Profiling Application-Specific Properties of Irregular Graph Algorithms on GPUs

Bishal Sharma
Texas State University
San Marcos, Texas, USA
dxa6@txstate.edu

Martin Burtscher
Texas State University
San Marcos, Texas, USA
burtscher@txstate.edu

ABSTRACT

With the availability of sophisticated profiling tools for GPUs such as NVIDIA's Nsight Compute and Nsight Systems, programmers tend to overlook the level of insight that can be gained from simple profiling techniques. For instance, the basic profiling approach of manually adding counters to source code is able to expose important application-specific behavior that general-purpose profilers cannot capture. Analyzing global or thread-local counts of certain events can help developers better reason about program behaviors that are crucial for detecting performance bottlenecks, validating key assumptions, and guiding effective optimizations. In this paper, we demonstrate on the example of 5 high-performance GPU graph-analytics codes how we used this profiling approach to uncover interesting application behaviors and to develop performance optimizations based on some of them.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; • **Software and its engineering** → *Software performance*.

KEYWORDS

GPU graph analytics, irregular applications, parallel algorithms

ACM Reference Format:
Bishal Sharma and Martin Burtscher. 2025. Profiling Application-Specific Properties of Irregular Graph Algorithms on GPUs. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3731599.3767443

1 INTRODUCTION

Graphics Processing Units (GPUs) are widely used to accelerate regular codes—applications with predictable memory access patterns and control flow [23]. Many regular codes operate on highly data-parallel workloads, allowing the same operation to be applied across many data elements with minimal synchronization [25]. For example, in image processing, fixed-size arrays enable threads to independently compute their share in parallel. This predictability in data access and execution flow makes regular codes well-suited for GPUs, often delivering significant speedups over multicore CPUs [14]. Due to the widespread use of regular codes, profilers for GPUs have been built primarily with such programs in mind.

Irregular codes, in contrast, have less predictable data access patterns and control flow. They are characterized by the use of irregular data structures such as graphs, trees, and hash tables. Graphs, for instance, may have an arbitrary number of vertices and edges. Since different vertices can have widely differing degrees, some require substantially more computation than others, making efficient load balancing a challenge [21]. The behavior of irregular programs is thus dynamic and can vary significantly between inputs. Moreover, the irregularity in the data processing pattern leads to unpredictable control flow [13]. Due to this unpredictability, it is often challenging to exploit parallelism in irregular codes [32].

Nevertheless, irregular codes are becoming prevalent in problem domains like scientific simulations [8, 29], prediction and forecasting [35], data mining [37], social network analysis [20], compiler design [2], system modeling [31], and so on. Irregular data structures can more efficiently represent the underlying problem in these domains. However, high-performance parallel implementations of irregular codes generally require program-specific optimization and parallelization, necessitating a thorough understanding of the program behavior. Profiling is crucial in this context. Unfortunately, existing profiling tools focus on general performance information that tends to have less relevance for irregular codes.

Profilers assess the performance of a program by collecting information related to its execution, like function and kernel runtimes, memory allocations and deallocations, data transfer rates, register usage, etc. With these tools, the behavior of regular codes can often be easily understood as their execution typically remains consistent across, for example, the execution of a loop or a function. However, the same cannot be said for irregular codes, whose behavior can vary significantly even within a single loop. Moreover, standard profilers are limited in their ability to provide application-specific information. For instance, the number of times a particular computation step conflicts with another is information that standard profilers do not capture. Yet, such information can help programmers gain key insight into the operation of critical code sections and hence find clues to better parallelize or optimize them.

We investigate the merits of profiling irregular CUDA codes by manually inserting counters into each of the five graph analytics codes from the ECL suite [26, 27]: Connected Components (CC) [22], Graph Coloring (GC) [3], Maximal Independent Set (MIS) [12], Minimum Spanning Tree (MST) [17], and Strongly Connected Components (SCC) [4]. We selected this suite because it contains irregular codes that deliver state-of-the-art performance on GPUs. It is important to note that these implementations are already highly

SC Workshops '25, November 16–21, 2025, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767443

optimized, guided by sophisticated profiling tools, and therefore leave little opportunity for further improvement.

We embed counters, either per-thread or cumulative depending on need, into the source code to analyze kernel-specific operations. The intent is to verify that each kernel behaves as expected for the given input. In cases where the collected metrics reveal unexpected statistics, we investigate the underlying cause and, where meaningful, devise optimizations based on our findings. We then implement these optimizations and evaluate their impact. Although analyzing code using manually inserted counters is a well-established practice, we demonstrate that this simple approach remains effective for uncovering additional insights, even in highly optimized codes.

The counters expose interesting behavior not captured by standard profilers in all five programs as highlighted in the result section. For three of the programs, they helped us devise optimizations to improve the performance even though the codes in the ECL suite are already highly optimized. Our modifications to the MST code only yields an average improvement of 0.14%. The SCC code shows both performance gains and losses, with speedups as well as slowdowns of up to 27%, depending on the input. Finally, the CC code exhibits runtime improvements ranging from 0% to 16%, illustrating that simple manual profiling can reveal otherwise missed information.

A key point to note is that, while manual profiling may not always translate into actionable optimizations, it consistently provides useful insight into the behavior of irregular programs. These insights lead to tangible performance improvements in some codes and shed light on the inner workings of others. That said, this work makes the following main contributions.

- We manually instrument the CUDA source codes of five graph analytics programs with counters to study application-specific behavior that general-purpose profilers cannot catch.
- We statistically and visually analyze the code-specific metrics to reveal interesting and unexpected behavior of irregular programs that has never before been described.
- We draw conclusions from our analysis and discuss useful insights regarding program performance.
- We devise optimizations based on these insights that yield speedups even for state-of-the-art codes.

The rest of the paper is organized as follows. Section 2 reviews the five irregular codes we study. Section 3 describes the code-specific events we profile and the metrics we use to evaluate them. Section 4 summarizes related work. Section 5 presents the experimental methodology. Section 6 analyzes the program behavior based on the profiling results and discusses optimizations. Section 7 summarizes our findings and draws conclusions.

