Fast Parallel Connected Components Algorithms on GPUs

Guojing Cong[1] and Paul Muzio[2]

[1] IBM TJ Watson research center, Yorktown Heights, NY 10598, USA
gcong@us.ibm.com
[2] CUNY High Performance Computing Center, Staten Island, New York, 10324
p.muzio@csi.cuny.edu

Abstract. We study parallel connected components algorithms on GPUs in comparison with CPUs. Although straightforward implementation of PRAM algorithms performs relatively better on GPUs than on CPUs, the GPU memory subsystem performance is poor due to non-coalesced random accesses.

We argue that generic sort-based access coalescing is too costly on GPUs. We propose a new coalescing technique and a new meta algorithm to improve locality and performance. Our optimization achieves up to 2.7 times speedup over the straightforward implementation. Interestingly, our optimization also works well on CPUs.

Comparing the best-performing algorithms on GPUs and CPUs, we find our new algorithm is the fastest on GPUs and the second fastest on CPUs, while the parallel Rem's algorithm is the fastest on CPUs but does not perform well on GPUs due to path divergence.

Keywords: Multi-core, GPU, CPU, Connected Components.

1 Introduction

GPUs have become alternative platforms to traditional CPUs for algorithms with substantial data parallelism. We study connected components (CC) algorithms with large, sparse inputs on GPUs in comparison with CPUs. 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 (e.g., see [12,10,15]). Our experiments with CC confirm the performance advantage of GPUs for the straightforward implementation of PRAM algorithms. The memory subsystem performance is nonetheless still poor due to non-coalesced random accesses. We argue that generic sort based coalescing is ineffective for improving performance on GPUs unless the hardware coalescing width is increased.

Several architectural features (see Section 2 for details) make it challenging for locality optimization on GPUs. The amount of GPU device memory is limited, and the overhead to improve locality is large while the performance gain

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 153–164, 2014.
© Springer International Publishing Switzerland 2014

is modest. Large asymptotic overhead in most cache-friendly and I/O-efficient algorithms makes it unlikely for them to perform well on GPUs.

We propose a low-cost technique to improve coalescing for CC on GPUs. We aslo present a new meta algorithm motivated by evolution random graph theory that further improves locality for graft-and-shortcut based algorithms. Our implementation achieves up to 2.7 times speedup over the straightforward implementation. Interestingly, the same optimizations apply on our target CPU, and our implementation beats the best prior cache-friendly implementation.

We rank the performance of the optimized algorithms on GPUs and CPUs. Our new algorithm is the fastest on the target GPU and the second fastest on the target CPU, while parallel Rem's algorithm is the fastest on the CPU but does not perform very well on the GPU. On average the best algorithms on the two platforms have comparable performance over a range of input graphs.

Our study focuses primarily on random graphs and scale-free graphs [11]. Both have small diameters. They are the most challenging inputs in terms of memory performance. The input graph is represented as $G = (V, E)$, with $|V| = n$ and $|E| = m = O(n)$. We create a random graph with n vertices and m edges by randomly adding m unique edges to the vertex set. Scale-free graphs are generated using the R-MAT model [6] with a=0.45, b=0.15, c=0.15, d=0.25. In ranking the best implementations on the two platforms, we also include a large-diameter graph and a real-life twitter graph. We defer the introduction of these graphs to Section 6.

The rest of the paper is organized as follows. Section 2 introduces the target platforms and the baseline algorithm. Section 3 evaluates prior techniques for improving locality on GPUs. Section 4 presents an optimization for the graft-and-shortcut approach, and section 5 introduces the meta algorithm that further improves coalescing for CC. Section 6 ranks the best implementations on the two platforms. In section 7 we give our conclusion and future work.

2 The Platforms and the Base-Line Algorithm

GPUs and CPUs are markedly different. CPU cores typically run at higher frequency than GPU cores, and exploit instruction level parallelism through out-of-order execution. Large caches, sophisticated prefetching, and branch prediction are common in mainstream CPUs. In contrast, the streaming multiprocessor (SM) in GPUs is relatively simple. Each SM has a single fetch unit and multiple scalar units. An instruction is fetched and executed in parallel on all scalar units for a group of data elements (a warp). *Path divergence* occurs on a conditional branch where threads take different paths. The threads on diverging paths are serialized. While each hardware thread is relatively light-weight and weak, GPU employs a large number of them to hide memory latency.

