Graphicionado
A High-Performance and Energy-Efficient Graph Analytics Accelerator

Tae Jun Ham
Lisa Wu
Narayanan Sundaram
Nadathur Satish
Margaret Martonosi

MICRO 2016

Slide: http://tiny.cc/graphicionado
Graph Analytics
– A key workload of modern-day computing

- Application Domains
  - Social Networks, Bioinformatics, Healthcare, Cybersecurity, Simulation, etc.

- Algorithms
  - Path Analysis (e.g., BFS, SSSP)
  - Centrality Analysis (e.g., PageRank, Betweenness Centrality)
  - Recommendation systems (e.g., Collaborative Filtering)
  - Community Analysis
Graph Analytics is **difficult to code**

- Graph Analytics has …
  - Irregular data access patterns
  - Extremely low computation-to-communication ratio
  - Poor spatial/temporal data locality
  - Difficult-to-extract parallelism
  - Memory-bound performance

- **Solution: Software Graph Processing Frameworks**
  - Programmers express graph algorithms using a **programming abstraction** (i.e., vertex program)
  - Frameworks orchestrate data movements for given computation using **efficient backend SW**

<table>
<thead>
<tr>
<th>Industry</th>
<th>Academia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Pregel</td>
<td>Pegasus (ICDM 09)</td>
</tr>
<tr>
<td>facebook GraphMat</td>
<td>TurboGraph (KDD 13)</td>
</tr>
<tr>
<td>Intel</td>
<td>Galois (SOSP 13)</td>
</tr>
<tr>
<td>NVIDIA PGX</td>
<td>GraphX (OSDI 14)</td>
</tr>
<tr>
<td>IBM</td>
<td>Ligra (PPoPP 13)</td>
</tr>
<tr>
<td>Microsoft Graph Engine</td>
<td>Ligra (PPoPP 16)</td>
</tr>
<tr>
<td></td>
<td>X-Stream (SOSP 13)</td>
</tr>
</tbody>
</table>
Software Frameworks have **limitations**

= Insufficient tailoring to HW

- Application-oblivious memory system in conventional processors
  - Caches *blindly* cache all data at a *coarse granularity* including ...
    - Low temporal locality data
    - Low spatial locality data (i.e., only 4B out of 64B is utilized)

  **Our measurement shows that SW frameworks consume 2x-27x more memory bandwidth** than the optimal communication case

- Expensive data movement in conventional processors
  - Graph analytics has extremely low computation-to-communication ratio

  **Most algorithms: < 6% of instructions are used for actual computation; 94% used for data movements; results in huge energy consumption.**
Graphicionado Approach

Graphicionado: A high-performance, energy-efficient graph analytics HW accelerator which overcomes the limitations of software frameworks while retaining the programmability benefit of SW frameworks

- Retains Programmability
  - Specialized-while-flexible HW pipeline

- Overcomes Limitation
  - Application-specific memory system
  - Application-specific pipeline design
Presentation Outline

- Why Graphicionado?
- Vertex Programming Abstraction
- Constructing Graphicionado Pipeline from Abstraction
- Optimizing Graphicionado
  - Optimizing Data Movement
  - Scaling On-Chip Memory Usage
  - Parallelization
- Performance/Energy Evaluation
- Conclusions
Graph Data Structure

- Graph consists of vertices & edges
  - Each Vertex has an ID and a property (or state)
  - Each Edge is defined as a tuple (SRC ID, DST ID, edge property)

- Graph analytics: an iterative computation to calculate the desired vertex property for each vertex
**Vertex Programming Abstraction**

1. For each active Vertex U
2. For each outgoing edge E(U,V)
3. \( \text{Res} = \text{Process\_Edge}(E_{\text{weight}}, U_{\text{prop}}) \)
4. \( V_{\text{temp}} = \text{Reduce}(V_{\text{temp}}, \text{Res}) \)
5. End
6. End

7. For each Vertex V
8. \( V_{\text{prop}} = \text{Apply}(V_{\text{temp}}, V_{\text{prop}}) \)
9. End

- **Vertex programming abstraction** is commonly used in SW frameworks (e.g., Intel GraphMat, Google Pregel, etc.)
  - Programmers express graph analytics algorithm with three custom functions

- **Graphicionado** uses the same abstraction to retain the programmability of SW frameworks
### Vertex Programming Abstraction

