Scalable GPU Graph Traversal

Duane Merrill
University of Virginia
Charlottesville
Virginia
USA
dgm4d@virginia.edu

Michael Garland
NVIDIA Corporation
Santa Clara
California
USA
mgarland@nvidia.com

Andrew Grimshaw
University of Virginia
Charlottesville
Virginia
USA
grimshaw@virginia.edu

Abstract

Breadth-first search (BFS) is a core primitive for graph traversal and a basis for many higher-level graph analysis algorithms. It is also representative of a class of parallel computations whose memory accesses and work distribution are both irregular and data-dependent. Recent work has demonstrated the plausibility of GPU sparse graph traversal, but has tended to focus on asymptotically inefficient algorithms that perform poorly on graphs with non-trivial diameter.

We present a BFS parallelization focused on fine-grained task management constructed from efficient prefix sum that achieves an asymptotically optimal $O(|V|+|E|)$ work complexity. Our implementation delivers excellent performance on diverse graphs, achieving traversal rates in excess of 3.3 billion and 8.3 billion traversed edges per second using single and quad-GPU configurations, respectively. This level of performance is several times faster than state-of-the-art implementations both CPU and GPU platforms.

Categories and Subject Descriptors G.2.2 [**Discrete Mathematics**]: Graph Theory – Graph Algorithms; D.1.3 [**Programming Techniques**]: Concurrent programming; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – Computations on discrete structures, Geometrical problems and computations

General Terms Algorithms, performance

Keywords Breadth-first search, GPU, graph algorithms, parallel algorithms, prefix sum, graph traversal, sparse graph

1. Introduction

Algorithms for analyzing sparse relationships represented as graphs provide crucial tools in many computational fields ranging from genomics to electronic design automation to social network analysis. In this paper, we explore the parallelization of one fundamental graph algorithm on GPUs: breadth-first search (BFS). BFS is a common building block for more sophisticated graph algorithms, yet is simple enough that we can analyze its behavior in depth. It is also used as a core computational kernel in a number of benchmark suites, including Parboil [26], Rodinia [10], and the emerging Graph500 supercomputer benchmark [29].

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

Contemporary processor architecture provides increasing parallelism in order to deliver higher throughput while maintaining energy efficiency. Modern GPUs are at the leading edge of this trend, provisioning tens of thousands of data parallel threads.

Despite their high computational throughput, GPUs might appear poorly suited for sparse graph computation. In particular, BFS is representative of a class of algorithms for which it is hard to obtain significantly better performance from parallelization. Optimizing memory usage is non-trivial because memory access patterns are determined by the structure of the input graph. Parallelization further introduces concerns of contention, load imbalance, and underutilization on multithreaded architectures [3, 21, 32]. The wide data parallelism of GPUs can be particularly sensitive to these performance issues.

Prior work on parallel graph algorithms has relied on two key architectural features for performance. The first is multithreading and overlapped computation for hiding memory latency. The second is fine-grained synchronization, specifically atomic read-modify-write operations. Atomic mechanisms are convenient for coordinating the dynamic placement of data into shared data structures and for arbitrating contended status updates. [3–5]

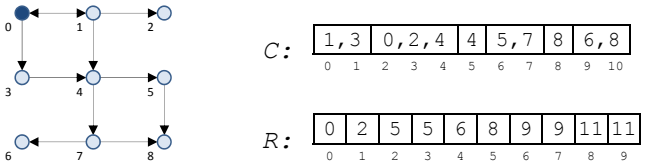
Modern GPU architectures provide both. However, serialization from atomic synchronization is particularly expensive for GPUs in terms of efficiency and performance. In general, mutual exclusion does not scale to thousands of threads. Furthermore, the occurrence of fine-grained and dynamic serialization within the SIMD width is much costlier than between overlapped SMT threads.

For machines with wide data parallelism, we argue that software prefix sum [7, 17] is often a more suitable approach to data placement. Prefix sum is a bulk-synchronous algorithmic primitive that can be used to compute scatter offsets for concurrent threads given their dynamic allocation requirements. Efficient GPU prefix sums [24] allow us to reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph traversal.

Our work as described in this paper makes contributions in the following areas:

Parallelization strategy. We present a GPU BFS parallelization that performs an asymptotically optimal linear amount of work. It is the first to incorporate fine-grained parallel adjacency list expansion. We also introduce local duplicate detection techniques for avoiding race conditions that create redundant work. We demonstrate that our approach delivers high performance on a broad spectrum of structurally diverse graphs. To our knowledge, we also describe the first design for multi-GPU graph traversal.

Empirical performance characterization. We present detailed analyses that isolate and analyze the expansion and contraction



Traversal from source vertex v_0		
BFS Iteration	**Vertex frontier**	**Edge frontier**
1	{0}	{1,3}
2	{1,3}	{0,2,4,4}
3	{2,4}	{5,7}
4	{5,7}	{6,8,8}
5	{6,8}	{}

Fig. 1. Example sparse graph, corresponding CSR representation, and frontier evolution for a BFS beginning at source vertex v_0.

Algorithm 1. The simple sequential breadth-first search algorithm for marking vertex distances from the source s.

Input: Vertex set V, row-offsets array R, column-indices array C, source vertex v_s
Output: Array $dist[0..n\text{-}1]$ with $dist[v_i]$ holding the distance from v_s to v_i
Functions: *Enqueue*(*val*) inserts *val* at the end of the queue instance. *Dequeue*() returns the front element of the queue instance.

```
1    Q := {}
2    for i in 0 .. |V|-1:
3      dist[i] := ∞
4    dist[s] := 0
5    Q.Enqueue(s)
6    while (Q != {}) :
7      i = Q.Dequeue()
8      for offset in R[i] .. R[i+1]-1 :
9        j := C[offset]
10       if (dist[j] == ∞)
11         dist[j] := dist[i] + 1;
12         Q.Enqueue(j)
```

aspects of BFS throughout the traversal process. We reveal that serial and warp-centric expansion techniques described by prior work significantly underutilize the GPU for important graph genres. We also show that the fusion of neighbor expansion and inspection within the same kernel often yields worse performance than performing them separately.

High performance. We demonstrate that our methods deliver excellent performance on a diverse body of real-world graphs. Our implementation achieves traversal rates in excess of 3.3 billion and 8.3 billion traversed edges per second (TE/s) for single and quad-GPU configurations, respectively. In context, recent state-of-the-art parallel implementations achieve 0.7 billion and 1.3 billion TE/s for similar datasets on single and quad-socket multicore processors [3].

2. Background

Modern NVIDIA GPU processors consist of tens of multiprocessor cores, each of which manages on the order of a thousand hardware-scheduled threads. Each multiprocessor core employs data parallel SIMD (single instruction, multiple data) techniques in which a single instruction stream is executed by a fixed-size grouping of threads called a *warp*. A *cooperative thread array* (or CTA) is a group of threads that will be co-located on the same multiprocessor and share a local scratch memory. Parallel threads are used to execute a single program, or *kernel*.

2.1 Breadth-first search

We consider graphs of the form $G = (V, E)$ with a set V of n vertices and a set E of m directed edges. Given a source vertex v_s, our goal is to traverse the vertices of G in breadth-first order starting at v_s. Each newly-discovered vertex v_i will be labeled by (a) its distance d_i from v_s and/or (b) the predecessor vertex p_i immediately preceding it on the shortest path to v_s. For simplicity, we identify the vertices v_0 .. v_{n-1} using integer indices. The pair (v_i, v_j) indicates a directed edge in the graph from $v_i \rightarrow v_j$, and the adjacency list $A_i = \{v_j \mid (v_i, v_j) \in E\}$ is the set of neighboring vertices incident on vertex v_i. We treat undirected graphs as symmetric directed graphs containing both (v_i, v_j) and (v_j, v_i) for each undirected edge. In this paper, all graph sizes and traversal rates are measured in terms of directed edge counts.

