficult-sized instances where the baseline

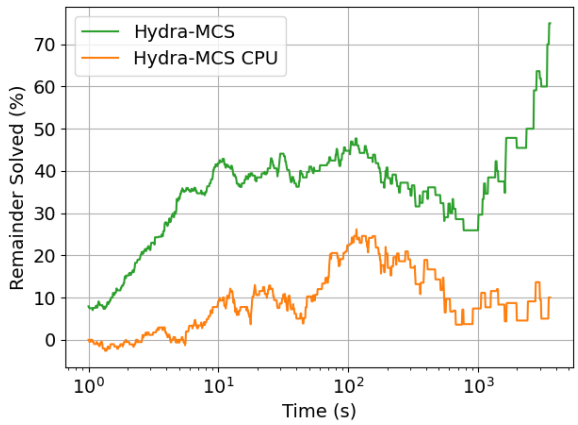


Fig. 8. Percentage of instances deemed intractable by the McSplit baseline that were successfully solved by competing approaches.

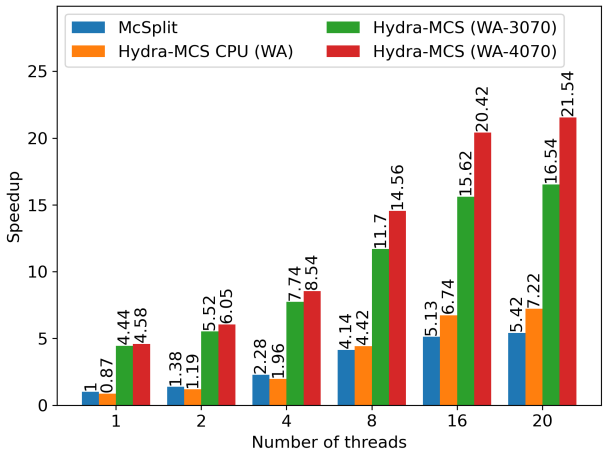
McSplit runtime exceeds one second. In simpler cases, CPU-GPU synchronization and transfer overhead may dominate execution time, making the original CPU-only McSplit preferable.

D. Algorithm scaling

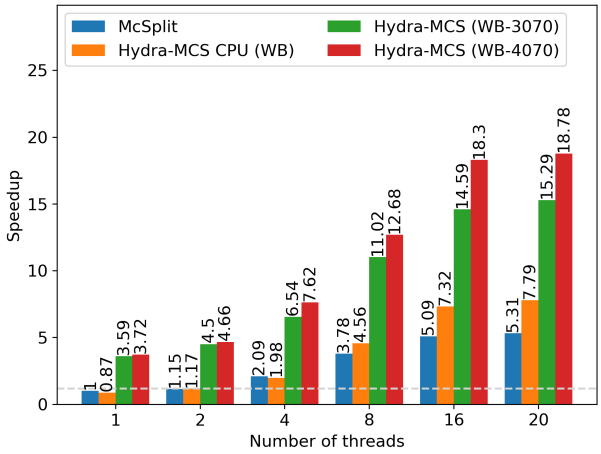
This section presents a detailed analysis of the algorithm's scalability, specifically examining the correlation between execution runtime and the number of allocated CPU threads. To isolate the specific impact of different architectural constraints and to better understand how underlying resources influence computational throughput, we conducted a rigorous evaluation across four distinct hardware configurations:

- **Original setup**: Intel Core i9-10900KF (with 10 cores and 20 threads), 64 GB DDR4 RAM, and NVIDIA GeForce RTX 4070 Super GPU. We call this configuration Workstation-A-4070 or WA-4070.
- **Older GPU**: Intel Core i9-10900KF (with 10 cores and 20 threads), but with an NVIDIA GeForce RTX 3070 GPU instead of the 4070 Super. We call this configuration Workstation-A-3070 or WA-3070.
- **Older CPU**: AMD Ryzen 2700x (with 8 cores and 16 threads), 32 GB DDR4 RAM, and NVIDIA GeForce RTX 4070 Super GPU. We call this configuration Workstation-B-4070 or WB-4070.
- **Older CPU and GPU**: AMD Ryzen 2700x (with 8 cores and 16 threads), 32 GB DDR4 RAM, and NVIDIA GeForce RTX 3070 GPU. We call this configuration Workstation-B-3070 or WB-3070.