We use NVIDIA Tesla S2050 and IBM P755 as our GPU based platform and CPU based platform, respectively. These machines were introduced to the market at approximately the same time. P755 has 4 Power7 chips. Each chip has 8 cores running at 3.61 GHz, and each core is capable of four-way simultaneous

multithreading (SMT). P755 supports up to 128 threads. Each Power7 core has 32KB L1, 256KB L2, and 4MB on-chip L3 caches. The Tesla S2050 has four Fermi GPUs running at 1.15GHz. We use one GPU as we study only shared-memory algorithms. Each GPU has 14 SMs, 448 cores, and 2GB global memory. Fermi has 64 KB configurable shared memory and L1 cache. The shared L2 is 768K.

We use a variant of the classic Shiloach-Vishkin algorithm (SV) [19] as our baseline algorithm on GPUs. We denote this algorithm CC when in no danger of confusion with the problem it solves. CC was shown to run faster than SV on CPUs [4].

CC uses m processors. It starts with n isolated vertices. Each processor inspects an edge, and tries to graft the larger endpoint (by index) to the smaller one. Grafting creates $k \geq 1$ connected components, and each component is then shortcut to a single super-vertex. Grafting and shortcutting continue on the reduced graph $G' = (V', E')$ with V' being the set of super-vertices and E' being the set of edges among super-vertices until no grafting is possible. The formal description of one graft-and-shortcut iteration in CC is shown in Algorithm 1.[1].

Algorithm 1. CC(El, D), El is the input edge list, $D[i]$ is the current component vertex i belongs to

```
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
```

Figure 1 shows the best performance of CC, with up to 128 threads on P755 and 14 SMs, 448 cores on S2050, on three inputs − a random graph with 50M vertices, 200M edges, a random graph with 100M vertices, 200M edges, and a scale-free graph with 20M vertices, 200M edges. For these inputs, CC is about 2.5 to 4 times faster on S2050 than on P755.

In our experiments, unless noted otherwise, we use maximum input sizes that fit in the GPU global memory. In Figure 1 the performance on P755 peaked with 32 threads instead of 128 threads. The performance on S2050 peaked at 12 SMs instead of 14 SMs. Our experiments confirm that straightforward implementation of PRAM algorithms tends to run faster on GPUs than on CPUs.

The memory subsystem on neither platform delivers data fast enough to keep all processors busy. This is largely due to the indirect, random accesses in CC. As profiling shows that *graft* dominates the execution time (on both platforms

[1] For simplicity and following the tradition of SV, Algorithm 1. assumes that each edge (u,v) appears twice in the edge list as $< u, v >$ and $< v, u >$. In our implementation each edge appears once to limit memory consumption.

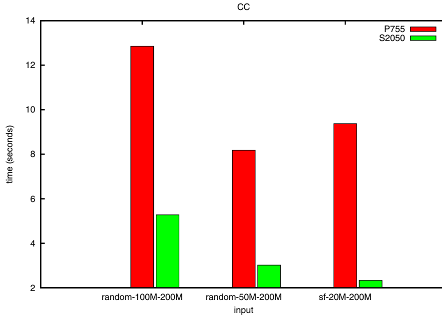

Fig. 1. Straightforward implementation

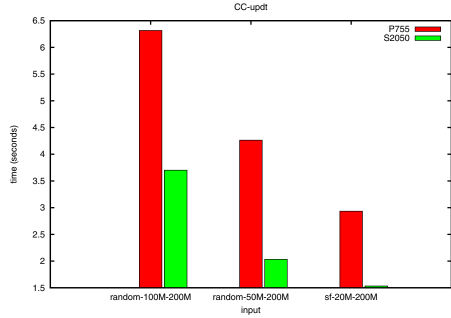

Fig. 2. With coalescing

between 90 and 95 percent of time is spent on *graft*), in our study we focus our optimization effort on *graft*.

3 Locality Optimization on GPUs

Coalescing, that is, merging multiple accesses to memory locations within a short range into one transaction, is critical to performance on GPUs. We consider improving locality to facilitate coalescing (improved locality also results in better cache performance).

