Fast parallel connected components algorithms on GPUs

Guojing Cong

IBM TJ Watson Research Center
1101 Kitchawan Road, Yorktown Heights, NY, 10598

HeteroPar
August 25, 2014
Porto, Portugal

Introduction

- GPUs have become alternative platforms to traditional CPUs for algorithms with substantial data parallelism
- CC is representative of graph problems with fast theoretic parallel algorithms that oftentimes perform poorly on cache-based machines due to irregular memory accesses
- Prior studies show that several parallel graph algorithms perform better on GPUs than on CPUs
- Are GPUs better for graph algorithms?

Outline

- Algorithms, platforms, and inputs
- Straightforward implementations
- Locality optimizations
  - Generic optimizations
  - Algorithm specific optimization
    - Optimization for CC
    - A meta-algorithm for graft-and-shortcut
- Asynchronous algorithm – Rem's
- Rankings
- Conclusion and future work

Algorithm

Algorithm 1: CC(El, D)

Data: El: edge list of size m, D of size n

Result: $D[i]$ is the component for vertex i

```
1: for 1 ≤ i ≤ m parallel do {graft}
2:    if D[El[i].u] < D[El[i].v] then
3:       D[D[El[i].v]] ← D[El[i].u]
4:    end if
5: end for
6: for 1 ≤ i ≤ n parallel do {shortcut}
7:    while D[i] ≠ D[D[i]] do
8:       D[i] ← D[D[i]]
9:    end while
10: end for
```

Based on the Classic Shiloach-Vishkin Algorithm (SV) ; runs in $O(\log^2 n)$ time with $O(n+m)$ processors.

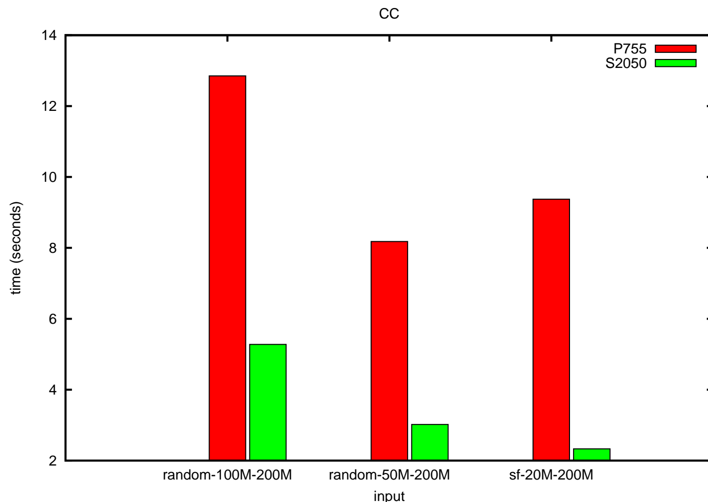
Platforms

- NVIDIA Tesla S2050
  - Four Fermi GPUs running at 1.15GHz (use one)
  - Each GPU has 14 SMs, 448 cores, and 2GB global memory
  - Fermi has 64 KB configurable shared memory and L1 cache
- IBM P755
  - 4 Power7 chips
  - Each chip has 8 cores running at 3.61 GHz, and each core with SMT4
  - 32KB L1, 256KB L2, and 4MB on-chip L3 caches

Inputs and instance sizes

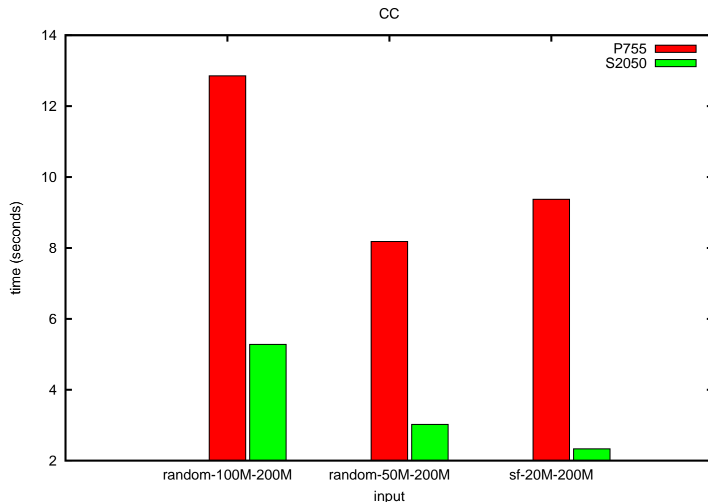
- Random graph by Erdős-Rényi model
- Scale-free graph by RMAT model with a=0.45, b=0.15, c=0.15, d=0.25
- Real-world instances
- All inputs fit in the GPU memory (sparse graph of about 50 - 200M vertices)

