Fast Community Detection in Graphs with Infomap Method using Accelerated Sparse Accumulation

Md Abdul M Faysal*, Maximilian Bremer†, Shaikh Arifuzzaman*, Doru Popovici†, John Shalf†, Cy Chan†
*University of Nevada, Las Vegas, †Lawrence Berkeley National Laboratory,
faysal@unlv.nevada.edu, shaikh.arifuzzaman@unlv.edu, {mb2010, dtpopovici, jshalf, cychan}@lbl.gov

***Abstract*—Information-theoretic community discovery method (popularly known as *Infomap*) is known for delivering better quality results in the Lancichinetti–Fortunato–Radicchi (LFR) benchmark compared to modularity-based algorithms. Parallel algorithms have been developed for Infomap due to the computational challenge of analyzing massive graphs resulting from the tremendous growth of information in bio-sciences, social sciences, business, and other domains. The state-of-the-art techniques on information-theoretic community discovery use hash tables for storing vertex neighborhood flow information, which can be computationally expensive due to collision handling operations and CPU branch mispredictions. The Accelerated Sparse Accumulation (ASA) hardware accelerator for hash accumulation has been developed recently for sparse matrix-matrix multiplication (SpGEMM). We generalize the interface of the ASA accelerator and demonstrate that for state-of-the-art parallel *Infomap*, the accelerator for hash accumulation with fast on-chip memory can overcome the performance bottlenecks of software hash tables and can achieve a speedup of $5.56\times$ while reducing the number of branch mispredictions by 59%, the CPI rate by 21%, and the total number of instructions by 24%.**

***Keywords*—Infomap, Community Discovery, Accelerator, Hash Accumulation, Sparse Graphs**

I. Introduction

Community discovery is a widely used application for grouping or clustering entities of similar categories [9], [12], [16], [17], [24], [25], [27], [28], [30], [33]. Some examples include finding groups of people having similar interests in social networks, marketing products to groups of consumers based on their categories, clustering similar kinds of proteins and recognizing the functionality of unknown proteins, web spam detection in the cyber-security domain, and so on. The tremendous growth of social, biological, professional, and traffic networks in recent years has contributed to the trend of research for parallel algorithm designs for community discovery [3]–[5], [13], [26], [29], [35], [37], [40], [41]. Discovering communities using an *information-theoretic* approach, popularly known as *Infomap* [33], has been found to deliver better quality of the discovered communities by separate studies [1], [18] and experimental (LFR) benchmark [19]. Infomap does not have the *resolution limit* problem [15] present in modularity-based algorithms [9]. There are several shared-memory and distributed memory-based parallel algorithms developed for Infomap [4], [5], [13], [14], [40].

The work [14] designed a parallel Infomap, combining both shared memory and distributed memory parallelism. The implementation achieved a $25\times$ speedup compared to the sequential version of Infomap [33]. A crucial step to decide the community membership of a vertex is to compute and accumulate information about the neighboring vertices. All of the sequential [33] or parallel implementations [4], [5], [13], [14] of Infomap use software hash tables to store the information about the neighboring vertices. We will demonstrate in a later section that the software hash accumulation takes up to $50-65\%$ of the total execution time and a major performance bottleneck in hardware resource utilization (i.e., stalls due to branch misprediction).

We demonstrate that an accelerator for hash accumulation with fast content addressable memory (CAM) can address the challenges in the software hash table and close the gap between achievable and utilized hardware resources. To the best of our knowledge, none of the existing works [4], [5], [13], [14], [33], [40], [41] on Infomap community detection have considered hardware acceleration to speed up hash operations. Chao et al. [42] proposed an accelerator for hash accumulation (ASA) designed for the SpGEMM computation. In this paper, we generalize the interface of ASA [42] outside of its' original SpGEMM formulation such that any application with a high volume of hash lookup and accumulation can benefit from ASA. To demonstrate that, we are augmenting the parallel Infomap [14] implementation with ASA-accelerated hashing operations and show that the hash operations achieve $3.28\times - 5.56\times$ speedups by using ASA. The limited capacity of the CAM can be a concern for storing hash tables for large networks. However, real-world social and metagenome networks exhibit the power-law degree distributions and are sparse as we show in Figure 4 in Section IV for social networks. Applications for metagenome assembly [23] or clustering of similar kinds of protein sequences [22] deal with similar kinds of networks. We demonstrate in Section IV that the limited capacity of the accelerator's memory is not an issue in processing the sparse networks, as 99% of the adjacency list (vertex neighbors) fits entirely within a CAM size of $8KB$. We summarize the contributions of our work as follows:

- We generalize the ASA interface for accelerating SpGEMM computation by Chao et al. [42] and demonstrate the applicability for an application with a high volume of hash operations. To the best of our knowledge, this is the first work where an accelerator is used for speeding up hash operation for Infomap community discovery.

- For the Infomap application, ASA decreases branch misprediction by 59%, CPI rate by 21%, and the total number of instructions by 24% by eliminating expensive software hash accumulation and collision handling operations.
- We demonstrate ASA's limited capacity of on-chip CAM is not an issue in handling big social and biological networks. We show in section IV that more than 99% of the vertices can be processed by only $8KB$ of CAM per core.

II. Background

In this section, we provide some background on the application of the information-theoretic approach to community detection and the motivation for an accelerator for hash accumulation.

A. Definitions

We list some definitions and notations used in the paper below.

Definition 1 *(Graph (Network)).* A graph is defined as $G = (V, E)$, where V is a set of vertices v and E is a set of edges (links) (u, v), with $u, v \in V$.

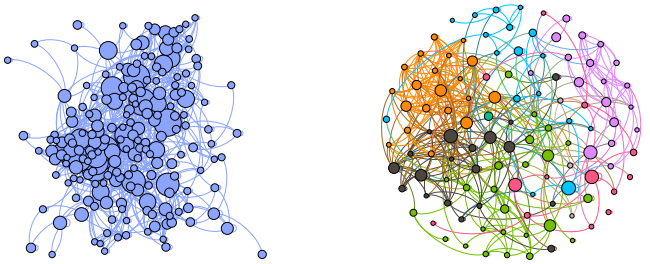
In this paper, we use the terms *graphs* and *networks* interchangeably. Even though community detection is an important task in graph analysis, there is no universally accepted definition or formalization for the term community in a graph $G(V, E)$. One of the main reasons behind the complexity of the problem is its ill-formalization. The related literature intuitively describes communities as sets of nodes closer among them than with the rest of the network. We provide a simplistic but working definition below.

