Comparing Graph Algorithm Styles on NVIDIA and AMD GPUs

Avery VanAusdal
Texas State University
San Marcos, Texas, USA
arv107@txstate.edu

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

ABSTRACT

Graph algorithms are important for many domains, and GPUs can be used to accelerate them. Unfortunately, CUDA code can only be run on NVIDIA GPUs. In this study, we port the CUDA graph codes from the Indigo3 benchmark suite to HIP. This enables them to run on both AMD and NVIDIA GPUs. In addition, it allows the study of performance differences between compiled CUDA and HIP codes that are otherwise identical. Since the Indigo3 codes are written in a variety of different implementation and parallelization styles, this also allows us to study the performance of AMD GPUs on these styles and compare the results with the NVIDIA-based style trends.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**;
• **Theory of computation** → *Graph algorithms analysis.*

KEYWORDS

Graph analytics, parallelization and implementation styles, GPUs

ACM Reference Format:
Avery VanAusdal and Martin Burtscher. 2025. Comparing Graph Algorithm Styles on NVIDIA and AMD 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, 5 pages. https://doi.org/10.1145/3731599.3767444

1 BACKGROUND

The Indigo3 benchmark suite [7] describes 13 implementation and parallelization styles for graph analytics codes, where each style represents a choice to be made when writing code. Indigo3 combines these styles in hundreds of ways for 7 important graph problems for CPUs and NVIDIA GPUs. For example, the parallelization styles determine at what granularity the computation is parallelized. GPUs have multiple levels of hardware parallelism that can be exploited when processing the neighbors of vertices in a graph [12], and the GPU granularity style addresses that.

The simplest granularity option is Thread-level, which assigns one thread to each element that needs processing. The downside is the vulnerability to load imbalance if some elements require more work to process than others, such as vertices with more neighbors. The next granularity option is Warp-level, which assigns an entire warp to each vertex that needs processing. A warp is a set of 32 or more contiguous threads, so the threads in the same warp work together to process the neighbors of a single vertex, reducing the effect of load imbalance. Threads in the same warp can communicate quickly without going through memory by using warp-level primitives, but this may require more complex synchronization than the Thread-level option. We can take this one step further with Block-level granularity, which assigns an entire block of threads to each vertex. Threads in the same block can use fast shared memory to communicate, but this may require even more complex synchronization than Warp-level granularity.

While the parallelization styles determine how work is divided, other styles refer to how the computations are implemented. For example, the Push or Pull style [3] determines the direction of the data flow. Push-style codes read the private data of active vertices and update their neighbors with it, leading to possibly overlapping updates. Conversely, Pull-style codes read their neighbors' data and use it to update their active vertices' private data.

The described styles are mostly independent sets of choices that can be combined in many ways. For example, with just the Push-Pull and Thread-Warp-Block styles mentioned, we can already generate six combinations: push-thread, push-warp, push-block, pull-thread, pull-warp, and pull-block. By using several styles on several graph problems, hundreds of codes can be generated.

Xie et al. [11] compare the performance of three graph algorithms on NVIDIA GPUs and AMD-like GPUs. They find that the relatively simple BFS algorithm prefers AMD-like GPUs' larger wavefront sizes and independent shared memory, whereas more complex and branch-heavy algorithms such as TC prefer NVIDIA GPUs. In contrast, our work focuses on how the choice in GPU manufacturer affects style decisions made within graph algorithms.

2 APPROACH

In this work, we study 330 CUDA codes from 6 graph problems from the Indigo3 suite: Breadth-First-Search (BFS), Maximal Independent Set (MIS), Minimum Spanning Tree (MST), PageRank (PR), Single Source Shortest Path (SSSP), and Triangle Counting (TC).

2.1 Styles

Table 1 shows the 9 styles we investigate and marks which style options apply to each graph problem with a "+" symbol. The last row shows the total number of combinations for each problem, ranging from 16 to 84 codes. The styles are as follows.

- **Vertex-based or Edge-based:** This style determines if the code iterates across the vertices or the edges of the graphs.
- **Topology-driven or Data-driven** [10]**:** Topology-driven codes process all elements each iteration, while Data-driven codes only process the elements on a worklist.