2 CODES INSTRUMENTED

This section reviews the five irregular CUDA codes from the baseline ECL suite that we profiled for this study.

2.1 Connected Components (CC)

The CC algorithm identifies CCs in an undirected graph, where a CC is a maximal set of vertices such that there exists a path between any two vertices in the set. ECL-CC is a high-performance GPU implementation for computing CCs. It uses a union-find data structure to efficiently manage and merge components. The algorithm has three main stages: initialization, computation, and finalization.

In the initialization stage, each vertex is initialized with the ID of the first neighbor in its adjacency list that has a smaller ID; if no such neighbor exists, it keeps its own ID. This heuristic leads to less work in the next phase compared to just using the vertex ID.

The main computation stage runs asynchronously and consists of 3 CUDA kernels that are customized for different vertex degrees (low, medium, and high) to balance the load across the threads. Each kernel performs union-find operations using intermediate pointer jumping. It interleaves hooking (union) and pointer jumping and uses atomicCAS to update parent pointers without locking.

The finalization stage applies a final pointer-jumping pass to ensure each vertex points directly to its component representative.

2.2 Graph Coloring (GC)

Graph coloring is the process of assigning colors to the vertices of an undirected graph so that no two adjacent vertices share the same color and a minimal number of colors is used. ECL-GC is a fast GPU-based graph-coloring algorithm that increases parallelism using shortcut techniques without compromising coloring quality. It has two main stages of operation: initialization and coloring.

The initialization stage prepares the input graph by imposing a vertex ordering based on the Largest-Degree-First (LDF) heuristic. This converts the undirected graph into a Directed Acyclic Graph (DAG), where edges point from higher-priority (higher-degree) to lower-priority vertices. Each vertex is assigned a bitmap representing its possible colors, with the number of possible colors initialized based on its in-degree. This setup defines the partial ordering needed for the Jones-Plassmann (JP) coloring strategy.

In the coloring stage, the algorithm colors the vertices in parallel while using two optimizations called shortcuts. These shortcuts safely relax the strict dependency on higher-priority neighbors by using the bitmaps' contents to infer when it is safe to color a vertex earlier than normally allowed. Shortcut 1 allows coloring a vertex if its best possible color is no longer under consideration by any higher-priority neighbor. Shortcut 2 removes dependency edges when the sets of possible colors for two connected vertices no longer overlap, further increasing parallelism.

2.3 Maximal Independent Set (MIS)

The MIS of a graph is a set of non-adjacent vertices that cannot be extended by including any other vertex. The ECL-MIS code combines aspects of Luby's random-selection and random-permutation algorithms and adds optimizations to achieve high performance.

The ECL-MIS implementation operates in two stages: initialization and selection. In the initialization stage, each vertex is assigned a compact, one-byte value encoding both its status and priority. The priority is based on a function that favors low-degree vertices and uses vertex IDs to break ties. This setup creates a deterministic partial permutation that boosts the MIS size while minimizing memory usage and avoiding the need for separate status and priority arrays.

In the selection stage, threads asynchronously process undecided vertices. If a vertex has the highest priority among its neighbors, it is marked "in" and its neighbors are set to "out." The algorithm uses

short-circuit checks to reduce work and requires no synchronization as status updates are monotonic. The threads continue until all of their assigned vertices are decided.

2.4 Minimum Spanning Tree (MST)

An MST is a subset of the edges in a weighted, connected graph that connects all vertices with the smallest possible total edge weight and without cycles. The ECL-MST implementation operates in two main stages: initialization and iterative MST construction.

During initialization, each vertex is assigned to its own set, edges are marked as unused, and the worklist is populated with all unique edges. This enables fast union-find operations using disjoint sets.

In the MST construction stage, the algorithm iteratively processes edges to identify and merge disjoint sets via their lightest connecting edges. It uses atomic operations to track the lightest edge for each set and includes only those in the MST that are the lightest for at least one endpoint. This continues until no more merges are possible. For denser graphs, a filtering step removes redundant edges early. Through edge-centric processing, implicit path compression, and hybrid parallelism, ECL-MST achieves high performance across diverse graph types.

2.5 Strongly Connected Components (SCC)

An SCC of a directed graph is a maximal subset of vertices such that every vertex in the subset is reachable from every other vertex in the subset. ECL-SCC is a GPU code that computes the SCCs present in a given directed graph using a new parallel algorithm. It progresses through three main stages in each iteration: signature initialization, maximum-value propagation, and edge removal.

In the initialization stage, each vertex is assigned two signature values, v_{in} and v_{out}, both set to its own unique ID. These values are used to track the highest-reachable vertex IDs through incoming and outgoing paths, respectively. This setup enables all vertices to concurrently act as "pivots", ensuring high initial parallelism.

The second stage involves maximum-value propagation, where the kernel propagates the maximum v_{in} and v_{out} values along the edges. For each edge, it updates the v_{out} of the source with the maximum of its current value and the v_{out} of the destination, and similarly updates the v_{in} of the destination with that of the source. This continues iteratively until a fixed point is reached.

In the edge removal stage, edges between vertices with differing signatures are removed, as they cannot belong to the same SCC. These three stages repeat on the pruned graph until all vertices have matching v_{in} and v_{out} values, at which point each vertex's signature uniquely identifies the SCC to which it belongs.

3 APPROACH

We instrument the discussed programs by augmenting their CUDA sources in a manner that does not affect overall functionality. Specifically, we add counter variables to track the number of times important events happen, which are either thread-local or global, depending on the granularity we need. The thread-local counters show the number of times a specific event occurred for each thread, whereas a global counter shows the total number of times an event occurred across all threads.

As with virtually all profiling tools, our approach introduces overhead and, hence, affects the execution time of the application. However, all programs we study are deterministic, producing identical outputs for identical inputs regardless of whether profiling is enabled. It is worth noting that some codes (e.g., ECL-MIS) are deterministic in their final results but exhibit internal non-determinism, meaning certain intermediate results depend on thread timing and may differ between runs. Adding counters can alter the timing but does not affect the overall non-determinism of these codes.

