MultiPIM: A Detailed and Configurable Multi-Stack Processing-In-Memory Simulator

Chao Yu, Sihang Liu, Samira Khan

Abstract—Processing-in-Memory (PIM) has being actively studied as a promising solution to overcome the memory wall problem. Therefore, there is an urgent need for a PIM simulation infrastructure to help researchers quickly understand existing problems and verify new mechanisms. However, existing PIM simulators do not consider architectural details and the programming interface that are necessary for a practical PIM system. In this letter, we present MultiPIM, a PIM simulator that models microarchitectural details that stem from supporting multiple memory stacks and massively-parallel PIM cores. On top of the detailed simulation infrastructure, MultiPIM provides an easy-to-use interface for configuring PIM hardware and adapting existing workloads for PIM offloading.

Index Terms—Processing-in-memory, simulator, memory network.

1 Introduction

Modern workloads, such as scientific computing, graph processing, data mining, and machine learning pose an ever-growing demand for larger data. Despite the rapid increase in processor and memory performance, due to the limited bandwidth between the processors and the memories, the memory wall becomes a major performance bottleneck. To mitigate the memory wall, the notion of processing-in-memory (PIM) or near-data-processing (NDP) has been proposed, which moves processing to the memory system for lower memory access latency and better utilization of the internal memory bandwidth. Following this direction, there have been a myriad of PIM systems [1–4]. Meanwhile, due to the lack of real PIM hardware infrastructure, there is a demand for an accurate PIM modeling and simulation platform.

There have been existing simulators, such as PIMSim [2] and Ramulator-PIM [3, 5], that model and simulate PIM architectures. As the research on PIM moves forward, we find there are new demands for the simulation infrastructure. Prior works focus on a single memory stack, without considering the complexities in practical PIM systems. However, to meet the need for high memory capacity and bandwidth, practical PIM systems will consist of multiple memory stacks, and therefore, the complexity increases. In a multi-stack PIM system, issues such as local/remote-stack latency and coherence among PIM processors are key performance factors. Virtual memory is another practical problem in unified memory systems. As accelerators and GPUs are moving toward the unified memory [6] for better programmability, we expect PIM systems will also follow the same trend. To reflect these practical considerations, the simulator needs to accurately model and simulate these additional supports. On the other hand, the increasingly growing research on PIM has new demands for customizability. First, a simulator should be configurable to model different PIM architectures, such as PIM systems with different multi-stack memory topologies. Second, a simulator should also have a flexible interface for programmers to offload code regions to PIM processors. Table 1 compares MultiPIM with the existing PIM simulators in terms of the simulated components (Comp.) and the flexibility (Flex.).

• C. Yu, S. Liu and S. Khan are with the University of Virginia, Charlottesville, VA, 22903.
E-mail: {yc9uf, sihangliu, samirakhan}@virginia.edu

TABLE 1. Features Comparison.

	Feature	PIMSim	Ramulator-PIM	MultiPIM
Comp.	**Multi-stack**	No	No	Yes
	PIM core cohere.	No	No	Yes
	Virtual memory	No	No	Yes
Flex.	**Offloading interface**	Trace file	Source code	Source code
	Parallel offloading	No	Yes	Yes
	End-to-end sim.	No	No	Yes

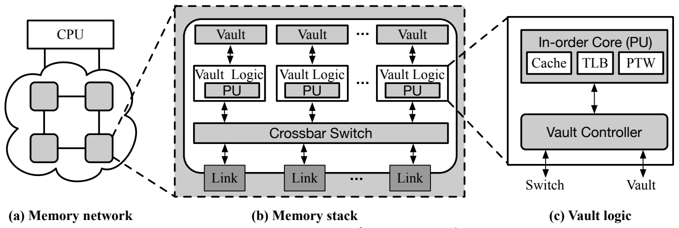


Fig. 1: An overview of PIM architecture.