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

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767444

- **Duplicates on WL or No duplicates on WL** [9]**:** This style only applies to Data-driven codes and determines if duplicate elements are allowed on the worklist. Not allowing them incurs overhead to ensure the element has not already been added but can be more efficient and allows smaller worklists.
- **Push or Pull** [3]**:** As mentioned in Section 1, this style determines if vertices push data out to their neighbors or pull data in from their neighbors.
- **Read-Write or Read-Modify-Write** [8]**:** Read-Write codes read from a neighbor and then write to it as two separate memory accesses, while Read-Modify-Write codes use atomic operations to read, modify, and write to a neighbor in a single (albeit more expensive) transaction.
- **Deterministic or Non-deterministic** [2]**:** Deterministic codes read from and write to distinct data structures, which enforces internal determinism for easier debugging, while non-deterministic codes perform their reads and writes to the same data structure.
- **Persistent or Non-persistent** [5]**:** The Persistent thread style launches exactly as many threads as the GPU can run simultaneously and splits up the work among them, with each thread likely processing multiple elements. Non-persistent codes simply launch as many threads as are needed to cover each element, with each thread only processing one element.
- **Thread, Warp, or Block** [12]**:** This GPU granularity style, as mentioned in Section 1, determines the level of GPU hardware parallelism that the code exploits.
- **Global-add, Block-add, or Reduction-add:** This GPU reduction style only applies to PR and TC. In those codes, each thread computes a partial sum, and this style determines how they are combined into the overall sum. In Global-add, every thread simply updates the same global memory location using atomic addition. In Block-Add, the threads in each block use shared memory to compute block-wide partial sums and a single designated thread atomically adds it to the global total. In Reduction-Add, each warp calculates a sum, each block combines its warps' sums into a block-wide sum, and a designated thread per block adds it to the global total.

2.2 Porting CUDA to HIP

Unlike CUDA codes, which only work on NVIDIA GPUs, AMD designed the HIP API to be portable and run on both AMD and NVIDIA GPUs. HIP has very similar syntax to CUDA, mostly renaming variables and functions that include the word "cuda" to include "hip". HIP also offers HIPify tools to semi-automatically port CUDA source codes to HIP. However, differences in the GPU architecture between AMD and NVIDIA GPUs can necessitate some changes beyond syntax to ensure the code is truly portable. For example, AMD GPUs do not all have the same warp size like NVIDIA GPUs, so HIP source codes should not assume a specific warp size. There are also some CUDA functions that do not currently exist for AMD GPUs but cannot be removed from the source code without breaking it on NVIDIA GPUs. We also discovered that AMD GPUs cannot launch more than 2^{32} total threads at the same time, while NVIDIA GPUs do not have the same restriction.

All NVIDIA GPUs have exactly 32 threads per warp, so CUDA codes often hardcode that value for simplicity and compiler optimization reasons. They will often declare a constant value for the warp size and manually unroll loops that rely on that value to avoid loop overhead. Since AMD GPUs have 32 or 64 threads per warp depending on the GPU, portable HIP code should support both. To do so, we replaced any constant warp size values with the built-in *warpSize* constant available for both NVIDIA and AMD GPUs. In addition, we re-roll any unrolled loops to use the *warpSize* value.

Warps execute in lockstep on current AMD GPUs, so they do not need (and do not define) some CUDA functions like *__syncwarp* and the *_sync* variants of the *__shfl*, *__any*, and *__all* functions. To handle this, we use preprocessor macros to check if the code is being compiled for an AMD device and disable or replace those missing functions if needed. We define *__syncwarp* as an empty function to effectively disable it, and we replace functions like *__shfl_sync* with their non-sync variants like *__shfl*. This allows the same source code to compile and work on both AMD and NVIDIA GPUs.

3 EXPERIMENTAL METHODOLOGY

Table 1 lists the codes we study along with the styles applied to each code. In total, we tested 330 CUDA codes and 330 HIP codes on 1 AMD GPU and 2 NVIDIA GPUs. The AMD GPU is an RX 7900 XTX with 24 GB of memory. The NVIDIA GPUs are an RTX 4090 with 24 GB of memory and an A100 with 40 GB of memory. To compile the codes for the AMD GPU, we used HIP version 6.2.41134 and ROCm 6.2.3. For the RTX 4090 and A100, we used HIP version 6.3.42134 for both and nvcc versions 12.6 and 12.0, respectively.