Subsections 3.1 and 3.2 list general and application-specific metrics that we measure in multiple codes. To the best of our knowledge, no general-purpose profiler tracks these metrics. We measure them to capture application-specific events that standard instrumentation tools cannot expose. A direct comparison between these tools and our approach is, therefore, not possible.

3.1 General Metrics

3.1.1 Load balance. This metric shows how well the workload is distributed across threads. It is embedded in each kernel to see how much of the overall work is being handled by each thread.

3.1.2 Iterations. This metric tracks the number of iterations performed by the threads in cases where individual threads repeat the same set of instructions until a given condition is met. How we define an iteration is application dependent, but it typically refers to an iteration of the outermost loop in the main computation kernel.

3.1.3 Idle threads. This metric tracks the number of threads that are not actively participating in the computation. Threads can end up idle in two ways. When launching as many thread blocks as needed to run a thread per vertex or edge, some of the threads in the last block may not have any work assigned to them. The second way is when a particular vertex or edge does not fulfill a given condition, the assigned thread may not have to do anything.

3.1.4 Active threads. This metric measures the number of threads that actively compute. It is the counterpart to the number of idle threads. We use this metric to study resource utilization.

3.1.5 Atomic updates. This metric tracks the outcome of atomic operations. There are two distinct types of atomic instructions. Specialized CUDA atomics like atomicMin and atomicMax always execute successfully and guarantee completion without retries, but they may not update the target value if it is already the minimum (or maximum) of the two compared values. In contrast, the generic atomicCAS may fail if the target value does not match the expected value, often requiring a retry loop. AtomicCAS is essential for more complex synchronization tasks. Learning about the outcomes of atomic operations (e.g., whether they updated the target or not) is critical for finding bottlenecks related to parallel updates of shared memory locations as well as thread contention.

3.2 Algorithm-specific Metrics

General metrics like the ones discussed above account for only some of the events we track. We also use algorithm-specific metrics to measure computations that are specialized to the given code. For instance, in ECL-CC, the current representative (the smallest vertex ID reachable) of a vertex is captured by calling a function. The

number of times the function is called, and the number of times the return value is smaller (or greater) than the old representative, is information that is specific to this code. We embed such algorithm-specific counters in all tested codes.

4 RELATED WORK

Profiling and performance analysis of GPU programs have been studied extensively, with numerous tools and frameworks developed to support performance characterization, optimization, and architectural modeling. The majority of existing approaches focus on IR-level or ISA-level instrumentation and do not support the metrics we are interested in.

General-purpose profiling and modeling tools such as *TAU* [34] and *HPCToolkit* [1] provide runtime analysis and statistical summaries for parallel programs. TAU supports a wide range of architectures and offers event-based profiling and tracing but is primarily CPU-focused and does not expose low-level GPU kernel execution behavior. NVIDIA's CUDA Profiling Tools Interface *CUPTI* [30] is widely used by tools like Nsight Systems and Visual Profiler to collect detailed kernel execution metrics. It exposes hardware performance counters such as instruction throughput, memory bandwidth, occupancy, and warp execution efficiency. These counters are valuable for understanding overall GPU utilization patterns. However, CUPTI operates primarily at the level of whole kernels and aggregated statistics per warp or SM, and it lacks instrumentation capabilities to capture thread-level event counts specific to algorithmic behavior of irregular workloads.

Dynamic instrumentation and compilation frameworks like *Ocelot* [16], *Lynx* [18], and *NVBit* [9] aim to improve profiling flexibility. Ocelot enables dynamic binary translation of PTX to multiple backends, allowing runtime performance tuning and architectural exploration without source recompilation. Lynx provides transparent, low-overhead injection of profiling routines into GPGPU applications at the PTX level, making it suitable for examining memory access patterns and control divergence in data-parallel workloads. NVBit, developed by NVIDIA, provides a lightweight, low-overhead binary instrumentation framework for dynamically inserting custom analysis code at the SASS instruction level. It supports runtime inspection and transformation of GPU instructions and is compatible with all CUDA applications without requiring recompilation. All three frameworks offer more flexibility than fixed-function profilers, but they do not have access to high-level program semantics nor were they designed for irregular codes.

Instrumentation for CPUs has also influenced GPU tooling. *Pin* [28], a dynamic instrumentation framework for x86 binaries, enables detailed runtime analysis via user-defined instrumentation functions. While not GPU-specific, Pin's architecture has inspired similar GPU-level tools. For instance, *Barra* [15] provides functional simulation at the PTX level, supporting flexible introspection into memory access and thread execution behavior. It facilitates studying kernel internals independent of the hardware, but it incurs high overhead and lacks support for newer GPU features.

Analytical and predictive GPU performance models offer a different approach. The adaptive performance modeling tool for GPU architectures [6] estimates GPU kernel execution time by considering control divergence, shared memory bank conflicts, and memory coalescing. This model is validated against a variety of workloads and shows good accuracy, though it assumes a regular execution flow. The quantitative performance analysis model for GPU architectures [38] presents a microbenchmark-driven model for GPU pipeline behavior, mapping the effects of memory latency, bandwidth, and computation intensity to performance outcomes. These models help developers understand performance bottlenecks but are limited when applied to irregular codes.

Simulator-integrated visualization tools [5] provide time-series views of warp scheduling, memory traffic, and pipeline usage using GPGPU-Sim as a backend. This enables fine-grained tracing of control divergence and bottlenecks over time, offering insight into low-level architectural dynamics. Similarly, detailed kernel analysis through simulation [7] captures occupancy, divergence, and throughput patterns in a variety of CUDA workloads to facilitate kernel classification based on performance behavior. However, both techniques are limited by simulation speed and scalability issues.