Definition 2 *(Community).* A community in a network is a set of entities that share some closely correlated action with the other entities of that set. A direct edge/link is considered a particular and very important kind of action. See Fig. 1.

Discovering communities in graphs (e.g., social, biological, and communication networks) is an important problem in many scientific domains [9], [16], [27], [28], [33]. The problem of community detection thus pertains to seeking community assignment C_u for each vertex $u \in V$ in a graph G. There are several variants of the community discovery problem–most algorithms [5], [9] find disjoint communities (where each vertex belongs to a single community) while some other algorithms [21] discover overlapping communities (a vertex can belong to multiple communities). The application of information-theoretic community discovery deals with the problem of discovering disjoint communities in a network. In separate studies, Lancichinetti et al. [18] and Aldecoa et al. [1] identified and showed the information-theoretic approach of community discovery, *Infomap*, delivers better solution quality than many other community detection algorithms.

B. Information-Theoretic Community Discovery

Infomap uses a dynamic process (random walk) to reveal the community structure within a network. Infomap exploits the duality of compressing a dataset and extracting a structural pattern in that dataset. That duality is discussed in a branch of statistics called Minimum Description Length (MDL). While the random walker traverses the network, its infinite traversal path can be traced by assigning Huffman coding to reveal structural regularity. More frequent traversal paths receive shorter code length while less frequent traversal paths receive longer code length. A straightforward way to think about how it can discover communities is to divide the vertices into different groups (communities) and then compute the Huffman code (compressed code length) for each of the combinations. For a network of n number of vertices, there are possibly 2^n such combinations, computing Huffman coding for each of those combinations and then finding the most compressed one is an NP-complete problem. Rosvall et al. [33] showed that one can find the theoretical limit of maximum compression for a given partitioning of the network by using Shanon's entropy theorem [36] without explicitly assigning Huffman coding for a traversal path. Rosvall formulated an equation based on the principle of minimum entropy theorem which he named the *Map Equation*.



(a) (b)

Fig. 1: (a) A protein-protein interaction network in yeast [7]. The vertices (blue circles) represent different proteins and the arcs (blue lines) represent interactions between those proteins. (b) Grouping of proteins by community detection based on their functional similarities. Proteins within the same group have the same color and share similar biological properties. The visualizations are generated using Gephi [6].

$$L(M) = q_{\curvearrowright} H(Q) + \sum_{i=1}^{m} p^i_{\circlearrowright} H(\rho^i) \tag{1}$$

Equation (1) has two parts, the first part $q_{\curvearrowright} H(Q)$ of the right side of the equation represents the movement of the random walk between the modules whereas the other part $\sum_{i \in m} p^i_{\circlearrowright} H(\rho^i)$ represents the movement of the random walk within a module. In this paper, the terms *module* and *community* represent the same meaning and will be used interchangeably. The term $q_{\curvearrowright}$ is the probability of the random walk exiting a module. The term $H(Q)$ is the entropy of the module, i.e., the average code length of the movements between the modules where Q stands for the probability distribution of the module entering rate. In the other part of the equation, the term $p^i_{\circlearrowright}$ stands for the stay probability of the random walk within module i. The parameter $p^i_{\circlearrowright}$ can be calculated by summing the vertex visit probability (PageRank) and the exit probability of the random walk for that module.

The term $H(\rho^i)$ is the entropy within the module, i.e., the average code length of the random walk within module i. The term ρ^i is the probability distribution of the code of module i. For any vertex α, the vertex visit rate, i.e., the PageRank p_α can be computed taking teleportation τ into account.

Algorithm 1: FindBestCommunity

```
Data: A vertex/supernode v_i of graph G(V, E)
Result: New community m_new for vertex v_i
1  std :: unordered_map < pair < int, double >> outFlowtoModules
2  std :: unordered_map < pair < int, double >> inFlowFromModules
3  numModlinks ← 0
4  for (linkIt ← v_i.outLinks.begin() to v_i.outLinks.end()) do
5      newModId ← node.at(linkIt → first).modId
6      if (outFlowtoModules.count(newModId) > 0) then
7          outFlowtoModules[newModId]+ = linkIt → second
8      else
9          outFlowtoModules[newModId] ← linkIt → second
10         inFlowFromMod[newModId] ← 0.0
11         numModLinks ← numsModLinks + 1
12     end
13 end
14 Accumulate incoming flow in inFlowFromModules (as in Ln. 4 − 13)
15 bestDiffCodelen ← 0.0
16 for (it ← outFlowtoModules.begin() to outFlowtoModules.end()) do
17     newModId ← it → first
18     outFlowToNewMod ← it → second
19     inFlowFromMod ← inFlowFromModules[newModId]
20     diffCodeLen ← calc(outFlowToNewMod, inFlowFromMod)
21     if (diffCodelen < bestDiffCodelen) then
22         bestDiffCodelen ← diffCodelen
23         bestModuletoMove ← newModId
24     end
25 end
26 return bestModuletoMove
```

C. Components of A Parallel Infomap Algorithm

An efficient parallel implementation of Infomap, known as *HyPC-Map*, is presented in [14]. HyPC-Map has four major compute kernels–we briefly introduce them next to clarify our subsequent methods and contributions.

PageRank: This kernel computes the ergodic vertex visit probability (PageRank) for all of the vertices taking teleportation into account. The PageRank [10] is computed using the power iteration method. The ergodic vertex visit frequencies are used to compute the module stay probability $p^i_{\circlearrowleft}$ as well as the exit probability of a vertex $q_{i\curvearrowright}$ from module i.