We represent the graph using an adjacency matrix **A**, whose rows are the adjacency lists A_i. The number of edges within sparse graphs is typically only a constant factor larger than n. We use the well-known compressed sparse row (CSR) sparse matrix format to store the graph in memory consisting of two arrays. Fig. 1 provides a simple example. The column-indices array C is formed from the set of the adjacency lists concatenated into a single array of m integers. The row-offsets R array contains $n + 1$ integers, and entry $R[i]$ is the index in C of the adjacency list A_i. We store graphs in the order they are defined and do not perform any preprocessing in order to improve locality or load balance.

Algorithm 1 presents the standard sequential BFS method. It operates by circulating the vertices of the graph through a FIFO queue that is initialized with v_s [11]. As vertices are dequeued, their neighbors are examined. Unvisited neighbors are labeled with their distance and/or predecessor and are enqueued for later processing. This algorithm performs linear $O(m+n)$ work since each vertex is labeled exactly once and each edge is traversed exactly once.

2.2 Parallel breadth-first search

The FIFO ordering of the sequential algorithm forces it to label vertices in increasing order of depth. Each depth level is fully explored before the next. Most parallel BFS algorithms are *level-synchronous*: each level may be processed in parallel as long as the sequential ordering of levels is preserved. An implicit race condition can exist where multiple tasks may concurrently discover a vertex v_j. This is generally considered benign since all such contending tasks would apply the same d_j and give a valid value of p_j.

Structurally different methods may be more suitable for graphs with very large diameters, e.g., algorithms based on the method of Ullman and Yannakakis [30]. Such alternatives are beyond the scope of this paper.

As illustrated in Fig. 1, each iteration of a level-synchronous method identifies both an edge and vertex *frontier*. The edge-frontier is the set of all edges to be traversed during that iteration or, equivalently, the set of all A_i where v_i was marked in the previous iteration. The vertex-frontier is the unique subset of such neighbors that are unmarked and which will be labeled and expanded for the next iteration. Each iteration logically (1) expands vertices into an edge-frontier, i.e., neighbor expansion; and then (2) contracts them to a vertex-frontier, i.e., status-lookup and filtering.

Quadratic parallelizations. The simplest parallel BFS algorithms inspect every edge or, at a minimum, every vertex during every iteration. These methods perform a quadratic amount of work. A vertex v_j is marked when a task discovers an edge $v_i \rightarrow v_j$ where v_i has been marked and v_j has not. Vertex-oriented variants must subsequently expand and mark the neighbors of v_j. Their work complexity is $O(n^2+m)$ as there may n BFS iterations in the worst case.

Quadratic parallelization strategies have been used by almost all prior GPU implementations. The static assignment of tasks to vertices (or edges) trivially maps to the data-parallel GPU machine model. Each thread's computation is completely independent from that of other threads. Harish *et al*. [16] and Hussein *et al*. [20] describe vertex-oriented versions of this method. Deng *et al*. present an edge-oriented implementation [12].

Hong *et al*. [18] describe a vectorized version of the vertex-oriented method that is similar to the CSR sparse matrix-vector (SpMV) multiplication approach by Bell and Garland [6]. Rather than threads, warps are mapped to vertices. During neighbor expansion, the SIMD lanes of an entire warp are used to strip-mine the corresponding adjacency list.

These quadratic methods are isomorphic to iterative SpMV in the algebraic semi-ring where the usual (+, ×) operations are replaced with (min, +), and thus can also be realized using generic implementations of SpMV [14].

Linear parallelizations. A work-efficient parallel BFS algorithm should perform $O(n+m)$ work. To achieve this, each iteration should examine only the edges and vertices in that iteration's logical edge and vertex-frontiers, respectively.

Frontiers may be maintained *in core* or *out of core*. An in-core frontier is processed online and never wholly realized. On the other hand, a frontier that is managed out-of-core is fully produced in off-chip memory for consumption by the next BFS iteration after a global synchronization step. Implementations typically prefer to manage the vertex-frontier out-of-core. Less global data movement is needed because the average vertex-frontier is smaller by a factor of $\bar{d}$ (average out-degree). For each iteration, tasks are mapped to unexplored vertices in the input vertex-frontier queue. Their neighbors are inspected and the unvisited ones are placed into the output vertex-frontier queue for the next iteration.

The typical approach for improving utilization is to reduce the task granularity to a homogenous size and then evenly distribute these smaller tasks among threads. This is done by expanding and inspecting neighbors in parallel. The implementation can either: (a) spawn all edge-inspection tasks before processing any, wholly realizing the edge-frontier out-of-core; or (b) carefully throttle the parallel expansion and processing of adjacency lists, producing and consuming these tasks in-core.

Leiserson and Schardl [21] designed an implementation for multi-socket systems that incorporates a novel multi-set data structure for tracking the vertex-frontier. Bader and Madduri [4] describe an implementation for the Cray MTA-2 using the hardware's full-empty bits for efficient queuing into an out-of-core vertex frontier. Both approaches perform parallel adjacency-list expansion, relying on runtimes to throttle edge-processing tasks in-core.

Luo *et al*. [22] present an implementation for GPUs that relies upon a hierarchical scheme for producing an out-of-core vertex-frontier. To our knowledge, theirs is the only prior attempt at designing a work-efficient BFS algorithm for GPUs. Threads perform serial adjacency list expansion and use an upward propagation tree of child-queue structures in an effort to mitigate the contention overhead on any given atomically-incremented queue pointer.

Distributed parallelizations. It is often desirable to partition the graph structure amongst multiple processors, particularly for very large datasets. The typical partitioning approach is to assign each processing element a disjoint subset of V and the corresponding adjacency lists in E. For a given vertex v_i, the inspection and marking of v_i as well as the expansion of v_i's adjacency list must occur on the processor that owns v_i. Distributed, out-of-core edge queues are used for communicating neighbors to remote processors. Incoming neighbors that are unvisited have their labels marked and their adjacency lists expanded. As adjacency lists are expanded, neighbors are enqueued to the processor that owns them. The synchronization between BFS levels occurs after the expansion phase.

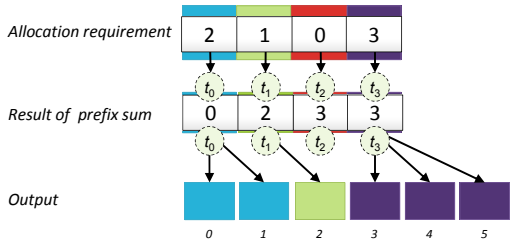


Fig. 2. Example of prefix sum for computing scatter offsets for run-length expansion. Input order is preserved.

It is important to note that distributed BFS implementations that construct predecessor trees will impose twice the queuing I/O as those that construct depth-rankings. These variants must forward the full edge pairing (v_i, v_j) to the remote processor so that it might properly label v_j's predecessor as v_i.

Yoo *et al.* [33] present a variation for BlueGene/L that implements a two-dimensional partitioning strategy for reducing the number of remote peers each processor must communicate with. Xia and Prasanna [32] propose a variant for multi-socket nodes that provisions more out-of-core edge-frontier queues than active threads, reducing the contention at any given queue and flexibly lowering barrier overhead.

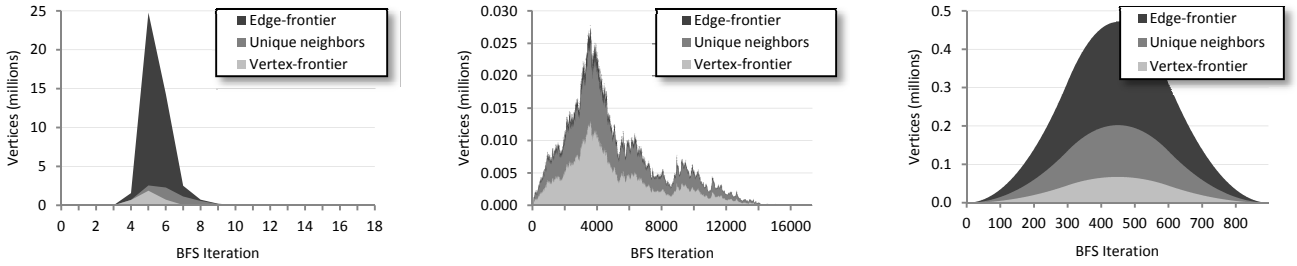
Agarwal *et al.* [3] describe an implementation for multi-socket systems that implements both out-of-core vertex and edge-frontier queues for each socket. Scarpazza *et al.* [27] describe a similar hybrid variation for the Cell BE processor architecture where DMA engines are used instead of threads to perform parallel adjacency list expansion.