Figure 9 shows the speedups of the original McSplit algorithm (*McSplit*), our CPU implementation (*Hydra-MCS CPU*), and our hybrid version (both with NVIDIA RTX 3070, *Hydra-MCS (3070)*, and NVIDIA RTX 4070 Super, *Hydra-MCS (4070)*), as the number of CPU threads used varies. Figure 9a presents the results obtained on the workstation with the Intel Core i9-10900KF CPU. Finally, Figure 9b shows the corresponding experiments on the AMD Ryzen 2700x configuration.



(a) Workstation A (3070 and 4070), with an Intel Core CPU.



(b) Workstation B (3070 and 4070), with an AMD CPU.

Fig. 9. Scalability analysis of the proposed CPU-only and hybrid CPU-GPU algorithms. The plot demonstrates performance as a function of the number of available computation threads, evaluated across different hardware configurations to highlight architectural impacts. The horizontal line represents the performance of Workstation A, which serves as the baseline, and it is $\sim 1.2\times$ faster than Workstation B.

For each configuration, we selected 10 sample graphs with long run times (over 10 seconds), and each test was repeated 10 times to minimize measurement uncertainty. We use McSplit running on a single thread as our baseline for evaluating all speedups.

As observed, our CPU-only implementation is less efficient than the original on extremely easy benchmarks, particularly when the number of threads is small (four or fewer). This phenomenon is due to the additional overhead inherited from the improved multithreaded parallelization logic. However, the new scheme's efficacy becomes increasingly evident with increasing parallelism, enabling our algorithm to surpass the original baseline as the thread count rises. The hybrid variant, in particular, capitalizes on the GPU's computational throughput to establish an immediate advantage. It achieves significantly superior performance metrics from a single CPU thread (thanks to device offloading) and maintains this upward trajectory, scaling effectively as additional host threads are provisioned. An in-depth analysis of the proposed methods suggests the following considerations. With few threads, the

CPU cannot generate enough partial solutions to saturate the GPU, leaving it underutilized. Conversely, when the number of threads increases and the GPU becomes the bottleneck, the CPU can continue to operate independently, processing the remaining branches and performing efficient pruning.

These dynamics are also confirmed by the data in Figures 9. Compared with the CPU-only implementation, the hybrid version exhibits lower scaling efficiency as the number of CPU threads increases, because the GPU already provides part of its computational capability even when only a few CPU threads are active. Nevertheless, the hybrid implementation maintains substantially higher absolute performance. Scaling is also more effective with a faster GPU and on systems where the CPU is less powerful relative to the GPU. This behavior suggests that the performance of the hybrid approach depends on the balance between host and device computational capabilities. As CPU performance increases relative to GPU performance, the GPU progressively becomes the limiting factor, and the marginal benefit of adding further CPU threads decreases.

To quantify these effects more clearly, it is important to distinguish between **absolute speedup** and **matched-thread speedup**. All speedups reported in Figure 9 are normalized to the execution time of McSplit using a single CPU thread. Such values therefore represent the overall acceleration of a given configuration relative to the sequential baseline. In contrast, a matched-thread comparison evaluates different implementations using the same number of CPU threads. It provides a more direct measure of the GPU's contribution and the CPU-side improvements introduced by Hydra-MCS.

Consider Workstation A equipped with the RTX 4070 Super. With 16 CPU threads, McSplit, Hydra-MCS CPU, and the hybrid Hydra-MCS configuration achieve absolute speedups of $5.13\times$, $6.74\times$, and $20.42\times$, respectively, relative to single-threaded McSplit. Holding the number of CPU threads fixed, the hybrid implementation is therefore approximately $(20.42/6.74 \simeq 3.03\times)$ faster than Hydra-MCS CPU and $(20.42/5.13 \simeq 3.98\times)$ faster than parallel McSplit.

The same behavior is observed with 20 CPU threads. In this configuration, McSplit reaches a speedup of $5.42\times$, Hydra-MCS CPU reaches $7.22\times$, and the hybrid Hydra-MCS implementation reaches $21.54\times$. Consequently, GPU integration provides an additional speedup of $(21.54/7.22 \simeq 2.98\times)$ over the Hydra-MCS CPU. In contrast, the complete hybrid implementation is $(21.54/5.42 \simeq 3.97\times)$ faster than McSplit using the same number of CPU threads. These results show that, under matched CPU resources, the GPU provides an additional performance improvement of approximately $3\times$ over our CPU-only implementation. In comparison, the complete Hydra-MCS system is approximately $4\times$ faster than parallel McSplit on this workload.