Performance comparison



Memory subsystem does not deliver data fast enough on either system

Performance comparison



Memory subsystem does not deliver data fast enough on either system

Taming random accesses

Graph algorithms are memory access intensive; for sparse graphs the accesses are random

- On SMPs, NUMA systems, and clusters:
  - Layout the graph to match access pattern
  - Software prefetching
  - An unified approach for improving temporal, spatial, and geographical locality for cache performance, remote transfers, and messaging overhead
- On GPUs?

Taming random accesses

Graph algorithms are memory access intensive; for sparse graphs the accesses are random

- On SMPs, NUMA systems, and clusters:
  - Layout the graph to match access pattern
  - Software prefetching
  - An unified approach for improving temporal, spatial, and geographical locality for cache performance, remote transfers, and messaging overhead
- On GPUs?

Taming random accesses

Graph algorithms are memory access intensive; for sparse graphs the accesses are random

- On SMPs, NUMA systems, and clusters:
  - Layout the graph to match access pattern
  - Software prefetching
  - An unified approach for improving temporal, spatial, and geographical locality for cache performance, remote transfers, and messaging overhead
- On GPUs?

Taming random accesses

Graph algorithms are memory access intensive; for sparse graphs the accesses are random

- On SMPs, NUMA systems, and clusters:
  - Layout the graph to match access pattern
  - Software prefetching
  - An unified approach for improving temporal, spatial, and geographical locality for cache performance, remote transfers, and messaging overhead
- On GPUs?

Taming random accesses

Graph algorithms are memory access intensive; for sparse graphs the accesses are random

- On SMPs, NUMA systems, and clusters:
  - Layout the graph to match access pattern
  - Software prefetching
  - An unified approach for improving temporal, spatial, and geographical locality for cache performance, remote transfers, and messaging overhead
- On GPUs?

Locality optimization on GPUs

- Coalescing – merging multiple accesses to memory locations (within a short range) into one transaction – is critical to performance
- A sort-based PRAM simulation approach by Chiang *et al.*
- Each edge $(u, v) \in El$ is augmented as an edge $(u, v, u', v') \in El'$
- El' is first sorted with u as key, and all $El[i].u' = D[El[i].u]$ $(1 \leq i \leq m)$ are retrieved; then El' is sorted again with v as key, and all $El[i].v' = D[El[i].v]$ $(1 \leq i \leq m)$ are retrieved
- After two rounds of sorting, $u' = D[u]$ and $v' = D[v]$ for each edge $(u, v, u', v') \in El'$.

Locality optimization on GPUs

- Coalescing – merging multiple accesses to memory locations (within a short range) into one transaction – is critical to performance
- A sort-based PRAM simulation approach by Chiang *et al.*
- Each edge $(u, v) \in El$ is augmented as an edge $(u, v, u', v') \in El'$
- El' is first sorted with u as key, and all $El[i].u' = D[El[i].u]$ $(1 \leq i \leq m)$ are retrieved; then El' is sorted again with v as key, and all $El[i].v' = D[El[i].v]$ $(1 \leq i \leq m)$ are retrieved
- After two rounds of sorting, $u' = D[u]$ and $v' = D[v]$ for each edge $(u, v, u', v') \in El'$.

Locality optimization on GPUs

- Coalescing – merging multiple accesses to memory locations (within a short range) into one transaction – is critical to performance
- A sort-based PRAM simulation approach by Chiang *et al.*
- Each edge $(u, v) \in El$ is augmented as an edge $(u, v, u', v') \in El'$
- El' is first sorted with u as key, and all $El[i].u' = D[El[i].u]$ $(1 \leq i \leq m)$ are retrieved; then El' is sorted again with v as key, and all $El[i].v' = D[El[i].v]$ $(1 \leq i \leq m)$ are retrieved
- After two rounds of sorting, $u' = D[u]$ and $v' = D[v]$ for each edge $(u, v, u', v') \in El'$.