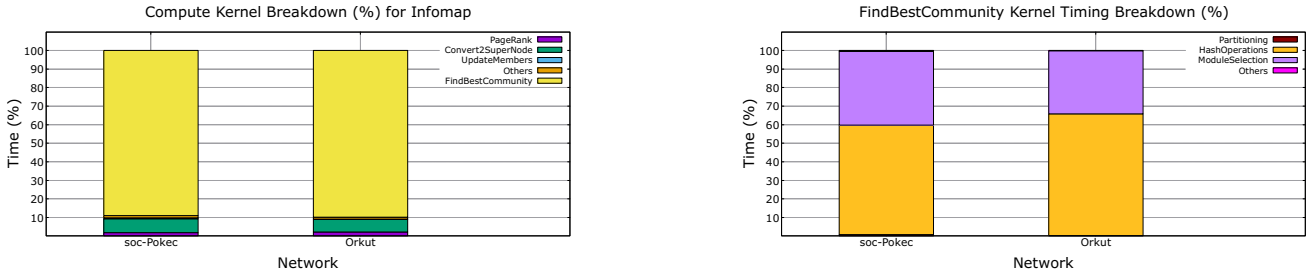
FindBestCommunity: This compute kernel is responsible for finding the optimal community for each vertex in a greedy manner. This kernel operates in the vertex level phase as well as in the super node level phase. During the vertex level phase, for each vertex, it greedily chooses the merge with the neighboring vertex that reduces the MDL in equation (1) the most. In the super node level phase, the generated groups of vertices from the vertex level phase are passed to this kernel in the form of a structure called a super node.

Convert2SuperNode: The groups of vertices generated in the vertex level phase in the *FindBestCommunity* kernel are represented by the structure called a super node. In a super node, the member components are all the vertices belonging to one group. The member vertices may be connected to other super nodes with edges. If multiple vertices of one super node are connected to another super node, a single super edge is created with accumulated edge weights.

UpdateMembers: After the *FindBestCommunity* kernel finds the new community membership for a vertex or a group of vertices, the community membership field for each of the vertices is updated.

D. Motivation for Accelerator

To understand the breakdown of computing costs of the major kernels of parallel Infomap, we experimented on some large networks. The results are shown in Fig. 2a. It is evident that *FindBestCommunity* is the most time-consuming kernel (yellow bar) in *Infomap* taking 70% to 90% of the complete application. Further, we observe that software hash operations (orange bar) take as much as 50% to 65% of the *FindBestCommunity* kernel (see Fig. 2b). For simplicity, all the plots illustrated in Fig. 2 are single-core execution of the application. This heavy use of software hash tables stems from the way how Infomap implementation makes the decision to change modules of an arbitrary vertex. From the discussion of the extended version of the *Map Equation* in [13], [33], it is sufficient to keep track of the exit probability $q_{i\curvearrowright}$ of a module i and the stay probability of the module $\sum_{\alpha \in i} p_\alpha$ to decide the next possible move for a vertex or super node. The exit probability and the stay probability are updated by the incoming and outgoing flow information from one module to another. A closer look into *FindBestCommunity* kernel is provided in Algorithm 1. During the module selection process for each vertex in Algorithm 1, each vertex/supernode maintains a pair of hash tables for storing the incoming flow and outgoing flow from/to the neighboring vertices/supernodes respectively. This flow information is used to compute the best reduction in MDL while processing any vertex. Because of this frequent use of the hash table, it becomes an expensive phase (orange bar) of the *FindBestCommunity* kernel. The software hash table, the C++ standard library unordered map, suffers from high latency-bound memory access due to branch



(a) Infomap

(b) FindBestCommunity

Fig. 2: The kernel breakdown of the Infomap application in native execution for large networks (Pokec and Orkut) in figure 2a. The majority of time is spent on *FindBestCommunity* kernel. A further breakdown in Fig. 2b shows hash operations taking 50% to 65% of that kernel computation time.

misprediction and collision chaining. At the beginning phase of FindBestCommunity (in Algorithm 1), each vertex belongs to its own community/module. As the algorithm progresses through multiple iterations of FindBestCommunity, the vertices of one module may move to another module. This is done in a greedy optimization fashion. A vertex can move to another module by following any of the connecting edges or by random teleportation. A vertex may have neighboring vertices or no neighbors at all. The neighboring vertices may belong to the same or different modules. A vertex may follow any of the edges to move to the module of another vertex if that move results in the maximum compression of the minimum description length described in Equation (1). The minimization/change in the MDL for a move of a vertex to a module is a function of edge flow to/from other neighboring vertices [33]. The flow to/from a module for a vertex is expressed as a function of the ergodic node visit frequency of the vertex itself and the edge weight to/from that module. In Algorithm 1, lines $1-2$ declare 2 hash tables for storing the outgoing flow to other modules and incoming flow from other modules. Lines $4-13$ describe iterating over the adjacency neighbors and storing/accumulating the module ID and corresponding outgoing flow as a $< key, value >$ pair. Line 14 refers to doing similar actions for the incoming flow from the modules to the current vertex. For simplicity, we omit to present the flow coming from teleportation in the algorithm snippet. Lines $16-25$ describe iterating over the $< key, value >$ pair and computing the difference in code length for a module (*newModId*). The difference in code length is computed by the function $calc(outFLowToNewMod, inFlowFromMod)$ in line 20. If moving to a module reduces the code length by more than the reduction observed so far, the change in code length is recorded along with the $moduleId$. From the description of the algorithm for *FindBestCommunity*, it is evident that most of the operations are hash table insertions, lookups, and accumulations of the flow value.

E. Pin and ZSim

To simulate our hardware accelerator ASA, we use ZSim [34], a Pin-based simulation infrastructure and tool. Pin is a program for instrumenting executables built in Linux, Windows, and macOS for Intel (R) IA-32, Intel64, and Itanium (R) processors. Pin is a dynamic instrumentation tool that intercepts the application binary during execution and injects instrumentation code snippets at desired locations, and allows inspection of program context information such as register states. It stores and restores context information when necessary so that the original execution flow is not affected by the instrumentation. Pintools are commonly used for hardware simulation as they can capture a natively executed instruction stream and then replay it on a simulated architecture.

Further, we made some changes to ZSim to model ASA. A software-implementation of the ASA architecture is controlled via custom instrumentation of the xchg instruction, which is not typically emitted by x86 compilers. We insert xchg instructions with the different registers to differentiate between inserting key-value pairs into the CAM (content addressable memory) and loading data from the CAM. These operations have associated latencies and use relevant ports within ZSim's out-of-order core model to correctly model the time spent executing ASA instructions. Lastly, the register values are read by ZSim to update the CAM state. This is important for example when determining whether or not an ASA insertion will cause overflow.

III. Methodology

We presented the *FindBestCommunity* kernel with software hash in Section II. In this section, we present the changes in design for the *FindBestCommunity* kernel using ASA. We provide the corresponding pseudocode in Algorithm 2. Further, in Fig. 3, we present the generalized block diagram of the ASA micro-architecture. The original work on ASA [42] discussed the ASA micro-architecture in detail for the SpGEMM computation. Here, we only discuss the API calls relevant to our context.