Custom instrumentation frameworks extend profiling capabilities with greater flexibility. *CUDA Flux* [10] allows developers to rewrite PTX code to insert instrumentation for collecting runtime metrics such as warp divergence and memory coalescing. It avoids reliance on hardware counters and works across architectures. *CUDAAdvisor* [33] is an LLVM-based framework that performs static analysis on CUDA programs to infer control flow, reuse distances, and divergence across basic blocks. It enables hardware-independent profiling and feedback across CUDA versions. Similarly, *SASSI* [36] is a profiling system that supports fine-grained metric collection per thread or warp and allows developer-defined metrics to be embedded directly in the code. These tools enable profiling that is both flexible and accurate, but they still typically lack semantic awareness of algorithm-level behavior.

Many of the profilers and instrumentation frameworks discussed above are highly sophisticated and can greatly aid with and simplify the assessment of GPU application performance. Tools like Nsight Systems and Nsight Compute provide intuitive visualizations and timeline views that help identify kernel bottlenecks and resource underutilization. CUPTI enables access to hardware counters for low-level performance analysis. Dynamic instrumentation frameworks such as NVBit, Lynx, and Ocelot allow runtime code modification and targeted inspection without recompilation, while static analysis tools like CUDAAdvisor enable compiler-level reasoning about control flow and memory usage. Simulator-based approaches, including GPGPU-Sim and Barra, offer fine-grained introspection into kernel behavior and pipeline activity, and analytical models help estimate performance based on architectural parameters. Together, these tools form a powerful suite for diagnosing inefficiencies and optimizing data-parallel GPU workloads.

However, they inevitably leave small but important gaps, particularly when applied to irregular programs, as they typically operate at the level of compiled code or hardware instructions and are not designed to look into high-level or algorithm-specific behavior. For instance, information related to program semantics (e.g., the number of label updates of a vertex) could be insightful when assessing the performance of graph analytics codes. However, general-purpose tools cannot measure such metrics because they do not know what is important in a specific program. Moreover, they do not provide certain low-level information (e.g., the failure

rate of atomics) even though such metrics could probably be supported. Hence, we recommend that programmers not forget about the simple approach of manually adding counters to CUDA code, like we have done, to complement existing profilers by capturing additional information, in particular application-specific metrics.

5 EXPERIMENTAL METHODOLOGY

5.1 Hardware and Software

We ran the instrumented ECL suite codes on an NVIDIA GeForce RTX 4090 GPU. This GPU is based on the Ada Lovelace architecture and includes 16,384 CUDA cores organized into 128 streaming multiprocessors (SMs). Each SM includes 128 kB of L1 cache/shared memory, and the entire GPU has 72 MB of L2 cache. The card is equipped with 24 GB of GDDR6X memory that has a peak memory bandwidth of 1008 GB/s. We used the `nvcc` compiler with the `-O3` flag to enable optimizations and the `-arch=compute_89` flag to target the 8.9 compute capability of the GPU.

The host system runs Fedora Linux 41 with kernel version 6.12.8 and is powered by an AMD Ryzen Threadripper 2950X CPU, with 16 cores and 32 hardware threads. It includes 48 GB of DDR4 main memory. We used CUDA Toolkit version 12.6 and NVIDIA driver version 545.29 and compiled the codes with `nvcc` version 12.6.85.

5.2 Benchmarks

We evaluated the 5 graph codes on the inputs listed in Table 1 [11]. The inputs vary significantly in size, type, and degree. They are the same as Liu et al. used for their work that introduces the ECL suite [27]. Four of the codes—MIS, CC, GC, and MST—operate on undirected graphs, the MST code uses weighted graphs, and SCC processes directed graphs. The upper block of Table 1 lists the inputs for MIS, CC, MST, and GC, and the lower block for SCC.

We only use mesh graphs for ECL-SCC because it was developed for meshes [4]. All input graphs are stored in compressed-sparse-row (CSR) format [19]. We run each code nine times per input and report results from the run yielding the median runtime.

Table 1: Input graphs. Abbreviations: InTopo = Internet Topology, PubCit = Publication Citation, PatCit = Patent Citation

Graph Name	Edges	Vertices	Type	d-avg	d-max
2d-2e20.sym	4,190,208	1,048,576	grid	4.0	4
amazon0601	4,886,816	403,394	co-purchases	12.1	2,752
as-skitter	22,190,596	1,696,415	InTopo	13.1	35,455
citationCiteseer	2,313,294	268,495	PubCit	8.6	1,318
cit-Patents	33,037,894	3,774,768	PatCit	8.0	793
coPapersDBLP	30,491,458	540,486	PubCit	56.4	3,299
delaunay_n24	100,663,202	16,777,216	triangulation	6.0	26
europe_osm	108,109,320	50,912,018	roadmap	2.1	13
in-2004	27,182,946	1,382,908	weblinks	19.7	21,869
internet	387,240	124,651	InTopo	3.1	151
kron_g500-logn21	182,081,864	2,097,152	Kronecker	86.8	213,904
r4-2e23.sym	67,108,846	3,888,608	random	8.0	26
rmat16.sym	967,866	65,536	RMAT	14.8	569
rmat22.sym	65,660,814	4,194,304	RMAT	15.7	3,687
soc-LiveJournal1	85,702,474	4,847,571	community	20.3	20,333
USA-road-d.NY	730,100	264,346	roadmap	2.8	8
USA-road-d.USA	57,708,624	23,947,347	roadmap	2.4	9
toroid-wedge	485,564	196,608	mesh	2.47	4
star	654,080	327,680	mesh	2.00	2
toroid-hex	4,684,142	1,572,864	mesh	2.98	4
cold-flow	6,295,558	2,112,512	mesh	2.98	5
klein-bottle	18,793,715	8,388,608	mesh	2.24	4

6 RESULTS

6.1 Program Behavior Analysis

In this section, we present our profiling results, analyze the observed behavior, and derive insights. Note that we only show a small portion of the data gathered and only highlight the most important characteristics of each code.