Cache-friendly sequential algorithms on CPUs abound in literature (e.g., see [1,13]). Cache-friendly *parallel* algorithms are proposed for modern multicore systems (e.g., see [5,2,8,3]). In practice these parallel algorithms are oftentimes too complex to implement, and have very large algorithmic overhead. Cong and Makarychev proposed *coordinated scheduling* of parallel irregular accesses to improve locality [9]. On cache-based platforms their implementation (*Schd*) achieved superior performance over the straightforward implementation for CC and other graph problems. Unfortunately, in our experiments even Schd incurs too much overhead relative to its performance gain on GPUs.

Chiang *et al.* proposed a sort-based PRAM simulation approach for designing graph algorithms with good locality [7]. A generic sort based approach to order memory accesses for coalescing works as follows for *graft*. Each edge $(u, v) \in El$ is augmented as an edge $(u, v, u', v') \in El'$. Here u' and v' store the component that u and v belong to, respectively. 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 $(u, v, u', v') \in El'$. Note that the retrievals of $D[u]$ and $D[v]$ corresponding to line 2 in Algorithm 1. are coalesced. Random accesses at line 3 can be handled similarly.

We analyze the overhead and performance gain of sorting relative to line 2 in Algorithm 1.. We argue that even the fastest integer sort on current GPUs (i.e., radix sort [18]) is too costly for improving coalescing.

Assume on a GPU $w > 1$ is the coalescing width, that is, random accesses to a range of w words are coalesced. Let T_{orig} be the memory access time for retrieving Ds on line 2 of Algorithm 1., and T_{sort} be the memory access time used in the sort based approach, respectively.

Theorem 1. *For a graph of n vertices and $m = O(n)$ edges, it is necessary that $w > 2^{8/41 \log n}$ for $T_{sort} < T_{orig}$.*

Proof. Let S and R be the time of sequentially accessing $2m$ words and the time of randomly accessing $2m$ words, respectively. $T_{orig} \geq R$.

In the sort-based approach, as $w > 1$, a linear ordering on the accesses is not necessary for coalescing. The n vertices are partitioned into blocks of size w. El' is sorted with the block *id*s of the endpoints as keys. First $El[i].u/w$ is used as key for the retrieval of $D[El[i].u]$, then $El[i].v/w$ is used ask key for the retrieval of $D[El[i].v]$. Each pass of radix sort works with b key bits. It takes $\frac{\log(n/w)}{b}$ passes to sort m keys ranging between 1 and n/w. Computing the histogram in each pass requires one round of sequential accesses to $4m$ words (u' and v' are carried along) that costs $2S$. Copying to the target locations takes $\lceil 2^b/w \rceil S$ time. Thus T_{sort} is at least

$$2\left(\frac{\log \frac{n}{w}}{b}\right)\left(1+\left\lceil\frac{2^b}{w}\right\rceil\right)2S = 4(\log n - logw)\frac{(1+\lceil\frac{2^b}{w}\rceil)}{b}S$$

For any given n and w, T_{sort} is minimized at $b = \log w$ with the value $8(\log n - \log w)S/\log w$. In order that $T_{sort} < T_{orig}$, we need $8(\log n - \log w)S/\log w < R < wS$, and we have $w > 2^{8/41 \log n}$.

For a sparse graph with $n = 200M$ vertices and $m = 400M$ edges that fits in memory, we need $w > 73.01$ for the sort-based approach to beat the straightforward implementation. On most GPUs $w \leq 32$. Unless the coalescing width is increased, sort-based techniques that have been shown to work on CPUs do not improve performance on GPUs.

Software prefetching is another technique to improve the memory performance of parallel graph algorithms on CPUs [9]. 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 (e.g., see[14]). 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.

4 Optimizing Graft-and-Shortcut

We have argued that generic sort-based approaches are too costly on current GPUs. We now explore optimizations specific to the graft-and-shortcut pattern in CC.

Accesses to D at lines 2-3 in Algorithm 1. are irregular and not coalesced. Line 2 compares the current components that u and v are in, and line 3 makes a union

of the two components by grafting the tree rooted at the larger endpoint to the one rooted at the smaller endpoint. While accesses to $D[u]$s and $D[v]$s determined by edges $(u,v) \in E$ are random, the D values evolve in a pattern that can be exploited for coalescing. 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, and the number of unique D values (hence the number of unique super-vertices) also decreases. Instead of retrieving the current components using u and v as indices, we introduce an *update* step after *shortcut* that replaces each edge (u, v) with $(D[u], D[v])$. The revised algorithm CC-updt is shown in Algorithm 2.. The *update* step is done at lines 11-13.

