High-Performance Graph Primitives on the GPU: Design and Implementation of Gunrock

Yangzihao Wang

University of California, Davis

yzhwang@ucdavis.edu

March 24, 2014
Problem Overview

There is a significant demand for efficient parallel framework for processing and analytics on large-scale graphs.

Applications in fields such as: Social Media, Science and Simulation, Advertising, and Web.
Graph Processing on the CPU

Higher Level of Performance

Galois
Ligra
PowerGraph
Pregel
PBGL

Higher Level of Programmability

Yangzihao Wang (yzhwang@ucdavis.edu)
Gunrock CUDA Library
March 24, 2014
Graph Processing on the GPU

Initial results show that GPU computing is promising for future processing of large-scale graphs. Most current research on GPU graph processing focuses on the challenge of irregular memory accesses and work distribution. However, two gaps in graph computation on the GPU are overlooked.

▶ Algorithmic and Implementation Gap: few algorithms have efficient single-node GPU implementations when compared to the state of the art on CPUs, and multi-node GPU implementations are even rarer.

▶ Programmability Gap: the implementations of these algorithms are complex and require expert programmers. The resulting code couples graph computation to parallel graph traversal, and is difficult to maintain and extend.
Graph Processing on the GPU

Initial results show that GPU computing is promising for future processing of large-scale graphs. Most current research on GPU graph processing focuses on the challenge of irregular memory accesses and work distribution. However, two gaps in graph computation on the GPU are overlooked.

▶ **Algorithmic and Implementation Gap:** few algorithms have efficient single-node GPU implementations when compared to the state of the art on CPUs, and multi-node GPU implementations are even rarer.
Initial results show that GPU computing is promising for future processing of large-scale graphs. Most current research on GPU graph processing focuses on the challenge of irregular memory accesses and work distribution. However, two gaps in graph computation on the GPU are overlooked.

- **Algorithmic and Implementation Gap:** few algorithms have efficient single-node GPU implementations when compared to the state of the art on CPUs, and multi-node GPU implementations are even rarer.

- **Programmability Gap:** the implementations of these algorithms are complex and require expert programmers. The resulting code couples graph computation to parallel graph traversal, and is difficult to maintain and extend.
Graph Processing on the GPU

The main goal of Gunrock’s programming model is to enable end-users to express, develop, and refine iterative graph algorithms with a high-level, programmable, and high-performance abstraction.
The main goal of Gunrock’s programming model is to enable end-users to express, develop, and refine iterative graph algorithms with a high-level, programmable, and high-performance abstraction.

- **Programmability**: To be expressive enough to represent a wide variety of graph computations.
Graph Processing on the GPU

The main goal of Gunrock’s programming model is to enable end-users to express, develop, and refine iterative graph algorithms with a high-level, programmable, and high-performance abstraction.

- **Programmability**: To be expressive enough to represent a wide variety of graph computations.
- **Performance**: To leverage the highest-performing GPU computing primitives.
Common Characteristics of Graph Problems

- Large-Scale: Facebook has over 1 billion nodes, 144 billion friendships and Twitter has 500 million nodes and over 15 billion follower edges.

- Heterogenous Node Degree Distribution: Several real-world graphs have such scale-free structure whose degree distribution follows a power law, at least asymptotically, which brings challenge on load-balancing.

- Iterative Convergent Process: Start with a working queue contains a subset of the graph, iteratively do the computation, will finally converge.
Common Characteristics of Graph Problems

- **Large-Scale:** Facebook has over 1 billion nodes, 144 billion friendships and Twitter has 500 million nodes and over 15 billion follower edges.
Common Characteristics of Graph Problems

- **Large-Scale**: Facebook has over 1 billion nodes, 144 billion friendships and Twitter has 500 million nodes and over 15 billion follower edges.

