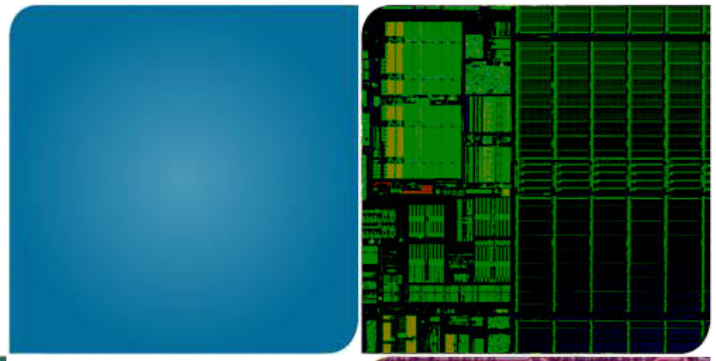
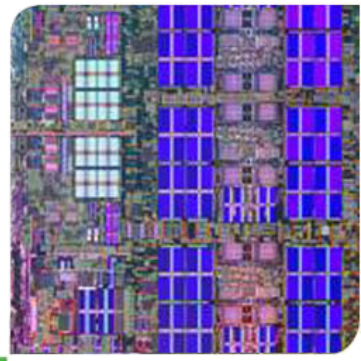
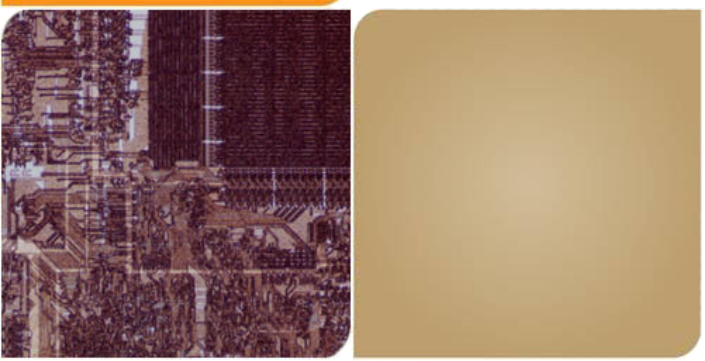



INSTRUCTION OFFLOADING WITH HMC 2.0 STANDARD –– A CASE STUDY FOR GRAPH TRAVERSALS

Lifeng Nai

Hyesoon Kim



Georgia Tech

comparch

OUTLINE

Graph computing is important

- however it doesn't perform well on conventional architectures
- Processing-in-memory can help