Table 1: List of styles used, "+" symbol indicates compatibility with listed style option

Style Options	BFS	MIS	MST	PR	SSSP	TC
Vertex-based, Edge-based	+, +	+, +	+, +	+, -	+, +	+, +
Topology-driven, Data-driven	+, +	+, +	+, +	+, -	+, +	+, -
Duplicates on WL, No duplicates on WL	+, +	-, +	-, -	-, -	+, +	-, -
Push, Pull	+, +	+, +	-, -	+, +	+, +	+, -
Read-write, Read-modify-write	+, +	-, +	-, -	-, +	+, +	-, +
Deterministic, Non-deterministic	+, +	+, +	-, -	+, +	+, +	+, -
Persistent, Non-persistent	+, +	+, +	+, +	+, +	+, +	+, +
Thread, Warp, Block	+, +, +	+, +, +	+, +, +	+, +, +	+, +, +	+, +, +
Global-add, Block-add, Reduction-add	-, -, -	-, -, -	-, -, -	+, +, +	-, -, -	+, +, +
Number of codes	84	56	16	54	84	36

Table 2: Input graphs and their characteristics

Name	Type	Origin	Vertices	Edges	d-avg	d-max	d≥32	d≥512	Diameter
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	4.0	4	0.0%	0.000%	2047
coPapersDBLP	publication	SMC	540,486	30,491,458	56.4	3,299	52.5%	0.092%	24
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	15.7	3,689	12.4%	0.045%	19
soc-LiveJournal	community	SNAP	4,847,571	85,702,474	17.7	20,333	14.0%	0.125%	20
USA-road-d.NY	road map	Dimacs	264,346	730,100	2.8	8	0.0%	0.000%	721

Table 3: Geometric mean of HIP throughput over CUDA throughput on NVIDIA GPUs

GPU	BFS	MIS	MST	PR	SSSP	TC
4090	1.006	0.998	1.000	0.998	1.003	1.012
A100	0.999	1.000	1.001	1.000	1.000	1.000

We use the 5 inputs listed in Table 2 to evaluate the codes' performance. These graphs vary in type, size, and degree distribution and include synthetic grids, social networks, and a road map.

As our performance metric, we use throughput in giga-edges per second, which is calculated by taking the number of edges in the graph, dividing it by the execution time, and then dividing by one billion. To compare the performance of two alternatives, we report throughput ratios, i.e., dividing one throughput by another. For example, to compare the performance of Push against Pull codes with only the GPU granularity style present, we would present the ratios calculated by dividing the throughputs of Push+Thread by Pull+Thread, Push+Warp by Pull+Warp, and Push+Block by Pull+Block. A ratio of 1.0 means the performance is the same.

4 RESULTS

4.1 CUDA versus HIP on NVIDIA GPUs

First, we compared the performance of each CUDA code with its HIP port on the same GPU. Table 3 lists the geometric mean of the throughput ratios for each code on each NVIDIA GPU. As we can see, the performance is practically identical, meaning the HIP port does not significantly change the execution time.

4.2 Style Performance Trends on NVIDIA versus AMD GPUs

Figures 1 to 9 show the performance trends for each style, with the results for CUDA codes on the RTX 4090 NVIDIA GPU on the left and for HIP codes on the RX 7900 XTX AMD GPU on the right. The figures only show the algorithms that utilize the style per Table 1. Figures 1 to 7 plot the throughput ratios of one option over the other using boxen plots to help visualize the distribution, with wider boxes representing a larger number of samples in the given range. The widest box represents the middle 50% of ratios and includes a horizontal line for the median ratio. Any outliers are plotted as circles. The dotted blue line indicates a ratio of 1.0. Figures 8 and 9 cover styles with three options, so their throughputs are directly presented for a side-by-side comparison instead of using ratios.

Figure 1 shows the trends for the Vertex/Edge style. On the NVIDIA GPU, BFS and SSSP have median ratios close to 1, so we cannot say in general that either the edge- or vertex-based codes perform better. MIS, however, has a median ratio close to 3 and favors vertex-based code in general. This is likely because MIS typically only needs to visit a few neighbors per vertex, making the vertex-based approach quite load balanced. MST and TC generally prefer Edge-based codes. These trends hold true for the AMD GPU as well, with the general preferences staying the same.

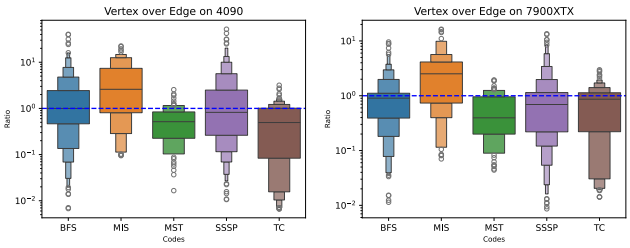

Figure 1: Throughput ratios for Vertex versus Edge codes

Figure 2 presents the throughput ratios for Topology-driven codes that do not use worklists over Data-driven codes that allow duplicates on their worklists. Figure 3 presents the ratios for Topology-driven codes over Data-driven codes that do not allow duplicates on their worklists. There are no versions of MIS that