Our parallelization strategy. In comparison, our BFS strategy expands adjacent neighbors in parallel; implements out-of-core edge and vertex-frontiers; uses local prefix sum in place of local atomic operations for determining enqueue offsets; and uses a best-effort bitmask for efficient neighbor filtering. We further describe the details in Section 5.

2.3 Prefix sum

Given a list of input elements and a binary reduction operator, *prefix scan* produces an output list where each element is computed to be the reduction of the elements occurring earlier in the input list. *Prefix sum* connotes a prefix scan with the addition operator. Software-based scan has been popularized as an algorithmic primitive for vector and array processor architectures [7–9] and as well as for GPUs [13, 24, 28].

Prefix sum is a particularly useful mechanism for implementing cooperative allocation, i.e., when parallel threads must place dynamic data within shared data structures such as global queues. Given a list of allocation requirements for each thread, prefix sum computes the offsets for where each thread should start writing its output elements. Fig. 2 illustrates prefix sum in the context of run-length expansion. In this example, the thread t_0 wants to produce two items, t_1 one item, t_2 zero items, and so on. The prefix sum computes the scatter offset needed by each thread to write its output element. Thread t_0 writes its items at offset zero, t_1 at offset two, t_3 at offset three, etc. In the context of parallel BFS, parallel threads use prefix sum when assembling global edge frontiers from expanded neighbors and when outputting unique unvisited vertices into global vertex frontiers.



(a) ***wikipedia-20070206*** **(b)** ***europe.osm*** **(c)** ***grid7pt.300***

Fig. 3. Sample frontier plots of logical vertex and edge-frontier sizes during graph traversal.

3. Benchmark Suite

3.1 Graph datasets

Our benchmark suite is composed of the thirteen graphs listed in Table 1. We generate the square and cubic Poisson lattice graph datasets ourselves. The *random.2Mv.128Me* and *rmat.2Mv.128Me* datasets are constructed using GTgraph [15]. The *wikipedia-20070206* dataset is from the University of Florida Sparse Matrix Collection [31]. The remaining datasets are from the 10th DIMACS Implementation Challenge [1].

One of our goals is to demonstrate good performance for large-diameter graphs. The largest components within these datasets have diameters spreading five orders of magnitude. Graph diameter is directly proportional to average search depth, the expected number of BFS iterations for a randomly-chosen source vertex.

3.2 Logical frontier plots

Although our sparsity plots reveal a diversity of locality, they provide little intuition as to how traversal will unfold. Fig. 3 presents sample *frontier plots* of logical edge and vertex-frontier sizes as functions of BFS iteration. Such plots help visualize workload expansion and contraction, both within and between iterations. The ideal numbers of neighbors expanded and vertices labeled per iteration are constant properties of the given dataset and starting vertex.

Frontier plots reveal the concurrency exposed by each iteration. For example, the bulk of the work for the *wikipedia-20070206* dataset is performed in only 1-2 iterations. The hardware can easily be saturated during these iterations. We observe that real-world datasets often have long sections of light work that incur heavy global synchronization overhead.

Finally, Fig. 3 also plots the duplicate-free subset of the edge-frontier. We observe that a simple duplicate-removal pass can perform much of the contraction work from edge-frontier down to vertex-frontier. This has important implications for distributed BFS. The amount of network traffic can be significantly reduced by first removing duplicates from the expansion of remote neighbors.

We note the direct application of this technique does not scale linearly with processors. As p increases, the number of available duplicates in a given partition correspondingly decreases. In the extreme where $p = m$, each processor owns only one edge and there are no duplicates to be locally culled. For large p, such decoupled duplicate-removal techniques should be pushed into the hierarchical interconnect. Yoo *et al*. demonstrate a variant of this idea for BlueGene/L using their MPI set-union collective [33].

Name	Sparsity Plot	Description	n (10^6)	m (10^6)	$\bar{d}$	Avg. Search Depth
europe.osm		European road network	50.9	108.1	2.1	19314
grid5pt.5000		5-point Poisson stencil (2D grid lattice)	25.0	125.0	5.0	7500
hugebubbles-00020		Adaptive numerical simulation mesh	21.2	63.6	3.0	6151
grid7pt.300		7-point Poisson stencil (3D grid lattice)	27.0	188.5	7.0	679
nlpkkt160		3D PDE-constrained optimization	8.3	221.2	26.5	142
audikw1		Automotive finite element analysis	0.9	76.7	81.3	62
cage15		Electrophoresis transition probabilities	5.2	94.0	18.2	37
kkt_power		Nonlinear optimization (KKT)	2.1	13.0	6.3	37
coPapersCiteseer		Citation network	0.4	32.1	73.9	26
wikipedia-20070206		Links between Wikipedia pages	3.6	45.0	12.6	20
kron_g500-logn20		Graph500 RMAT (A=0.57, B=0.19, C=0.19)	1.0	100.7	96.0	6
random.2Mv.128Me		$G(n, M)$ uniform random	2.0	128.0	64.0	6
rmat.2Mv.128Me		RMAT (A=0.45, B=0.15, C=0.15)	2.0	128.0	64.0	6

Table 1. Suite of benchmark graphs

4. Microbenchmark Analyses

A linear BFS workload is composed of two components: $O(n)$ work related to vertex-frontier processing, and $O(m)$ for edge-frontier processing. Because the edge-frontier is dominant, we focus our attention on the two fundamental aspects of its operation: *neighbor-gathering* and *status-lookup*. Although their functions are trivial, the GPU machine model provides interesting challenges for these workloads. We investigate these two activities in the following analyses using NVIDIA Tesla C2050 GPUs.

4.1 Isolated neighbor-gathering

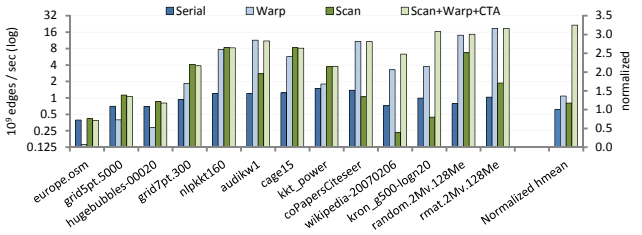
This analysis investigates serial and parallel strategies for simply gathering neighbors from adjacency lists. The enlistment of threads for parallel gathering is a form task scheduling. We evaluate a spectrum of scheduling granularity from individual tasks (higher scheduling overhead) to blocks of tasks (higher underutilization from partial-filling).

(a) serial

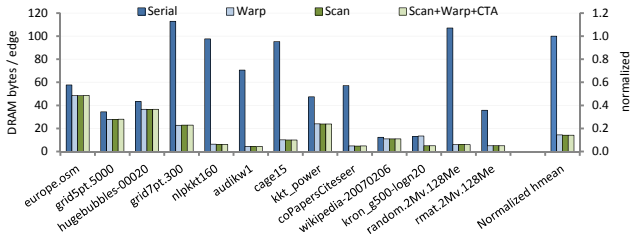
(b) coarse-grained, warp-based cooperative expansion (emphasis on controlling thread)

(c) fine-grained, scan-based cooperative expansion

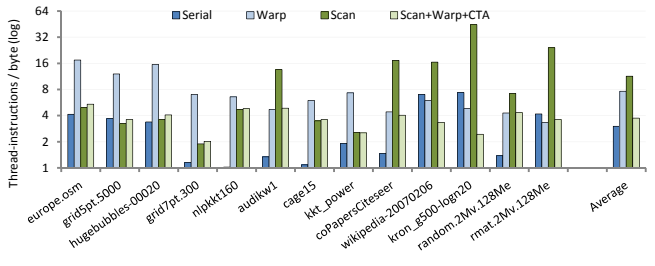
Fig. 4. Alternative strategies for gathering four unexplored adjacency lists having lengths 2, 1, 0, and 3.



