Accelerating Graph Computation with Racetrack Memory and Pointer-Assisted Graph Representation

Eunhyuk Park, Sungjoo Yoo, Sunggu Lee and Helen Li*
Embedded System Architecture Lab, POSTECH
University of Pittsburgh*

Abstract

The poor performance of NAND Flash memory, such as long access latency and large granularity access, is the major bottleneck of graph processing. This paper proposes an intelligent storage for graph processing which is based on fast and low cost racetrack memory and a pointer-assisted graph representation. Our experiments show that the proposed intelligent storage based on racetrack memory reduces total processing time of three representative graph computations by 40.2%~86.9% compared to the graph processing, GraphChi, which exploits sequential accesses based on normal NAND Flash memory-based SSD. Faster execution also reduces energy consumption by 39.6%~90.0%. The in-storage processing capability gives additional 10.5%~16.4% performance improvements and 12.0%~14.4% reduction of energy consumption.

1. Introduction

The graph represents the relationship between objects using vertices and edges. It is used in many areas like web mining, social network, chemical compounds, DNA gene analysis, etc. As more data are being represented by graphs in the above areas, graph computation is expected to become more and more important. However, graph computation is challenging in several aspects. Above all, graph computation suffers from long disk access latency due to large data sets and random (and fine-grained) memory accesses (as will be explained in detail in Section 3). In addition, the ratio of computation to data transfer is very low so it is often the case that storage I/O and related functions (e.g., graph data sorting and re-arrangement to better utilize sequential traffic performance in the storage) dominate total execution cycles.

In this paper, we propose an intelligent storage for graph computation which is based on (1) a low-cost, fast and byte-addressable non-volatile memory, namely, racetrack memory [2][9] and (2) an optimization of graph representation exploiting the fast byte-addressability, i.e., a pointer-assisted graph representation. Racetrack memory is a new non-volatile memory and can provide large capacity due to the small size of memory cell. It provides low access latency in both read and write operations and can be accessed at a fine granularity, e.g., 8-byte data. As a result, the racetrack memory can improve the performance of random traffic-dominated graph computation. Other new memory technologies can also be candidates in our proposed intelligent storage. We select racetrack memory in terms of area (PCM provides about $4F^2$ cells) and latency (the latest ReRAM prototype gives 230μs of write latency [3]).

The pointer-assisted graph avoids expensive sorting, rearrangement and search operations which occupy a significant portion of graph computation. In addition, the proposed intelligent storage can perform local operations for simple graph computations, e.g., pagerank. It can improve the performance of graph computation, especially, by eliminating traffics from storage to main memory.

978-3-9815370-2-4/DATE14/©2014 EDAA

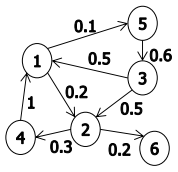
2. Related Work

In this section we review previous work in both categories of active storage and graph computation. Kang et al. [4] and Cho et al. [5] propose in-storage processing which treats SSD as a processing unit using an internal controller or an FPGA accelerator. Both works show that in-storage processing can improve performance and reduce power consumption by exploiting the full bandwidth of the storage device as well as avoiding host traffics.

In GraphChi [6], Kyrola et al. propose a method called parallel sliding window (PSW). It tries to exploit the characteristics that the storage gives higher performance in sequential accesses than in random ones. GraphChi enables a multi-core machine to give comparable performance to large-scale graph processing engines. However, it requires additional steps in graph computation, sorting and data re-arrangement, which renders its performance improvement limited.

3. Problem

In order to examine the current problem in graph computation with the SSD, we first introduce a graph example in Figure 1 (a). For performing a graph computation with the graph, the edges are expressed as a tabular form as shown in Figure 1 (b) (in-edge sorted in this case). Each entry in the table contains two vertex IDs ('in' for destination and 'out' for source) connected to the edge and edge value (edge weight in this example).