The *goal* of this work is to design a PIM simulator that is not only highly-configurable and easy-to-use but also precise in modeling architectural details of practical PIM systems. In this work, we implement *MultiPIM*[1], a simulator that meets these demands. We next introduce the high-level design of MultiPIM in two aspects: architecture modeling and usability. MultiPIM models a multi-stack PIM system connected by a memory network, as illustrated in Figure 1, where each memory stack is further divided into vaults. Each vault contains a PIM processor, and these processors across vaults are connected through the on-chip network (i.e., via crossbar switches). MultiPIM precisely models the interconnects by placing cycle-accurate routers among the memory stacks, and crossbars among the vaults. For better usability, the interconnect topology and the timing parameters can be easily configured by modifying configuration files, without changing the simulator's source code. To simplify the adaption of workloads, MultiPIM provides simple interfaces that allow users to annotate PIM-offload code regions. MultiPIM automatically recognizes POSIX and OpenMP threads within the annotated regions and simulates them on the massively-parallelized in-memory processors. The contributions of this work are the followings:

- This work is the first open-sourced simulator that supports a configurable multi-memory-stack PIM system.

1. Available at https://github.com/Systems-ShiftLab/MultiPIM.

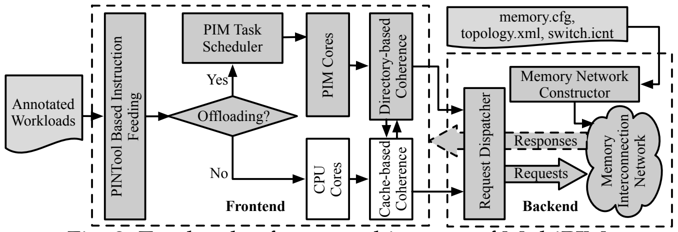

Annotated Workloads
PINTool Based Instruction Feeding
PIM Task Scheduler
Yes
Offloading?
No
PIM Cores
CPU Cores
Directory-based Coherence
Cache-based Coherence
Frontend
memory.cfg, topology.xml, switch.icnt
Memory Network Constructor
Request Dispatcher
Responses
Requests
Memory Interconnection Network
Backend

Fig. 2: Top-level software architecture of MultiPIM.

- MultiPIM precisely simulates PIM architectural details, including the multi-stack interconnect, crossbar switches, PIM-core coherence, and virtual memory, which are essential to a practical PIM system.
- MultiPIM comes with an easy-to-use interface that allows the users to not only configure the architecture but also adapt existing workloads for evaluation.

2 MultiPIM: Simulator Design

The high-level structure of MultiPIM consists of two parts, the frontend and the backend, as illustrated in Figure 2. The frontend performs instruction simulation while the backend handles memory access simulation. These two parts are connected through a predefined interface and work in collaboration as a complete simulator. The decoupled frontend and backend of the simulator make it easier to extend or replace each part. For example, the current ZSim-based frontend can be replaced by other simulators (e.g., GEM5 and GPGPU-Sim), by adapting the new front end to the existing backend interface. When executing a workload with MultiPIM, PIM code regions are executed on PIM cores and the rest are executed on the host CPU. To support the multithreaded PIM execution, MultiPIM uses a PIM task scheduler that assigns tasks to PIM cores. Next, we describe the frontend and backend in detail.

Frontend. The frontend of MultiPIM handles non-memory instructions and cache accesses. As they typically have fixed latencies, MultiPIM uses a simple timing model in the frontend. Whereas, the latency of memory requests (i.e., memory instructions that miss the cache) varies. Therefore, MultiPIM simulates cache miss events using the precise backend for better accuracy.

Backend. The Ramulator-based [3, 5] backend simulates the actual latency of memory requests issued from the frontend by routing packets through the memory network, where the connections among the multiple memory stacks are constructed from a user-defined configuration file (discussed in Section 3.1). The frontend memory requests, issued by PIM cores or the CPU, are all gathered at a request dispatcher. Then, the request dispatcher issues memory requests to the corresponding component in the backend (i.e., link, crossbar switch or vault) according to the requester type and destination (e.g., a request from a PIM core goes through its crossbar switch to a remote stack). Once a memory request is serviced by a memory stack, the corresponding response is routed back to the requester.

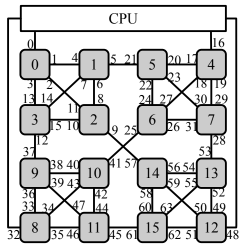
3 Detailed Design and Features

3.1 Multiple Memory Support

Modern applications such as scientific computing and data analytics are becoming increasingly memory intensive. Thus, having multiple memory stacks is a necessity to satisfy the need for both memory capacity and bandwidth. One of the key features of MultiPIM is the support for accurate simulation of a memory network that connects multiple memory stacks. Our implementation takes memory timing parameters from the HMC specification [7]. Note that other memory systems can also be easily integrated, as the Ramulator-based backend already supports various memory technologies.

```
<memtopology>
  <memnodes num="16" linkspernode="4">
    <node id="0"><link id="0" tocpu="true"/></node>
    <node id="4"><link id="16" tocpu="true"/></node>
    ...
  </memnodes>
  <meminterconnections>
    <interconnection from="1" to="4"/>
    <interconnection from="3" to="13"/>
    ...
  </meminterconnections>
</memtopology>
```

(a) XML based memory interconnection definition



(b) Generated memory topology

Fig. 3: Format of (a)memory interconnection definition and (b)the corresponding memory topology.