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

1: **for** $1 \leq i \leq m$ parallel **do** {graft}
2: **if** $El[i].u < El[i].v$ **then**
3: $D[El[i].v] \leftarrow El[i].u$
4: **end if**
5: **end for**
6: **for** $1 \leq i \leq n$ parallel **do** {shortcut}
7: **while** $D[i] \neq D[D[i]]$ **do**
8: $D[i] \leftarrow D[D[i]]$
9: **end while**
10: **end for**
11: **for** $1 \leq i \leq m$ parallel **do** {update}
12: $El[i].u \leftarrow D[El[i].u]$, $El[i].v \leftarrow D[El[i].v]$
13: **end for**

In comparison with CC (Algorithm 1.), at first glance CC-updt simply shifts the random accesses from line 2 to line 12. Assuming in each iteration the same grafting choices are made in CC and CC-updt (races among processors may result in different drafting patterns), the same amount of random memory accesses appear to occur in both algorithms. However, as D values become more and more regular, we show accesses to D become increasingly more likely to be coalesced.

Theorem 2. *On average in each iteration CC-updt issues at least $n/2$ fewer random accesses than CC.*

Proof. In the first iteration both algorithms have the same number of random accesses. The number of unique active super-vertices (where grafting can happen) is reduced at least by half by each iteration in CC-updt. After the first iteration, at least $n/2$ edges have two endpoints within the same super-vetex, and for them the accesses to D are coalesced in the second iteration. The number of random accesses is thus reduced by at least $n/2$. The reduction is at least $n/2+n/4$ in the third iteration, and at least $n/2 + n/4 + \cdots + n/2^{i-1}$ in the i-th $(1 \leq i \leq \log n)$ iteration. In the last iteration where no grafting is possible, the reduction is $m \geq n = 2(n/2)$. Thus on average each iteration issues at least $n/2$ fewer random accesses.

We evaluate the performance of CC-updt on S2050. The results with a scale-free graph of 20M vertices and 200M edges are shown in Figure 2. Speedups

between 1.75 and 1.83 are achieved. The observed performance improvement is due to better coalescing and possibly better cache performance. Performance study of CC-updt in comparison to CC is also done on P755. The speedups are between 1.66 and 3.0, as shown in Figure 2. Comparing Figures 1 and 2, we notice that the performance gap between P755 and S2050 is reduced for CC-updt.

5 A Meta Algorithm for Further Improvement

We propose a new meta algorithm motivated by the following result from evolution random graph theory [16] that further improves coalescing.

Theorem 3. *Under the Erdös-Rényi model there is a unique giant component of order $f(c)n$ in the graph when $m \sim cn$ with $c > 1/2$.*

Function $f(c)$ approaches 1 as c increases[2]. We exploit the giant component for coalescing. The algorithm, *Stages*, is shown in Algorithm 3..

Stages first permutes the edges in El, and then divides them into groups, $El_1, El_2, \cdots, El_g$, with $|El_i| > n/2$ $(1 \leq i \leq g-1)$ except possibly for El_g. Next for each group El_i $(1 \leq i \leq g)$ Stages invokes a connected components algorithm, say, CC, with El_i and D, and updates the endpoints of each edge in El_{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.

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**

Theorem 4. *Algorithm 3. computes connected components.*

Proof. By induction. By the correctness of CC, the connected components of the induced graph with El_1 are computed and contracted. Each vertex i $(1 \leq i \leq n)$ belongs to a component represented by $D[i]$. Assume after processing k groups $El_1, El_2, \cdots, El_k$ of edges $(1 \leq k < g)$, the connected components of the induced

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

graph are computed and contracted, and $D[i]$ is the connected component for vertex i $(1 \leq i \leq n)$. The *update* step in Algorithm 3. transforms the edges in E_{k+1} of the original graph into edges of the current contracted graph. Subsequent computation in CC is with edges in E_{k+1} on super-vertices with $i = D[i]$ $(1 \leq i \leq n)$. All other vertices have $D[i] < i$, that is, each of them shoots a pointer to its super-vertex. The components for these vertices are updated in the *shortcut* step in CC when it sets $D[i] \leftarrow D[D[i]]$ for all vertices. After CC, connected components on edges induced by $El_1, El_2, \cdots, El_{k+1}$ are computed with $D[i]$ representing the current component for each vertex i $(1 \leq i \leq n)$.