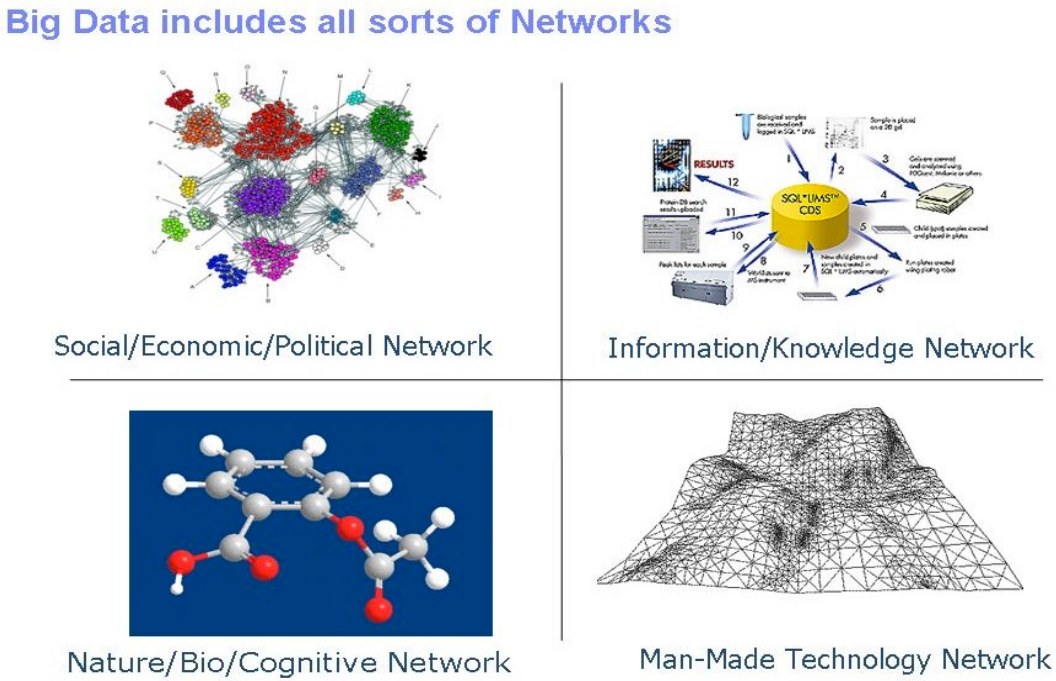
Nice Feature from HMC 2.0 specification

- Atomic instruction → processing-in-memory
- By utilizing HMC atomic instructions, we can improve graph computing applications

GRAPH COMPUTING

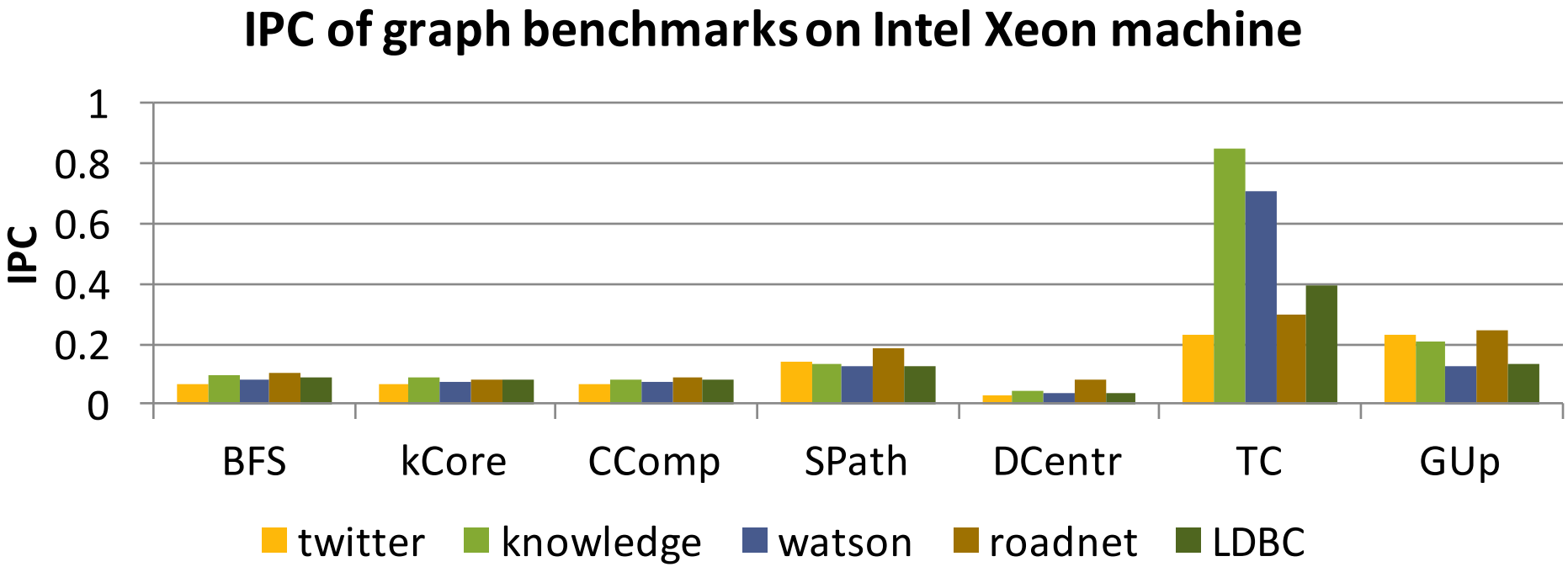
- Big data is Big Linked Data

- Graph computing is important for processing network data.



GRAPH COMPUTING

- Graph computing doesn't perform on conventional architectures.