(a)

in	out	weight
1	3	0.5
1	4	1
2	3	0.5
2	1	0.2
3	5	0.6
4	2	0.3
5	1	0.1
6	2	0.2

(b)

Figure 1 A graph example and its tabular representation

Pagerank [7] is an algorithm to determine the importance of web page using its connection information. Figure 2 gives a pseudo code of a key function in pagerank. The function iterates multiple times until the vertex weights converge.

```
1 for each vertex_x in graph
2   for each in-edge in vertex_x
3     sum += in-edge→data
4   vertex_x→value = CalculatePageRank(sum, in-edge number)
5   for each out-edge in vertex_x
6     out-edge→data = vertex_x→value
```

Figure 2 Pagerank

In order to explain how random accesses are generated in graph computation, assume the pagerank function is applied to the graph in Figure 1. For instance, when calculating the weight of vertex 2, the pagerank function first reads the edge weight of its in-edges (3^{rd} and 4^{th} entries in Figure 1 (b)), calculates the vertex weight and updates its out-edges with the new vertex weight. In order to do that, the pagerank function accesses the 6^{th} and 8^{th} entries in Figure 1 (b), which incurs random accesses. They are

also fine-grained accesses since only the weights (in a few bytes) of associated edges are accessed.

Random accesses to the graph data in the storage can significantly degrade the performance of graph computation since the state-of-the-art storage adopts long-latency memory, hard disk (10ms of read latency) or NAND Flash memory (more than 50μs for read latency). Thus, each random and fine-grained access can take such a long latency. Note that the benefit of caching is limited in graph computation due to the large size of graph.

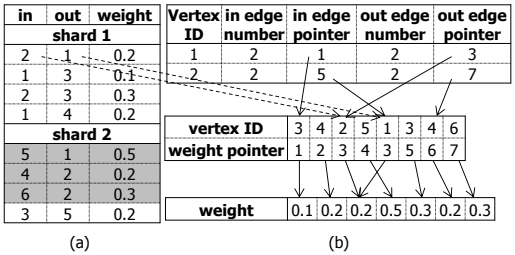


Figure 3 Avoiding random accesses in graph processing

In GraphChi [6], the authors re-organize graph representation in order to avoid random accesses. Figure 3 illustrates the basic concept of GraphChi method utilizing the graph in Figure 1 (a). The graph is partitioned into sub-graphs called shards. Each shard has the same number (four in Figure 3) of in-edges and includes all the vertices of those edges. In a shard, in-edges are first sorted in terms of their source vertex indexes (column 'out' in the figure). When performing pagerank in Figure 3 (a), for instance, the vertices 1 and 2 in shard 1 are first processed. To be specific, the host fetches the in-edge data in the shard. Since all the edge data are required for the vertices in the first shard, the host also fetches their out-edge data which are stored in other shards. The out-edge data of the shard 1 are shaded in Figure 3 (a). The computation for the first shard results in updates in the associated edges. Then, the other shards are processed in the same way.

GraphChi is effective in that random accesses are reduced by utilizing shards. However, it incurs another problem of pre-/post-processing overhead. Figure 3 (b) illustrates the internal structures containing vertex information (upper table), the relationship between vertex ID and weight (center table), and an array of vertex weights (lower table) in GraphChi. They need to be created for every shard processing thereby incurring pre-/post-processing overhead. According to our investigation, it can occupy up to 45.5 % of total runtime in graph computations.

As the above example shows, graph computation incurs lots of random and fine-grained accesses. A recent improvement to avoid random accesses suffers from pre-/post-processing overhead. In this paper, we advocate (1) adopting new high-density and low-latency memory and (2) exploiting the low latency of new memory by utilizing a pointer-based graph representation for the problems.

4. Proposed Storage Structure

Racetrack memory is a spintronics-based non-volatile memory based on domain wall motion (DWM) and giant-magneto resistance (GMR) [9]. Racetrack memory consists of a strip of ferromagnetic material (called racetrack), magnetic tunneling junction (MTJ) and an access transistor. The read and write operations are the same as those of spin-transfer torque RAM (STT-RAM). When both ferromagnetic layers in the MTJ have the same (different) direction(s) of magnetization, the MTJ has low (high) resistance and the resistance level is sensed by applying a small read voltage across the MTJ [2]. For a write operation, the magnetization direction of free layer in the MTJ is changed depending on the write bit data by applying a high current through the MTJ. The direction of write current determines magnetization direction to be stored in the domain. In order to access (read or write) a domain which is not in the MTJ, shift operation(s) is performed by injecting shift current to the racetrack as shown in Figure 4 (a) to move the magnetic domains, which is called domain wall motion [9]. The shift operation typically takes 0.5ns for each domain shift, i.e., one bit shift [2].