6.1.1 ECL-MIS. Table 2 lists per-thread statistics of 3 aspects of the main kernel in ECL-MIS: the number of vertices assigned, the number of iterations, and the number of vertices finalized. Each metric is measured separately for every thread and averaged across all threads (196,608 on the RTX 4090) and all iterations. On the europe_osm graph, each thread, on average, processes around 259 vertices and finalizes just over 28 vertices in about 2 iterations.

Vertices are assigned to threads in a round-robin fashion to balance the workload. Hence, all threads get the same number of vertices ±1, which is why we only show the average in the table. We list this number as a point of reference for the other two metrics.

The number of iterations reflects the kernel's asynchronous nature: each thread repeatedly processes its vertices until all of them have been decided. The average number of iterations executed by most threads is small and correlates somewhat ($r = 0.64$) with the ratio of the maximum degree over the average degree of the vertices. (A high such ratio is indicative of a power-law graph, which tends to have a low diameter [24].) This finding is surprising, as graph algorithms that propagate information along edges typically require more iterations on higher-diameter graphs. Yet, ECL-MIS exhibits the opposite behavior. The reason is that the MIS algorithm makes local decisions based on neighboring vertices and hardly ever propagates information far in the graph.

The maximum number of iterations executed by any thread also exhibits an interesting behavior. On our smallest input (internet), this number is higher than on some of our largest inputs (e.g., europe_osm), which have orders of magnitude more vertices. In fact, there is a small negative correlation with the number of vertices in the graph ($r = -0.37$). This is because each thread only processes a single vertex on our smallest inputs, meaning there is little work per iteration. Hence, each thread rapidly checks a few conditions over and over until they finally change (due to updates from neighboring threads), explaining the high maximum iteration counts.

A thread eventually finalizes all of its vertices that are 'in' the MIS. This number is generally much lower than the number of assigned vertices, since few are included in the MIS per iteration. Finalizing a vertex also marks its neighbors as 'out'. The average and maximum number of vertices finalized per thread both correlate strongly with the number of vertices in the input ($r \geq 0.98$), indicating a balanced vertex distribution and little load imbalance.

Since ECL-MIS is asynchronous, some of its behavior is thread-timing dependent [12], making it internally non-deterministic (but the final result is deterministic). To account for this internal non-determinism, we measured the number of iterations executed by each thread several times for each input. Table 3 lists the results.

As we can see, the iteration counts are a little different for every run, but the general trends remain the same. We found other metrics to behave similarly (not shown). Hence, despite its lock-free

Table 2: ECL-MIS metrics

Graph	Iterations		Vertices	Finalized	
	Avg	Max	Avg	Avg	Max
2d-2e20.sym	2.28	42	5.34	0.71	6
amazon0601	2.16	53	2.05	0.26	3
as-skitter	6.46	140	8.67	1.62	9
cit-Patents	1.89	29	19.20	4.38	18
citationCiteseer	1.84	100	1.38	0.40	2
coPapersDBLP	4.46	215	2.76	0.08	3
delaunay_n24	1.96	19	85.34	8.65	42
europe_osm	2.04	15	258.95	38.42	141
in-2004	4.12	235	7.03	1.60	8
internet	1.34	52	0.73	0.28	1
kron_g500-logn21	2.05	42	10.70	2.53	11
r4-2e23.sym	2.16	28	42.66	4.17	26
rmat16.sym	1.74	33	0.55	0.08	1
rmat22.sym	2.12	36	21.33	2.79	18
soc-LiveJournal1	4.35	79	24.66	3.99	22
USA-road-d.NY	1.76	80	1.36	0.31	2
USA-road-d.USA	1.89	11	121.80	22.32	81

Table 3: Measurements across multiple runs of ECL-MIS

Graph	Number of iterations					
	Run 1		Run 2		Run 3	
	Avg	Max	Avg	Max	Avg	Max
2d-2e20.sym	2.28	42	2.32	49	2.26	37
amazon0601	2.16	53	2.11	51	2.11	58
as-skitter	6.46	140	5.79	128	6.35	140
cit-Patents	1.89	29	1.90	31	1.91	31
citationCiteseer	1.84	100	1.82	70	1.85	79
coPapersDBLP	4.46	215	4.38	223	4.36	169
delaunay_n24	1.96	19	1.97	21	1.95	20
europe_osm	2.04	15	2.04	15	2.04	14
in-2004	4.12	235	4.11	238	4.19	226
internet	1.34	52	1.34	56	1.36	47
kron_g500-logn21	2.05	42	2.05	46	2.04	45
r4-2e23.sym	2.16	28	2.17	26	2.18	27
rmat16.sym	1.74	33	1.76	35	1.78	40
rmat22.sym	2.12	36	2.14	36	2.11	31
soc-LiveJournal1	4.35	79	4.27	79	4.18	74
USA-road-d.NY	1.76	80	1.78	72	1.74	54
USA-road-d.USA	1.89	11	1.89	12	1.89	13

asynchronous implementation, this code exhibits quite consistent behavior. Nevertheless, it is important to note that the same program and input may have to be profiled multiple times, even if the overall code is deterministic. Moreover, since the execution of profiling instructions affects thread timing, the measurement of thread-timing-sensitive metrics may be perturbed.

6.1.2 ECL-SCC. Figure 1, generated using the star.mesh input, illustrates how the ECL-SCC algorithm progressively identifies SCCs. The plot legends list the values of m and n, where m is the outer loop counter and n is the inner loop counter. Hence, n represents the current signature-propagation iteration, in which the maximum vertex IDs reachable along the edges are updated. Propagation continues iteratively until no further changes occur. The outer loop counter m is incremented, and n is reset to 1 (reflecting a do-while loop) after removing all edges with non-matching signatures and resetting all vertex signatures to their vertex IDs. This process repeats until the two signature values are equal in all vertices. Note that signature propagation occurs at the thread-block level: if any thread in a block performs an update, all threads in that block remain active and continue checking for further updates in the next local iteration. Likewise, if any block detects an update, the entire