<table>
<thead>
<tr>
<th>Processing Phase</th>
<th>Apply Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. For each active Vertex U</td>
<td>7. For each Vertex V</td>
</tr>
<tr>
<td>2. For each outgoing edge E(U,V)</td>
<td>8. $V_{\text{prop}} = \min(V_{\text{temp}}, V_{\text{prop}})$ — Apply</td>
</tr>
<tr>
<td>3. $\text{Res} = E_{\text{weight}} + U_{\text{prop}}$ — Process_Edge</td>
<td></td>
</tr>
<tr>
<td>4. $V_{\text{temp}} = \min(V_{\text{temp}}, \text{Res})$ — Reduce</td>
<td></td>
</tr>
<tr>
<td>5. End</td>
<td>9. End</td>
</tr>
</tbody>
</table>

$V_{\text{prop}}$: minimum distance to V on last iteration

$V_{\text{temp}}$: minimum distance to V on this iteration

---

**Single Source Shortest Path (SSSP) Example**

**Process_Edge:** $E_{\text{weight}} + U_{\text{prop}}$

**Reduce:** $\min(V_{\text{temp}}, \text{Res})$

**Apply:** $\min(V_{\text{temp}}, V_{\text{prop}})$
**Vertex Programming Abstraction**

1. **For each active Vertex U**
2. **For each outgoing edge E(U,V)**
3. \[ \text{Res} = E_{\text{weight}} + U_{\text{prop}} \] —— Calculate a distance to V through this edge
4. \[ V_{\text{temp}} = \min(V_{\text{temp}}, \text{Res}) \] —— Update the current minimum distance to V
5. End
6. End

---

7. **For each Vertex V**
8. \[ V_{\text{prop}} = \min(V_{\text{temp}}, V_{\text{prop}}) \] —— Compare the current minimum distance with the value from the last iteration; If it changes its value, add V to the active vertex set for the next iteration
9. End

---

\( V_{\text{prop}} \): minimum distance to V on last iteration
\( V_{\text{temp}} \): minimum distance to V on this iteration

---

**Single Source Shortest Path (SSSP) Example**

**Process_Edge:** \( E_{\text{weight}} + U_{\text{prop}} \)  
**Reduce:** \( \min(V_{\text{temp}}, \text{Res}) \)  
**Apply:** \( \min(V_{\text{temp}}, V_{\text{prop}}) \)
From abstraction to HW

```c
for (i=0; i<ActiveVertex.size(); i++) {
    vid = ActiveVertexID[i];
    vprop = ActiveVertexProp[i];
    eptr = PtrToEdgeList[vid];
    for (e = Edges[eptr]; e.src == vid; e = Edges[++eptr]) {
        res = Process_Edge(e.weight, vprop);
        temp = TempVertexProp[e.dst];
        temp = Reduce(temp, res);
        TempVertexProp[e.dst] = temp;
    }
}
// Apply Phase updates ActiveVertexProp with TempVertexProp
```

Processing Phase Block Diagram

1. Read Active SRC Property
2. Read Edge Pointer
3. Read Edges for given SRC
4. Process Edge
5. Reduce
6. Write Temp. DST Property
7. Read Temp. DST Property
8. Reduce
9. Write Temp. DST Property
From abstraction to HW

```cpp
1 for (i=0; i<ActiveVertex.size(); i++) {
2   vid = ActiveVertexID[i];
3   vprop = ActiveVertexProp[i];
4   eptr = PtrToEdgeList[vid];
5   for (e = Edges[eptr]; e.src == vid; e = Edges[++eptr]) {
6     res = Process_Edge(e.weight, vprop);
7     temp = TempVertexProp[e.dst];
8     temp = Reduce(temp, res);
9     TempVertexProp[e.dst] = temp;
10    }
11 } // Apply Phase updates ActiveVertexProp with TempVertexProp
```

Processing Phase Block Diagram

- **Line 2-3**: Read Active SRC Property
- **Line 4**: Read Edge Pointer
- **Line 5**: Read Edges for given SRC
- **Line 6**: Process Edge
- **Line 7**: Read Temp. DST Property
- **Line 8**: Reduce
- **Line 9**: Write Temp. DST Property

**Memory Access Units**
- **Sequential Memory Access Unit**
- **Non-Sequential Memory Access Unit**
- **Custom Computation Unit**
From abstraction to HW

```cpp
for (i=0; i<ActiveVertex.size(); i++) {
    vid = ActiveVertexID[i];
    vprop = ActiveVertexProp[i];
    eptr = PtrToEdgeList[vid];
    for (e = Edges[eptr]; e.src == vid; e = Edges[++eptr]) {
        res = Process_Edge(e.weight, vprop);
        temp = TempVertexProp[e.dst];
        temp = Reduce(temp, res);
        TempVertexProp[e.dst] = temp;
    }
}
// Apply Phase updates ActiveVertexProp with TempVertexProp
```