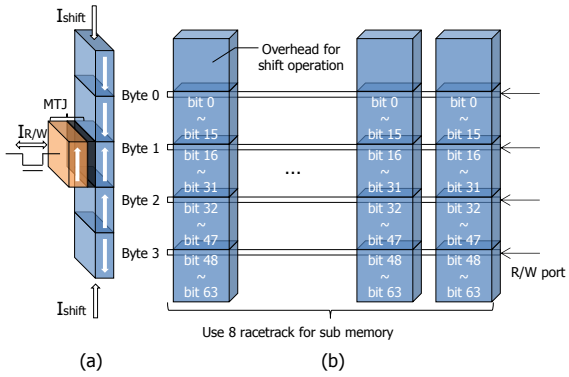


Figure 4 (a) Side view of racetrack and (b) sub-memory

Racetrack memory provides several important benefits. First, it can give large capacity comparable to vertical NAND Flash memory because (1) many bit data (domains) share one access transistor and (2) it can be implemented vertically [9]. Second, it provides fast access at the order of less than tens of nanosecond [2]. In addition, the data in multiple racetracks can be accessed in parallel providing high bandwidth. In Figure 4 (b), eight racetracks each of which has 4 MTJs can be accessed in parallel enabling 32-bit access at a time. In order to access more data, e.g., 64-bit data, the racetracks can be shifted in parallel to access adjacent data. The third benefit is its non-volatility which enables low standby power consumption. One drawback is the latency and power incurred by shift operations.

Figure 5 shows our proposed graph representation. A vertex is represented by a tuple of vertex value (e.g., vertex weight in pagerank), in-edge and out-edge pointers. An in-edge pointer points to an array of in-edge information. Each entry in the array consists of source vertex ID and edge value (e.g., edge weight in pagerank). An out-edge pointer in the vertex data points to an array of out-edge information. Each entry of the array contains destination vertex ID and edge offset (offset in the in-edge array of the destination vertex). Note that each edge has its weight information only at a single location, in its entry of in-edge array (edge value).

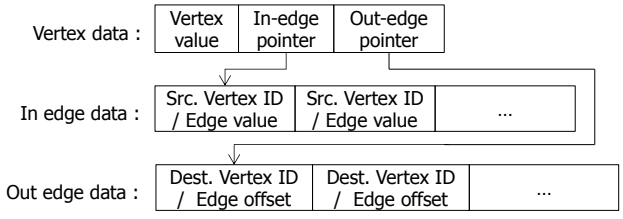


Figure 5 Pointer-assisted graph representation

Note that the benefits of racetrack memory, low latency and fine-grained access enable us to utilize pointers in the graph

representation. As illustrated in Figure 3, the conventional graph computation requires data re-arrangements. Compared with this, our proposed pointer-assisted graph representation enables us to access, with low latency, only the required data from the storage without additional expensive data re-arrangement steps.

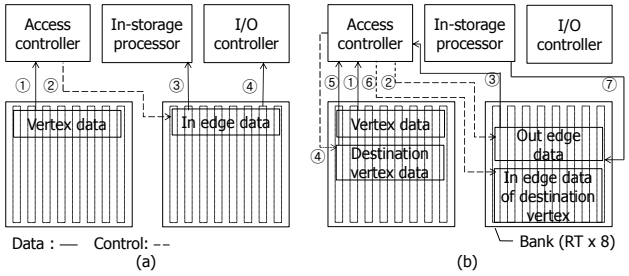


Figure 6 (a) Reading in-edge data (b) Accessing out-edge data

Figure 6 shows the organization of our proposed intelligent storage based on racetrack memory. It consists of racetrack memory and controller sub-system. In the figure, the racetrack memory consists of two banks which can be accessed in parallel. The controller sub-system consists of access controller, in-storage processor, and I/O controller. The access controller receives requests from the host and issues read/write commands to access racetrack memory, data transfer commands to the I/O controller and, if specified in the host request, in-storage computation commands to the in-storage processor. On the command from the access controller, the in-storage processor receives data and performs local computation. The I/O controller transfers data (racetrack memory data or the data in the in-storage processor) between the host and the intelligent storage.