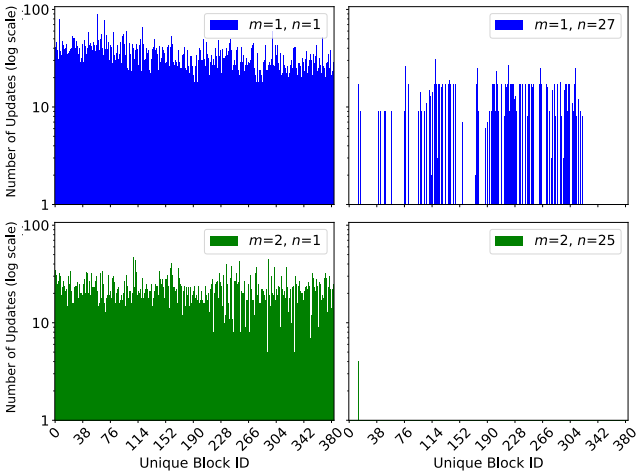

Figure 1: ECL-SCC code progression

grid of blocks re-enters the outer propagation loop to check for additional changes. To analyze this behavior in detail, we track the number of updates performed by each thread block during every signature-propagation iteration (n).

The x-axis of the plots lists the block IDs (384 total blocks with 512 threads per block), and the y-axis lists the number of signature updates in each thread-block for each signature-propagation iteration (m). We show plots for two different outer iterations: m=1 and m=2 out of 10 total. The upper plots show the number of block-wide updates in the first outer iteration (m=1). Of the 43 total signature-propagation iterations, the 1st and 27th are shown. In the first plot (upper left), we see around 70 updates in each block, meaning the threads in each block collectively updated their `in` and `out` signatures around 70 times. These updates may trigger further updates in subsequent iterations, but they diminish quickly, leading to an increase in the number of inactive blocks. The upper right plot illustrates this, with approximately 10 updates for the majority of the blocks and no updates in others. Signature propagations thus appear to remain largely localized within thread blocks.

Similar behavior is observed in the second outer iterations (m=2, bottom plots). In the second-to-last iteration (lower right plot), only one thread block is active, with only 3 updates, possibly from just one thread. However, all threads in the block still check for updates, as any change triggers another propagation step. Additional plots from other inputs (not shown) show comparable trends. These observations suggest that smaller block sizes may lower the average number of iterations per block and improve performance.

6.1.3 ECL-CC. Table 4 shows profiling data from the `init` kernel in ECL-CC, which accounts for 10-20% of the total runtime. It includes two metrics: the number of vertices initialized and the number of vertices traversed to find the first smaller neighbor. As discussed in Section 2.1, ECL-CC initializes the label of each vertex to the ID of the first neighbor in its adjacency list with a smaller ID.

The number of vertices initialized matches the total number of vertices in each input graph as all vertices are initialized and the counts are aggregated across all threads. We include this value as

Table 4: ECL-CC init kernel profiling data

Graph	Vertices initialized	Vertices traversed	Graph	Vertices initialized	Vertices traversed
2d-2e20.sym	1.05×10^6	1.68×10^6	internet	1.25×10^5	1.65×10^5
amazon0601	4.03×10^5	4.04×10^5	kron_g500-logn21	2.10×10^6	2.59×10^6
as-skitter	1.70×10^6	1.70×10^6	r4-2e23.sym	8.39×10^6	8.39×10^6
citationCiteseer	2.68×10^5	4.00×10^5	rmat16.sym	6.55×10^4	10.35×10^4
cit-Patents	3.77×10^6	8.21×10^6	rmat22.sym	4.19×10^6	6.20×10^6
coPapersDBLP	5.40×10^5	6.00×10^5	soc-LiveJournal1	4.85×10^6	4.87×10^6
delaunay_n24	1.68×10^7	2.74×10^7	USA-road-d.NY	2.64×10^5	2.94×10^5
europe_osm	5.09×10^7	5.29×10^7	USA-road-d.USA	2.39×10^7	3.23×10^7
in-2004	1.38×10^6	2.58×10^6			

a reference for the second metric. For most inputs, the number of vertices traversed is only slightly higher than, and often nearly equal to, the number of vertices initialized. A small difference between the two metrics indicates that most vertices find a smaller neighbor after visiting very few neighbors (e.g., as-skitter). Some inputs result in a large difference (e.g., cit-Patents), meaning that initialization requires searching through many neighbors.

Per-thread data (not shown) corroborates that the number of vertices traversed is either 1 or equal to the vertex's degree, indicating a full traversal of the adjacency list when a smaller neighbor cannot be found. This behavior stems from the fact that the adjacency lists are sorted, placing the smallest neighbor first. Hence, failing to find a smaller neighbor ID right away results in unnecessary work since all remaining neighbors have an even higher ID.

6.1.4 ECL-MST. Figure 2 presents key profiling metrics for the ECL-MST code on the amazon0601 input, focusing on the main computation kernel (K1). For each iteration, it shows the fraction of launched threads that end up having work to do (i.e., process edges connecting distinct components), the percentage of conflicting threads (attempting atomic updates to the same memory location), and the fraction of useless atomics (`atomicCAS` failures and `atomicMin` operations with no effect).

The x- and y-axes in the figure represent percentages and iterations, respectively, with each set of three bars showing metrics from one iteration. The code distinguishes between "Regular" iterations, which process the light edges (with weights below a threshold), and "Filter" iterations, which handle heavier edges (if needed).

Except in the first iteration of both kinds, the percentage of threads doing useful work is quite low. This trend also holds for other inputs (not shown). Upon inspecting the code, we found part of the reason for this behavior to be the launch configuration (i.e., the number of blocks), which is not updated correctly.

ECL-MST builds the MST by successively merging connected components (CCs) along their lightest edge until just one CC remains. Early iterations involve many small CCs connected to others. Interestingly, our results show that thread conflicts tend to decrease with increasing iteration count. We expected the initial abundance of CCs to distribute atomic updates and result in fewer conflicts.