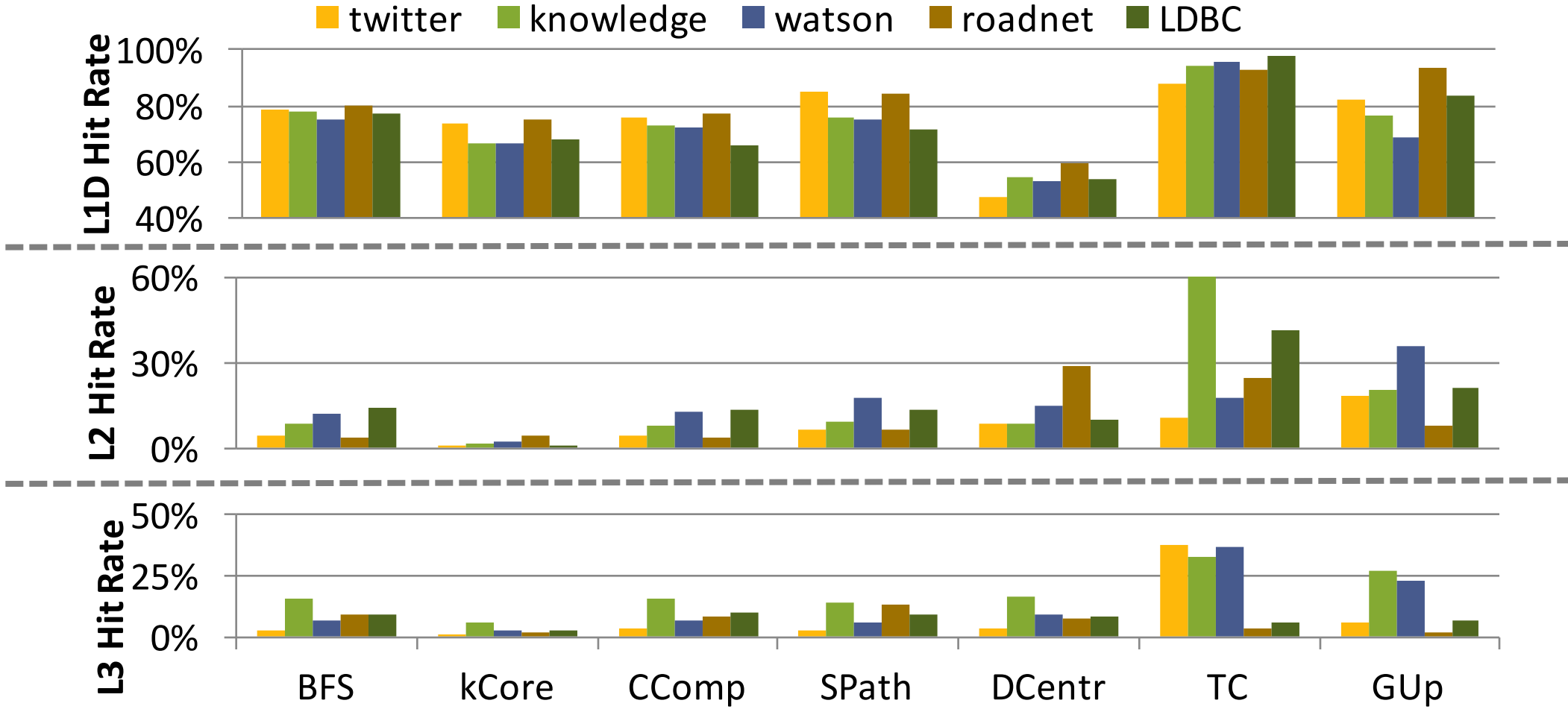
GraphBIG Benchmark Suite (https://github.com/graphbig/graphBIG)

"GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions" (to appear in SC'15)

Georgia Tech comparch

GRAPH COMPUTING

Problem: cache performance [GraphBIG SC'15]



Georgia Tech comparch

GRAPH COMPUTING

- Graph computing doesn't perform on conventional architectures.
- Problem: cache performance
- Offloading -> Processing-in-memory
- HMC 2.0 standard provides that capability

HMC 2.0

Hybrid Memory Cube Specification 2.0

- Released @ 2014 by Hybrid Memory Cube Consortium
- (although hardware is not available yet)

New feature: Atomic Request Commands

- In-memory processing capability
- Read-Update-Write atomic operation
- 16 byte/8 byte granularity
- Integer data only

HMC ATOMIC INSTRUCTION

Type	Data Size	Operation
Arithmetic	Dual 8B	Add immediate
	Single 8B	Add immediate
	8-byte	Increment
Bitwise	16-byte	Swap
	8-byte	Bit write
Boolean	16-byte	AND
	16-byte	NAND
	16-byte	OR
	16-byte	XOR
Comparison	8-byte	CAS/if equal
	16-byte	CAS/if zero
	8/16-byte	CAS/if greater
	8/16-byte	CAS/if less

HMC ATOMIC INSTRUCTION

HMC Atomic Instruction

- Instruction-level offloading

Limitations

- One-memory location
- No floating-point
- No indirect access
- Simple operations only

How to utilize this feature for graph?