The larger speedup values reported in Figures 9 answer a different question. On Workstation A, Hydra-MCS CPU increases from $0.87\times$ with one CPU thread to $6.74\times$ with 16 threads. In contrast, the original McSplit implementation increases from the $1\times$ single-thread baseline to $5.13\times$ over the same range. The hybrid Hydra-MCS configuration with the RTX 4070 Super, instead, already achieves an absolute speedup of $4.58\times$ with one CPU thread and reaches $20.42\times$ with 16 CPU threads. Therefore, the $20.42\times$ value represents the **overall speedup of the complete hybrid system relative to single-threaded McSplit**. It combines the effects of CPU parallelism, the improved work-sharing strategy adopted by Hydra-MCS, and GPU offloading, and it should not be interpreted as the isolated contribution of the GPU. The matched-thread comparisons reported above provide the appropriate measure for the latter contribution.

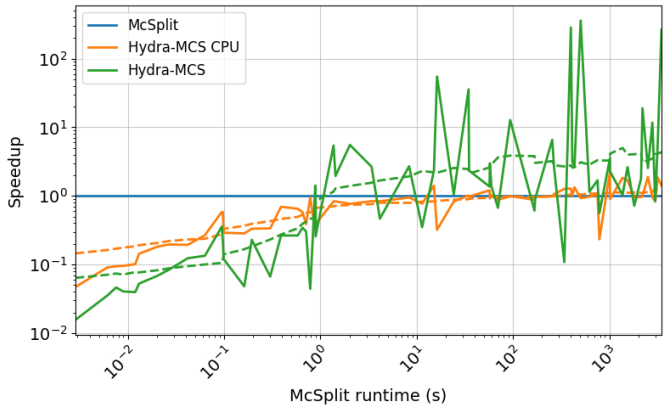


Fig. 10. Speedup of CPU-only and hybrid Hydra-MCS over multithreaded McSplit on the 66 solved graph pairs containing from 60 to 100 nodes. Solid curves show per-instance speedups; dashed curves show moving geometric means.

The results across the four hardware configurations further emphasize that effective CPU-GPU collaboration depends on the relative computational capabilities of the two devices. If the host CPU is insufficiently powerful, it cannot generate partial solutions quickly enough to enable the GPU to be utilized, resulting in full device underutilization. Conversely, if the GPU is underpowered relative to the host, it becomes the primary bottleneck: the CPU may spend computational effort preparing and transferring work to a device that processes it too slowly to provide a sufficient benefit. Hydra-MCS therefore achieves its best performance when the CPU can generate work at a rate that keeps the GPU active without making device execution the dominant bottleneck.

E. Scalability to Larger Graphs

We further evaluate graph-size scalability for more than 400 graph pairs with 60 to 100 nodes. We use the Workstation-A-4070 setup, the same algorithm as in the preceding experiments, and a $3,600$-second timeout, with multithreaded McSplit as the baseline. Only 66 graph pairs were solved within the timeout and could therefore be included in the exact speedup comparison. The remaining instances could not be included in the exact runtime-speedup comparison because at least one of the compared implementations did not complete within the one-hour timeout.

Figure 10 follows the presentation of Figure 5. All instances are ordered by increasing McSplit runtime, solid curves report per-instance speedups, and dashed curves show moving geometric means.

Consistent with the 40-50-node results in Figure 5, hybrid Hydra-MCS provides increasing average speedups as Mc-

Method	Runtime speedup	Node speedup
McSplitDAL	0.337×	0.737×
RRSplit	0.971×	2.591×

TABLE IV
GEOMETRIC-MEAN RUNTIME AND EXPLORED-NODE SPEEDUPS OVER SINGLE-THREADED MCSPLIT ON THE 12 SOLVED 60-100-NODE INSTANCES.