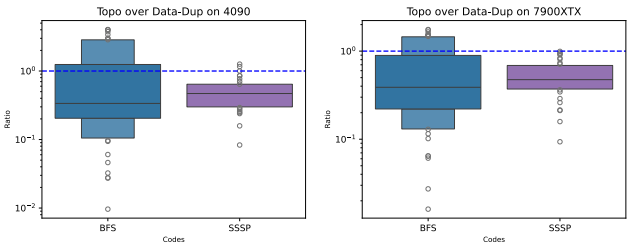

Figure 2: Throughput ratios for Topology versus Data codes with duplicates allowed on the worklist

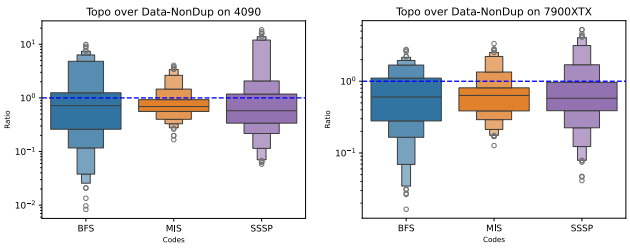

Figure 3: Throughput ratios for Topology versus Data codes without duplicates on the worklist

allow duplicates on the worklist, so MIS appears in Figure 3 but not in Figure 2. Overall, the trends for these styles look very similar between the NVIDIA and AMD GPUs. The data-driven codes are usually faster than their topology-driven counterparts.

Figures 4 to 7 present the throughput ratios for the Push/Pull, ReadWrite/ReadModifyWrite, NonDeterm/Determ, and NonPersist/Persist styles. PR behaves differently for the NonDeterm/Determ style shown in Figure 6 since only the PR's Pull-style codes support the non-deterministic option. These styles, like the earlier ones, show very similar trends between the NVIDIA and AMD GPUs.

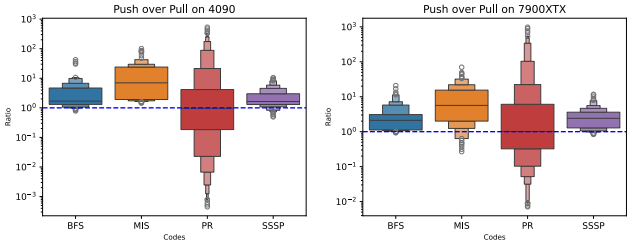

Figure 4: Throughput ratios for Push versus Pull codes

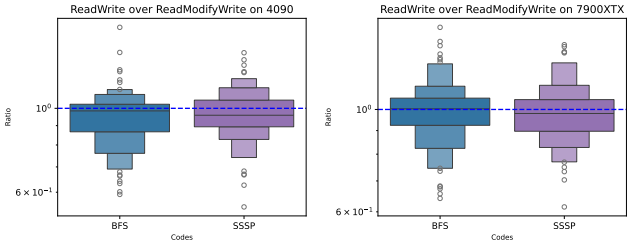

Figure 5: Throughput ratios for Read-Write versus Read-Modify-Write codes

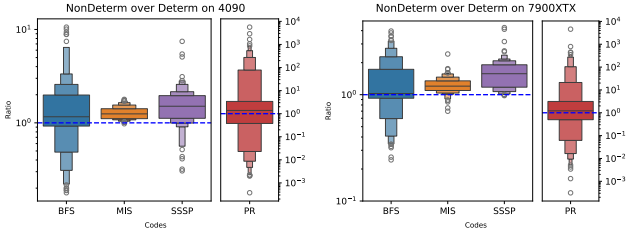

Figure 6: Throughput ratios for Non-deterministic codes versus Deterministic codes

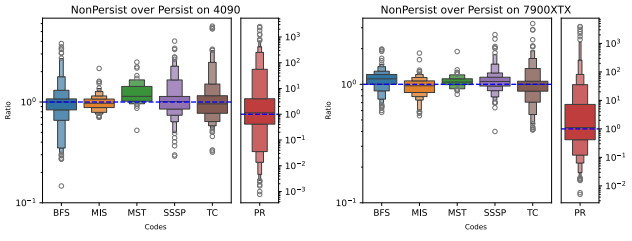

Figure 7: Throughput ratios for Non-Persistent thread versus Persistent thread codes