The reason for this behavior is that each thread non-atomically checks if its edge is lighter than the current minimum for the CCs it connects. Only if it passes this check does it attempt an atomic update. Early on, many threads pass the check, yielding more conflicts. In later iterations, with fewer CCs and edges left, most threads fail the check and skip the atomic update, preventing conflicts.

Figure 2: Bar plot of metrics derived from ECL-MST profiling data for the amazon0601 input. Error bars show 95% confidence intervals around the median.

The percentage of failed atomics increases with the iteration count. This is expected as the threads that pass the check try to update one of the remaining CCs, and there are exponentially fewer CCs in later iterations, yielding a higher percentage of failures.

6.1.5 ECL-GC. Table 5 presents per-vertex statistics of two counters in the `runLarge` kernel, which colors high-degree vertices ($degree > 31$). Hence, the table excludes inputs that only have vertices with degrees below this threshold. The first counter, "Best available color changed," records how often a vertex's best available color is invalidated because a higher-priority neighbor has claimed it. The second counter, "Color assignment not yet possible," tracks how often a vertex cannot be colored because some higher-priority neighbor might still affect which color the vertex gets. This counter shows how many times a vertex is processed unsuccessfully.

Using coPapersDBLP as an example, which has a high average degree, the table shows that every vertex has its best color invalidated 40.3 times and is processed unsuccessfully 186.4 times on average. In contrast, inputs with low average degrees such as amazon0601, citationCiteseer, and internet yield much lower counts. These results suggest that dense inputs increase the likelihood of color invalidations and of unsuccessful coloring attempts. Indeed, the averages in Table 5 correlate moderately ($r \approx 0.62$) with the average degree of the graphs, reinforcing that the density is an important factor affecting the efficiency of the coloring heuristic.

6.2 Optimizations based on Observations

In this section, we discuss code-specific optimizations for ECL-SCC, ECL-CC, and ECL-MST that we derived from what we learned in

Table 5: Per-vertex statistics of the ECL-GC runLarge kernel

Graph	Best available color changed		Color assignment not yet possible	
	Avg	Max	Avg	Max
amazon0601	0.12	10	0.26	45
as-skitter	1.86	61	27.09	1316
cit-Patents	0.97	10	3.70	68
citationCiteseer	0.09	10	0.25	41
coPapersDBLP	40.31	342	186.39	3363
in-2004	14.52	446	25.04	4632
internet	0.00	4	0.00	6
kron_g500-logn21	10.22	322	61.52	1057
rmat16.sym	0.12	11	0.84	58
rmat22.sym	8.57	29	82.80	163
soc-LiveJournal1	20.18	289	120.92	1620

Table 6: Speedups for a few ECL-SCC block sizes (original=512)

Graph	Threads Per Block			
	64	128	256	1024
toroid-wedge	0.85	0.98	1.03	0.94
star	0.99	1.14	1.12	0.76
toroid-hex	0.79	0.95	1.08	0.77
cold-flow-sponge	0.89	0.95	0.99	0.89
klein-bottle	1.02	1.28	1.02	0.76

Table 7: Performance improvement in ECL-CC

Graph	Speedup
as-skitter	1.05
coPapersDBLP	1.03
in-2004	1.16
kron_g500-logn21	1.14
rmat22.sym	1.03

Section 6.1. We then apply these optimizations and present performance results. Note that the codes in the ECL suite are already highly optimized. The goal of this section is merely to demonstrate that counter-based profiling can, in some cases, reveal additional optimization potential even on state-of-the-art codes.

6.2.1 ECL-SCC. Based on the findings discussed in Section 6.1.2, we experimented with optimizing the number of threads per block to minimize wasted work. Table 6 presents the speedups for different threads-per-block configurations on the RTX 4090.

With a small block size (e.g., 64 threads), it is more likely that a higher number of thread blocks remain idle even if some work is still left, which means that some blocks may stop early and work gets pushed to outer iterations. With a large block size (e.g., 1024 threads), the opposite happens: even a single active thread keeps the entire block alive, forcing many idle threads to participate in block-wide synchronizations and increasing inner-loop overhead. The optimal block size balances these effects, allowing updates to converge efficiently without over- or under-propagating across blocks. Hence, moderate block dimensions with 128 or 256 threads yield the best performance.

6.2.2 ECL-CC. As outlined in Section 6.1.3, the `init` kernel needlessly traverses the adjacency lists of the vertices that do not have a neighbor with a lower ID. We optimized the code to only access the first neighbor. Table 7 reports the effect of this change on overall runtime. It only lists the inputs that yield a noticeable speedup.

An improvement is expected for the input graphs with large differences between their two values in Table 4. However, not all such inputs exhibit better runtimes (e.g., cit-Patents). This is likely because the neighbor traversal constitutes only a small portion of the overall compute time for these inputs. Nevertheless, other inputs run up to 16% faster after the optimization. These improvements result from reducing unnecessary neighbor traversals during the search for smaller neighbors when assigning initial representatives to vertices.

Table 8: Runtime change of updated ECL-MST code

Graph	Runtime % change	Graph	Runtime % change
2d-2e20.sym	0.43	internet	3.33
amazon0601	0.83	kron_g500-logn21	-3.35
as-skitter	0.10	r4-2e23.sym	0.05
cit-Patents	0.04	rmat16.sym	-0.52
citationCiteseer	-1.32	rmat22.sym	0.05
coPapersDBLP	0.30	soc-LiveJournal1	0.02
delaunay_n24	0.14	USA-road-d.NY	0.87
europe_osm	0.25	USA-road-d.USA	0.36
in-2004	0.75		

6.2.3 ECL-MST. The analysis in Section 6.1.4 revealed that two of the three computation-intensive kernels are launched with too many thread blocks. We corrected this issue and show the percentage improvement over the original version in Table 8.

The results indicate little to no improvement on average and, in a few cases, even a degradation in overall performance despite launching significantly fewer threads per iteration. This is likely because the overhead of recomputing the number of thread blocks before each kernel launch largely offsets, or even exceeds, the savings from launching extra threads (which is done by the GPU hardware in essentially no time) that immediately terminate. Nevertheless, the optimization can improve performance by up to 3.33%.