(a) Average gather rate (log)



(b) Average DRAM overhead



(c) Average computational intensity (log)

Fig. 5. Neighbor-gathering behavior. Harmonic means are normalized with respect to serial-gathering.

For a given BFS iteration, our test kernels simply read an array of preprocessed row-ranges that reference the adjacency lists to be expanded and then load the corresponding neighbors into local registers[1]. The gathered neighbors are not output into a global edge frontier (which would require extra overhead from prefix sum and scatter).

Serial gathering. Each thread obtains its preprocessed row-range bounds for the vertex it is to expand and then serially acquires the corresponding neighbors from the column-indices array *C*. Fig. 4a illustrates four threads assigned to gather four unexplored adjacency lists having lengths 2, 1, 0, and 3. Graphs having non-uniform degree distributions can impose significant load imbalance between threads within the same warp.

Coarse-grained, warp-based gathering. This approach performs a coarse-grained redistribution of gathering workloads. Instead of processing adjacency lists individually, each thread will enlist its entire warp to gather its assigned adjacency list. Consider our example adjacency lists as being assigned to threads from different warps. Fig. 4b illustrates three warps gathering the three non-empty adjacency lists in "broadside" parallel fashion, each under the control of a specific thread.

Enlistment operates by having each thread attempt to vie for control of its warp by writing its thread-identifier into a single word shared by all threads of that warp. Only one write will succeed, thus determining which is subsequently allowed to command the warp as a whole to gather its corresponding neighbors. The enlistment process repeats until all threads have all had their adjacent neighbors gathered.

Although it provides better workload balance, this approach can suffer underutilization within the warp. Many datasets have an average adjacency list size that is much smaller than the warp width, leaving warp read transactions under filled. Furthermore, there may also be load imbalance between warps when threads within one warp have significantly larger adjacency lists to expand than those in others.

Fine-grained, scan-based gathering. This approach performs a fine-grained redistribution of gathering workloads. Threads construct a shared array of column-indices offsets corresponding to a CTA-wide concatenation of their assigned adjacency lists. For our running example, the prefix sum in Fig. 2 illustrates the cooperative expansion of column-indices offsets into a shared gather vector. As illustrated in Fig. 4c, we then enlist the entire CTA to gather the referenced neighbors from the column-indices array *C* using this perfectly packed gather vector. This assignment of threads ensures that no SIMD lanes are unutilized during global reads from *C*.

Compared to the two previous strategies, the entire CTA participates in every read. Any workload imbalance between threads is not magnified by expensive global memory accesses to *C*. Instead, workload imbalance can occur in the form of underutilized cycles during offset-sharing. The worst case entails a single thread having more neighbors than the gather buffer can accommodate, resulting in the idling of all other threads while it alone shares gather offsets.

Scan+warp+CTA gathering. We can mitigate this imbalance by supplementing fine-grained scan-based expansion with coarser CTA-based and warp-based expansion. CTA-wide gathering is similar to warp-based gathering, except threads vie for control of the entire CTA for strip-mining very large adjacency lists. Then we apply warp-based gathering to acquire adjacency smaller than the CTA size, but greater than the warp width. Finally we perform scan-based gathering to efficiently acquire the remaining "loose ends".

This hybrid strategy limits all forms of load imbalance from adjacency list expansion. The fine-grained work redistribution of

[1] For full BFS, we do not perform any preprocessing

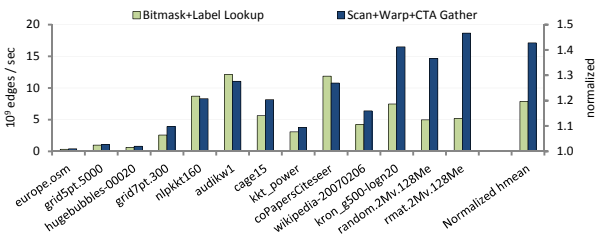


Fig. 6. Comparison of status-lookup with neighbor-gathering.

scan-based gathering limits imbalance from SIMD lane underutilization. Warp enlistment limits offset-sharing imbalance between threads. CTA enlistment limits imbalance between warps. And finally, any imbalance between CTAs can be limited by oversubscribing GPU cores with an abundance of CTAs or implementing coarse-grained tile-stealing mechanisms for CTAs to dequeue tiles at their own rate. We implement both CTA-scheduling policies, binding one or the other for each kernel as an architecture-specific tuning decision.

Analysis. We performed 100 randomly-sourced traversals of each dataset, evaluating these kernels on the logical vertex-frontier for every iteration. Fig. 5a plots the average edge-processing throughputs for each strategy in log-scale. The datasets are ordered from left-to-right by decreasing average search depth.

The serial approach performs poorly for the majority of datasets. Fig. 5b reveals it suffers from dramatic over-fetch. It plots bytes moved through DRAM per edge. The arbitrary references from each thread within the warp result in terrible coalescing for SIMD load instructions.

The warp-based approach performs poorly for the graphs on the left-hand side having average ≤ 10. Fig. 5c reveals that it is computationally inefficient for these datasets. It plots a log scale of computational intensity, the ratio of thread-instructions versus bytes moved through DRAM. The average adjacency lists for these graphs are much smaller than the number of threads per warp. As a result, a significant number of SIMD lanes go unused during any given cycle.

Fig. 5c also reveals that that scan-based gathering can suffer from extreme workload imbalance when only one thread is active within the entire CTA. This phenomenon is reflected in the datasets on the right-hand size having skewed degree distributions. The load imbalance from expanding large adjacency lists leads to increased instruction counts and corresponding performance degradation.

Combining the benefits of bulk-enlistment with fine-grained utilization, the hybrid scan+warp+CTA demonstrates good gathering rates across the board.

4.2 Isolated status-lookup and concurrent discovery

Status-lookup is the other half to neighbor-gathering; it entails checking vertex labels to determine which neighbors within the edge-frontier have already been visited. Our strategy for status-lookup incorporates a bitmask to reduce the size of status data from a 32-bit label to a single bit per vertex. CPU parallelizations have used atomically-updated bitmask structures to reduce memory traffic via improved cache coverage [3, 27].

Because we avoid atomic operations, our bitmask is only a conservative approximation of visitation status. Bits for visited vertices may appear unset or may be "clobbered" due to false-sharing within a single byte. If a status bit is unset, we must then check the corresponding label to ensure the vertex is safe for

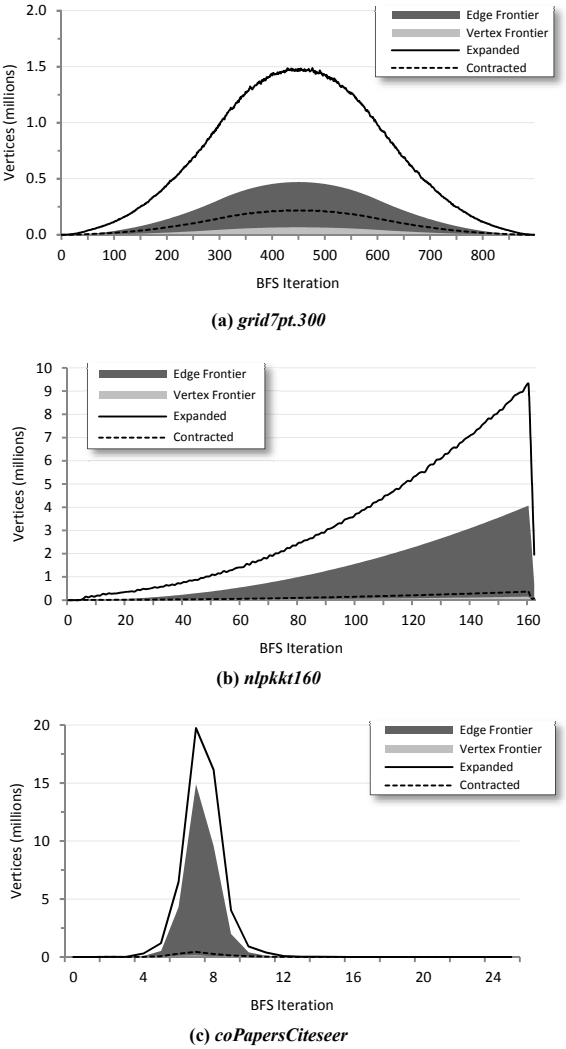


Fig. 7. Actual expanded and contracted queue sizes without local duplicate culling, superimposed over logical frontier sizes. The redundant expansion factors are 2.6x, 1.7x, and 1.1x for the *grid7pt.300*, *nlpkkt160*, and *coPapersCiteseer* datasets, respectively.

marking. This scheme relies upon capacity and conflict misses to update stale bitmask data within the read-only texture caches.