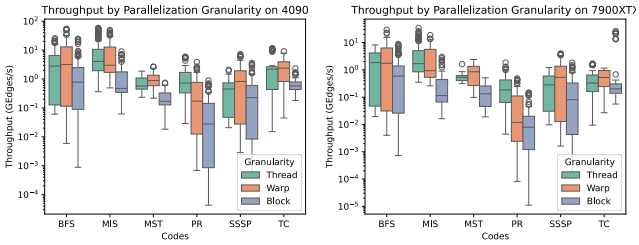

Figure 8: Throughputs in giga-edges per second by GPU granularity style

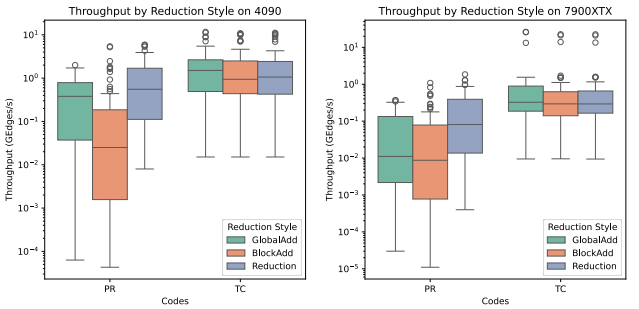

Figure 9: Throughputs in giga-edges per second by GPU reduction style

Figure 8 shows the throughputs of the GPU granularity style options for each code. On both GPUs, the Block granularity is the slowest on our inputs for all codes. For each code, the fastest granularity between Thread or Warp is the same on both GPUs.

Figure 9 presents the throughputs of the GPU reduction style options for the PR and TC codes. The biggest deviation in trends is visible in GlobalAdd for PR. GlobalAdd, where each vertex result is independently added to the same global sum, performs noticeably worse on the AMD than the NVIDIA GPU. While NVIDIA GPUs automatically aggregate atomicAdd ops from the same warp [1, 4], this suggests that AMD GPUs do not. Overall, however, the best style options for each algorithm remain the same on both devices.

5 SUMMARY & CONCLUSIONS

We study 330 CUDA codes from the Indigo3 benchmark suite and their HIP counterparts. We evaluate the performance of the CUDA and HIP codes on 1 AMD GPU and 2 NVIDIA GPUs using 5 input graphs from different domains. Our results show that HIP ports are no slower than their original CUDA versions on NVIDIA GPUs. We analyze the performance of the codes' styles and find that the overall trends are similar on AMD and NVIDIA GPUs despite the unknown underlying architectural differences. These results indicate that guidelines for writing graph analytics codes, such as those found in Indigo2 [6], likely apply to AMD GPUs as well.

ACKNOWLEDGMENTS

This work has been supported by the NSF under Award #1955367 and by an equipment donation from NVIDIA Corporation.

REFERENCES

[1] Andrew Adinets. 2017. *CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics*. https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

[2] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2012. Efficient system-enforced deterministic parallelism. *Commun. ACM* 55, 5 (2012), 111–119.

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, IEEE, New York, NY, USA, 1–10.

[4] Brandon Alexander Burtchell and Martin Burtscher. 2024. Characterizing CUDA and OpenMP Synchronization Primitives. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*. 295–308. https://doi.org/10.1109/IISWC63097.2024.00034

[5] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. IEEE, San Jose, CA, USA, 1–14. https://doi.org/10.1109/InPar.2012.6339596

[6] Yiqian Liu, Noushin Azami, Avery Vanausdal, and Martin Burtscher. 2023. Choosing the Best Parallelization and Implementation Styles for Graph Analytics Codes: Lessons Learned from 1106 Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 92, 14 pages. https://doi.org/10.1145/3581784.3607038

[7] Yiqian Liu, Noushin Azami, Avery Vanausdal, and Martin Burtscher. 2024. Indigo3: A Parallel Graph Analytics Benchmark Suite for Exploring Implementation Styles and Common Bugs. *ACM Trans. Parallel Comput.* 11, 3, Article 13 (Aug. 2024), 29 pages. https://doi.org/10.1145/3665251

[8] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-Free Irregular Computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (Houston, Texas, USA) *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2458523.2458533

[9] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, New York, NY, USA, 463–474. https://doi.org/10.1109/IPDPS.2013.28

[10] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 12–25.

[11] Peichen Xie, Zhigao Zheng, Yongluan Zhou, Yang Xiu, Hao Liu, Zhixiang Yang, Yu Zhang, and Bo Du. 2025. GPU Architectures in Graph Analytics: A Comparative Experimental Study. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*. https://www.researchgate.net/publication/389691514_GPU_Architectures_in_Graph_Analytics_A_Comparative_Experimental_Study

[12] Yang Zhang, Jie Shen, Zhen Xu, Shikai Qiu, and Xuhao Chen. 2019. Architectural Implications in Graph Processing of Accelerator with Gardenia Benchmark Suite. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 1329–1339.