GRAPH WORKLOAD

Zoom-in -> Details

Meta data: small size

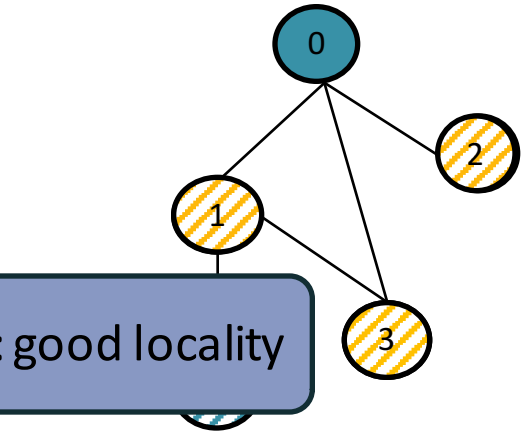
Graph structure: good locality

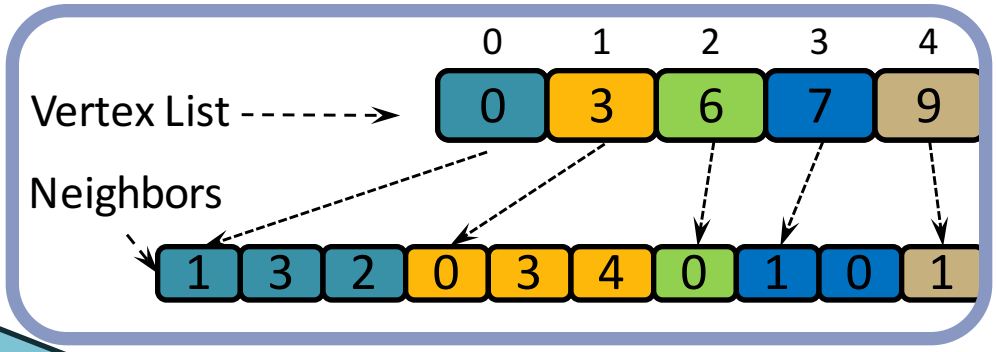
Operation: simple integer check & set

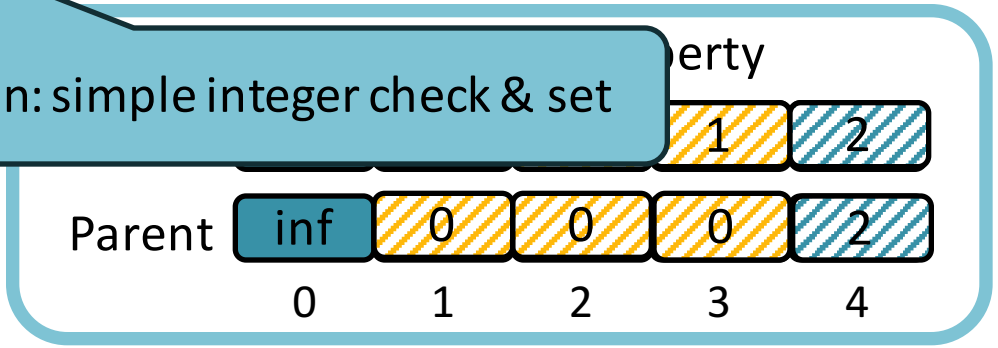
Graph property: large size, no locality

```
Q-in.insert(source);
while(! Q-in.empty()){
    Q-out.clear(); step++;
    for all u in Q-in in parallel
        for each v in u.neighbors {
            if (level[v] == infinity){
                level[v] = step;
                parent[v] = u;
                Q-out.insert(v);
            }
        }
```