A. Hash Accumulation

The software hash accumulation in lines $4-13$ of Algorithm 1 is replaced by the ASA accumulation call in line 7 of Algorithm 2. Since each thread has its own core-local CAM, the *accumulate* API call takes four parameters, the thread ID (*tid*), the module ID (*k*), the hashed module ID (*hash(k)*) to

Algorithm 2: FindBestCommunity_ASA

Data: A vertex/supernode v_i of graph $G(V, E)$
Result: New community m_{new} for vertex v_i

1 $std :: vector < pair < key, value >> nonoverflowed\_pairs$
2 $std :: vector < pair < key, value >> overflowed\_pairs$
3 $tid \leftarrow omp\_get\_thread\_num()$
4 $numModlinks \leftarrow 0$
5 **for** $(linkIt \leftarrow v_i.outLinks.begin()$ to $v_i.outLinks.end())$ **do**
6 $\quad k \leftarrow node.at(linkIt \rightarrow first).modId$
7 $\quad accumulate(tid, hash(k), k, linkIt \rightarrow second)$
8 **end**
9 $gather\_CAM(tid, nonoverflowed\_pairs, overflowed\_pairs)$
10 **if** $(!overflowed\_pairs.empty())$ **then**
11 $\quad sort\_and\_merge(nonoverflowed\_pairs, overflowed\_pairs)$
12 **end**
13 **Accumulate** incoming flow (as in Ln. $5-12$)
14 **Iterate** over the merged vector and record the module ($bestModuletoMove$) that minimizes code length most
15 **return** $bestModuletoMove$

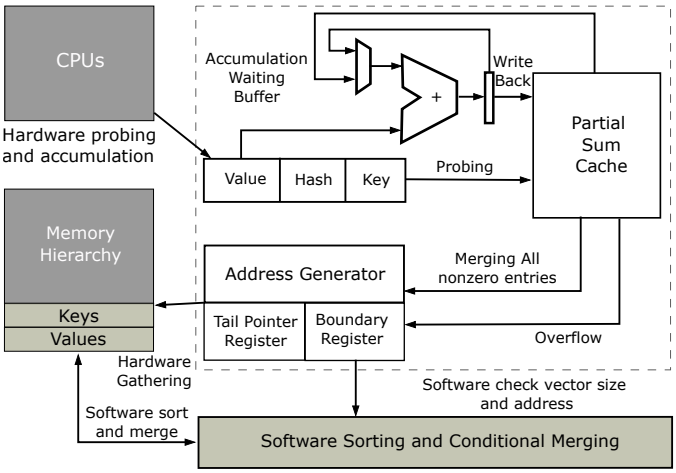


Fig. 3: Block diagram of the generalized ASA micro-architecture. Different modules of the architecture and their functionalities are described in Chao et al. [42] and therefore skipped here for brevity.

index the CAM entry, and the flow value ($linkIt \rightarrow second$) to accumulate in the corresponding CAM entry. There can be three possible outcomes due to the call to *accumulate*. If the key (*k*) is hashed to a unique index, a new entry is created in the cache. If the key already exists in the cache, the passed argument value is added to the partial sum. If the key is not found and there is no space available in the cache, an entry is evicted based on an LRU policy and stored in a queue buffer.

B. Gather CAM Entries

The API call *gather_CAM* in line 9 of Algorithm 2, takes the thread ID (*tid*) and the references to the vectors for copying the CAM entries back to the memory. Vector $nonoverflowed\_pairs$ gets the contents of CAM whereas vector $overflowed\_pairs$ gets the content of the overflowed queue buffer.

C. Sorting and Merging

In some cases, there might be an overflow due to the limited cache entries (CAM size). If the overflow buffer $overflowed\_pairs$ is not empty, then the overflowed pairs are pushed to the end of the $nonoverflowed\_pairs$ and then sorted by the keys. After that, the values of the same keys are merged. This process is presented in lines $10-12$ of Algorithm 2.

IV. Evaluation

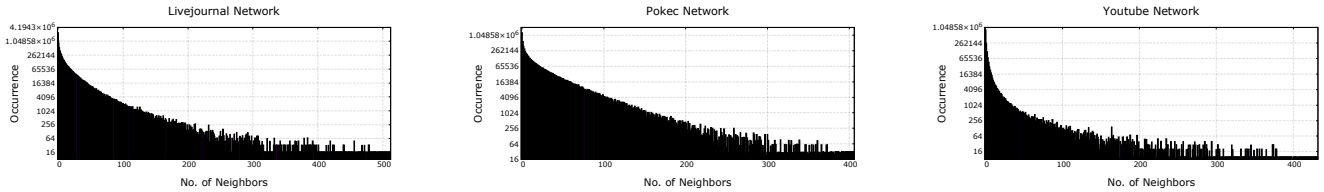
We use C++ for implementation and g++ 7.5.0 compiler for building and incorporating the ASA accelerator with HyPC-Map. The experiments and simulations are run on a Linux system with an Intel 2.6 GHz $64-$bit processor with 16 physical cores, 2 sockets, and 8 physical cores in each of the sockets. To match up the native configuration's CPU clock frequency to that of ZSim's simulated CPUs, the *scaling_governor* is set to *performance*. This ensures uniform (non-turbo mode) native CPU clock frequency. We use ZSim [34] as the hardware architecture simulation tool. The data sets used in this paper are collected from SNAP [20] and listed in Table I.

TABLE I: Network dataset for our experiments. We used several social and information networks

Network	# Vertices	# Edges
Amazon	334863	925872
DBLP	317080	1049866
YouTube	1134890	2987624
soc-Pokec	1632803	30622564
LiveJournal	3997962	34681189
Orkut	3072441	117185083

A. Utilizing Limited CAM Capacity

There is a trade-off between the cost of the on-chip memory of the accelerator (ASA) and the number of hash table entries that can be accommodated. Fortunately, because of the power-law degree distribution of the scale-free networks, most of the vertices do not have more than a few neighbors. Only a few vertices can have more than thousands of neighbors. Figure 4 shows this behavior for the 3 large social networks. In Figure 5, we show that having $8KB$ of memory per core is sufficient to cover the neighborhood list of 99% of the vertices for all the social networks shown in the plots. This observation can be exploited for biological networks because those networks follow similar sparsity and degree distribution.