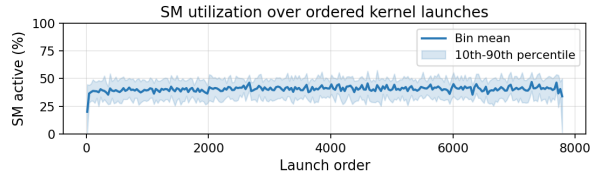
Split runtimes grow, peaking on the hardest solved instances. However, larger graphs exhibit higher variance. Individual instances range from modest slowdowns to speedups of up to 360×. These results indicate that the hybrid advantage can persist and, in some instances, increase with difficulty. In contrast, the CPU-only baseline remains more stable across runtimes, but its geometric mean stays close to baseline on large instances. Because the sheer computational weight of each branch naturally mitigates thread starvation, the load-balancing mechanism provides diminishing returns on these workloads. Larger graphs also demand more selective work-sharing. At 100 nodes, we restrict task sharing to mapping depths less than or equal to 10 pairs (10% of graph size) and increase the RC threshold to 75% of $|G_1| \cdot |G_2|$. Permissive thresholds over-generate small tasks and trap workers in narrow subtrees; selectively offloading branches with substantial remaining computation better preserves search diversity.

Among the graph pairs excluded from the exact speedup comparison, McSplit solved one instance close to the 3,600-second timeout that neither Hydra-MCS configuration completed. In contrast, hybrid Hydra-MCS solved one instance in approximately 20 seconds for which McSplit timed out. For the instances left unsolved by all approaches, the best incumbent solution at timeout was identical in the large majority of cases. Differences were rare: the original McSplit implementation occasionally found a larger solution, while such differences occurred somewhat more often in favor of our hybrid implementation. Overall, the timeout-limited results are broadly comparable, with a slight tendency to favor hybrid Hydra-MCS when incumbent sizes differ.

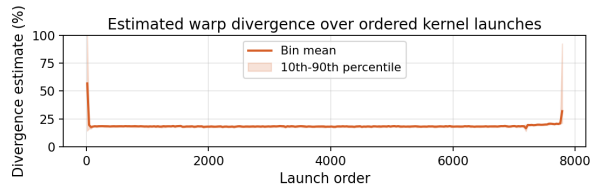
1) Recent Sequential Methods: We also compare the released sequential McSplitDAL and RRSplit implementations with single-threaded McSplit on all 12 60-100-node instances solved by all three approaches within the $3,600$-second timeout. All other tested instances remained unsolved by all three approaches. Table IV reports geometric means; values above one indicate faster execution or fewer explored nodes than McSplit.

McSplitDAL is approximately 2.96× slower than McSplit and explores about 1.36× more nodes. RRSplit explores approximately 2.59× fewer nodes, yet remains about 3% slower. Consistent with the 40-50-node results, this comparison shows that stronger pruning does not necessarily reduce runtime because its per-node overhead can offset the smaller search tree. Nonetheless, the small eligible sample size prevents broad generalization of performance.

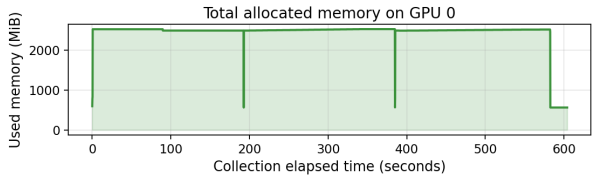
2) GPU Profile and Memory Scaling: We profile a representative long-running Hydra-MCS workload on the RTX 4070 Super under Ubuntu using NVIDIA Nsight Compute. Figure 11 preserves kernel-launch order and aggregates contiguous launches into bins. The SM-utilization and estimated-divergence curves report bin means, and the bands report the 10th-90th percentiles.



(a) SM utilization.



(b) Estimated warp divergence.



(c) Total allocated device memory.

Fig. 11. GPU profile of a representative long-running Hydra-MCS workload. Figure (a) streaming-multiprocessor (SM) utilization. Figure (b) estimated control-flow divergence proxy, i.e., an indication of of branch-path non-uniformity, consistent with the expected irregular control flow of branch-and-bound search. Figure (c) total allocated device memory.

Figure 11a shows sustained SM activity around 40%, with 10th-90th percentiles typically between 30-50%. Even with oversubscribed streams, irregular task granularities, and control-flow divergence, substantial compute headroom remains because irregular task sizes and control flow do not expose uniformly parallel work. Figure 11b corroborates this evidence. A persistent non-zero mean and occasional spikes reflect frequent thread divergence. This value is an estimate defined as 100 minus Nsight Compute's branch-target uniformity percentage, rather than a direct warp-divergence measurement.