Locality optimization on GPUs

- Coalescing – merging multiple accesses to memory locations (within a short range) into one transaction – is critical to performance
- A sort-based PRAM simulation approach by Chiang *et al.*
- Each edge $(u, v) \in El$ is augmented as an edge $(u, v, u', v') \in El'$
- El' is first sorted with u as key, and all $El[i].u' = D[El[i].u]$ $(1 \leq i \leq m)$ are retrieved; then El' is sorted again with v as key, and all $El[i].v' = D[El[i].v]$ $(1 \leq i \leq m)$ are retrieved
- After two rounds of sorting, $u' = D[u]$ and $v' = D[v]$ for each edge $(u, v, u', v') \in El'$.

Locality optimization on GPUs

- Coalescing – merging multiple accesses to memory locations (within a short range) into one transaction – is critical to performance
- A sort-based PRAM simulation approach by Chiang *et al.*
- Each edge $(u, v) \in El$ is augmented as an edge $(u, v, u', v') \in El'$
- El' is first sorted with u as key, and all $El[i].u' = D[El[i].u]$ $(1 \leq i \leq m)$ are retrieved; then El' is sorted again with v as key, and all $El[i].v' = D[El[i].v]$ $(1 \leq i \leq m)$ are retrieved
- After two rounds of sorting, $u' = D[u]$ and $v' = D[v]$ for each edge $(u, v, u', v') \in El'$.

Analysis

Let T_{orig} be the time of CC (in terms of memory access), T_{sort} be the time of simulating CC using the generic sorting approach, w be the coalescing width

Theorem

For a graph of n vertices and $m = O(n)$ edges, it is necessary that $w > 2^{8/41 \log n}$ for $T_{sort} < T_{orig}$.

For a graph with $n = 200M$ vertices and $m = 400M$ edges, in order that $T_{sort} < T_{orig}$, $w > 73.01$ is necessary. On most GPUs $w \leq 32$.

Analysis

Let T_{orig} be the time of CC (in terms of memory access), T_{sort} be the time of simulating CC using the generic sorting approach, w be the coalescing width

Theorem

For a graph of n vertices and $m = O(n)$ edges, it is necessary that $w > 2^{8/41 \log n}$ for $T_{sort} < T_{orig}$.

For a graph with $n = 200M$ vertices and $m = 400M$ edges, in order that $T_{sort} < T_{orig}$, $w > 73.01$ is necessary. On most GPUs $w \leq 32$.

Analysis

Let T_{orig} be the time of CC (in terms of memory access), T_{sort} be the time of simulating CC using the generic sorting approach, w be the coalescing width

Theorem

For a graph of n vertices and $m = O(n)$ edges, it is necessary that $w > 2^{8/41 \log n}$ for $T_{sort} < T_{orig}$.

For a graph with $n = 200M$ vertices and $m = 400M$ edges, in order that $T_{sort} < T_{orig}$, $w > 73.01$ is necessary. On most GPUs $w \leq 32$.

Software prefetching

Software prefetching improves the memory performance of parallel graph algorithms on CPUs.
In our experiments we did not observe any performance improvement on S2050 with software prefetching.
Inter-thread prefetching shows modest performance improvement (e.g., 16%) on simulators.
In practice, as thread and thread block scheduling is not deterministic, no studies on actual hardware as we know have shown any significant performance improvement for graph algorithms.

Algorithm-specific optimizations:Optimizing graft-and-shortcut

Observation: While accesses to $D[u]$s and $D[v]$s are random, the D values evolve in a pattern.

First, $D[i]$ is non-increasing for each vertex i ($1 \leq i \leq n$) from one iteration to the next. In fact, for most vertices their D values steadily decrease.

Second, the number of unique D values (hence the number of unique super-vertices) also decreases

using u and v as indices, we introduce an *update* step after *shortcut* that replaces each edge (u, v) with $(D[u], D[v])$

Algorithm-specific optimizations:Optimizing graft-and-shortcut

Observation: While accesses to $D[u]$s and $D[v]$s are random, the D values evolve in a pattern.

First, $D[i]$ is non-increasing for each vertex i ($1 \leq i \leq n$) from one iteration to the next. In fact, for most vertices their D values steadily decrease.

