

High-Performance and Scalable GPU Graph Traversal

DUANE MERRILL and MICHAEL GARLAND, NVIDIA Corporation
and ANDREW GRIMSHAW, University of Virginia

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 nontrivial diameter.

We present a BFS parallelization focused on fine-grained task management constructed from efficient prefix sum computations that achieves an asymptotically optimal $O(|V| + |E|)$ gd 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 on 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: Design, Algorithms, Performance

Additional Key Words and Phrases: Breadth-first search, GPU, graph algorithms, parallel algorithms, prefix sum, graph traversal, sparse graphs

ACM Reference Format:

Merrill, D., Garland, M., and Grimshaw, A. 2015. High-performance and scalable GPU graph traversal. *ACM Trans. Parallel Comput.* 1, 2, Article 14 (January 2015), 30 pages.

DOI: <http://dx.doi.org/10.1145/2717511>

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 article, 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 simple enough so 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 [Stratton et al. 2012], Rodinia [Che et al. 2009], and the emerging Graph500 supercomputer benchmark [Graph500 List 2011].

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.

D. Merrill was supported in part by a NVIDIA Graduate Fellowship.

Authors' addresses: D. Merrill (corresponding author), M. Garland, NVIDIA Corporation; email: duane.merrill@gmail.com; A. Grimshaw, University of Virginia, Charlottesville, VA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

2015 Copyright is held by the author/owner(s). 2329-4949/2015/01-ART14

DOI: <http://dx.doi.org/10.1145/2717511>

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 nontrivial because memory access patterns are determined by the structure of the input graph, and parallelization further introduces concerns of contention, load imbalance, and underutilization on multithreaded architectures [Agarwal et al. 2010; Leiserson and Schardl 2010; Xia and Prasanna 2009]. 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, and 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 [Agarwal et al. 2010; Bader and Madduri 2006a; Bader et al. 2005].

Modern GPU architectures provide both of these features, however, serializations from atomic collisions are particularly expensive for GPUs in terms of efficiency and performance. In general, mutual exclusion does not scale to thousands of parallel processing elements. Furthermore, the cost of fine-grained and dynamic serialization between threads within the same GPU SIMD unit is greater than that of more traditional, overlapped SMT threads. For example, all SIMD lanes may be made to wait while the collisions of only a few are serialized.

For machines with wide data parallelism, we argue that software-based prefix sum computations [Blelloch 1990; Hillis and Steele 1986] are a more suitable approach for cooperative 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 [Merrill and Grimshaw 2009] allow to reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph traversal.

Our work as described in this article 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 and, 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 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, and 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 [Agarwal et al. 2010]. Our implementations are publicly available via the B40C Project [Merrill 2011].

2. BACKGROUND

Modern NVIDIA GPU processors consist of tens of multiprocessor cores, each managing 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*. A sequence of kernel invocations is bulk synchronous: each kernel is initially presented with a consistent view of the results from the previous.

The efficiency of GPU architecture stems from the bulk-synchronous and SIMD aspects of the machine model. They facilitate excellent processor utilization on uniform workloads having regularly structured computation. When the computation becomes dynamic and varied, mismatches with the underlying architecture can result in significant performance penalties. For example, performance can be degraded by irregular memory access patterns that cannot be coalesced, or that result in arbitrary memory bank conflicts, control-flow divergences between SIMD warp threads that result in thread serialization, and load imbalances between barrier synchronization points that result in resource underutilization [Owens et al. 2008]. In this work, we make extensive use of local prefix sum computation as a foundation for reorganizing sparse and uneven workloads into dense and uniform ones.

2.1. Breadth-First Search

BFS is a graph traversal algorithm that systematically explores every reachable vertex from a given source, where closer vertices are visited first. Fundamental uses of BFS include: identifying all of the connected components within a graph, finding the diameter of a tree, and testing a graph for bipartiteness [Cormen et al. 2001]. More sophisticated problems incorporating BFS include: identifying the reachable set of heap items during garbage collection [Cheney 1970], belief propagation in statistical inference [Gonzalez et al. 2009], finding community structure in networks [Newman and Girvan 2004], and computing the maximum flow/minimum cut for a given graph [Hussein et al. 2007].

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 $v_0 \dots v_{n-1}$ 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 $A_i = \{v_j | (v_i, v_j) \in E\}$ 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 article, all graph sizes and traversal rates are measured in terms of directed edge counts.

We represent the graph using an adjacency matrix \mathbf{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) format to store the graph in memory consisting of two arrays.

Figure 1 provides a simple example, where 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 .

