Graphene: Fine-Grained IO Management for Graph Computing

Hang Liu       H. Howie Huang
The George Washington University

March 2, 2017
Graphs are Everywhere ...
Graph Algorithms are Insightful

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Functionality</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle completing</td>
<td>Friend recommendation</td>
<td></td>
</tr>
<tr>
<td>Shortest path</td>
<td>Navigation</td>
<td></td>
</tr>
<tr>
<td>PageRank</td>
<td>Webpage searching</td>
<td></td>
</tr>
<tr>
<td>Matrix factorization</td>
<td>Product recommendation</td>
<td></td>
</tr>
</tbody>
</table>
BIG Graph

Facebook, #active users

Graph size continues to grow ...

*source: statista
Related Work

Programmability

Easy

Hard

Graphene

External-memory system
X-Stream, GridGraph, FlashGraph

Distributed system
PowerGraph

Shared memory system
Ligra, Galois

Price

Low

High

Time

Long

Short
Agenda

❖ Background
❖ Graphene
  ❖ Programming model
  ❖ Bitmap based IO management
  ❖ Row-column balanced partition
  ❖ HugePage
❖ Evaluation
❖ Conclusion
Solid-State Drive

- Affordable
  - SSD ~$0.3/GB vs. DRAM ~$10/GB
- Bandwidth
  - Sequential IO 540 MB/s
  - Random IO ~300 MB/s (4KB IO block)
- Energy efficient
Difficult to Program

```c
fd = open(neighborListFile, O_DIRECT);
IObuffer = posix_memalign(4KB aligned);
IOsetup(IOcontext);

while true
    foreach vertex v
        if isActive(v)
            begin = v.neighborBeginPostion();
            end = v.neighborEndPostion();
            beginAligned = IOalign(begin);
            endAligned = IOalign(end);
            IOsubmit(IOcontext, IObuffer, fd, beginAligned, endAligned);
            IOpoll(IOcontext, timeout);
            if (IOcontext == error){Errorhandler(IOcontext);}
        neighborList = neighborParser(IObuffer);
        Compute (v, neighborList)
        level++;
```

Open file descriptor
Allocate aligned IO buffer
Setup kernel IO context, lack error support
Align the IO offset and size
Submit IO request, lack error support
Poll IO request, lack error support
Define IO error handler
Parse IO buffer to filter out useful neighbors
IO Centric Graph Processing

Philosophy:
❖ Graphene knows which neighbors to load
❖ Pushes data to algorithms for computation

```
IOIterator it = IOIterator (neighborListFile, isActive);
while true
    while it -> hasMore()
        vertex v = it -> current();
        Compute (v, it -> getNeighbor(v));
    level++;
```

APSP (LOC = 90)  
BFS (60)  
PageRank (80)  
k-Core (50)  
WCC (89)  
SpMV (70)  

Request  
Block  
SSD  

...  

Request  
Block  
SSD  

...  

Request  
Block  
SSD  

...  

Request  
Block  
SSD  

...  

Request  
Block  
SSD
Linux Pluglist

Active vertices

IO requests

Pluglist

Sort

Neighbor list file

Small Pluglist size limits adjustment range

Sorting is costly

Load redundant blocks

Random IO
**Novel Bitmap Solution**

**Active vertices**

**Bitmap**

**Neighbor list file**

- Global range IO adjustment
- Hashing for sort
- Sequential IO
- No redundant IO
Conventional 2D Partition

Each partition has the same number of vertices.

Balance #edges should be the goal!
Row-Column Balanced 2D Partition

2.5x speedup over conventional 2D partition on Twitter graph
IO vs Compute

Before IO optimization

After IO optimization
Cache Optimization

❖ Last-level cache (LLC)
  ❖ Page coloring has some benefits

❖ Translation lookaside buffer (TLB)
  ❖ Use HugePage for buffering IO, managing metadata, etc.
Evaluation

- Intel Xeon E5-2620 CPUs
  - 12 cores, supporting 24 physical threads with hyper-threading
  - 2MB, 1GB HugePage
  - 15MB LLC
- 256GB main memory
- 16 SSDs (Samsung 850 EVO SSDs with PCI-HBAs).
- Linux kernel 4.4.0, g++ 4.8.5
# Datasets Specifications

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster</td>
<td>Friendster social</td>
<td>68M</td>
<td>2.6B</td>
<td>20GB</td>
</tr>
<tr>
<td>Twitter</td>
<td>Twitter social</td>
<td>53M</td>
<td>2.0B</td>
<td>15GB</td>
</tr>
<tr>
<td>Clueweb</td>
<td>ClueWeb12</td>
<td>978M</td>
<td>42.6B</td>
<td>336GB</td>
</tr>
<tr>
<td>EU</td>
<td>EU domains</td>
<td>1071M</td>
<td>92B</td>
<td>683GB</td>
</tr>
<tr>
<td>Gsh</td>
<td>Gsh webpage</td>
<td>988M</td>
<td>33.8B</td>
<td>252GB</td>
</tr>
<tr>
<td>UK</td>
<td>UK domains</td>
<td>788M</td>
<td>48B</td>
<td>270GB</td>
</tr>
<tr>
<td>Kron30</td>
<td>Graph500 Generator</td>
<td>1B</td>
<td>32B</td>
<td>256GB</td>
</tr>
<tr>
<td>Kron31</td>
<td>Graph500 Generator</td>
<td>2B</td>
<td>1T</td>
<td>8TB</td>
</tr>
</tbody>
</table>
Graphene vs. State-of-the-art

External memory system: FlashGraph 3.6X, X-Stream 16X, GridGraph 6.9X

Distributed system: PowerGraph 20X.

Shared memory system: Galois 5% and Ligra 16% (no BFS)
Graphene vs. State-of-the-art (Cont’d)

External memory system: FlashGraph 3.6X, X-Stream 16X, GridGraph 6.9X

Distributed system: PowerGraph 20X.

Shared memory system: Galois 5% and Ligra 16% (no BFS)
Disk Bandwidth Utilization

Throughput (MB/s)

Time elapse (second)

Min
Median
Max

Min: The minimum throughput observed over time.
Median: The middle value of the throughput distribution.
Max: The maximum throughput observed over time.
Conclusion

❖ Easy programming: “IO centric graph processing”

❖ IO optimizations:
  ❖ Bitmap, Asynchronous direct IO
  ❖ Fine grained IO

❖ Balancing:
  ❖ Row-column based partition
  ❖ Workload stealing

❖ HugePage optimization

Despite external memory, Graphene performs comparably with in-memory solutions