Second, the number of unique D values (hence the number of unique super-vertices) also decreases

using u and v as indices, we introduce an *update* step after *shortcut* that replaces each edge (u, v) with $(D[u], D[v])$

Algorithm-specific optimizations:Optimizing graft-and-shortcut

Observation: While accesses to $D[u]$s and $D[v]$s are random, the D values evolve in a pattern.
First, $D[i]$ is non-increasing for each vertex i ($1 \leq i \leq n$) from one iteration to the next. In fact, for most vertices their D values steadily decrease.
Second, the number of unique D values (hence the number of unique super-vertices) also decreases
using u and v as indices, we introduce an *update* step after *shortcut* that replaces each edge (u, v) with $(D[u], D[v])$

Algorithm-specific optimizations:Optimizing graft-and-shortcut

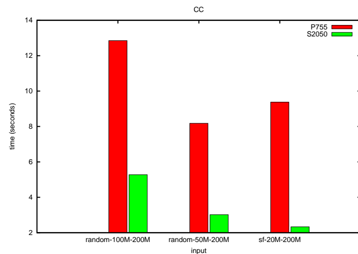
Observation: While accesses to $D[u]$s and $D[v]$s are random, the D values evolve in a pattern.
First, $D[i]$ is non-increasing for each vertex i $(1 \leq i \leq n)$ from one iteration to the next. In fact, for most vertices their D values steadily decrease.
Second, the number of unique D values (hence the number of unique super-vertices) also decreases
using u and v as indices, we introduce an *update* step after *shortcut* that replaces each edge (u, v) with $(D[u], D[v])$

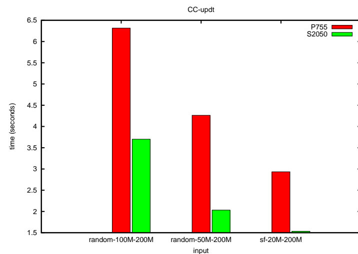
CC-update

Algorithm 2: CC-updt(El, D), $|El| = m$, $|D| = n$

```
1:  for 1 ≤ i ≤ m parallel do {graft}
2:      if El[i].u < El[i].v then
3:          D[El[i].v] ← El[i].u
4:      end if
5:  end for
6:  for 1 ≤ i ≤ n parallel do {shortcut}
7:      while D[i] ≠ D[D[i]] do
8:          D[i] ← D[D[i]]
9:      end while
10: end for
11: for 1 ≤ i ≤ m parallel do {update}
12:     El[i].u ← D[El[i].u], El[i].v ← D[El[i].v]
13: end for
```

Performance (in comparison)





A meta-algorithm

Motivated by evolutionary random-graph theory:

Theorem

Under the Erdös-Rényi model there is a unique giant component of order $f(c)n$ in the graph when $m \sim cn$ with $c > 1/2$.

$f(c) = 1 - \frac{1}{2c}\sum_{k=1}^{\infty}\frac{k^{k-1}}{k!}(2ce^{-2c})^k$ approches 1 as c increases.

A meta algorithm

Stages first permutes the edges in EI, and then divides them into groups, $EI_1, EI_2, \cdots, EI_g$, with $|EI_i| > n/2$ $(1 \leq i \leq g-1)$ except possibly for EI_g.
Next for each group EI_i $(1 \leq i \leq g)$ Stages invokes a connected components algorithm, say, CC, with EI_i and D, and updates the endpoints of each edge in EI_{i+1} with the current components they belong to.
When Stages terminates, $D[i]$ $(1 \leq i \leq n)$ is the connected component for vertex i.

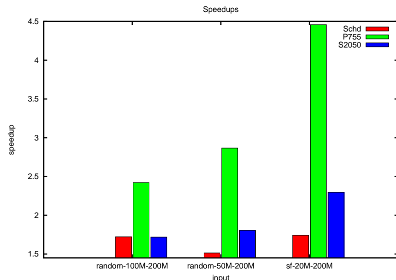
Stages

Algorithm 3: Stages(El, D), $|El| = m$, $|D| = n$, $1/2 < q < m/n$

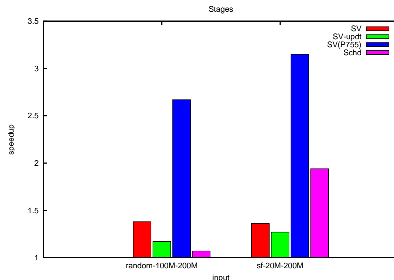
1: randomly permute El
2: divide El into groups $El_1, El_2, \cdots, El_g$, with $|El_i| = qn$, $1 \leq i < g$, $1/2 < q \leq m/n$, and $|El_g| = m - (g-1)qn$
3: **for** $1 \leq i \leq g$ **do**
4: Call CC (El_i,D)
5: **if** $i < g$ **then** {update}
6: **for** $1 \leq j \leq |El_{i+1}|$ parallel **do**
7: $El_{i+1}[j].u \leftarrow D[El_{i+1}[j].u]$, $El_{i+1}[j].v \leftarrow D[El_{i+1}[j].v]$
8: **end for**
9: **end if**
10: **end for**

