Gunrock: A Fast and Programmable Multi-GPU Graph Processing Library

November 19, 2015, GPU Technology Theater @ SC 15
Yuechao Pan with Yangzihao Wang, Yuduo Wu, Carl Yang, Leyuan Wang, Andy Riffel and John D. Owens
University of California, Davis
ychpan@ucdavis.edu
## Why use GPUs for Graph Processing?

### Graphs
- Found everywhere
  - Road & social networks, web, etc.
- Require fast processing
  - Memory bandwidth, computing power and GOOD software
- Becoming very large
  - Billions of edges
- Irregular data access pattern and control flow
  - Limits performance and scalability

### GPUs
- Found everywhere
  - Data center, desktops, mobiles, etc.
- Very powerful
  - High memory bandwidth (288 GBps) and computing power (4.3 Tflops)
- Limited memory size
  - 12 GB per NVIDIA K40
- Hard to program
  - Harder to optimize

### Scalability

### Performance

### Programmability
What we want to achieve with Gunrock?

Performance

- High performance GPU computing primitives
- High performance framework
- Optimizations
- Multi-GPU capability

Programmability

- A data-centric abstraction designed specifically for the GPU
- Simple and flexible interface to allow user-defined operations
- Framework and optimization details hidden from users, but automatically applied when suitable
Idea: Data-Centric Abstraction & Bulk-Synchronous Programming

**Data-centric abstraction**
- Operations are defined on a group of vertices or edges ≡ a frontier
  => Operations = manipulations of frontiers

**Bulk-synchronous programming**
- Operations are done one by one, in order
  - Within a single operation, computing on multiple elements can be done in parallel, without order
Gunrock’s Operations on Frontiers

Generation

**Advance**: visit neighbor lists

**Filter**: select and reorganize

Computation

**Compute**: per-element computation, in parallel can be combined with advance or filter
Example: BFS with Gunrock

Advance + Compute (+1, AtomicCAS)

0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13
Example: BFS with Gunrock

- **Advance + Compute** (+1, AtomicCAS)
  - 3 4 2
- **Filter**
  - 3 4 2
Example: BFS with Gunrock

1
Advance + Compute (+1, AtomicCAS)
3  4  2
Filter

3  4  2
Advance + Compute (+1, AtomicCAS)

1  2  5  6  7  8  9  10  1  8  1  3  5  8
Example: BFS with Gunrock

P: uneven neighbor list lengths (v4 vs. v3)
P: Concurrent discovery conflict (v5,8)

Advance + Compute

3 4 2

Filter

3 4 2

Advance + Compute (+1, AtomicCAS)

1 2 5 6 7 8 9 10 1 8 1 3 5 8
Example: BFS with Gunrock

P: uneven neighbor list lengths (v4 vs. v3)
P: Concurrent discovery conflict (v5,8)

1
Advance + Compute
3 4 2
Filter
3 4 2
Advance + Compute (+1, AtomicCAS)

Filter
6 7 9 10 8 5

Gunrock @ GPU Technology Theater, Nov. 19, 2015
Example: BFS with Gunrock

P: uneven neighbor list lengths (v4 vs. v3)
P: Concurrent discovery conflict (v5,8)
P: From many to very few (v5,6,7,8,9,10 -> v11, 12)

Advance + Compute
3  4  2
Filter
3  4  2
Advance + Compute (+1, AtomicCAS)

1  2  5  6  7  8  9  10  1  8  1  3  5  8
Filter
6  7  9  10  8  5
Advance + Compute, Filter
11  12
Optimizations: Workload mapping and load-balancing

P: uneven neighbor list lengths
S: trade-off between extra processing and load balancing
First appeared in various BFS implementations, now available for all advance operations

Load-Balanced Partitioning [3]

Block cooperative Advance of large neighbor lists;
Warp cooperative Advance of medium neighbor lists;
Pre-thread Advance of small neighbor lists.
Per-thread fine-grained, Per-warp and per-CTA coarse-grained [4]
Optimizations: Idempotence

P: Concurrent discovery conflict (v5,8)
S: Idempotent operations (frontier reorganization)
- Allow multiple concurrent discoveries on the same output element
- Avoid atomic operations
First appeared in BFS [4], now available to other primitives
Optimizations: Pull vs. push traversal