CC takes $O(\log^2 n)$ time with $O(m+n)$ processors under CRCW PRAM. With p processors CC takes $O\left(\frac{m+n}{p}\log^2 n\right)$ time. Graft-and-shortcut in Stages takes $O\left(\frac{m}{pqn}(qn+n)\log^2 n\right)$ time, while *update* takes $O\left(\frac{m}{p}\log n\right)$ time. CC and Stages have the same asymptotic complexity when $qn = \Theta(m)$. Stages degenerates into CC when $qn = m$.

Let T_{cc} and T_{stgs} be the (worst-case) number of non-coalesced random accesses in *graft* for CC and Stages, respectively.

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

Proof. As CC takes at most $\log n$ arounds of *graft* with $2m$ random accesses each, $T_{cc} = 2m\log n$. Similarly, in Stages the first group of qn edges incur at most $2qn\log n$ random accesses. A component of size $f(q)n$ forms after the first group of edges are processed. For the second group the probability that an edge is contained in the giant component is $f^2(q)$. The expected number of random accesses is at most $2(1-f^2(q))qn\log((1-f(q))n+1)$. Before the jth $(1 < j \leq g)$ group of qn edges, the giant component is of size $f((j-1)q)n$, and the number of random accesses in processing the jth group is at most $2(1 - f^2((j-1)q))qn\log((1-f((j-1)q))n+1)$. Let $f(0) = 0$, we have

$$\begin{aligned} T_{stgs} \leq & \sum_{k=0}^{\frac{m}{qn}-1}(1-f^2(kq))2qn\log((1-f(kq))n+1) \\ \leq & \ 2qn\left(\log n + \sum_{k=1}^{\frac{m}{qn}-1}(1-f^2(q))\log((1-f(q))n+1)\right) \\ = & \ 2qn\left(\log n + \left(\frac{m}{qn}-1\right)(1-f^2(q))\log((1-f(q))n+1)\right) \end{aligned}$$

$$\begin{aligned} T_{cc} - T_{stgs} \geq & \ 2m\log n - 2qn\left(\log n + \left(\frac{m}{qn}-1\right)(1-f^2(q))\log((1-f(q))n+1)\right) \\ > & \left(2m - 2qn - \frac{m}{qn}(1-f^2(q))\right)\log n \end{aligned}$$

The extra cost of random memory accesses in *update* is at most $2m - 2qn$. $T_s > T_{cc} - T_{stgs}$ when n is large enough and $m = O(n)$.

Our results are derived for random graphs. The experiments below show the technique is effective for other graphs.

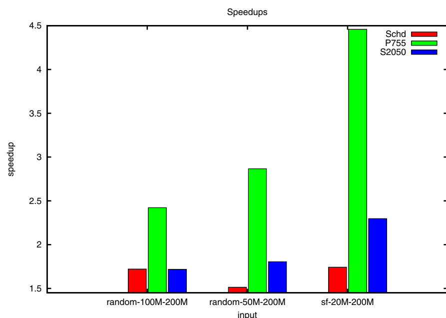

Fig. 3. CC-updt speedups

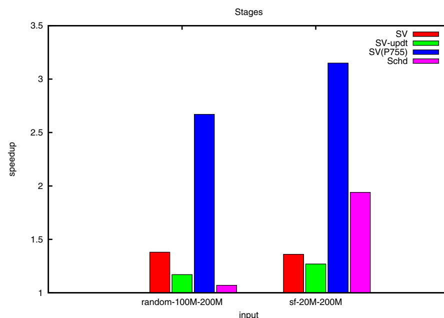

Fig. 4. Stages speedups

Figure 3 shows the speedups of Stages over CC for three different inputs on P755 and S2050. We use 128 hardware threads on P755 and 14 SMs on S2050. In our implementation $q = 1$. The speedups achieved on S2050 are between 1.7 to 2.3. Stages is much faster than CC on P755. The speedups are between 2.4 and 4.5. In the figure *Schd* shows the speedup of coordinated scheduling over CC on P755. Schd is the best prior locality optimized implementation of CC. On P755 Stages clearly outperforms Schd.

As a meta algorithm, Stages can be used to optimize other parallel connected components algorithms. Figure 4 shows for a random graph and a scale-free graph the speedup of stages over various connected components algorithms. SV, SV-updt (with an additional *update* step), and Schd are called at line 4 of Algorithm 3. instead of CC. For each input, the two bars on the left are the speedups of Stages over SV and SV-updt on S2050, and the two bars on the right are the speedups over SV and Schd on P755. We use 14 SMs on S2050 and 128 threads on P755. Stages improves the performance of all algorithms studied. The speedups on S2050 are between 1.07 to 2.6, and the speedups on P755 are between 1.3 and 3.1.