- **Heterogenous Node Degree Distribution**: Several real-world graphs have such scale-free structure whose degree distribution follows a power law, at least asymptotically, which brings challenge on load-balancing.
Common Characteristics of Graph Problems

- **Large-Scale**: Facebook has over 1 billion nodes, 144 billion friendships and Twitter has 500 million nodes and over 15 billion follower edges.

- **Heterogenous Node Degree Distribution**: Several real-world graphs have such scale-free structure whose degree distribution follows a power law, at least asymptotically, which brings challenge on load-balancing.

- **Iterative Convergent Process**: Start with a working queue contains a subset of the graph, iteratively do the computation, will finally converge.
Design Choices

▶ Bulk Synchronous Parallel (BSP) Model:
Large amount of independent computations can be parallelized when traverse the graph. Programming simplicity and scalability advantages.

▶ Queue-based Iterative Method:
Workload efficient. Small cost of keeping a compact working queue with GPU primitives: scan + stream compact. Could easily expand to support priority scheduling.

▶ Hybrid Push-vs-Pull style of graph traversal:
Start graph traversing step from either the current frontier (push) or the unvisited set (pull) to achieve the highest performance.

▶ Idempotent Operation:
Whether to permit a vertex to appear multiple times in frontier(s) from one or more iterations.

Yangzihao Wang (yzhwang@ucdavis.edu)
Design Choices

- **Bulk Synchronous Parallel (BSP) Model:** Large amount of independent computations can be parallelized when traverse the graph. Programming simplicity and scalability advantages.

- **Queue-based Iterative Method:** Workload efficient. Small cost of keeping a compact working queue with GPU primitives: scan + stream compact. Could easily expand to support priority scheduling.

- **Hybrid Push-vs-Pull style of graph traversal:** Start graph traversing step from either the current frontier (push) or the unvisited set (pull) to achieve the highest performance.

- **Idempotent Operation:** Whether to permit a vertex to appear multiple times in frontier(s) from one or more iterations.
Design Choices

- **Bulk Synchronous Parallel (BSP) Model:** Large amount of independent computations can be parallelized when traverse the graph. Programming simplicity and scalability advantages.

- **Queue-based Iterative Method:** Workload efficient. Small cost of keeping a compact working queue with GPU primitives: scan + stream compact. Could easily expand to support priority scheduling.
Design Choices

- **Bulk Synchronous Parallel (BSP) Model:** Large amount of independent computations can be parallelized when traverse the graph. Programming simplicity and scalability advantages.

- **Queue-based Iterative Method:** Workload efficient. Small cost of keeping a compact working queue with GPU primitives: scan + stream compact. Could easily expand to support priority scheduling.

- **Hybrid Push-vs-Pull style of graph traversal:** Start graph traversing step from either the current frontier (push) or the unvisited set (pull) to achieve the highest performance.
Design Choices

- **Bulk Synchronous Parallel (BSP) Model:** Large amount of independent computations can be parallelized when traverse the graph. Programming simplicity and scalability advantages.

- **Queue-based Iterative Method:** Workload efficient. Small cost of keeping a compact working queue with GPU primitives: scan + stream compact. Could easily expand to support priority scheduling.

- **Hybrid Push-vs-Pull style of graph traversal:** Start graph traversing step from either the current frontier (push) or the unvisited set (pull) to achieve the highest performance.

- **Idempotent Operation:** Whether to permit a vertex to appear multiple times in frontier(s) from one or more iterations.
Graph Data Representation

Gunrock uses compressed sparse row (CSR) format for vertex-centric operations and edge list for edge-centric operations.
Workload Mapping Strategies (What makes Gunrock traversal fast?)

- **The Problem:** Write various-length neighbor lists of vertices into an output queue.
- **Per-thread+Per-warp and Per-CTA:** Develop specialized workload mapping strategies according to neighbor list’s size.

![Diagram of warp and block structures](image)

one warp (t4 is the controlling thread)

one block (256 threads, with t42 is the controlling thread)
Workload Mapping Strategies (What makes Gunrock traversal fast?)

- **Partitioned Load-balancing**: Organize groups of edges into equal-length chunks and assigning each chunk to a block.