Figure 11c reports a consistent occupation of < 3 GiB while the application is active. In fact, Hydra-MCS currently relies on statically preallocated GPU buffers for bidomain state, reserving a conservative worst-case capacity of approximately $(|G_1| \cdot |G_2|)/2$ entries per structure. This deliberately conservative structure accommodates the worst-case bidomain-partitioning scenario. Although a single bidomain requires only 32B of memory to represent, as both graph sizes grow together, the total device requirement grows quadratically; the total cost further scales with the number of concurrently preallocated structures.

Fixed provisioning for graphs with at most 100 nodes explains why GPU memory usage is expected to remain nearly flat across the 40-100-node experiments, as observed for most tested instances, while still imposing a VRAM scaling limit beyond that range.

Two complementary strategies could alleviate this memory bottleneck:

- Cooperative execution: A warp or thread block could share a single bidomain allocation for a task rather than maintaining per-thread worst-case copies, trading intra-group synchronization and task-level concurrency for a drastically lower memory footprint.
- Dynamic pooled storage: Tasks could allocate variable-length slices from a reusable device memory pool based on active bidomain counts, reclaiming regions upon backtracking and reserving an overflow path for rare worst-case partitions.

Compact vertex indexing and host-backed fallback buffers offer further avenues to reduce device memory pressure.

V. Conclusions and Future Works

In this work, we present Hydra-MCS, an algorithm for the Maximum Common Subgraph problem specifically designed to fully exploit CPU parallelism and the computational power of a modern GPU. We base our implementations on an estimate of the computational work required to achieve the desired solution in the future. We use this metric to determine when the RC is sufficient to warrant spawning a new task and offloading it to another execution unit, rather than processing it locally, thereby maintaining load balancing and minimizing unnecessary task sharing between threads. Our algorithms perform optimally for small- to medium-sized graphs, where computing an exact solution is feasible within a practical time frame. On the scaling workload, with an equal allocation of 20 CPU threads, GPU integration provides approximately a $3\times$ speedup over Hydra-MCS CPU and approximately a $4\times$ speedup over parallel McSplit; the larger absolute speedups relative to single-thread McSplit reflect the combined effects of CPU parallelism and GPU acceleration.

Our primary exact-runtime evaluation focuses on graphs with 40-50 nodes, for which a sufficiently large number of instances can be solved within the one-hour timeout. To investigate behavior beyond this regime, we additionally evaluated more than 400 graph pairs containing 60-100 nodes. Only 66 of these instances were solved to optimality by both the multithreaded McSplit baseline and the compared Hydra-MCS configurations within the timeout and could therefore be included in the exact runtime-speedup comparison. These results confirm that Hydra-MCS can retain substantial advantages on larger instances, while also showing that exact MCS computation rapidly becomes impractical as graph size increases and that performance becomes increasingly instance-dependent.

Notwithstanding the encouraging results of this study, inherent limitations persist, most notably in scalability. Indeed, the prospect of extending our current methodology to accommodate significantly larger graph structures introduces unique computational hurdles. These constraints, while challenging, serve as a ground for further investigation and define the primary objectives for our future work:

- As the graph size increases, the current buffer-based approach becomes inefficient: Partial solutions scheduled for the GPU may remain in the buffer for too long without being processed. Replacing the buffer with a priority queue accessible by both the CPU and GPU could improve task distribution and responsiveness.
- For larger graphs, GPU memory usage may become a critical resource. Grouping threads into blocks or warps to solve a single instance collaboratively, rather than assigning one instance per thread, could help manage memory and improve efficiency.
- Since it is often infeasible to solve large graphs exactly within reasonable time limits, integrating advanced heuristics from recent literature could enable more aggressive search-space pruning or guide the algorithm toward high-quality solutions, even if it cannot explore the entire space.

Additionally, other works propose heuristics for dynamically selecting the order in which vertices are processed or the graph to branch from at each step, rather than relying on a fixed order determined at the start of execution. Incorporating such strategies could further enhance the performance and the solution quality.