6 Ranking the Algorithms

We now rank the performance of the algorithms on S2050. Due to their similarity to CC and CC-updt, we do not include SV and SV-updt in our rankings. We include parallel Rem's algorithm (Rem) by Patwary *et al.* [17] in our study. In comparison to CC and SV, Rem is not based on the PRAM model. It does not shortcut the trees, and uses union-find structures to determine whether two vertices are in the same component. Rem is largely asynchronous and resolves data races through a verification algorithm. Rem is efficient for a moderate number of threads. One drawback of Rem is that increasing the available parallelism can result in many rounds of verification and thus degrades performance. Rem is faster than CC on current CPUs [17]. Our optimizations do not apply to Rem as it does not shortcut the trees.

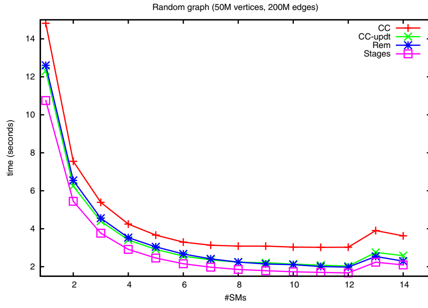

Fig. 5. Performance of four implementations on a random graph

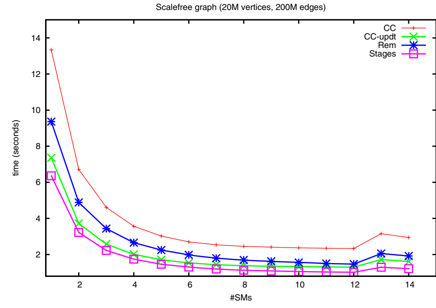

Fig. 6. Performance of four implementations on a scale-free graph

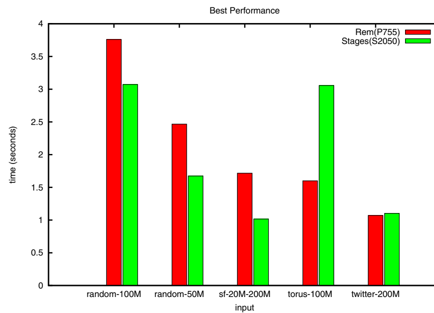

Fig. 7. Performance comparison on P755 and S2050

Figures 5 and 6 show on S2050 the performance of CC, CC-updt, Stages, and Rem with the same inputs, that is, a random graph with 50M vertices, 200M edges, and a scale-free graph with 20M vertices, 200M edges, respectively. CC-updt is always faster than CC, and Stages is always faster than CC-updt. Rem is slower than CC-updt for the scale-free graph.

Our new algorithm, Stages, is the fastest on S2050, while Rem (with software prefetching) is the fastest on P755 (due to limited space, we do not show the performance plots for CC, CC-updt, Stages, Schd, and Rem on P755). The relative poor performance of Rem on S2050 is largely due to path divergence resulted from its complex control flow in union-find.

We compare the performance of Stages on S2050 and Rem on P755. In addition to random graphs and scale-free graphs, we include a 2D torus and a sample of the twitter network in our test. Torus represents graphs (such as VLSI circuits) of regular topology with long diameters, and the twitter network is an example of real life social networks. The torus has about 100M vertices. The twitter sample has 35M vertices and 200M edges.

Figure 7 shows the performance of the fastest implementations on the two platforms. We use 128 threads on P755 and 14 SMs on S2050. The two algorithms have roughly comparable performance for small-world graphs. Stages on S2050

is faster than Rem on P755 with the random graphs and the scale-free graph. Rem on P755 is faster than Stages on S2050 with the torus. They have similar performance with the twitter graph.

7 Conclusion and Future Work

We present our study of optimizing connected components algorithms on GPUs in comparison with CPUs. We show that 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.

We argue that generic techniques to improve locality for better performance are too costly on GPUs. We propose a low-cost coalescing optimization for CC. We further present a meta algorithm that improves coalescing for several connected components algorithms. Interestingly, these optimizations also improve performance on CPUs. In fact, our implementation beats the best prior locality-optimized implementation on P755.