P: From many to very few (v5,6,7,8,9,10 -> v11, 12)
S: Pull vs. push operations (frontier generation)
- Automatic selection of advance direction based on ratio of undiscovered vertices
First appeared in DO-BFS [5], now available to other primitives
Optimizations: Priority queue

P: A lot of redundant work in SSSP-like primitives
S: Priority queue (frontier reorganization)
- Expand high-priority vertices first
First appeared in SSSP[3], now available to other primitives
Idea: Multiple GPUs

P: Single GPU is not big and fast enough
S: use multiple GPUs
-> larger combined memory space and computing power

P: Multi-GPU program is very difficult to develop and optimize
S: Make algorithm-independent parts into a multi-GPU framework
-> Hide implementation details, and save user's valuable time

P: Single GPU primitives can’t run on multi-GPU
S: Partition the graph, renumber the vertices in individual sub-graphs and do data exchange between super steps
-> Primitives can run on multi-GPUs as it is on single GPU
Multi-GPU Framework (for programmers)

Recap: Gunrock on single GPU

Input frontier

Associative data (label, parent, etc.)

Iterate till convergence

Output frontier

Single GPU
Multi-GPU Framework (for programmers)

Dream: just duplicate the single GPU implementation
Reality: it won’t work, but good try!

GPU 0

Input frontier

Iterate till convergence

Output frontier

GPU 1

Input frontier

Iterate till convergence

Output frontier

Associative data (label, parent, etc.)
Multi-GPU Framework (for programmers)

Now it works

Partition

Iterate till all GPUs convergence

GPU 0

GPU 1

Associative data (label, parent, etc.)

Local input frontier

Remote input frontier

Remote input frontier

Local input frontier

Remote output frontier

Remote output frontier

Local output frontier

Remote output frontier

Local output frontier
Multi-GPU Framework (for programmers)

Legend:
- Parameters required from user
- User provided operations
- Single GPU data flow
- Multi GPU data flow

Package data
Data package
Push to peer

Input graph
Partitioner
Partition table
Sub-graph builder
Sub-graphs

Received data package
Unpackage
Remote input frontier
Local input frontier
Sub-queue kernels
Sub-queue kernels
Output sub-frontier
Output sub-frontier

Merged frontier
Merged frontier
Full-queue kernels
Full-queue kernels
Output frontier
Local output frontier

Separate
Converged?
Finish

GPU0
GPU1

Converged?
Separate

Gunrock @ GPU Technology Theater, Nov. 19, 2015
Multi-GPU Framework (for end users)

gunrock_executable input_graph --device=0,1,2,3 other_parameters
Graph partitioning

- Distribute the vertices
- Host edges on their sources’ host GPU
- Duplicate remote adjacent vertices locally
- Renumber vertices on each GPU

-> Primitives no need to know peer GPUs
-> Local and remote vertices are separated
-> Partitioning algorithm not fixed

P: Still looking for good partitioning algorithm /scheme
Graph partitioning

|V| = 13
|E| = 44

GPU 0
|V| = 11
|E| = 23

GPU 1
|V| = 12
|E| = 21

Original vertices
Local vertices
Remote vertices (with local replicas)
Local V-id
Remote V-id

Gunrock @ GPU Technology Theater, Nov. 19, 2015
Optimizations: Multi-GPU Support & Memory Allocation

P: Serialized GPU operation dispatch and execution

**S: Multi CPU threads and multiple GPU streams**

≥1 CPU threads with multiple GPU streams to control each individual GPUs

-> overlap computation and transmission

-> avoid false dependency

P: Memory requirement only known after advance / filter

**S: Just-enough memory allocation**

check space requirement before every possible overflow

-> minimize memory usage

-> can be turned off for performance, if requirements are known (e.g. from previous runs on similar graphs)
Results: Single GPU Gunrock vs. Others

* 17x (avg.) vs. BGL [6], a single thread CPU graph library;
* 2.4x (avg.) vs. Ligra [8], a multi-thread CPU graph library;
* beats Cusha [7] with bitcoin dataset;
* comparable with hardwired GPU implementations,
  some speed-up from applying optimizations across primitives;
* 10x (avg.) vs. MapGraph [9], especially for CC

Gunrock @ GPU Technology Theater, Nov. 19, 2015
## Results: Multi-GPU Gunrock vs. Others (BFS)