**Processing Phase Pipeline**

1. **Line 2-3**: Read Active SRC Property
2. **Line 4**: Read Edge Pointer
3. **Line 5**: Read Edges for given SRC
4. **Line 6**: Process Edge
5. **Line 7**: Read Temp. DST Property
6. **Line 8**: Reduce
7. **Line 9**: Write Temp. DST Property

---

**Sequential Memory Access Unit**

**Non-Sequential Memory Access Unit**

**Custom Computation Unit**

**Atomic Update Control Unit**

- **Read-Modify-Write Update needs to be atomic for the same destination vertex (e.dst)**
Optimizing Data Movement

Processing Phase Pipeline

1. Non-sequential access; low-spatial-locality
   - Only use 4-16B out of 64B off-chip memory access granularity; leads to off-chip memory BW waste

2. High-temporal locality
   - Data will be used again for edges with the same destination vertex

Use fine-grained on-chip SPM for data used for these modules
1. Non-sequential access; low-spatial-locality
   - Only use 4-16B out of 64B off-chip memory access granularity; leads to off-chip memory BW waste

2. High-temporal locality
   - Data will be used again for edges with the same destination vertex

Use fine-grained on-chip SPM for data used in these modules
Optimizing Data Movement

1. Sequentially accessed; high-spatial locality
   - Has predictable access pattern
   - All 64B loaded from the off-chip memory will be used

2. Low-temporal locality
   - Each active vertex or an edge is processed only once. This data will not be accessed again for the current iteration.

Use prefetcher; Do not use on-chip SPM for data used for these modules
Scaling On-chip Memory Usage

- Graphicionado utilizes an on-chip storage to avoid wasting off-chip BW
  - Often requires 4-16B on-chip storage per vertex

  Not enough on-chip storage for 10M+ vertices

- Solution: Partition a graph before the execution; Then, process each subgraph at a time
  - Assign edges to different slices based on their destination vertices
Scaling On-chip Memory Usage

- Graphicionado utilizes an on-chip storage to avoid wasting off-chip BW
  - Often requires 4-16B on-chip storage per vertex

  → Not enough on-chip storage for 10M+ vertices

- Solution: Partition a graph before the execution; Then, process each subgraph at a time
  - Assign edges to different slices based on their destination vertices

*Paper has additional scaling techniques*
- **Naive Parallelization** Approach
  - Replicate Pipeline
  - Each pipeline stream processes a portion of the active (SRC) vertices (e.g., a modulo of the SRCID determines which stream for processing)

  Different streams will try to read/write the same memory address (TempVertexProp[]) at the same time on S6 & S8

  Design complexity (SPM port count) and performance degradation

Two modules trying to read the same address (TempVertexProp[3])
- **Graphicionado Approach**
  - Each pipeline stream is separated to two pieces:
    - SRC access / DST access portion
  - N x N switch routes data to the correct DST stream (e.g., a modulo of the DSTID)

Now each unit accesses an exclusive portion of the scratchpad memory.

> Reduction in design complexity and improvement in throughput
Presentation Outline

- Why Graphicionado?
- Vertex Programming Abstraction
- Constructing Graphicionado Pipeline from Abstraction
- Optimizing Graphicionado
  - Optimizing Data Movement
  - Scaling On-Chip Memory Usage
  - Parallelization
- Performance/Energy Evaluation
- Conclusions
- Graphicionado achieves ~3x speedup over state-of-the-art software framework (Intel GraphMat) across different workloads
  - Software framework was run on a 32-core Haswell Xeon server
  - Both system were provisioned the same theoretical peak memory BW (78GB/s)
Graphicionado Energy Consumption

- Graphicionado consumes < 2% of the energy (50x-100x) compared to the software processing framework
  - Most of the energy (70%+) was spent for the eDRAM static energy
  - 20-25x power saving & 2-5x speedup
The parallelization and optimization of Graphicionado achieves up to 27x speedup over the baseline single-stream pipeline.

- Without effective parallelizations and optimizations, simply using HW does not necessarily bring any benefit.
Conclusions

- Software graph processing frameworks has limitations
  - Inefficient on-chip memory (cache) usage
  - Expensive data movement

- Graphicionado: Specialized HW accelerator for graph analytics
  - Efficient use of on-chip memory
  - Specialized pipeline tailored for graph analytics data movement
  - Effective parallelism

- ~3x speedup and 50x+ energy saving over state-of-the-art software framework
Graphicionado
A High-Performance and Energy-Efficient
Graph Analytics Accelerator

Tae Jun Ham
Lisa Wu
Narayanan Sundaram
Nadathur Satish
Margaret Martonosi

MICRO 2016