Similar to the neighbor-gathering analysis, we isolate the status-lookup workload using a test-kernel that consumes the logical edge-frontier at each BFS iteration. The filtered neighbors are not output into a global vertex frontier (which would require extra overhead from prefix sum and scatter). Fig. 6 compares the throughputs of lookup versus gathering workloads. We observe that status-lookup is generally the more expensive of the two. This is particularly true for the datasets on the right-hand side having high average vertex out-degree. The ability for neighbor-gathering to coalesce accesses to adjacency lists increases with , whereas accesses for status-lookup have arbitrary locality.

Concurrent discovery. The effectiveness of status-lookup during frontier contraction is influenced by the presence of duplicate vertex identifiers within the edge-frontier. Duplicates are representative of different edges incident to the same vertex. This can pose a problem for implementations that allow the benign race condition. When multiple threads concurrently discover the same vertices via these duplicates, the corresponding adjacency lists will be expanded multiple times.

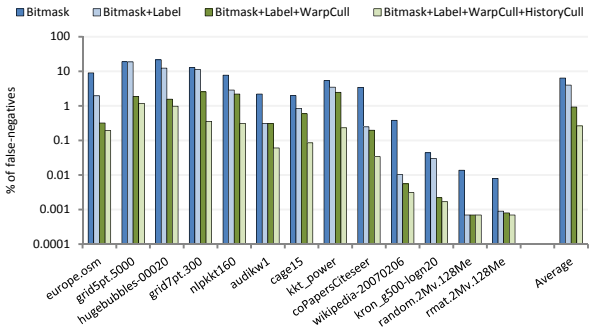


Fig. 8 Percentages of false-negatives incurred by status-lookup strategies.

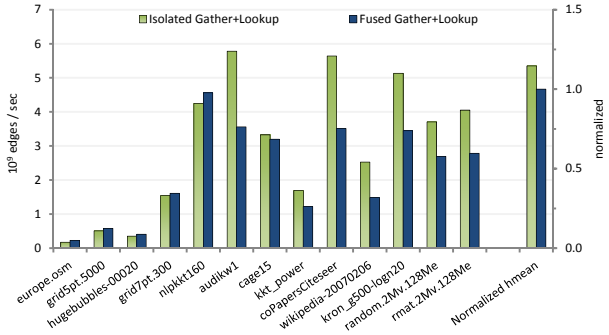


Fig. 9. Comparison of isolated versus fused neighbor-gathering and lookup.

Prior CPU parallelizations have noted the potential for redundant work, but concluded its manifestation to be negligible [21]. Concurrent discovery on CPU platforms is rare due to a combination of relatively low parallelism (~8 hardware threads) and coherent L1 caches that provide only a small window of opportunity around status-inspections that are immediately followed by status updates.

The GPU machine model, however, is much more vulnerable. If multiple threads within the same warp are simultaneously inspecting same vertex identifier, the SIMD nature of the warp-read ensures that all will obtain the same status value. If unvisited, the adjacency list for this vertex will be expanded for every thread.

We illustrate the effects of redundant expansion upon overall workload for several datasets using a simplified version of the *two-phase* BFS implementation described in Section 5.3. These expansion and contraction kernels make no special effort to curtail concurrent discovery. Fig. 7 plots the actual numbers of vertex identifiers expanded and contracted for each BFS iteration alongside the corresponding logical frontiers. The deltas between these pairs reflect the generation of unnecessary work. We define the *redundant expansion factor* as the ratio of neighbors actually enqueued versus the number of edges logically traversed.

The problem is severe for spatially-descriptive datasets. These datasets exhibit nearby duplicates within the edge-frontier due to their high frequency of convergent exploration. For example, simple two-phase traversal incurs 4.2x redundant expansion for the 2D lattice *grid5pt.5000* dataset. Even worse, the implementation altogether fails to traverse the *kron_g500-logn20* dataset which encodes sorted adjacency lists. The improved locality enables the redundant expansion of ultra-popular vertices, ultimately exhausting physical memory when filling the edge queue.

This issue of redundant expansion appears to be unique to GPU BFS implementations having two properties: (1) a work-efficient traversal algorithm; and (2) concurrent adjacency list expansion. Quadratic implementations do not suffer redundant work because vertices are never expanded by more than one thread. In our evaluation of linear-work serial-expansion, we observed negligible concurrent SIMD discovery during serial inspection due to the independent nature of thread activity.

In general, the issue of concurrent discovery is a result of false-negatives during status-lookup, i.e., failure to detect previously-visited and duplicate vertex identifiers within the edge-frontier. Atomic read-modify-write updates to visitation status yield zero false-negatives. As alternatives, we introduce two localized mechanisms for reducing false-negatives: (1) *warp culling* and (2) *history culling*.

Warp culling. This heuristic attempts to mitigate concurrent SIMD discovery by detecting the presence of duplicates within the warp's immediate working set. Using shared-memory per warp, each thread hashes in the neighbor it is currently inspecting. If a collision occurs and a different value is extracted, nothing can be determined regarding duplicate status. Otherwise threads write their thread-identifier into the same hash location. Only one write will succeed. Threads that subsequently retrieve a different thread-identifier can safely classify their neighbors as duplicates to be culled.

History culling. This heuristic complements the instantaneous coverage of warp culling by maintaining a cache of recently-inspected vertex identifiers in local shared memory. If a given thread observes its neighbor to have been previously recorded, it can classify that neighbor as safe for culling.

Analysis. We augment our isolated lookup tests to evaluate these heuristics. Kernels simply read vertex identifiers from the edge-frontier and determine which should not be allowed into the vertex-frontier. For each dataset, we record the average percentage of false negatives with respect to $m - n$, the ideal number of culled vertex identifiers.

Fig. 8 illustrates the progressive application of lookup mechanisms. The bitmask heuristic alone incurs an average false-negative rate of 6.4% across our benchmark suite. The addition of label-lookup (which makes status-lookup safe) improves this to 4.0%. Without further measure, the compounding nature of redundant expansion allows even small percentages to accrue sizeable amounts of extra work. For example, a false-negative rate of 3.5% for traversing *kkt_power* results in a 40% redundant expansion overhead.

The addition of warp-based culling induces a tenfold reduction in false-negatives for spatially descriptive graphs (left-hand side). The history-based culling heuristic further reduces culling inefficiency by a factor of five for the remainder of high-risk datasets (middle-third). The application of both heuristics allows us to reduce the overall redundant expansion factor to less than 1.05x for every graph in our benchmark suite.

4.3 Coupling of gathering and lookup

A complete BFS implementation might choose to fuse these workloads within the same kernel in order to process one of the frontiers online and in-core. We evaluate this fusion with a derivation of our scan+warp+CTA gathering kernel that immediately inspects every gathered neighbor using our bitmap-assisted lookup strategy. The coupled kernel requires $O(m)$ less overall data movement than the other two put together (which effectively read all edges twice).