BFS Sample Code







Georgia Tech comparch

INSTRUCTION OFFLOADING

Incorporating HMC Atomic Instruction needs only little effort

```
for each v in u.neighbors{
    if (level[v] == infinity){
        level[v] = step;
        parent[v] = u;
        Q-out.insert(v);
    }
}
```

```
if (level[v] == infinity)
    level[v] = step;
if (parent[v] == infinity){
    parent[v] = u;
    Q-out.insert(v);}
```

CAS_equal

CAS_equal &
if success
Q-out.insert

```
bool flag[SZ]={false};
for each v in u.neighbors{
    CAS_equal(&(level[v]), infinity, step);
    flag[i] = CAS_equal(&(parent[v]), infinity, u);
}
for each flag[i] if (flag[i]) Q-out.insert(v);
```

CAS_equal(ptr, T, N) :

```
if (*ptr == T)
    {*ptr = N; return true;}
else return false;
```

DISCUSSIONS

- Bandwidth
- Applicability
- Cache policy

BANDWIDTH DISCUSSION

Bandwidth without using HMC atomic instructions

- $Bandwidth\ (in\ FLITs) \cong 24E_i(1 - H_p)$

Bandwidth when using HMC atomic instructions

- $Bandwidth\ (in\ FLITs) \cong 8E_i$

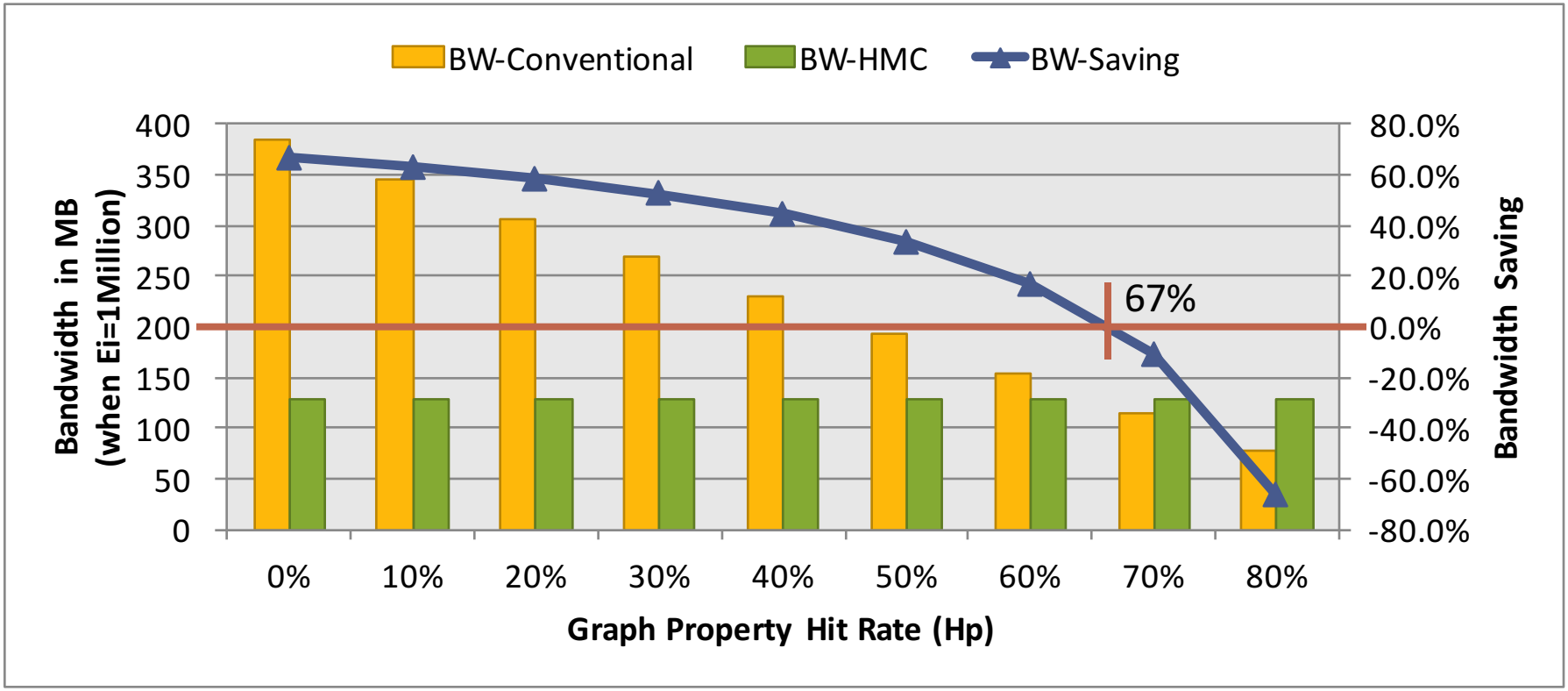
- E_i – # of traversed edges in step i
- H_p – graph property cache hit rate