Flexible Memory Interconnection Definition. To support different memory topologies, MultiPIM generates the memory network from a user-defined XML configuration file. Figure 3 showcases its format and the generated memory topology. The number of memory stacks and the number of links per memory stack are defined by the num and $linkspernode$ attributes of the $memnodes$ tag. According to num and $linkspernode$, each memory node/stack is assigned a unique $node_{id}$ (from 0 to $num-1$), and each of its link is assigned a unique $link_{id}$ (from $node_{id} * linkspernode$ to $(node_{id}+1) * linkspernode - 1$). To define the type of each link, i.e., whether it connects to the CPU or other memory stacks, users need to specify the $tocpu$ attribute of the link: a $true$ value indicates it connects to the CPU; a $false$ value (default case) indicates it connects to other memory stacks. Then, to configure the connection among memory stacks, the user needs to set the $interconnection$ tag: a pair of $link_{id}$s represents an interconnection, and a link only connects two memory stacks. Further, the interconnection type can be configured as $undirected$ (default) or $directed$. In an example of $link_a$ to $link_b$, the $directed$ configuration allows packets to be transmitted only from $link_a$ to $link_b$; the $undirected$ configuration allows a bidirectional transmission.

Memory Network Timing Modeling. The timing model of the memory network mainly consists of the crossbar switches and the links. MultiPIM uses a cycle-accurate interconnection network simulator, BookSim2 [8], to model the timing of crossbar switches. This design makes MultiPIM extensible to different NoC topologies (e.g., mesh, dragonfly, and torus). In MultiPIM, each crossbar switch has a separate sub-net, and all vault controllers and links in a memory stack are connected to its sub-net as different nodes. When a packet is being transmitted between two vaults, MultiPIM models the crossbar switch latency by first injecting the packet to the crossbar switch from the source node and then ejecting it from the target node. Every cycle, the crossbar switch first ejects packets for all vault controllers and links, and then injects packets from those vault controllers and links. Each link and its vault controllers avoid injecting and ejecting packets within the same cycle to avoid bus conflicts. Besides, MultiPIM also supports an optimization that divides the vaults in a memory stack into groups (or *quadrants* [7]) and associates each vault group with a link, where the number of groups is the same as the number of links in a memory stack. This optimization enables packets to be transmitted directly in a *quadrant*, without going through the crossbar switch.

MultiPIM precisely models the timing of memory links. During transmission, MultiPIM first calculates its transmission time (in cycles) $linkcycle_{packet}$ based on its length and the configured link speed. Upon completion of transmission, MultiPIM adds this time to the next transmission $linkcycle_{next}$, indicating the earliest transmission time of the next packet. This way, MultiPIM precisely maintains the timing of packet transmission through the memory network.

Request Dispatcher. The request dispatcher receives memory

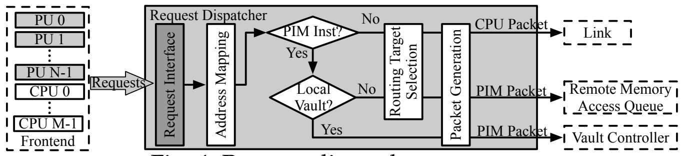


Fig. 4: Request dispatcher structure.

requests from all requesters (i.e., PIM cores and CPU cores) in the frontend, and then dispatches them to different components in the backend. Figure 4 illustrates the architecture of the request dispatcher. The frontend sends requests to the request dispatcher by calling the request interface. Upon an interface call, the physical-address of a request is disassembled and mapped to the internal memory address, including cube (or stack), vault, bank, and DRAM address, by the predefined address mapping scheme. For example, the address can be mapped as RW:CH:BK:CB:VT:CL:BO or as RW:BK:CH:CB:VT:CL:BO.[2] Afterward, the request is dispatched to a certain component (i.e., link, crossbar switch or vault). If the requester is a CPU core, the memory network first selects a link that connects to the CPU, and then generates a packet to the destination memory stack, according to the request and the link ID. If the requester is a PIM core, the memory network dispatches the request based on the following rules. A request that targets the local vault will directly perform local memory access; other requests will first enter the memory stack's crossbar and then be routed to the destination vault or memory stack. For better accuracy, MultiPIM simulates the queuing latency during routing.