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad \begin{array}{l} C = [0, 1, 1, 2, 0, 2, 3, 1, 3] \\ R = [0, 2, 4, 7, 9] \end{array}$$

Fig. 1. Example CSR representation: column-indices array C and row-offsets array R comprise the adjacency matrix \mathbf{A} .

ALGORITHM 1 The simple sequential breadth-first search algorithm for marking vertex distances from the source s . Alternatively, a shortest-paths search tree can be constructed by marking i as j 's predecessor in line 11.

Input: Vertex set V , row-offsets array R , column-indices array C , source vertex s

Output: Array $dist[0..n-1]$ with $dist[v]$ holding the distance from s to v

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 V:
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)

```

We store graph edges in the order they are defined. We do not perform any offline pre-processing in order to improve locality of reference, improve load balance, or eliminate sparse memory references. Such strategies might include sorting neighbors within their adjacency lists, sorting vertices into a space-filling curve and remapping their corresponding vertex identifiers, splitting up vertices having large adjacency lists, encoding adjacency row-offset and length information into vertex identifiers, removing duplicate edges, singleton vertices, and self-loops, etc.

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 [Cormen et al. 2001]. As vertices are dequeued, their neighbors are examined, and unvisited neighbors labeled with their distance and/or predecessor and 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, where each depth level is fully explored before the next. Most parallel BFS algorithms are *level-synchronous*, that is, each level may be processed in parallel, so 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, such as, algorithms based on the method of Ullman and Yannakakis [1990]; such alternatives are beyond the scope of this article.

As illustrated in Figure 2, 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 this iteration or, equivalently, the set of all A_i where v_i was marked in

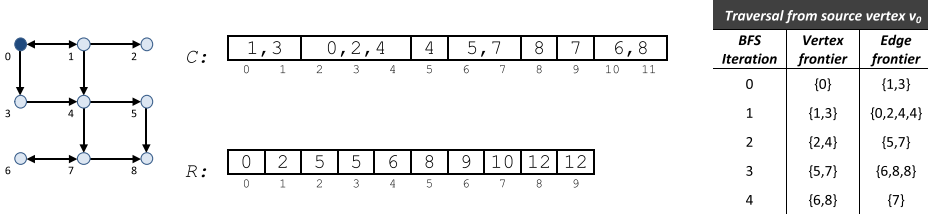


Fig. 2. Example BFS frontier evolution from source vertex v_0 . For each vertex, the distance from the source is the BFS iteration in which it appeared in the vertex frontier.

ALGORITHM 2 A simple quadratic work vertex-oriented BFS parallelization.

Input: Vertex set V , row-offsets array R , column-indices array C , source vertex s
Output: Array $dist[0..n-1]$ with $dist[v]$ holding the distance from s to v

```

1  parallel for (i in V) :
2      dist[i] := ∞
3      dist[s] := 0
4      iteration := 0
5      do :
6          done := true
7          parallel for (i in V) :
8              if (dist[i] == iteration)
9                  done := false
10                 for (offset in R[i] .. R[i+1]-1) :
11                     j := C[offset]
12                     dist[j] = iteration + 1
13             iteration++
14         while (!done)

```

ALGORITHM 3 A linear work BFS parallelization constructed using a global vertex-frontier queue.

Input: Vertex set V , row-offsets array R , column-indices array C , source vertex s , queues

Output: Array $dist[0..n-1]$ with $dist[v]$ holding the distance from s to v

Functions: *LockedEnqueue(val)* safely inserts val at the end of the queue instance