![Diagram of partitioned load-balancing](image-url)
Workload Mapping Strategies (What makes Gunrock traversal fast?)

![Graph showing performance comparison between Duane’s and Load-balanced strategies across different datasets.](image_url)

- **webbase**
  - Duane’s: 30.019
  - Load-balanced: 50.585
- **coAuthor**
  - Duane’s: 220.196
  - Load-balanced: 237.864
- **kron-500**
  - Duane’s: 525.376
  - Load-balanced: 949.445
- **soc-livj**
  - Duane’s: 769.877
  - Load-balanced: 1047.738

- **Yangzihao Wang** (yzhwang@ucdavis.edu)
- **Gunrock CUDA Library**
- **March 24, 2014**
Workload Mapping Strategies (What makes Gunrock traversal fast?)

Gunrock Performance Chart: BFS
A High-Level View of Programming Model

Bulk iterative processes whichiterate between graph traversal and computation using operators and functors.

- Enactor (CUDA Kernel Entry)
- Functor (Problem-spec Computation)

Graph Traverse Operator:
- Workload mapping strategies
- Idempotent operation switch
- Push-vs-Pull based operation switch
Graph Primitive Example: BFS

A breadth-first search (BFS) algorithm takes a graph and a source vertex $s$, and computes a breadth-first search tree rooted at $s$ containing all vertices reachable from $s$. Our implementation will compute the predecessor and the distance from source vertex for each vertex.
Graph Primitive Example: BFS

A breadth-first search (BFS) algorithm takes a graph and a source vertex $s$, and computes a breadth-first search tree rooted at $s$ containing all vertices reachable from $s$. Our implementation will compute the predecessor and the distance from source vertex for each vertex.
Graph Primitive Example: BFS

A breadth-first search (BFS) algorithm takes a graph and a source vertex $s$, and computes a breadth-first search tree rooted at $s$ containing all vertices reachable from $s$. Our implementation will compute the predecessor and the distance from source vertex for each vertex.
Graph Primitive Example: BFS

A breadth-first search (BFS) algorithm takes a graph and a source vertex $s$, and computes a breadth-first search tree rooted at $s$ containing all vertices reachable from $s$. Our implementation will compute the predecessor and the distance from source vertex for each vertex.
Graph Primitive Example: BFS

A breadth-first search (BFS) algorithm takes a graph and a source vertex $s$, and computes a breadth-first search tree rooted at $s$ containing all vertices reachable from $s$. Our implementation will compute the predecessor and the distance from source vertex for each vertex.

![Graph Example](image)
### Performance: Speedup

<table>
<thead>
<tr>
<th>Primitive</th>
<th>webbase1M</th>
<th>coAuthor</th>
<th>socLiveJournal</th>
<th>Kron_g500</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>NA(GPU:0.001s)</td>
<td>4</td>
<td>27</td>
<td>22</td>
</tr>
<tr>
<td>CC</td>
<td>3</td>
<td>10</td>
<td>13</td>
<td>NA(GPU:0.253s)</td>
</tr>
<tr>
<td>BC</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SSSP</td>
<td>2</td>
<td>15</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>PR</td>
<td>13</td>
<td>NA(GPU:0.1s)</td>
<td>NA(GPU:3.4s)</td>
<td>NA(GPU:29.6s)</td>
</tr>
</tbody>
</table>
What’s Next?

- More flexible operator set;
- More graph types and data structures;
- External memory and multi-node GPUs support;
- Non-regular Graph Operations/Advanced graph primitives.
Acknowledgement

- This work is done with Yuechao Pan and my advisor Prof. John Owens.
- Thanks to Andrew Davidson for his help with SSSP implementation.
- Thanks to Erich Elsen and Vishal Vaidyanathan from Royal Caliber.
- Thanks to Duane Merrill and Sean Baxter from NVIDIA Research.
- Thanks to Dr. Chris White and DARPA XDATA funding.

Thank you!