We find that Stages is consistently the fastest algorithm on S2050. Rem is fast on P755, but its performance suffers from path divergence on S2050. Rem has similar performance as CC-updt on S2050. Stages on S2050 and Rem on P755 on average have similar performance over a range of inputs.

In future work we will study graph algorithms on multi-GPUs and a cluster of GPUs. We will also study architectural support for efficient execution of graph algorithms on emerging architectures.

References

1. Arge, L., Bender, M.A., Demaine, E.D., Holland-Minkley, B., Munro, J.I.: Cache-oblivious priority queue and graph algorithm applications. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, Montreal, Canada, pp. 268–276 (2002)
2. Arge, L., Goodrich, M.T., Nelson, M., Sitchinava, N.: Fundamental parallel algorithms for private-cache chip multiprocessors. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA 2008, pp. 197–206. ACM, New York (2008)
3. Arge, L., Goodrich, M.T., Sitchinava, N.: Parallel external memory graph algorithms. In: 24th IEEE International Parallel & Distributed Processing Symposium, Atlanta, Georgia, USA (2010)
4. Bader, D.A., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, New Mexico (April 2004)
5. Blelloch, G.E., Chowdhury, R.A., Gibbons, P.B., Ramachandran, V., Chen, S., Kozuch, M.: Provably good multicore cache performance for divide-and-conquer algorithms. In: In Proc. 19th ACM-SIAM Sympos. Discrete Algorithms, pp. 501–510 (2008)

6. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Proc. 4th SIAM Intl. Conf. on Data Mining (April 2004)
7. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proceedings of the 1995 Symposium on Discrete Algorithms, pp. 139–149 (1995)
8. Chowdhury, R., Silvestri, F., Blakeley, B., Ramachandran, V.: Oblivious algorithms for multicores and network of processors. In: 24th IEEE International Parallel & Distributed Processing Symposium, Atlanta, Georgia, USA (2010)
9. Cong, G., Makarychev, K.: Optimizing large-scale graph analysis on multi-threaded, multi-core platforms. In: Proceedings of the 2012 IEEE International Parallel & Distributed Processing Symposium, IPDPS 2012, pp. 414–425. IEEE Computer Society, Washington, DC (2012)
10. Dehne, F., Yogaratnam, K.: Exploring the limits of GPUs with parallel graph algorithms. CoRR, abs/1002.4482 (2010)
11. Goh, K.-I., Oh, E., Jeong, H., Kahng, B., Kim, D.: Classification of scale-free networks. Proc. Natl. Acad. Sci. 99, 12583–12588 (2002)
12. Hong, S., Oguntebi, T., Olukotun, K.: Efficient parallel graph exploration on multi-core cpu and gpu. In: 2011 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 78–88 (October 2011)
13. Ladner, R., Fix, J.D., LaMarca, A.: The cache performance of traversals and random accesses. In: Proc. 10th Ann. Symp. Discrete Algorithms (SODA-1999), pp. 613–622. ACM-SIAM, Baltimore (1999)
14. Lee, J., Lakshminarayanaand, N.B., Hyesoon, K., Vuduc, R.: Many-thread aware prefetching mechanisms for GPGPU applications. In: 43rd Annual IEEE/ACM Int'l Symp on Microarchitecture (MICRO), pp. 213–224 (December 2010)
15. Luo, L., Wong, M., Hwu, W.: An effective gpu implementation of breadth-first search. In: 2010 47th ACM/IEEE Design Automation Conference (DAC), pp. 52–55 (June 2010)
16. Palmer, E.M.: Graphical evolution. Wiley-Interscience Series in Discrete Mathematics. Wiley (1985)
17. Patwary, M.A., Ref, P., Manne, F.: Multi-core spanning forest algorithms using the disjoint-set data structure. In: Proceedings of the 2012 IEEE International Parallel & Distributed Processing Symposium, IPDPS 2012, pp. 827–835. IEEE Computer Society Press, Washington, DC (2012)
18. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for many-core GPUs. In: Proceedings of the 2009 IEEE Int'l Symp. on Parallel&Distributed Processing, IPDPS 2009, pp. 1–10. IEEE Computer Society, Washington, DC (2009)
19. Shiloach, Y., Vishkin, U.: An $O(\log n)$ parallel connectivity algorithm. J. Algs 3(1), 57–67 (1982)