7 SUMMARY AND CONCLUSIONS

We manually instrumented the 5 high-performance GPU graph analytics codes from the ECL suite with counters to track application-specific events that general-purpose profiling tools cannot track. The gathered information, such as per-thread iteration counts and failure rates of atomic operations, reveals important behavioral patterns, some of which can be exploited to boost performance.

In ECL-CC, we found that adjacency list traversals during initialization are unnecessary. Avoiding them improved the overall runtime by up to 16%. In ECL-MST, we identified inefficient kernel launch configurations, but correcting them only led to minimal performance gain due to the extra cost of recalculating the launch configuration. In ECL-SCC, our manual profiling exposed excessive block-wide synchronizations, which we alleviated by tuning the thread-block size, leading to a runtime improvement of up to 28%.

These results highlight that the simple approach of manual code instrumentation can uncover important program-specific behavior where standard profilers fall short as they are unaware of program semantics. Therefore, we encourage programmers to not only rely on sophisticated tools but also on simple approaches such as inserting counters into source code, especially for irregular workloads. Such manual instrumentation complements the capabilities of sophisticated tools and can provide otherwise missing information needed for performance validation and optimization.

ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under Award #1955367.

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701. https://doi.org/10.1002/cpe.1553 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1553

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers: Principles, Tools, and Techniques.

[3] Ghadeer Alabandi, Evan Powers, and Martin Burtscher. 2020. Increasing the parallelism of graph coloring via shortcutting. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 262–275.

[4] Ghadeer Alabandi, William Sands, George Biros, and Martin Burtscher. 2023. A GPU algorithm for detecting strongly connected components. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[5] Aaron Ariel, Wilson WL Fung, Andrew E Turner, and Tor M Aamodt. 2010. Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 164–174.

[6] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. 2010. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 105–114.

[7] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 163–174.

[8] Josh Barnes and Piet Hut. 1986. A hierarchical O (N log N) force-calculation algorithm. *nature* 324, 6096 (1986), 446–449.

[9] Arnaud Benoit, Miguel Cebrian, Ignacio Guillen, Francisco Guzman, Shravan Pai, Wilson Fung, and David Gregg. 2019. NVBit: A dynamic binary instrumentation framework for CUDA applications. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 140–151.

[10] Lorenz Braun and Holger Fröning. 2019. CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 73–81. https://doi.org/10.1109/PMBS49563.2019.00014

[11] Martin Burtscher. 2023. ECL Graph Inputs. https://userweb.cs.txstate.edu/~burtscher/research/ECLgraph/. Accessed: 2025-06-11.

[12] Martin Burtscher, Sindhu Devale, Sahar Azimi, Jayadharini Jaiganesh, and Evan Powers. 2018. A high-quality and fast maximal independent set implementation for gpus. *ACM Transactions on Parallel Computing (TOPC)* 5, 2 (2018), 1–27.

[13] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 141–151.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. 2008. A Performance Study of General-purpose Applications on Graphics Processors Using CUDA. *J. Parallel and Distrib. Comput.* 68, 10 (October 2008), 1370–1380.

[15] Sylvain Collange, David Defour, and David Parello. 2009. Barra, a modular functional gpu simulator for gpgpu. *University de Perpignan, Tech. Rep* (2009).

[16] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 353–364.

[17] Alex Fallin, Andres Gonzalez, Jarim Seo, and Martin Burtscher. 2023. A High-Performance MST Implementation for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[18] Naila Farooqui, Andrew Kerr, Greg Eisenhauer, Karsten Schwan, and Sudhakar Yalamanchili. 2012. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, 58–67.

[19] Alan George and Joseph W Liu. 1981. *Computer solution of large sparse positive definite*. Prentice Hall Professional Technical Reference.

[20] Kirsten Hildrum and Philip S Yu. 2005. Focused community discovery. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*. IEEE, 4–pp.

[21] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. *Acm Sigplan Notices* 46, 8 (2011), 267–276.

[22] Jayadharini Jaiganesh and Martin Burtscher. 2018. A high-performance connected components implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 92–104.

[23] David B. Kirk and Wen mei W. Hwu. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Burlington, MA.

[24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 177–187.

[25] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008), 39–55.

[26] Yiqian Liu, Avery VanAusdal, and Martin Burtscher. 2024. ECL Graph Benchmark Suite. https://github.com/burtscher/ECL-suite. Accessed: 2025-06-23.

[27] Yiqian Liu, Avery VanAusdal, and Martin Burtscher. 2024. Performance Impact of Removing Data Races from GPU Graph Analytics Programs. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. 320–331. https://doi.org/10.1109/IISWC63097.2024.00036

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[29] Jayadev Misra. 1986. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)* 18, 1 (1986), 39–65.

[30] NVIDIA Corporation. 2025. CUPTI: CUDA Profiling Tools Interface. https://docs.nvidia.com/cupti/. Version 12.5.

[31] James L Peterson. 1977. Petri nets. *ACM Computing Surveys (CSUR)* 9, 3 (1977), 223–252.

[32] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Md. Mujeeb Hassaan, Rakesh Kaleem, Andrew Lee, Andrew Lenharth, Roman Manevich, et al. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 12–25.

[33] Du Shen, Shuaiwen Leon Song, Ang Li, and Xu Liu. 2018. CUDAAdvisor: LLVM-based runtime profiling for modern GPUs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 214–227. https://doi.org/10.1145/3168831

[34] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311.

[35] Keshav Srinivasan, Nolan Coble, Joy Hamlin, Thomas Antonsen, Edward Ott, and Michelle Girvan. 2022. Parallel machine learning for forecasting the dynamics of complex networks. *Physical Review Letters* 128, 16 (2022), 164101.

[36] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/2749469.2750375

[37] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. *Introduction to data mining*. Pearson Education India.

[38] Yao Zhang and John D Owens. 2011. A quantitative performance analysis model for GPU architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*. IEEE, 382–393.