Finally, our current implementation launches a new GPU kernel for each batch of solutions. Although this solution is simple and effective for synchronization, it introduces significant overhead. A more advanced design (in which a single kernel remains active and threads fetch new work from an internal queue) could reduce this overhead and improve overall efficiency. However, implementing such a scheme would require substantial changes to the underlying synchronization mechanisms and is considered a possible future line of investigation.

To summarize, while our approach yields strong results on small and medium-sized graphs, addressing the aforementioned challenges is crucial for scaling to larger instances and improving the efficiency and applicability of Hydra-MCS.

References

[1] Mariya I Skvortsova, Igor I Baskin, Ivan V Stankevich, Vladimir A Palyulin, and Nikolai S Zefirov. Molecular Similarity. 1. Analytical Description of the Set of Graph Similarity Measures. Journal of chemical information and computer sciences, 38(5):785–790, 1998.

[2] Maureen Heymans and Ambuj K Singh. Deriving Phylogenetic Trees from the Similarity Analysis of Metabolic Pathways. Bioinformatics, 19(suppl_1):i138–i146, 2003.

[3] Björn H Junker and Falk Schreiber. Analysis of Biological Networks, volume 2. Wiley Online Library, 2008.

[4] Andrew Dalke and Janna Hastings. Fmcs: A novel algorithm for the multiple mcs problem. Journal of Cheminformatics, 5:O6, 2013.

[5] Stanley Milgram. The Small World Problem. Psychology today, 2(1):60–67, 1967.

[6] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. Journal of the ACM (JACM), 46(5):604–632, 1999.

[7] Younghee Park and Douglas Reeves. Deriving common malware behavior through graph clustering. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS '11, page 497–502, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Jochem H. Rutgers, Pascal T. Wolkotte, Philip K.F. Hölzenspies, Jan Kuper, and Gerard J.M. Smit. An approximate maximum common subgraph algorithm for large digital circuits. In 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pages 699–705, 2010.

[9] Thomas Zimmermann and Nachiappan Nagappan. Predicting subsystem failures using dependency graph complexities. In The 18th IEEE International Symposium on Software Reliability (ISSRE '07), pages 227–236, 2007.

[10] SG Shirinivas, S Vetrivel, and NM Elango. Applications of graph theory in computer science an overview. International journal of engineering science and technology, 2(9):4610–4621, 2010.

[11] R Morpurgo. Un metodo euristico per la verifica dell'isomorfismo di due grafi semplici non orientati. Calcolo, 8(1):1–31, 1971.

[12] Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo, 9(4):341–352, December 1973.

[13] Bohdan Zelinka. On a Certain Distance between Isomorphism Classes of Graphs. Časopis pro pěstováni matematiky, 100(4):371–373, 1975.

[14] Julian R Ullmann. An algorithm for subgraph isomorphism. Journal of the ACM (JACM), 23(1):31–42, 1976.

[15] Harry G. Barrow and Rod M. Burstall. Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques. Inf. Process. Lett., 4(4):83–84, 1976.

[16] Michael M Cone, Rengachari Venkataraghavan, and Fred W McLafferty. Computer-aided interpretation of mass spectra. 20. molecular structure comparison program for the identification of maximal common substructures. Journal of the American Chemical Society, 99(23):7668–7671, 1977.

[17] James J McGregor. Backtrack search algorithms and the maximal common subgraph problem. Software: Practice and Experience, 12(1):23–34, 1982.

[18] Ciaran McCreesh, Patrick Prosser, and James Trimble. Heuristics and Really Hard Instances for Subgraph Isomorphism Problems. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI'16, pages 631–638. AAAI Press, 2016.

[19] Philippe Vismara and Benoît Valery. Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms. In Communications in Computer and Information Science, pages 358–368. Springer Berlin Heidelberg, 2008.

[20] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In Lecture Notes in Computer Science, pages 350–368. Springer International Publishing, 2016.

[21] P. B. Jayaraj, K. Rahamathulla, and G. Gopakumar. A gpu based maximum common subgraph algorithm for drug discovery applications. In 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 580–588, 2016.

[22] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. A learning based branch and bound for maximum common subgraph related problems. Proceedings of the AAAI Conference on Artificial Intelligence, 34(03):2392–2399, Apr. 2020.

[23] Jianrong Zhou, Kun He, Jiongzhi Zheng, Chu-Min Li, and Yanli Liu. A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem, 2022.