Figure 6 (a) shows how the in-edge information of vertex is accesed. First, the access controller issues a read request to read the data of a vertex (arrow ①). Vertex data are stored in an array and the size of each entry is the same. Thus, the desired vertex is localized with vertex ID. After obtaining the in-edge pointer in the read vertex data, the controller issues a read command to access the data of the desired in-edge (arrow ②). Note that in-edge data can be accessed in a sequential way by placing contiguously on the racetracks or in a parallel way by distributing them on multiple banks. Investigating efficient graph data placement on racetrack memory will be our future work. After reading all the in-edge data, there are two possibilities. In case of simple function, e.g., pagerank, the in-storage processor can perform graph computation (arrow ③). Otherwise, the in-edge data are transferred to the host via the I/O controller (arrow ④).

Figure 6 (b) illustrates how the out-edge information of vertex is accessed. First, the controller reads the vertex data to obtain the out-edge pointer. Then, it accesses the array of out-edge information in order to obtain the destination vertex IDs and the edge offset in the destination vertex (arrows ② and ③ in Figure 6 (b)). The controller reads the in-edge pointer of destination vertex (arrows ④ and ⑤) and accesses the out-edge data using previous read edge offset and in-edge pointer (arrows ⑥ and ⑦).

5. Experiments

We simulate graph computations on two designs: a baseline GraphChi design utilizing a conventional SSD [4][8] and our proposed racetrack memory-based intelligent storage. Both run with a high-performance host which consists of x86 out-of-order core system. Both host and storage are on a PCI-E 3.0 bus (supporting 1GBps) for higher storage I/O bandwidth than the popular SATA 3.0. We use a Pin-based event-driven architecture modeling framework, McSimA+ [10] for our simulations. Table 1 shows the architectural parameters.

Table 1 Parameters, energy and timing of models

host	parameters	device	energy	timing
CPU core	out-of-order x86 core, 3GHz	DRAM avg	154 mW [5]	-
L1 cache	I : 4 way, 32KB, d : 8 way, 32KB	NAND,8kB read	3.31 uJ [5]	A = 50 uS,
L2 cache	16 way, 4MB	NAND,8kB write	64.94 uJ [5]	B =1 ns/byte
main memory	DDR3 4GB, $t_{CL}:t_{RP}:t_{RCD} = 14:14:14\ ns$	RT, shift	0.62 nJ / bit [2]	0.5 ns
bus	PCI-EX 3.0 1x	RT, write	0.57 nJ / byte [2]	10ns
SSD/RT memory	DDR2 256MB, 6.4 GB/s	RT, read	0.074 nJ / byte [2]	0.5ns
host	**energy**	processor	192 pJ / inst. [5]	-
CPU static	idle : 9.25 W, load : 35.7 W [11]	ALU	2.11 pJ / op. [5]	-
CPU load	1.44 nJ / instruction [11]	MUL	67.6 pJ / op. [5]	-
DRAM static	1.03 W [12]	REG	4.23 pJ / access [5]	-
DRAM R/W	Max. 1.34 / 2.34 W [12]			
chipset	5.49 W [5]		-	
I/O	9.61 W [5]			

Our SSD timing model uses a linear model in [8] which expresses N-byte access time as A+N*B with a fixed time, A (due to Flash Translation Layer overhead such as mapping table accesses in the SSD controller) and a data size-dependent time, N*B. Our racetrack memory model is based on [2]. However, the number of R/W ports per racetrack is reduced for area cost reduction and the tag is removed since we model the main memory. The racetrack model uses 256 MB module which has a hierarchical and dense architecture of $1F^2$ cells as in [2]. The area cost of 256 MB module is estimated to be 6.6 mm^2 at 45 nm technology. In timing aspect, the read and write operation consists of many steps, like routing, row decoder, etc. Table 1 shows the latency parameters obtained by modeling those detailed steps.

Both SSD and racetrack memory models have static and dynamic components in energy consumption. The static energy is proportional to total execution cycles. In CPU case, instruction count is used for dynamic power estimation. In host memory case, DRAM energy is decomposed into static (proportional to runtime) and dynamic (proportional to the amount of data traffics) components. On device side, the energy models are based on [5] for the SSD and [2] for the racetrack as shown in Table 1.