```

1  parallel for (i in V) :
2      dist[i] := ∞
3      dist[s] := 0
4      iteration := 0
5      inQ := {}
6      inQ.LockedEnqueue(s)
7      while (inQ != {}) :
8          outQ := {}
9          parallel for (i in inQ) :
10             for (offset in R[i] .. R[i+1]-1) :
11                 j := C[offset]
12                 if (dist[j] == ∞)
13                     dist[j] = iteration + 1
14                     outQ.LockedEnqueue(j)
15             iteration++
16         inQ := outQ

```

the previous iteration. The vertex frontier, by contrast is the unique subset of such neighbors that are unmarked and which will be labeled and expanded for the next iteration. Each BFS iteration comprises two logical phases that realize these frontiers.

- (1) *Neighbor expansion*. The vertices in the vertex frontier are expanded into an edge frontier of neighboring vertex identifiers.
- (2) *Status lookup and filtering*. The neighbor identifiers in the edge frontier are contracted into a vertex frontier of unvisited vertices.

Quadratic Work 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. As Algorithm 2 illustrates, 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, and each thread's computation is completely

independent from that of other threads. Harish and Narayanan [2007] and Hussein et al. [2007] describe vertex-oriented versions of this method, while Deng et al. [2009] present an edge-oriented implementation.

Hong et al. [2011a] 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 [2009]. Rather than threads, warps are mapped to vertices. During neighbor expansion, the SIMD lanes of an entire warp are used to strip-mine¹ the corresponding adjacency list.

These quadratic methods are isomorphic to iterative SpMV in the algebraic semi ring, where the usual $(+, \times)$ operations are replaced with $(\min, +)$ and thus can also be realized using generic implementations of SpMV [Garland 2008].

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

As described in Algorithm 3, each BFS iteration maps parallel threads to unexplored vertices in the input vertex-frontier queue. Their neighbors are inspected and the unvisited ones placed into the output vertex-frontier queue for the next iteration.

Frontiers may be maintained *in core* or *out of core*. An in-core frontier is processed online and never wholly realized, while 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, since less global data movement is needed because the average vertex frontier is smaller by a factor of \bar{d} (average outdegree).

Research has traditionally focused on two aspects of this scheme: (1) improving hardware utilization via intelligent task scheduling; and (2) designing shared data structures that incur minimal overhead from insertion and removal operations.

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. Logically, the sequential for-loop in line 10 of Algorithm 3 is replaced with a parallel for-loop. 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 [2010] designed a Cilk-based implementation for multiset systems that incorporates a novel multiset data structure for tracking the vertex frontier. Bader and Madduri [2006a] 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 and rely upon run-time scheduling for the management and throttling of edge-processing tasks in-core.

Chhugani et al. [2012] present an implementation for multiset CPUs that avoids atomic read-modify-write updates and carefully arranges queues and status vectors to mitigate TLB misses and cross-socket traffic. Their implementation uses a single, global, best-effort bitmask to reduce the overhead of inspecting a given vertex's visitation status, similar to our own design.

Luo et al. [2010] 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. Their GPU kernels logically correspond to lines 10–13 of Algorithm 3. Threads perform serial

¹Strip-mining entails the sequential processing of parallel batches, where the batch size is typically the number of hardware SIMD vector lanes.

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.

Direction-Optimizing Parallelizations. Beamer et al. [2012] describe a hybrid strategy that combines “top-down” phases (linear work complexity) with “bottom-up” phases (quadratic work complexity). The idea is to first execute a few iterations of a queue-based linear work BFS parallelization for a few iterations—just enough to discover the high-degree vertices. Then the algorithm begins a bottom-up phase that discovers newly reachable vertices at each time step by searching all of the unlabeled vertices, to see whether any have incoming edges from previously discovered vertices. For any given vertex, the algorithm can early-terminate the search of incoming edges as soon as a predecessor from the previous level is found.

The strategy is primarily designed for large graphs having very small diameter, and has been employed with great success for traversing certain small-world datasets with ultrapopular vertices, such as RMAT-generated datasets where this short-circuiting behavior is a likely occurrence [Checconi et al. 2012; Hiragushi and Takahashi 2013; Satish et al. 2012]. The result is that many graph edges are never inspected, which can dramatically lower system bandwidth overheads.² The current highest ranking Graph500 implementation by Checconi and Petrini [2014] employs this method for BlueGene-Q systems.

Distributed Parallelizations. It is often desirable to partition the graph structure amongst multiple processors, particularly for datasets too large to fit within the physical memory of a single machine. Implementations for shared memory SMP platforms have also demonstrated advantages for partitioning the graph amongst the different CPU sockets; a given socket will have higher throughput to the specific memory managed by its local DDR channels [Agarwal et al. 2010].

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. Algorithm 4 describes the general method. Incoming neighbors that are unvisited have their labels marked and their adjacency lists expanded and, as adjacency lists are expanded, neighbors are enqueued to the processor that owns them. The synchronization between BFS levels occurs after the expansion phase.

It is important to note that distributed BFS implementations which 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. [2005] present a variation for BlueGene-L that implements a 2D partitioning strategy for reducing the number of remote peers with which each processor must communicate. Xia and Prasanna [2009] propose a variant for multisocket 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. [2010] describe an implementation for multisocket systems that implements both out-of-core vertex and edge-frontier queues for each socket. As a hybrid of Algorithms 3 and 4, only remote edges are queued out-of-core in global memory, while local edges are inspected and filtered in-core. After a global synchronization, a

²Measuring traversal performance in TE/s may be an inappropriate metric for this type of traversal, as many edges are never actually traversed.

ALGORITHM 4 A linear work, vertex-oriented BFS parallelization for a graph that has been partitioned across multiple processors. The scheme uses a set of distributed edge-frontier queues, one per processor.

Input: Vertex set V , row-offsets array R , column-indices array C , source vertex s , edge-destination frontier queues inQ and $outQ$ for each processor

Output: Array $dist[0..n-1]$ with $dist[v]$ holding the distance from s to v

Functions: *LockedEnqueue(val)* safely inserts val at the end of the queue instance