(a) LiveJournal (b) Soc-pokec (c) YouTube

Fig. 4: Illustrating the general nature of scale-free social networks displaying power-law degree distribution. A few vertices may have high neighbor counts whereas the majority of the vertices (in this case millions of those vertices) have 0, or a few neighbors for *LiveJournal* network in Figure 4a, social *Pokec* network in Figure 4b, and *YouTube* network in Figure 4c.

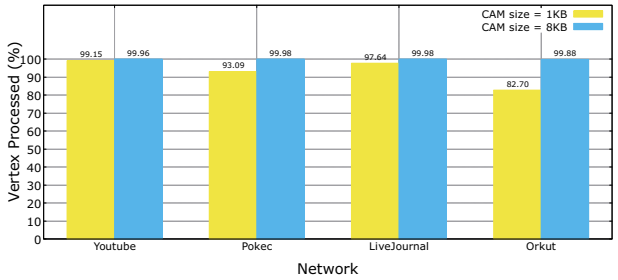


Fig. 5: Even small CAMs can store the majority of neighbor lists of graphs thanks to their power-law degree distributions nature. A core-local CAM of $1KB$ capacity can accommodate the neighbor lists of more than 82% of the vertices without overflowing. Using a capacity of $8KB$ covers more than 99% of the vertices of the social network data sets used in this paper.

B. Validation of Native vs Baseline

Before we compare the application performance of Infomap using hardware accelerated hash against software hash, we validate the *ZSim*-simulated performance of Infomap with software hash implementation [14] (which we call *Baseline*) against its performance from native execution on the same machine (without ZSim). *ZSim* [34] has been validated on Intel *Westmere* architecture with an average of $\sim 10\%$ error. For our experiments, we validate on an Intel *Ivy Bridge* architecture. We list the native hardware and ZSim simulation configurations in Table II in column 2, and column 3 respectively. Note that the $L3$ cache size of $20MB$ for *Native* configuration (column 2) cannot be replicated for *Baseline* (column 3) since ZSim requires the power of 2 for the cache sizes. The single core validation results for native execution vs Baseline are listed in Table III. The *FindBestCommunity* kernel is run for multiple iterations. We list the run time for each of the iterations in each row of the table for both the native and Baseline executions along with the percentage difference in column 4. The average error is $\sim 12.7\%$ between native and Baseline. Similarly, Table IV lists execution times for native *versus* Baseline execution for 2 processing cores. The difference in the run time between Native and Baseline could be coming from the difference in LLC ($L3$) cache sizes or the *ZSim*-induced simulation error.

TABLE II: Machine configurations for Native vs Baseline validation.

Item	Native	Baseline
Processor	8 cores, $2.6GHz$	8 cores, $2.6GHz$
L1 instruction cache	$32KB$	$32KB$
L1 data cache	$32KB$	$32KB$
L2	private $256KB$	private $256KB$
L3	shared $20MB$	shared $16MB$
Main Memory	$DDR3-1333MHz$, $CL\,1600MT/s$	$DDR3-1333MHz$, $CL10\,1600MT/s$

TABLE III: Runtime comparison in different iterations between Baseline and native using single processing core for the YouTube social network

Iteration no.	Native (sec)	Baseline (sec)	(% diff)
1	8.426	9.254	10
2	6.580	7.201	9
3	5.151	5.739	11
4	3.452	3.910	13
5	2.272	2.605	15
6	1.614	1.859	15
7	1.180	1.369	16

TABLE IV: Runtime comparison between baseline and native in different iterations using 2 processing cores for the YouTube social network

Iteration no.	Native (sec)	Baseline (sec)	(% diff)
1	5.676	5.572	2
2	4.072	4.055	1
3	3.275	3.186	3
4	2.048	2.026	1
5	1.238	1.466	18

TABLE V: Time spent on hash operations for Baseline vs ASA

Network	Baseline (sec)	ASA (sec)
Amazon	4.73	1.44
DBLP	7.35	1.86
YouTube	52.38	11.15
soc-Pokec	508.97	91.46
Orkut	1846.70	379.97

C. Performance Evaluation

We list the time spent on *HashOperations* in Table V– for *Baseline* in column 2 and *ASA* in column 3. Further, in

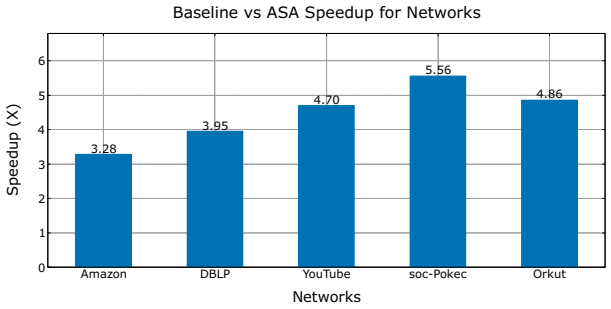


Fig. 6: ASA reduces the hash operations time compared to Baseline software hash. Running the experiments in single core for different networks exhibits $5.56\times$ speedup for *soc-Pokec* network, $4.86\times$ speedup for *Orkut* network, $4.70\times$ speedup for *YouTube* network, $3.95\times$ speedup for *DBLP* network, and $3.28\times$ speedup for *Amazon* network.

Fig. 6, we illustrate the speedup gain of *ASA* over *Baseline* for hash operations across different networks for single-core executions. We observe the highest single-core performance gain of $5.56\times$ for the *soc-Pokec* network. Similarly, *ASA* delivers $3.28\times$, $3.95\times$, $4.7\times$, and $4.86\times$ gain over *Baseline* for *Amazon*, *DBLP*, *YouTube*, and *Orkut* networks, respectively. Fig. 7 illustrates the performance breakdown of the computational kernels between Baseline and ASA for multi-core executions. We observe between $(68-70)\%$ reduction in *HashOperations* computation time from Baseline to ASA for multi-core execution for the *Amazon* network (Figure 7a). Similarly we observe between $(75-77)\%$ reduction in *HashOperations* time for the *DBLP* network (Figure 7b).