Fig. 9 compares this fused kernel with the aggregate throughput of the isolated gathering and lookup workloads performed separately. Despite the additional data movement, the separate kernels outperform the fused kernel for the majority of the benchmarks. Their extra data movement results in net slowdown, however, for the latency-bound datasets on the left-

hand side having limited bulk concurrency. The implication is that fused approaches are preferable for fleeting BFS iterations having edge-frontiers smaller than the number of resident threads.

The fused kernel likely suffers from TLB misses experienced by the neighbor-gathering workload. The column-indices arrays occupy substantial portions of GPU physical memory. Sparse gathers from them are apt to cause TLB misses. The fusion of these two workloads inherits the worst aspects of both: TLB turnover during uncoalesced status lookups.

5. Single-GPU Parallelizations

A complete solution must couple expansion and contraction activities. In this section, we evaluate the design space of coupling alternatives by constructing full implementations for processing BFS iterations. Further algorithmic detail can be found in our technical report [25].

5.1 Expand-contract (out-of-core vertex queue)

Our single-kernel *expand-contract* strategy is loosely based upon the fused gather-lookup benchmark kernel from Section 4.3. It consumes the vertex queue for the current BFS iteration and produces the vertex queue for the next. It performs parallel expansion and filtering of adjacency lists online and in-core using local scratch memory.

This kernel requires $2n$ global storage for input and output vertex queues. The roles of these two arrays are reversed for alternating BFS iterations. A traversal will generate $5n+2m$ explicit data movement through global memory. All m edges will be streamed into registers once. All n vertices will be streamed twice: out into global frontier queues and subsequently back in. The bitmask bits will be inspected m times and updated n times along with the labels. Each of the n row-offsets is loaded twice.

Each CTA performs three local prefix sums per block of dequeued input. One is computed during scan-based gathering. The other two are used for computing global enqueue offsets for valid neighbors during CTA-based and scan-based gathering. Although GPU cores can efficiently overlap concurrent prefix sums from different CTAs, the turnaround time for each can be relatively long. This can hurt performance for fleeting, latency-bound BFS iterations.

5.2 Contract-expand (out-of-core edge queue)

Our *contract-expand* strategy filters previously-visited and duplicate neighbors from the current edge queue. The adjacency lists of the surviving vertices are then expanded and copied out into the edge queue for the next iteration.

This kernel requires $2m$ global storage for input and output edge queues. Variants that label predecessors, however, require an additional pair of "parent" queues to track both origin and destination identifiers within the edge-frontier. A traversal will generate $3n+4m$ explicit global data movement. All m edges will be streamed through global memory three times: into registers from C, out to the edge queue, and back in again the next iteration. The bitmask, label, and row-offset traffic remain the same as for *expand-contract*.

Despite a much larger queuing workload, the c*ontract-expand* strategy is often better suited for processing small, fleeting BFS iterations. It incurs lower latency because CTAs only perform local two prefix sums per block: one each for computing global enqueue offsets during CTA/warp-based and scan-based gathering. We overlap these prefix sums to further reduce latency. By operating on the larger edge-frontier, the c*ontract-expand* kernel also enjoys better bulk concurrency in which fewer resident CTAs sit idle.

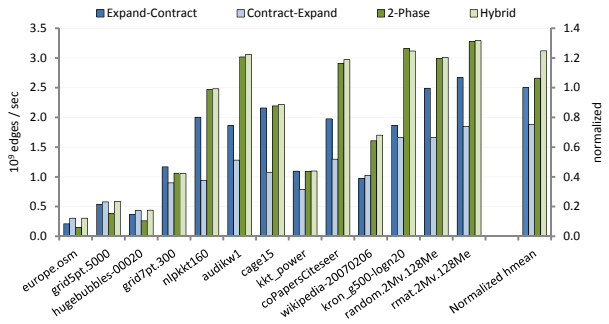


Fig. 10 BFS traversal performance. Harmonic means are normalized with respect to the *expand-contract* implementation.

5.3 Two-phase (out-of-core vertex and edge queues)

Our *two-phase* implementation isolates the expansion and contraction workloads into separate kernels. The expansion kernel employs the scan+warp+CTA gathering strategy to obtain the neighbors of vertices from the input vertex queue. As with the *contract-expand* implementation above, it performs two overlapped local prefix sums to compute scatter offsets for the expanded neighbors into the global edge queue.

The contraction kernel begins with the edge queue as input. Threads filter previously-visited and duplicate neighbors. The remaining valid neighbors are placed into the outgoing vertex queue using another local prefix sum to compute global enqueue offsets.

These kernels require $n+m$ global storage for vertex and edge queues. A *two-phase* traversal generates $5n+4m$ explicit global data movement. The memory workload builds upon that of *contract-expand*, but additionally streams n vertices into and out of the global vertex queue.

5.4 Hybrid

Our hybrid implementation combines the relative strengths of the *contract-expand* and *two-phase* approaches: low-latency turnaround for small frontiers and high-efficiency throughput for large frontiers. If the edge queue for a given BFS iteration contains more vertex identifiers than resident threads, we invoke the *two-phase* implementation for that iteration. Otherwise we invoke the *contract-expand* implementation. The hybrid approach inherits the $2m$ global storage requirement from the former and the $5n+4m$ explicit global data movement from the latter.

5.5 Strategy evaluation

In comparing these strategies, Fig. 10 plots average traversal throughput across 100 randomly-sourced traversals of each dataset using a single NVIDIA Tesla C2050. As anticipated, the *contract-expand* approach excels at traversing the latency-bound datasets on the left and the *two-phase* implementation efficiently leverages the bulk-concurrency exposed by the datasets on the right. Although the *expand-contract* approach is serviceable, the *hybrid* approach meets or exceeds its performance for every dataset.

The importance of work-compaction. With in-core edge-frontier processing, the *expand-contract* implementation is designed for one-third as much global queue traffic. The actual DRAM savings are substantially less. We only measured a 50% reduction in measured DRAM workload for datasets with large $\bar{d}$. Furthermore, the workload differences are effectively lost in excess over-fetch traffic for the graphs having small $\bar{d}$: they use large memory transactions to retrieve small adjacency lists.

Graph Dataset	CPU Parallel (linear-work)		GPU* (quadratic-work [18])	GPU* (linear-work *hybrid* strategy)	
	*Distance BFS rate*** [21]	*Predecessor BFS rate**** [3]	*Distance BFS rate*	*Distance BFS rate* (*sequential speedup* *****)	*Predecessor BFS rate* (*sequential speedup* *****)
europe.osm			0.00014	0.31 (11x)	0.31 (11x)
grid5pt.5000			0.00078	0.6 (7.4x)	0.57 (7.0x)
hugebubbles-00020			0.00061	0.43 (15x)	0.42 (15x)
grid7pt.300	0.12		0.012	1.1 (29x)	0.97 (26x)
nlpkkt160	0.47		0.21	2.5 (9.7x)	2.1 (8.2x)
audikw1			1.2	3.0 (4.6x)	2.5 (3.9x)
cage15	0.23		0.50	2.2 (18x)	1.9 (15x)
kkt_power	0.11		0.18	1.1 (23x)	1.0 (21x)
coPapersCiteseer			2.2	3.0 (6.0x)	2.5 (5.0x)
wikipedia-20070206	0.19		0.39	1.6 (25x)	1.4 (22x)
kron_g500-logn20			1.5	3.1 (13x)	2.5 (10x)
random.2Mv.128Me		0.50	1.2	3.0 (29x)	2.4 (23x)
rmat.2Mv.128Me		0.70	1.3	3.3 (22x)	2.6 (17x)

Table 2. Average single-socket graph traversal rates (10^9 TE/s). * NVIDIA 14-core 1.15 GHz Tesla C2050. ** Intel 4-core 2.5 GHz Core i7. *** Intel 8-core 2.7 GHz Xeon X5570. **** GPU speedup versus sequential method on Intel 3.4GHz Core i7 2600K.