$BW(metaData) \approx 0 \quad BW(graphStructure) \approx E_i \times 4byte$
No-HMC: $BW(graphProp_{read}) \approx 2E_i \times 64 \times (1 - H_p) \approx BW(graphProp_{writeback})$
No-HMC: $BW(total) \approx 256E_i(1 - H_p) = 4E_i(1 - H_p) \times 6\ FLIT = 24E_i(1 - H_p)\ FLIT$
HMC: $BW(total) \approx 2E_i \times 4FLIT = 8E_i\ FLIT$

BANDWIDTH DISCUSSION

HMC Atomic Instruction (assuming bypass cache)

- can save as much as 66% bandwidth
- lower Hp → higher bandwidth saving (no benefit if Hp > 67%)



Georgia Tech comparch

APPLICABILITY DISCUSSION

Computation Type	Workload	Applicable?
Computation on Graph Structure	Breadth-first search	YES
	Depth-first search	YES
	Degree centrality	YES
	Betweenness centrality	NO, floating-point operations
	Shortest path	YES
	K-core decomposition	YES
	Connected component	YES
Computation on Dynamic Graph	Graph construction	NO, complicated code/behaviors
	Graph update	NO, complicated code/behaviors
	Topology morphing	NO, complicated code/behaviors
Computation on Graph Property	Triangle count	YES
	Gibbs inference	NO, computation intensive & cache friendly

APPLICABILITY DISCUSSION

Computation on graph structure

- Mostly applicable
- Floating Point support can help

Computation on dynamic graph

- Not applicable
- Support of indirect access and multiple mem locations can help

Computation on graph property

- Not applicable
- Computation intensive, cache friendly

CACHE POLICY DISCUSSION

HMC Atomic-Inst Accessed Data

- Data Value is updated outside the host processor → data coherence issue

Bypass Cache or Not

- Regular Rd/Wr to the same address:
- *Bypass cache*: Mark the data as un-cacheable
- *Cache & flush*: Cache for reads, flush when write/atomic-inst happens
- *Cache & coherence*: Maintain host-memory coherence

- HMC Atomic-Inst:
- *Bypass cache*: Directly send request to memory
- *Cache check*: Check cache first, process-in-host when hit

CONCLUSION AND FUTURE WORK

Graph computing has significant performance degradations caused by irregular graph property accesses

HMC 2.0 atomic instructions show promising potentials to improve graph computing performance by offloading property accesses.

Future work

- Detailed performance analysis
- Trade-offs between different design choices of host processor
- Potential extensions on existing HMC atomic instructions

QUESTIONS?

BACKUP SLIDES

RELATED WORK

- Junwhan Ahn (ISCA'15) "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing"
  - Fully-programmable in-memory core

- Junwhan Ahn (ISCA'15) "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture"
  - General PIM instructions, some selected target applications
  - Focus on the architecture design of host processor side

- Our Work
  - Strictly follow HMC 2.0 → a more realistic design
  - Focus more on application side → analyze full-spectrum of graph computing

GRAPH WORKLOAD

Zoom-in -> Details

Meta data

Graph structure

Graph property

```
CUDA_KERNEL (graph_structure,
                graph_property, step){
    unsigned u = getThreadID();
    if (level[u]==step){
        for each v in u.neighbors {
            if (level[v] == infinity){
                level[v] = step+1;
                parent[v] = u;
            }
        }
    }
}
```

GPU BFS Code

GRAPH WORKLOAD

Data access types

- Meta data (small size) → L1 cache
- Graph structure (good locality) → L2/L3 cache
- Graph property (large size, no locality) → Memory

Operations on graph property

- Usually as simple as check-and-set
- Most workloads don't use floating-point