<table>
<thead>
<tr>
<th>rmat_n20_128</th>
<th>Ref.</th>
<th>Ref. hardware</th>
<th>Ref. performance</th>
<th>Our hardware</th>
<th>Our performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merrill et al. [4]</td>
<td>4x Tesla C2050</td>
<td>8.3 GTEPS</td>
<td>4x Tesla K40</td>
<td>11.2 GTEPS</td>
<td></td>
</tr>
<tr>
<td>rmat_n20_16</td>
<td>Zhong et al. [10]</td>
<td>4x Tesla C2050</td>
<td>15.4 ms</td>
<td>4x Tesla K40</td>
<td>9.29 ms</td>
</tr>
<tr>
<td>peak performance</td>
<td>Fu et al. [9]</td>
<td>16x Tesla K20</td>
<td>15 GTEPS</td>
<td>6x Tesla K40</td>
<td>22.3 GTEPS</td>
</tr>
<tr>
<td>peak performance</td>
<td>Fu et al. [11]</td>
<td>16x Tesla K20</td>
<td>29.1 GTEPS</td>
<td>6x Tesla K40</td>
<td>22.3 GTEPS</td>
</tr>
</tbody>
</table>

* ~ 35% faster than Merrill et al.’s results. Their results on > 3-year-old hardware are impressive, though only customized to BFS.

* > 50% faster than Medusa (Zhong et al.), another programmable graph framework.

* 6 GPU peak performance comparable to MapGraph (Fu et al.) using 16 GPU cluster
Results: Multi-GPU Scaling

* Traversed edges per sec (TEPS) for BFS
* Strong scaling on rmat_n22_48
* Weak scaling on R-MAT graphs (scale 48, each GPU hosting ~180M edges)
Things that we can improve on

* Partitioning
* Inter-iteration overhead
* Long tail / small frontier issue
Current Status

It has over 10 graph primitives
* traversal-based, node-ranking, global (CC, MST)
* LOC ≤ 10 to use a primitive
* LOC ≤ 300 to program a new primitive
* Good balance between performance and programmability

Multi-GPU framework under major revision
* use circular-queue for better scheduling and smaller overhead
* extendable onto multi-node usage

More graph primitives are coming
* graph coloring, maximum independent set, community detection, subgraph matching

Open source, available at
http://gunrock.github.io/
Future Work

* Multi-node support with NVLink
* Performance analysis and optimization
* Graph BLAS
* Asynchronized graph algorithms
* Fixed partitioning / 2D partitioning
* Global, neighborhood, and sampling operations
* More graph primitives
* Dynamic graphs
* Kernel fusion
* …
Acknowledgment

The Gunrock team

Onu Technology and Royal Caliber team
   Erich Elsen, Vishal Vaidyananthan, Oded Green and others
   For their discussion on library development and dataset generating code

All code contributors to the Gunrock library

NVIDIA
   For hardware support, GPU cluster access, and all other supports and discussions

The Gunrock project is funded by
* DARPA XDATA program under AFRL Contract FA8750-13-C-0002
* NSF awards CCF-1017399 and OCI-1032859
* DARPA STTR award D14PC00023
References


Questions?

Q: How can I find Gunrock?  
A: http://gunrock.github.io/

Q: Is it free and open?  
A: Absolutely (under Apache License v2.0)

Q: Papers, slides, etc.?  
A: https://github.com/gunrock/gunrock#publications

Q: Requirements?  
A: CUDA ≥ 5.5, GPU compute capability ≥ 3.0, Linux || Mac OS

Q: Language?  
A: C/C++, with a simple wrapper connects to Python

Q: … (continue)
Example python interface - breadth-first search

from ctypes import *

### load gunrock shared library - libgunrock
gunrock = cdll.LoadLibrary('..//..//build/lib/libgunrock.so')

### read in input CSR arrays from files
row_list = [int(x.strip()) for x in open('toy_graph/row.txt')]
col_list = [int(x.strip()) for x in open('toy_graph/col.txt')]

### convert CSR graph inputs for gunrock input
row = pointer((c_int * len(row_list))(*row_list))
col = pointer((c_int * len(col_list))(*col_list))
nodes = len(row_list) - 1
edges = len(col_list)

### output array
labels = pointer((c_int * nodes)())

### call gunrock function on device
gunrock.bfs(labels, nodes, edges, row, col, 0)

### sample results
print ' bfs labels (depth):',
for idx in range(nodes):
    print labels[0][idx],

Gunrock @ GPU Technology Theater, Nov. 19, 2015 | 34