The performance improvement observed from Baseline to ASA comes from two primary reasons. ASA reduces the average number of instructions over software hash implementation. Software hash tables incur collision chaining or linear probing logic to handle collisions, which requires executing a few additional instructions. ASA's extension to ISA provides a single CPU instruction for hash lookup and accumulation. In Fig. 8a, we observe up to 24% reduction of the total number of instructions for the *FindBestCommunity* kernel for some larger networks. Fig. 9 illustrates 12% (Fig. 9a for *Amazon*) and 15% (Fig. 9b for *DBLP*) reductions in the average number of instructions per core from Baseline to ASA for the *FindBestCommunity* kernel during multi-core executions. The above statistics for the reduced number of instructions in ASA include the instructions for handling overflow (lines $10-12$ of Algorithm 2). For the *soc-Pokec* network, it only takes 9.86% of the ASA computation time (Table V, column 3) and for the *Orkut* network, it only takes 13.31% of ASA computation time to handle overflow.

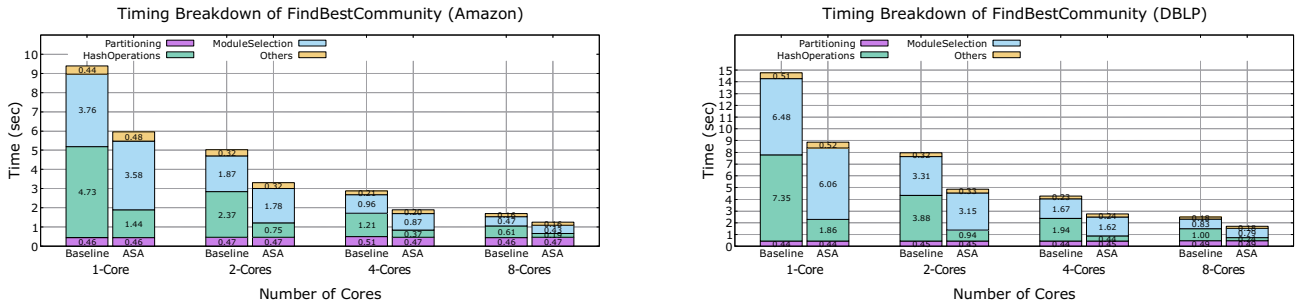
Furthermore, ASA's performance gain comes significantly from reducing the number of branch mispredictions in the software hash table. Branch mispredictions can be very expensive since the CPU core must flush all partially executed instructions from the incorrect branch from its pipeline and restart execution on the correct branch. Fig. 8b demonstrates up to 59% decrease in the number of mispredicted branches for larger networks. Fig. 10 demonstrates an average of 40% (Fig. 10a for *Amazon*) and 46% (Fig. 10b for *DBLP*) reductions in the number of average branch mispredictions per core across experiments running on different numbers of processing cores.

Moreover, resolving collisions in a software hash table often results in irregular memory access patterns that are difficult for hardware prefetchers to predict (e.g., to follow pointers connecting entries that hash to the same bucket), potentially causing memory latency stalls. Reducing the number of branch mispredictions and irregular memory accesses due to hash collisions results in lower CPI for ASA over Baseline. Fig. 8c demonstrates $(18-21)\%$ reduction in CPI for some larger networks (*YouTube, soc-Pokec, and Orkut*) in a single-core execution. Similarly, in a multi-core execution, Fig. 11 shows a 20% reduction in CPI rate for the *Amazon* network (Fig. 11a) and 21% reduction in CPI rate for the *DBLP* network (Fig. 11b) on average per core from Baseline to ASA.

V. Related Work

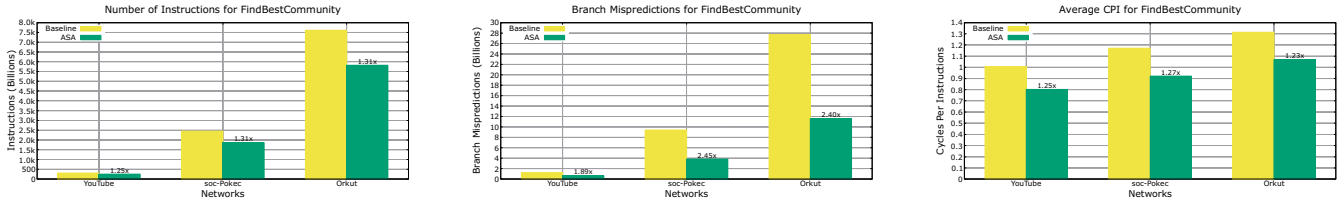
Using a hardware accelerator for efficient graph mining applications is an important design concept for software-hardware co-design. In general, the term graph data mining refers to the strategies of discovering structural information such as community, cliques, motifs, k-truss, and other patterns. Graph data mining on real-world data sets suffers from under-utilization of computing resources due to random access patterns of the irregular graph structure and significant load imbalance due to power-law degree distribution. The set of possible patterns can grow exponentially with the size of the input graph. All these motivate the hardware accelerator for graph mining. While for machine learning algorithms, there are a set of mature computational kernels and accelerators for different kernels, the area of graph data mining is yet to reach that point.

One graph mining accelerator, Gramer [39], is based on the observation that most random memory requests come from a small fraction of vertex and edge data. Gramer maintains a smart cache hierarchy where the most frequently used data/pattern is resident on a top level with a no-eviction policy and the second level is on-chip memory with a lightweight replacement policy. The difficulty in this approach lies in identifying and managing *valuable* data. Flexminer [11], another graph mining accelerator packaged with its compiler program that operates on dedicated on-chip storage, memoizes reusable connectivity information on a connectivity map (c-map) and does not address the issue of fast accumulation of values against keys. Another work IntersectX [31] designs a stream-based ISA extension following the observation that many graph pattern mining applications use the intersection of two edge lists as core computation. Another accelerator called SISA [8] identifies a similar set of operations (union, intersection) and designs a set-centric ISA extension to process with on-chip memory. Rao et al. [32] propose an accelerator *SparseCore* for sparse tensor and graph pattern computation for streaming data or sparse vector. Adam et al. [2] propose a