Packet Routing between Memory Stacks. As described in Section 3.1, all the vault controllers and links in a memory stack are regarded as individual nodes that connect to a crossbar switch. Therefore, to access a remote memory stack, either from the host CPU or a PIM core, packets need to go through the memory network that consists of these switches and links. By default, MultiPIM follows the user-defined routing policy (e.g., a shortest-path routing algorithm). It can also be extended to support other dynamic routing algorithms (e.g., a load-aware routing algorithm).

3.2 Practical System Considerations

Massive PIM cores. A practical PIM system may consist of dozens of memory stacks with thousands of PIM cores, which not only increases the parallelism of tasks but fully utilizes the terabytes-per-second internal bandwidth of the whole system. MultiPIM's frontend is implemented based on ZSim [9] to support fast simulation for massive PIM cores. The implementation of PIM cores is currently based on ZSim's *TimingCore* (also supports other types of ZSim cores). Besides, as described in Section 2, the frontend of MultiPIM can integrate other simulators (e.g., GEM5 and GPGPU-Sim) to mode PIM cores, by connecting to the predefined MultiPIM interfaces and sending memory requests to the request dispatcher in the backend.

Multi-PIM-core Coherence. The coherence problem arises as threads executing on different PIM cores may share data. Existing works only consider coherence between CPU cores and PIM cores [4], or rely on strict restrictions to avoid coherence between PIM cores [10], which hinders the use of PIM for general-purpose processing. As PIM cores are residing in different memory stacks, without a shared last-level cache (LLC), we implement a coherence directory (can be either shared or private) with the MESI protocol for PIM cores. MultiPIM supports both write-through and write-back cache policies. Our implementation has flexible abstract interfaces for future researchers to integrate other coherence protocols.

2. RW: Row, CH: Column High, CL: Column Low, BK: Bank, CB: Cube, VT: Vault, BO: Byte Offset

TABLE 2. Offloading interfaces.

Interface	Description
PIM Multi-Processing interface	
`pim_mp_begin()`	Annotate the begin of multiple PIM tasks(e.g., POSIX or OpenMP threads).
`pim_mp_end()`	Annotate the end of multiple PIM tasks
PIM Block interface	
`pim_blk_begin()`	Annotate the begin of a PIM code block
`pim_blk_end()`	Annotate the end of a PIM code block

Virtual Memory. Although placing PIM in a separate address space eliminates the need for address translation for PIM cores, it increases the complexity of programming and hinders the cooperation between CPU and PIM cores. A unified memory [6] simplifies programming and is becoming the trend in heterogeneous computing–devices such as GPUs and accelerators are moving toward a unified address space. Following this trend, MultiPIM supports virtual memory with different page sizes (e.g., 4KB page and 2MB page) following the Linux's buddy memory management mechanism. Both the CPU and PIM cores in MultiPIM can be configured with translation lookaside buffers (TLBs), both instruction-TLB and data-TLB, and a page table walker (PTW). PTWs in the CPU cores are connected to the LLC to reduce the page-table-walk overhead; however, as there is no LLC in PIM cores, PTWs in PIM cores access memory directly. All PTW requests are sent to the backend and simulated as normal memory requests for better accuracy.

3.3 Parallel Computing and Offloading Interfaces

The PIM architecture is intrinsically suitable for parallel computing because of its massive number of cores. Thus, to make full use of the PIM computing resources, MultiPIM supports two widely-used multithreading interfaces: POSIX and OpenMP threads. MultiPIM can automatically offload threads in the PIM regions to parallel PIM cores.