The *contract-expand* implementation performs poorly for graphs having large $\bar{d}$. This behavior is related to a lack of explicit workload compaction before neighbor gathering. It executes nearly 50% more thread-instructions during BFS iterations with very large contraction workloads. This is indicative of SIMD underutilization. The majority of active threads have their neighbors invalidated by status-lookup and local duplicate removal. Cooperative neighbor-gathering becomes much less efficient as a result.

5.6 Comparative performance

Table 2 compares the distance and predecessor-labeling versions of our *hybrid* strategy with prior BFS parallelizations for both GPU and multicore CPU architectures.

Distance vs. predecessor-labeling. Our performance disparity between the two BFS problem types is largely dependent upon average vertex degree $\bar{d}$. Smaller $\bar{d}$ incurs larger DRAM over-fetch which reduces the relative significance of added parent queue traffic. For example, the performance impact of exchanging parent vertices is negligible for *europe.osm*, yet is as high as 19% for *rmat.2Mv.128Me*.

Contemporary GPU parallelizations. In comparing our approach with the recent quadratic-work method of Hong *et al*. [18], we evaluated their implementation directly on our corpus of sparse graphs. We observed a 4.2x harmonic mean slowdown across all datasets. As expected, their implementation incurs particularly large overheads for high diameter graphs, notably a 2300x slowdown for *europe.osm*. At the other end of the spectrum, we measured a 2.5x slowdown for *rmat.2Mv.128Me*, the lowest diameter dataset.

The only prior published linear-work GPU performance evaluation is from Luo et al. [22]. In the absence of their hand-tuned implementation, we compared our implementation against the specific collections of 6-pt lattice datasets[2] and DIMACS road network datasets[3] referenced by their study. Using the same model GPU (a previous-generation NVIDIA GTX280), our *hybrid* parallelization respectively achieved 4.1x and 1.7x harmonic mean speedups for these two collections.

Contemporary multicore parallelizations. It is challenging to contrast CPU and GPU traversal performance. The construction of high performance CPU parallelizations is outside the scope of this work. Table 2 cites the recent single-socket CPU traversal rates by Leiserson et al. [21] and Agarwal et al. [3] for datasets common to our experimental corpus. With an NVIDIA C2050, we achieve harmonic mean speedups of 8.1x and 4.2x versus their respective 4-core and 8-core parallelizations.

To give perspective on the datasets for which we do not have published CPU performance rates, we note these two studies report sub-linear performance scaling per core. In this vein, we compare GPU traversal performance with our own efficient sequential implementation on a state-of-the-art Intel 4-core 3.4 GHz Core i7 2600K. Despite fewer memory channels on our newer CPU, the performance of our sequential implementation exceeds their single-threaded results.

With respect to this single-threaded implementation, we consider a 4x GPU speedup as being competitive with contemporary CPU parallelizations. As listed in Table 2, our C2050 traversal rates exceed this factor for all benchmark datasets. In addition, the majority of our graph traversal rates exceed 12x speedup, the perfect scaling of three such CPUs. At the extreme, our average *wikipedia-20070206* traversal rates outperform the sequential CPU version by 25x, i.e., eight CPU-equivalents.

Relative to the sequential CPU implementation, we also note that our methods perform equally well for large and small-diameter graphs alike. Our *hybrid* strategy provides traversal speedups of an order of magnitude for both the *europe.osm* and the *kron_g500-logn20* datasets.

6. Multi-GPU Parallelization

Communication between GPUs is simplified by a unified virtual address space in which pointers can transparently reference data residing within remote GPUs. PCI-express 2.0 provides each GPU with an external bidirectional bandwidth of 6.6 GB/s. Under the assumption that GPUs send and receive equal amounts of traffic, the rate at which each GPU can be fed with remote work is conservatively bound by 825×10^6 neighbors / sec, where neighbors are 4-byte identifiers. This rate is halved for predecessor-labeling.

6.1 Design

We implement a simple partitioning of the graph into equally-sized, disjoint subsets of V. For a system of p GPUs, we initialize each processor p_i with an (m/p)-element C_i and (n/p)-element R_i and $Labels_i$ arrays. Because the system is small, we can provision each GPU with its own full-sized n-bit best-effort bitmask.

We stripe ownership of V across the domain of vertex identifiers. Striping provides good probability of an even distribution of adjacency list sizes across GPUs, an important property for maintaining load balance in small systems. However, this method of partitioning progressively loses any inherent locality as the number of GPUs increases.

[2] Regular degree-6 cubic lattice graphs of size 1M, 2M, 5M, 7M, 9M, and 10M vertices

[3] New York, Florida, USA-East, and USA-West datasets from the 9th DIMACS Challenge corpus [2].

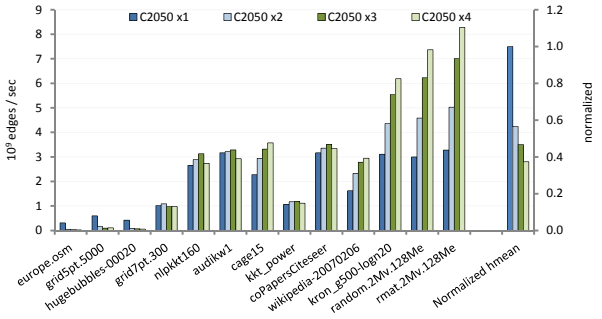


Fig. 11. Average multi-GPU traversal rates. Harmonic means are normalized with respect to the single GPU configuration.

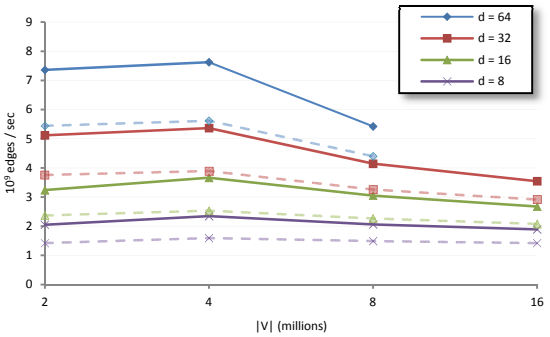


Fig. 12. Multi-GPU sensitivity to graph size and average out-degree for uniform random graphs using four C2050 processors. Dashed lines indicate predecessor labeling variants.

Graph traversal proceeds in level-synchronous fashion. The host program orchestrates BFS iterations as follows:

1. Invoke the *expansion* kernel on each GPU_i, transforming the vertex queue $Qvertex_i$ into an edge queue $Qedge_i$.
2. Invoke a fused *filter+partition* operation for each GPU_i that sorts neighbors within $Qedge_i$ by ownership into p bins. Vertex identifiers undergo opportunistic local duplicate culling and bitmask filtering during the partitioning process. This partitioning implementation is analogous to a three-kernel radix-sorting pass [23].
3. Barrier across all GPUs. The sorting must be completed on all GPUs before any can access their bins on remote peers. The host program uses this opportunity to terminate traversal if all bins are empty on all GPUs.
4. Invoke p-1 *contraction* kernels on each GPU_i to stream and filter the incoming neighbors from its peers. Kernel invocation simply uses remote pointers that reference the appropriate peer bins. This assembles each vertex queue $Qvertex_i$ for the next BFS iteration.

The implementation requires $(2m+n)/p$ storage for queue arrays per GPU: two edge queues for pre and post-sorted neighbors and a third vertex queue to avoid another global synchronization after Step 4.

6.2 Evaluation

Fig. 11 presents traversal throughput as we scale up the number of GPUs. We experience net slowdown for datasets on the left having average search depth > 100. The cost of global synchronization between BFS iterations is much higher across multiple GPUs.

We do yield notable speedups for the three rightmost datasets. These graphs have small diameters and require little global synchronization. The large average out-degrees enable plenty of opportunistic duplicate filtering during partitioning passes. This allows us to circumvent the PCI-e cap of 825×10^6 edges/sec per GPU. With four GPUs, we demonstrate traversal rates of 7.4 and 8.3 billion edges/sec for the uniform-random and RMAT datasets respectively.