[24] Yanli Liu, Jiming Zhao, Chu-Min Li, Hua Jiang, and Kun He. Hybrid learning with new value function for the maximum common subgraph problem, 2022.

[25] James Trimble. Partitioning algorithms for induced subgraph problems. PhD thesis, University of Glasgow, 2023.

[26] Kaiqiang Yu, Kaixin Wang, Cheng Long, Laks Lakshmanan, and Reynold Cheng. Fast Maximum Common Subgraph Search: A Redundancy-Reduced Backtracking Approach. Proceedings of the ACM on Management of Data, 3:1–27, 06 2025.

[27] Yunsheng Bai, Derek Xu, Ken Gu, Xueqing Wu, Agustin Marinovic, Christopher Ro, Yizhou Sun, and Wei Wang. Neural Maximum Common Subgraph Detection with Guided Subgraph Extraction, 2020.

[28] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. GLSearch: Maximum Common Subgraph Detection via Learning to Search. In Marina Meila and Tong Zhang, editors, Proceedings of the 38th International Conference on Machine Learning, volume 139 of Proceedings of Machine Learning Research, pages 588–598. PMLR, 18–24 Jul 2021.

[29] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. A Learning Based Branch and Bound for Maximum Common Subgraph Related Problems. Proceedings of the AAAI Conference on Artificial Intelligence, 34(03):2392–2399, April 2020.

[30] Ciaran McCreesh, Patrick Prosser, and James Trimble. A partitioning algorithm for maximum common subgraph problems. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17, IJCAI'17, pages 712–719. AAAI Press, 2017.

[31] Stefano Quer, Andrea Marcelli, and Giovanni Squillero. The maximum common subgraph problem: A parallel and multi-engine approach. Computation, 8(2):48, 05 2020.

[32] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. VSGM: View-Based GPU-Accelerated Subgraph Matching on Large Graphs. SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15, 2022.

[33] Lin Hu, Lei Zou, and M Tamer Özsu. GAMMA: A Graph Pattern Mining Framework for Large Graphs on GPU. In 2023 IEEE 39th International Conference on Data Engineering (ICDE), pages 273–286. IEEE, 2023.

[34] Andrea Calabrese, Lorenzo Cardone, Salvatore Licata, Marco Porro, and Stefano Quer. A web scraping algorithm to improve the computation of the maximum common subgraph. In International Conference on Software and Data Technologies, 2023.

[35] Buddhi W. Kothalawala, Henning Koehler, and Qing Wang. Learning to Bound for Maximum Common Subgraph Algorithms. In Maria Garcia de la Banda, editor, 31st International Conference on Principles and Practice of Constraint Programming (CP 2025), volume 340 of Leibniz International Proceedings in Informatics (LIPIcs), pages 22:1–22:18, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[36] Lorenzo Cardone and Stefano Quer. The multi-maximum and quasi-maximum common subgraph problem. Computation, 11(4), 2023.

[37] P. Foggia, C. Sansone, and M. Vento. A Database of Graphs for Isomorphism and Sub-Graph Isomorphism Benchmarking. In -, page 176–187, 2001.

[38] M. De Santo, P. Foggia, C. Sansone, and M. Vento. A Large Database of Graphs and its Use for Benchmarking Graph Isomorphism Algorithms. Pattern Recogn. Lett., 24(8):1067–1079, May 2003.

AUTHOR BIOGRAPHY

Lorenzo Cardone received the M.Sc. degree in Computer Science from Politecnico di Torino in 2021, where he is currently a Ph.D. candidate in Computer and Control Engineering. His research focuses on graph problems, software optimization, and parallel algorithms, with particular emphasis on multi-core and many-core architectures.

Stefano Quer received the M.S. degree in Electronic Engineering in 1991 and the Ph.D. degree in Computer Engineering in 1996. He is currently a Professor with the Department of Control and Computer Engineering at Politecnico di Torino, Italy. During his career, he has been a Visiting Faculty member at the University of California, Berkeley, and has gained industrial research experience at Digital Equipment Corporation, Intel Corporation, Synopsys Inc., and Compaq Computer Corporation, where he also served as a consultant. His research interests include computer-aided design for VLSI systems, formal methods and testing for hardware and software verification, and sequential and concurrent algorithms for solving complex problems under limited computational resources.