Analysis and performance

$$T_{cc} - T_{stgs} > \left(2m - 2qn - \frac{m}{qn}(1 - f^2(q)\right) \log n$$



CC-updt speedups



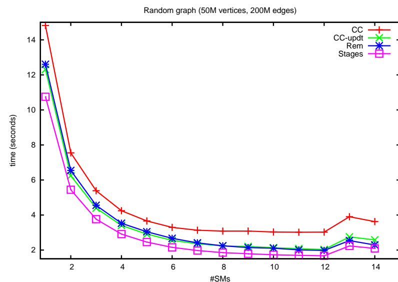
SV speedups

An asynchronous algorithm: Rem's

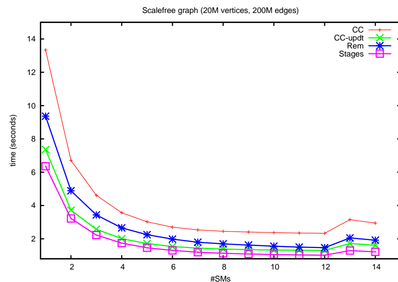
Algorithm 4: Rem(El, $|D|$), $|D| = n$, $|El| = m$

```
1:  for 1 ≤ i ≤ m do
2:      u ← El[i].u, v ← El[i].v
3:      while D[u] ≠ D[v] do
4:          if D[u] < D[v] then
5:              if u = D[u] then
6:                  D[u] ← D[v]; break
7:              end if
8:              z ← D[u]; D[u] ← D[v]; u ← z
9:          else
10:             if v = D[v] then
11:                 D[v] ← D[u]; break
12:             end if
13:             z ← D[v]; D[v] ← D[u]; v ← z
14:         end if
15:     end while
16: end for
```

Ranking the algorithms

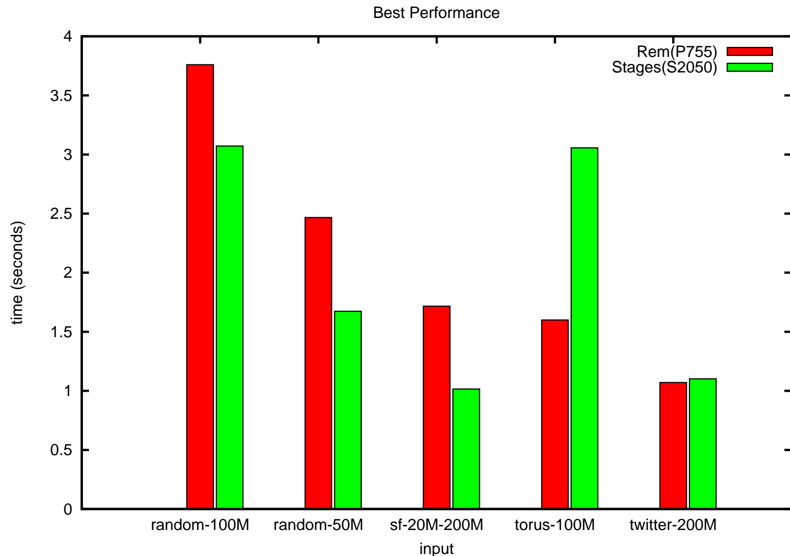


four implementations on a random graph



four implementations on a scale-free graph

GPU vs. CPU



Performance comparison on P755 and S2050

Conclusion and future work

- Straightforward implementation of PRAM algorithms performs relatively better on GPUs than on CPUs. However, the memory subsystem of GPUs does not deliver data fast enough keep all SMs busy
- Generic techniques to improve locality for better performance are too costly on GPU
- low-cost coalescing optimization for CC. We further present a meta algorithm that improves coalescing for several connected components algorithms
- Rem's algorithm performs best on CPUs but not very well on GPUs
- On average the fastest algorithms on the target CPU and GPU have roughly the same performance
- Algorithms for Multi-GPUs, a cluster of GPUs, and emerging architectures