Table 2 Graph examples

graph examples [13][14]			
name	**vertex number**	**edge number**	**average number of edges**
amazon0601	400727	3200440	7.98658
dblp-2010	326186	1615400	4.95239
uk-2007-05@100000	100000	3050615	30.5062
algorithm examples [1][6]			
	iterations		
	amazon0601	**dblp-2010**	**uk-2007-05@100000**
Community Detection	10	10	10
PageRank	5	5	5
SSSP	10	7	5

We use three representative graph computation algorithms, community detection (CD), single source shortest path (SSSP) [1] and pagerank [7]. Community detection finds groups of vertices called community in which the level of inter-vertex connections is higher than the average level of the entire graph. SSSP finds the shortest path from a given vertex to all the other vertices in the entire graph. As Table 2 shows, for each of the three graph algorithms, we use three real graphs which have millions of edges. The simulations take 30~60 hours to run tens of billions instructions for these graphs. We did not run simulations with larger graphs due to too long simulation runtime.

Figure 7 compares the runtime of GraphChi (denoted as 'SSD') and our method ('RT'). Our intelligent storage offers by 40.2% to 86.9% runtime improvement. In the CD case, the speed up is lower than others because the processing (by the host) dominates runtime. Thus, even though our intelligent storage

eliminates the overhead of preprocessing and reduces I/O traffics, the gain is limited. Our storage reduces I/O traffics in CD and SSSP, because GraphChi cannot deal with random vertex queries. To avoid random access, GraphChi requires additional disk data I/O and shard processing in this case. However, our racetrack model can provide the required random data with low latency, thereby significantly reducing I/O traffics.

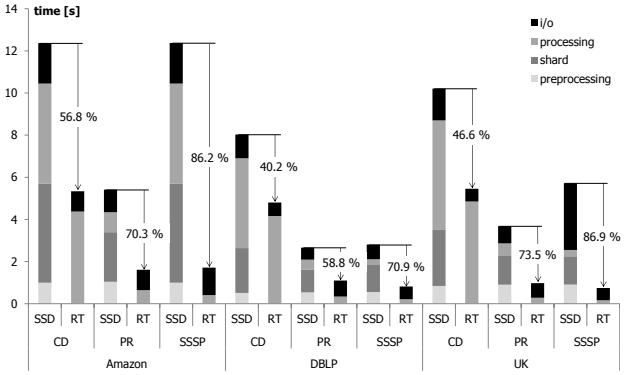


Figure 7 Runtime of graph processing

Figure 8 shows that the gain in total energy consumption is similar (average 39.6%~90.0%) to that of runtime in Figure 7. It is because the major components of energy consumption are the static energy of CPU and system, and these components are proportional to total runtime. However, because CPU has longer idle state in racetrack model, the difference is bigger in Figure 8 than that in Figure 7, except CD which has processing as a dominant factor of runtime. The energy consumption of SSD and racetrack memory is almost the same. In the racetrack memory, shift operations (~10x and ~14x more frequent than reads and writes, respectively) dominate total disk energy consumption because of a lot of random accesses.

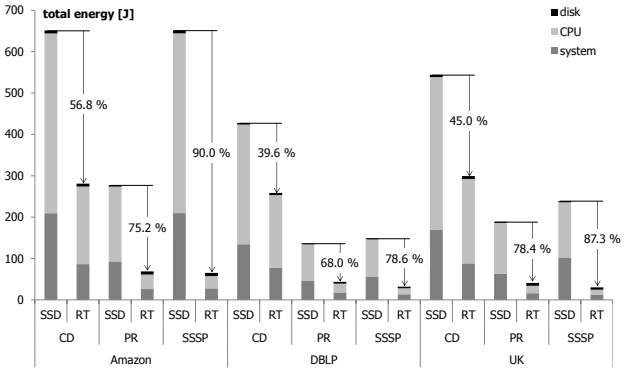


Figure 8 Total energy consumption

Figure 9 shows the effects of in-storage processing in our intelligent storage. In this case, the in-storage processor is implemented as a stream processor and performs pagerank. The intelligent storage gives performance benefits mostly by eliminating data transfer between storage and host. The comparison between the intelligent storage (i.RT in Figure 9) and the original racetrack-based storage (RT) shows that, in the two graphs of Amazon and DBLP, the in-storage processing reduces I/O traffics between host and storage while increasing processing time. Their net effect is reduction in total runtime. The graph, UK gives reductions in both I/O and processing time because our model finds edge data based on vertex. Thus, the high average number of edges in UK graph (about 30.5 in Table 2) reduces the amount of pointer operation per each edge, which enables the device to operate more efficiently compared to the other graphs thereby reducing processing time. The in-storage processing gives on average additional 13.8% and 13.1% improvements in runtime and energy consumption, respectively.