MultiPIM provides two easy-to-use offloading interfaces to annotate the PIM regions: A *multi-processing* interface and a *code-block* interface, as listed in Table 2. The multi-processing interface marks a large region of the source code and offloading all the parallel tasks within the region to PIM. Users only need to call the $pim\_mp\_begin()$ and $pim\_mp\_end()$ functions to select the region. Inside this region, all new threads (e.g., an OpenMP thread group) will be scheduled to different PIM cores. The code-block interface offloads a finer-grained code segment within a thread, using a pair of $pim\_blk\_begin()$ and $pim\_blk\_end()$. Using a combination of these two methods, the user can choose the optimal offloading strategy.

3.4 Limitations and Future Directions

Energy and Power Modeling. Existing open-sourced power models (e.g., McPat and DRAMPower) can be used in conjunction with MultiPIM. However, we leave power modeling as a future direction. Future researchers can plug MultiPIM's statistics into their energy/power models.

Heterogeneous Architectures. PIM and host cores in MultiPIM have the same ISA. However, different ISAs in PIM and CPU cores may provide better performance benefit. It is possible to extend MultiPIM to support a different ISA for PIM cores as long as the user can provide a trace for that ISA.

4 Evaluation and Validation

Configurations and Workloads. Table 3 lists the configuration of the simulated system. We use data-intensive workloads from gapbs [11] (CC, PR and TC), with three real-word input

TABLE 3. Simulation configurations.

	Parameter	Value
CPU	Core	16 out-of-order cores @ 3.2GHz, 4-issue
	L1 Cache	I-Cache: 32KB, 4-way; D-Cache: 32KB, 8-way
	L2 Cache & LLC	L2: 256KB, 8-way; LLC: 16MB, 16-way
	TLB	I-TLB: 256 entries; D-TLB: 256 entries
Memory	DRAM Timing	FR-FCFS, tCL=tRCD=tRP=17ns, tCWL=13ns
	HMC	4GB/cube, 32 vaults, 4 layers, 8 banks/vault, 4 links/cube, 16-lane/link, 30Gb/s per lane, clock @ 1.25GHz (links, logic layer, switch)
	Crossbar	2D-Torus topology
	Topologies	16 cubes, Dragonfly & Mesh, minimal routing
PIM	Core	1 in-order-stall core @ 2GHz per vault
	L1 Cache	I-Cache: 16KB, 4-way; D-Cache: 16KB, 4-way
	TLB	I-TLB: 128 entries; D-TLB: 128 entries

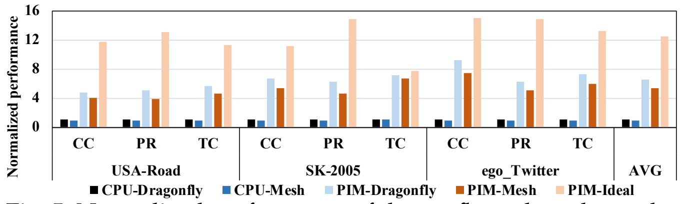


Fig. 5: Normalized performance of dragonfly and mesh topologies. All results are normalized to the CPU baseline.

graphs: USA-Road [12], SK-2005 [13], and ego-Twitter [14]. The OpenMP threads in these workloads are offloaded to PIM using the *multi-processing* offloading interface. The baseline performs computation on the CPU, where the CPU is connected to 16 memory stacks, configured with the dragonfly topology. We evaluate the performance impact of a PIM system with four PIM design choices: (1) multiple stacks with remote memory accesses, (2) memory network topology, (3) virtual memory support, and (4) cache write policies.

Evaluation. Figure 5 shows the performance of different memory topologies: dragonfly, mesh, and an ideal case without remote-access overhead. The performance numbers are normalized to the CPU baseline. First, both the dragonfly and mesh PIM designs have better performance than the CPU baseline (i.e., *CPU-Dragonfly*). And, the dragonfly topology (6.5× speedup over *CPU-Dragonfly*) is relatively more efficient than mesh (5.3× speedup over *CPU-Dragonfly*) as it reduces the number of hops when accessing the remote memory. Second, despite the speedup from PIM, there is an approximately 2× performance gap between the more efficient dragonfly topology and the ideal topology, indicating that remote memory access is a major performance bottleneck. Third, Figure 6(a) shows that a practical PIM system (*Dragonfly-VM*) with virtual memory translation exhibits a 25% overhead, as compared to the ideal design (*Dragonfly-IdealVM*). Thus, address translation also has a significant performance impact. Fourth, Figure 6(b) compares the performance of different cache policies: *Write-Back* (*Dragonfly-WB*) and *Write-Through* (*Dragonfly-WT*). Overall, the *Write-Back* cache has better performance than *Write-Through*. However, some workloads (TC+USA-Road, CC+ego_Twitter and TC+ego_Twitter) significantly benefit from *Write-Through*.