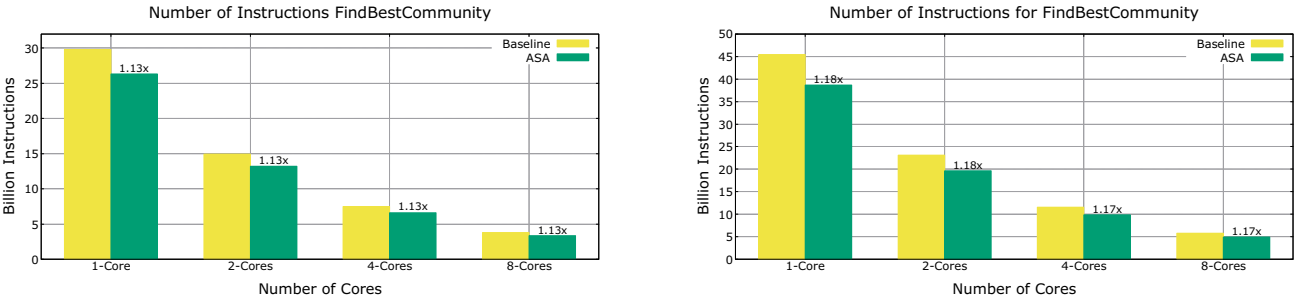
(a) Amazon (b) DBLP

Fig. 7: Timing breakdown of the simulated kernel (*FindBestCommunity*). For different numbers of processing cores, the reduction in execution time for *HashOperations* is presented. For single core, the *HashOperations* time reduces from 4.73 seconds to 1.44 seconds for the Amazon network (Fig. 7a), and for the DBLP network (Fig. 7b) the single core run time reduces from 7.35 seconds to 1.86 seconds.



(a) Total Instructions (b) Branch Misprediction (c) Average CPI

Fig. 8: The comparison between *Baseline* and *ASA* for big networks. Fig. 8a shows the total number of executed instructions reduced from *Baseline* to *ASA*. For the *soc-Pokec* network, the total number of instructions in *Baseline* is 2.4 trillion and reduced to 1.8 trillion. Fig. 8b shows a reduction of the number of mispredicted branches from *Baseline* to *ASA*. For the *Orkut* network, it reduces from 27.69 billion in *Baseline* to 11.55 billion in *ASA*. Finally, Fig. 8c shows $(18-21)\%$ reduction in the average number of cycles per instruction (CPI) from *Baseline* to *ASA* for the big networks.



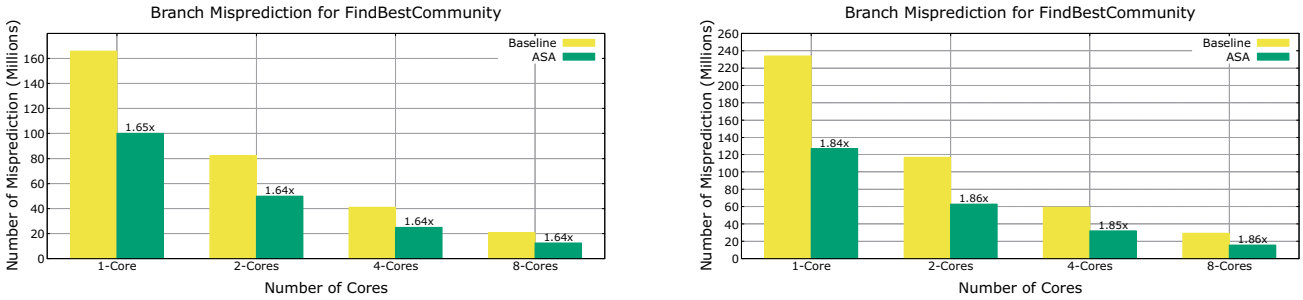
(a) Amazon (b) DBLP

Fig. 9: The average number of instructions per core reduced from Baseline to ASA. The reduction factor is consistent across multi-core executions for the Amazon network and for the DBLP network respectively

hardware accelerator with dedicated hardware units to handle the irregular data movements of graph computation in Graph Neural Network (GNN) that can be used to solve community detection problems. None of the literature [2], [8], [11], [31], [32], [39] deal with the problem of accelerating key-value matching and accumulation for vertex-neighborhood connectivity which is addressed in our work. Yang et al. [38] designed a hash accelerator using FPGA on-chip SRAM with scalability limited to 16 processing engines (PE). Zhang et al. [43] design a hash accelerator by ISA extensions and hardware changes with most operations handled by the accelerator and rare cases handled by software. The work [43] proposed two implementations of the hash accelerator, Flat-HTA, and Hierarchical HTA. Chao et al. [42] show both versions can be outperformed by ASA in their comparison of the SpGEMM computation.

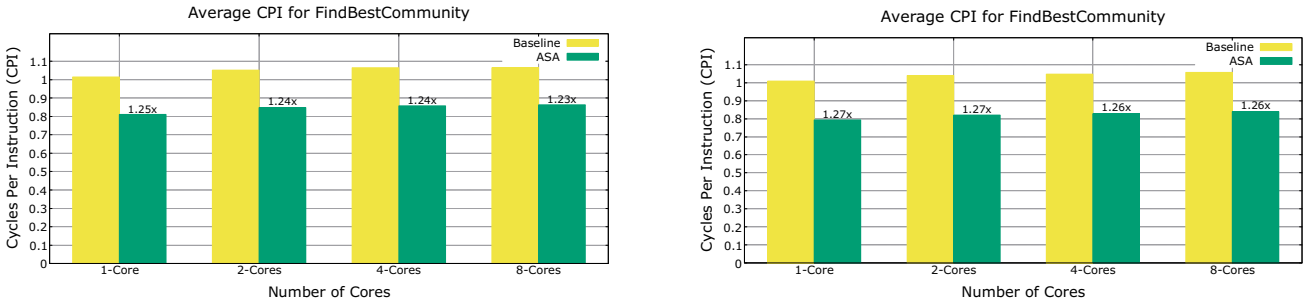
VI. Conclusion

Community discovery is a popular application for salient motif discovery on social and relational networks. Using hardware accelerators to speed up the computation of certain parts of the software kernel has a very high impact on hardware-software co-design. To the best of our knowledge, our work is



(a) Amazon (b) DBLP

Fig. 10: Number of branch misprediction reduced from Baseline to ASA. The reduction factor is consistent across multi-core executions for the Amazon network and the DBLP network respectively



(a) Amazon (b) DBLP

Fig. 11: Average CPI (the number of cycles retired per instruction) reduced from Baseline to ASA. The reduction factor is consistent across multi-core executions for the Amazon network and the DBLP network respectively

the first one to incorporate a hardware accelerator for sparse accumulation for information-theoretic community discovery. We have articulated in this study that existing implementations of *Infomap* with the regular software hash kernel are not well-served by the general-purpose CPUs. In this work, we demonstrate that in addition to the optimization from the algorithmic side, there remains an opportunity to gain higher performance throughput by using an accelerator that computes certain computations more efficiently than the general-purpose hardware. Existing accelerators for graph pattern mining fail to address accelerating key-value lookup and accumulation. Through this work, we show that the ASA accelerator can accelerate not only the SpGEMM kernel, but also other applications that heavily rely on hash lookup and accumulation by reducing the number of branch mispredictions, average CPI, and the total number of instructions.