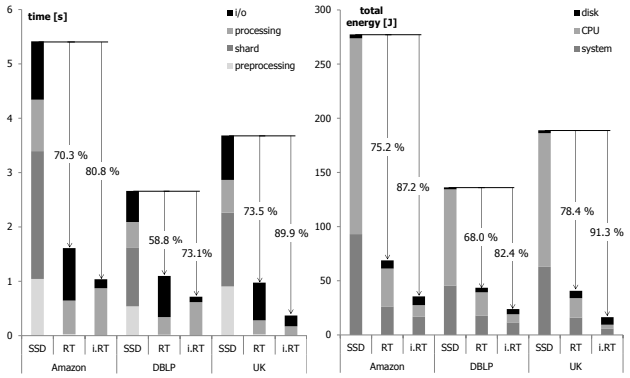


Figure 9 Runtime and energy comparison

6. Conclusion

Graph computation is characterized by random accesses to the storage. In order to overcome the limitations of current graph computation based on NAND Flash memory-based SSD, we propose (1) utilizing a cheap and low latency memory, racetrack memory and (2) exploiting its low latency benefit with a pointer-assisted graph representation. Our experiments show that the proposed storage offers 40.2%~86.9% improvement in graph computation time and similar improvements in energy consumption. The in-storage processing of simple graph computations can also give additional improvements by 13.8% (runtime) and 13.1% (energy) on average.

7. Acknowledgement

This work was supported by the IT R&D program MKE/KEIT (No.10041608, Embedded System Software for New Memory-based Smart Devices) and the Center for Integrated Smart Sensors funded by the Ministry of Science, ICT & Future Planning as Global Frontier Project (NRF-2011-0031863).

8. References

[1] L. d. F. Costa, et al., "Characterization of complex networks: A survey of measurements," Advances in Physics, 56(1):167–242, 2007.
[2] Z. Sun, et al., "Cross-layer racetrack memory design for ultra high density and low power consumption," Proc. DAC, 2013.
[3] T.-Y. Liu, et al., "A 130.7 mm^2 2-layer 32Gb ReRAM memory device in 24nm technology," Proc. ISSCC, 2013.
[4] Y. Kang, et al., "Enabling cost-effective data processing with smart ssd," Proc. Storage Conference, 2013.
[5] S. Cho, et al., "Active disk meets flash: a case for intelligent ssds," Proc. ICS, 2013.
[6] A. Kyrola, et al., "GraphChi: Largescale graph computation on just a pc," Proc. OSDI, 2012.
[7] L. Page, et al., "The pagerank citation ranking: bringing order to the web," Stanford InfoLab, 1999.
[8] K. El Maghraoui, et al., "Modeling and simulating flash based solid-state disks for operating systems," Proc. WOSP/SIPEW, 2010.
[9] L. Thomas, et al., "Racetrack memory: a high-performance, low-cost, non-volatile memory based on magnetic domain walls," Proc. IEDM, 2011.
[10] J. H. Ahn, et al., "McSimA+: A Manycore Simulator with Application-level Simulation and Detailed Microarchitecture Modeling," Proc. ISPASS, 2013.
[11] W. L. Bircher, et al., "Complete system power estimation: A trickle-down approach based on performance events," Proc. ISPASS, 2007.
[12] Micron Technology, Inc., "DDR3 SDRAM System-Power Calculator," http://www.micron.com/products/support/power-calc.html, 2013.
[13] P. Boldi, et al., "Ubicrawler: A scalable fully distributed web crawler," Software: Practice and Experience, 34(8):711–726, 2004.
[14] J. Leskovec, "Stanford large network dataset collection," http://snap.stanford.edu/data/index.html, 2011.