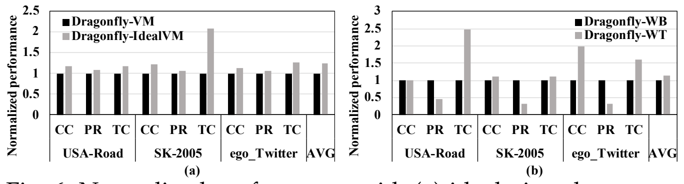


Fig. 6: Normalized performance with (a) ideal virtual memory (i.e. zero address translation overhead) and (b) different cache write policies.

We find that most memory accesses are reads (>99.9%) in these these workloads. As such, the *Write-Through* policy can avoid the cache coherence traffic in most scenarios, such as directory probing, caused by the *Write-Back* policy. This result shows that future PIM designs may need to consider dynamic policies for the best performance. Finally, we evaluate MultiPIM's simulation efficiency. On average, running these benchmarks on MultiPIM is about $10^5\times$ slower than on the real host.

Validation. MultiPIM is built upon the already validated simulation platforms: the frontend and backend are implemented based on ZSim [9] and Ramulator [3, 5], respectively; the crossbar switch is modeled with Booksim2 [8]. Thus, we focus on validating the memory network of MultiPIM against a memory-centric network design [15]. We use an identical configuration, including the same number of CPU cores, HMC cubes, memory interconnections, and the same minimal routing algorithm as the prior work [15]. We also replicate their binding of CPU cores to links to achieve the same effect of accessing remote memory. Then, we compare the average and the maximum hop count of dragonfly and mesh memory networks for validation. Our results show that the maximum hop counts in the dragonfly (4 hops) and mesh (7 hops) memory network designs are consistent with the prior work [15], and the average hop counts of dragonfly and mesh memory network designs have less than 6% and 1% differences, respectively.

The results above demonstrate that MultiPIM can help identify new research problems in PIM. Due to the limitation on the length, we evaluate four major aspects of PIM to prove MultiPIM's precision and configurability. Nonetheless, other aspects, such as packet routing algorithm, PIM task scheduling, and PIM-core coherence, can also be evaluated with the open-sourced MultiPIM.

5 Conclusion

In this paper, we introduce MultiPIM, an open-source and highly-configurable PIM simulator that supports multiple memory stacks. MultiPIM provides a flexible interface that supports various PIM architectural designs and can easily adapt existing workloads for PIM offloading. We expect that future works can better identify research problems and validate their designs through a more precise simulation of different PIM architectures using MultiPIM.

References

[1] J. Ahn and et al., "PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *ISCA*. IEEE, 2015.
[2] S. Xu and et al., "PIMSim: A flexible and detailed processing-in-memory simulator," *CAL*, 2018.
[3] G. Singh and et al., "NAPEL: Near-memory computing application performance prediction via ensemble learning," in *DAC*. IEEE, 2019.
[4] A. Boroumand and et al., "CoNDA: Efficient cache coherence support for near-data accelerators," in *ISCA*. IEEE, 2019.
[5] Y. Kim and et al., "Ramulator: A fast and extensible DRAM simulator," *CAL*, 2015.
[6] M. J. Nielsen and et al., "Unified memory computer architecture with dynamic graphics memory allocation," Aug. 15 2000, uS Patent 6,104,417.
[7] H. Consortium, "Hybrid memory cube specification 2.1," *hybridmemorycube.org*, 2014.
[8] N. Jiang and et al., "A detailed and flexible cycle-accurate network-on-chip simulator," in *ISPASS*. IEEE, 2013.
[9] D. Sanchez and et al., "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA*. IEEE, 2013.
[10] J. Ahn and et al., "A scalable processing-in-memory accelerator for parallel graph processing," in *ISCA*, 2015.
[11] S. Beamer and et al., "The GAP benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
[12] USA-Road. [Online]. Available: shorturl.at/oLV01
[13] SK-2005. [Online]. Available: shorturl.at/ruwxL
[14] ego-Twitter. [Online]. Available: shorturl.at/gmtyR
[15] G. Kim and et al., "Memory-centric system interconnect design with hybrid memory cubes," in *PACT*. IEEE, 2013.