Acknowledgements

This work has been partially supported by a National Science Foundation (NSF) grant (Award # 2132212) and a US Department of Energy/Berkeley Lab/University of California Subcontract Award # 7551418 (Prime Award # DE-AC02-05CH11231).

References

[1] R. Aldecoa and I. Marìn, "Exploring the limits of community detection strategies in complex networks," *Scientific Reports*, vol. 3, p. 2216, Jul 2013.

[2] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[3] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic Acids Research*, vol. 46, no. 6, pp. e33–e33, 01 2018.

[4] S.-H. Bae, D. Halperin, J. West, M. Rosvall, and B. Howe, "Scalable flow-based community detection for large-scale network analysis," in *2013 IEEE 13th International Conference on Data Mining Workshops*, Dec 2013, pp. 303–310.

[5] S.-H. Bae and B. Howe, "Gossipmap: a distributed community detection algorithm for billion-edge directed graphs," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12.

[6] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009.

[7] V. Batagelj, "Protein-protein interaction network in budding yeast."

[8] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 282–297.

[9] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, Oct 2008.

[10] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, APR 1998.

[11] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 581–594.

[12] A. Clauset, M. E. J. Newman, and C. Moore, "Finding community structure in very large networks," *Phys. Rev. E*, vol. 70, p. 066111, Dec 2004.

[13] M. A. M. Faysal and S. Arifuzzaman, "Distributed community detection in large networks using an information-theoretic approach," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 4773–4782.

[14] M. A. M. Faysal, S. Arifuzzaman, C. Chan, M. Bremer, D. Popovici, and J. Shalf, "Hypc-map: A hybrid parallel community detection algorithm using information-theoretic approach," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–8.

[15] S. Fortunato and M. Barthélemy, "Resolution limit in community detection," *Proceedings of the National Academy of Sciences*, vol. 104, no. 1, pp. 36–41, 2007.

[16] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[17] R. Guimerà, M. Sales-Pardo, and L. A. N. Amaral, "Modularity from fluctuations in random graphs and complex networks," *Phys. Rev. E*, vol. 70, p. 025101, Aug 2004.

[18] A. Lancichinetti and S. Fortunato, "Community detection algorithms: A comparative analysis," *Phys. Rev. E*, vol. 80, p. 056117, Nov 2009.

[19] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical Review E*, vol. 78, no. 4, oct 2008.

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[21] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining Social-Network Graphs*, 2nd ed. Cambridge University Press, 2014, p. 325–383.

[22] W. Li and A. Godzik, "Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences," *Bioinformatics*, vol. 22, no. 13, pp. 1658–1659, May 2006.

[23] A. Madhavan, R. Sindhu, B. Parameswaran, R. K. Sukumaran, and A. Pandey, "Metagenome analysis: a powerful tool for enzyme bioprospecting," *Applied Biochemistry and Biotechnology*, vol. 183, no. 2, pp. 636–651, Oct 2017.

[24] C. P. Massen and J. P. K. Doye, "Identifying communities within energy landscapes," *Phys. Rev. E*, vol. 71, p. 046101, Apr 2005.

[25] A. Medus, G. Acuña, and C. Dorso, "Detection of community structures in networks via global optimization," *Physica A: Statistical Mechanics and its Applications*, vol. 358, pp. 593–604, Dec 2005.

[26] M. Naim, F. Manne, M. Halappanavar, and A. Tumeo, "Community detection on the gpu," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 625–634.

[27] M. E. J. Newman, "Finding community structure in networks using the eigenvectors of matrices," *Physical Review E*, vol. 74, no. 3, Sep 2006.

[28] M. E. J. Newman, "Spectral methods for community detection and graph partitioning," *Physical Review E*, vol. 88, no. 4, Oct 2013.

[29] T. P. Peixoto, "Efficient monte carlo and greedy heuristic for the inference of stochastic block models," *Phys. Rev. E*, vol. 89, p. 012804, Jan 2014.

[30] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi, "Defining and identifying communities in networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. 9, pp. 2658–2663, 2004.

[31] G. Rao, J. Chen, and X. Qian, "Intersectx: An accelerator for graph mining," *ArXiv*, vol. abs/2012.10848, 2020.

[32] G. Rao, J. Chen, J. Yik, and X. Qian, "Sparsecore: Stream isa and processor specialization for sparse computation," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 186–199.

[33] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *Proceedings of the National Academy of Sciences*, vol. 105, no. 4, pp. 1118–1123, 2008.

[34] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, p. 475–486, jun 2013.

[35] N. S. Sattar and S. Arifuzzaman, "Parallelizing louvain algorithm: Distributed memory challenges," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing (DASC 2018), Athens, Greece, August 12-15, 2018*, 2018, pp. 695–701.

[36] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[37] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "Fast algorithm for modularity-based graph clustering," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, ser. AAAI'13. AAAI Press, 2013, p. 1170–1176.

[38] Y. Yang, S. R. Kuppannagari, A. Srivastava, R. Kannan, and V. K. Prasanna, "Fasthash: Fpga-based high throughput parallel hash table," *High Performance Computing*, vol. 12151, pp. 3 – 22, 2020.

[39] P. Yao, L. Zheng, Z. Zeng, Y. Huang, C. Gui, X. Liao, H. Jin, and J. Xue, "A locality-aware energy-efficient accelerator for graph mining applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 895–907.

[40] J. Zeng and H. Yu, "A distributed infomap algorithm for scalable and high-quality community detection," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: ACM, 2018, pp. 4:1–4:11.

[41] J. Zeng and H. Yu, "Effectively unified optimization for large-scale graph community detection," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 475–482.

[42] C. Zhang, M. Bremer, C. Chan, J. Shalf, and X. Guo, "Asa: Accelerating sparse accumulation in column-wise spgemm," *ACM Trans. Archit. Code Optim.*, may 2022, just Accepted.

[43] G. Zhang and D. Sanchez, "Leveraging caches to accelerate hash tables and memoization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 440–452.