As expected, this strong-scaling is not linear. For example, we observe 1.5x, 2.1x, and 2.5x speedups when traversing *rmat.2Mv.128Me* using two, three, and four GPUs, respectively. Adding more GPUs reduces the percentage of duplicates per processor and increases overall PCI-e traffic.

Fig. 12 further illustrates the impact of opportunistic duplicate culling for uniform random graphs up to 500M edges and varying out out-degree . Increasing yields significantly better performance. Other than a slight performance drop at n=8 million vertices when the bitmask exceeds the 768KB L2 cache size, graph size has little impact upon traversal throughput.

To our knowledge, these are the fastest traversal rates demonstrated by a single-node machine. The work by Agarwal et al. is representative of the state-of-the-art in CPU parallelizations, demonstrating up to 1.3 billion edges/sec for both uniform-random and RMAT datasets using four 8-core Intel Nehalem-based XEON CPUs [3]. However, we note that the host memory on such systems can further accommodate datasets having tens of billions of edges.

7. Conclusion

This paper has demonstrated that GPUs are well-suited for sparse graph traversal and can achieve very high levels of performance on a broad range of graphs. We have presented a parallelization of BFS tailored to the GPU's requirement for large amounts of fine-grained, bulk-synchronous parallelism.

Furthermore, our implementation performs an asymptotically optimal amount of work. While quadratic-work methods might be acceptable in certain very narrow regimes [18, 19], they suffer from high overhead and did not prove effective on even the lowest diameter graphs in our experimental corpus. Our linear-work method compares very favorably to state-of-the-art multicore implementations across our entire range of benchmarks, which spans five orders of magnitude in graph diameter.

Beyond graph search, our work distills several general themes for implementing sparse and dynamic problems for the GPU machine model:

- Prefix sum can serve as an effective alternative to atomic read-modify-write mechanisms for coordinating the placement of items within shared data structures by many parallel threads.
- In contrast to coarse-grained parallelism common on multicore processors, GPU kernels cannot afford to have individual threads streaming through unrelated sections of data. Groups of GPU threads should cooperatively assist each other for data movement tasks.
- Fusing heterogeneous tasks does not always produce the best results. Global redistribution and compaction of fine-grained tasks can significantly improve performance when the alternative would allow significant load imbalance or underutilization.
- The relative I/O contribution from global task redistribution can be less costly than anticipated. The data movement from reorganization may be insignificant in comparison to the

actual over-fetch traffic from existing sparse memory accesses.

- It is useful to provide separate implementations for saturating versus fleeting workloads. Hybrid approaches can leverage a shorter code-path for retiring underutilized phases as quickly as possible.

8. References

[1] 10th DIMACS Implementation Challenge: *http://www.cc.gatech.edu/dimacs10/index.shtml*. Accessed: 2011-07-11.

[2] 9th DIMACS Implementation Challenge: *http://www.dis.uniroma1.it/~challenge9/download.shtml*. Accessed: 2011-07-11.

[3] Agarwal, V. et al. 2010. Scalable Graph Exploration on Multicore Processors. *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (New Orleans, LA, USA, Nov. 2010), 1-11.

[4] Bader, D.A. and Madduri, K. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. *2006 International Conference on Parallel Processing (ICPP'06)* (Columbus, OH, USA), 523-530.

[5] Bader, D.A. et al. On the Architectural Requirements for Efficient Execution of Graph Algorithms. *2005 International Conference on Parallel Processing (ICPP'05)* (Oslo, Norway), 547-556.

[6] Bell, N. and Garland, M. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), 18:1–18:11.

[7] Blelloch, G.E. 1990. *Prefix Sums and Their Applications*. Synthesis of Parallel Algorithms.

[8] Blelloch, G.E. 1989. Scans as primitive parallel operations. *IEEE Transactions on Computers*. 38, 11 (Nov. 1989), 1526-1538.

[9] Chatterjee, S. et al. 1990. Scan primitives for vector computers. *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 1990), 666–675.

[10] Che, S. et al. 2009. Rodinia: A benchmark suite for heterogeneous computing. *2009 IEEE International Symposium on Workload Characterization (IISWC)* (Austin, TX, USA, Oct. 2009), 44-54.

[11] Cormen, T.H. et al. 2001. *Introduction to Algorithms*. MIT Press.

[12] Deng, Y. (Steve) et al. 2009. Taming irregular EDA applications on GPUs. *Proceedings of the 2009 International Conference on Computer-Aided Design* (New York, NY, USA, 2009), 539–546.

[13] Dotsenko, Y. et al. 2008. Fast scan algorithms on graphics processors. *Proceedings of the 22nd annual international conference on Supercomputing* (New York, NY, USA, 2008), 205–213.

[14] Garland, M. 2008. Sparse matrix computations on manycore GPU's. *Proceedings of the 45th annual Design Automation Conference* (New York, NY, USA, 2008), 2–6.

[15] GTgraph: A suite of synthetic random graph generators: *https://sdm.lbl.gov/~kamesh/software/GTgraph/*. Accessed: 2011-07-11.

[16] Harish, P. and Narayanan, P.J. 2007. Accelerating large graph algorithms on the GPU using CUDA. *Proceedings of the 14th international conference on High performance computing* (Berlin, Heidelberg, 2007), 197–208.

[17] Hillis, W.D. and Steele, G.L. 1986. Data parallel algorithms. *Communications of the ACM*. 29, 12 (Dec. 1986), 1170-1183.

[18] Hong, S. et al. 2011. Accelerating CUDA graph algorithms at maximum warp. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA, 2011), 267–276.

[19] Hong, S. et al. 2011. Efficient Parallel Graph Exploration for Multi-Core CPU and GPU. (New York, NY, USA, 2011), to appear.

[20] Hussein, M. et al. 2007. On Implementing Graph Cuts on CUDA. *First Workshop on General Purpose Processing on Graphics Processing Units* (Boston, MA, Oct. 2007).

[21] Leiserson, C.E. and Schardl, T.B. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2010), 303–314.

[22] Luo, L. et al. 2010. An effective GPU implementation of breadth-first search. *Proceedings of the 47th Design Automation Conference* (New York, NY, USA, 2010), 52–55.

[23] Merrill, D. and Grimshaw, A. 2011. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*. 21, 02 (2011), 245-272.

[24] Merrill, D. and Grimshaw, A. 2009. *Parallel Scan for Stream Architectures*. Technical Report #CS2009-14. Department of Computer Science, University of Virginia.

[25] Merrill, D. et al. 2011. *High Performance and Scalable GPU Graph Traversal*. Technical Report #CS2011-05. Department of Computer Science, University of Virginia.

[26] Parboil Benchmark suite: *http://impact.crhc.illinois.edu/parboil.php*. Accessed: 2011-07-11.

[27] Scarpazza, D.P. et al. 2008. Efficient Breadth-First Search on the Cell/BE Processor. *IEEE Transactions on Parallel and Distributed Systems*. 19, 10 (Oct. 2008), 1381-1395.

[28] Sengupta, S. et al. 2008. *Efficient parallel scan algorithms for GPUs*. Technical Report #NVR-2008-003. NVIDIA.

[29] The Graph 500 List: *http://www.graph500.org/*. Accessed: 2011-07-11.

[30] Ullman, J. and Yannakakis, M. 1990. High-probability parallel transitive closure algorithms. *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures - SPAA '90* (Island of Crete, Greece, 1990), 200-209.

[31] University of Florida Sparse Matrix Collection: *http://www.cise.ufl.edu/research/sparse/matrices/*. Accessed: 2011-07-11.

[32] Xia, Y. and Prasanna, V.K. 2009. Topologically Adaptive Parallel Breadth-first Search on Multicore Processors. *21st International Conference on Parallel and Distributed Computing and Systems (PDCS'09)* (Nov. 2009).

[33] Yoo, A. et al. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. *ACM/IEEE SC 2005 Conference (SC'05)* (Seattle, WA, USA), 25-25.