```

1  parallel for  $i$  in  $V$  :
2       $dist_{proc}[i] := \infty$ 
3       $iteration := 0$ 
4      parallel for ( $proc$  in  $0 .. processors-1$ ) :
5           $inQ_{proc} := \{\}$ 
6           $outQ_{proc} := \{\}$ 
7          if ( $proc == Owner(s)$ )
8               $inQ_{proc}.LockedEnqueue(s)$ 
9               $dist_{proc}[s] := 0$ 
10     do :
11          $done := true$ ;
12         parallel for ( $proc$  in  $0 .. processors-1$ ) :
13             parallel for ( $i$  in  $inQ_{proc}$ ) :
14                 if ( $dist_{proc}[i] == \infty$ )
15                      $done := false$ 
16                      $dist_{proc}[i] := iteration$ 
17                     for ( $offset$  in  $R[i] .. R[i+1]-1$ ) :
18                          $j := C[offset]$ 
19                          $dest := owner(j)$ 
20                          $outQ_{dest}.LockedEnqueue(j)$ 
21         parallel for ( $proc$  in  $0 .. processors-1$ ) :
22              $inQ_{proc} := outQ_{proc}$ 
23              $iteration++$ 
24     while (! $done$ )

```

second phase is performed to filter edges from remote sockets. Their implementation uses a single, global, atomically updated bitmask to reduce the overhead of inspecting a given vertex's visitation status.

Scarpazza et al. [2008] describe a similar hybrid variation for the Cell BE processor architecture. Instead of a separate contraction phase per iteration, processor cores perform edge expansion, exchange, and contraction in batches, and DMA engines are used instead of threads to perform parallel adjacency list expansion. Their implementation requires an offline preprocessing step that sorts vertex identifiers and encodes adjacency lists into segments packaged by the processor core.

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, a *prefix scan* produces an output list where each element is computed to be the reduction of those elements occurring earlier in the input list. A *prefix sum*, on the other hand, connotes a prefix scan with the addition operator. Software-based scan has been popularized as an algorithmic primitive for vector and array processor architectures [Blelloch 1989, 1990; Chatterjee et al. 1990] and as well as for GPUs [Dotsenko et al. 2008; Merrill and Grimshaw 2009; Sengupta et al. 2008].

Prefix sum computation is a particularly useful mechanism for implementing cooperative allocation, that is, when parallel threads must place dynamic data within shared

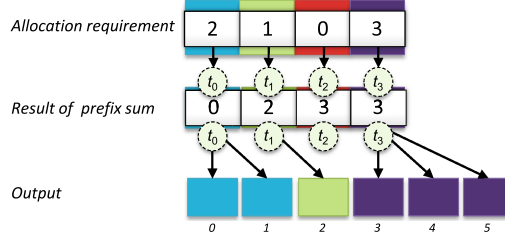


Fig. 3. Example of prefix sum for computing scatter offsets for run-length expansion. Input order is preserved.

data structures such as global queues. Given a list of allocation requirements for each thread, a prefix sum computes the offsets for where each thread should start writing its output elements. Figure 3 illustrates a 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 compute a prefix sum when assembling global edge frontiers from expanded neighbors and when outputting unique unvisited vertices into global vertex frontiers.

3. BENCHMARK SUITE

3.1. Graph Datasets

Our benchmark suite is composed of the 13 graphs listed in Table I. We generate the square and cubic Poisson lattice graph datasets ourselves. The *random.2Mv.128Me* and *rmat.2Mv.128Me* datasets are constructed using GTgraph [Bader and Madduri 2006b]. The *wikipedia-20070206* dataset is from the University of Florida Sparse Matrix Collection [Davis and Hu 2011], and the remaining datasets are from the 10th DIMACS Implementation Challenge [DIMACS 2012].

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, that is, 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. Figure 4 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, Figure 4 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, which 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.

Table I. Suite of Benchmark Graphs

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

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. [2005] demonstrate a variant of this idea for BlueGene-L using their MPI set-union collective.

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 *filtering*. 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.

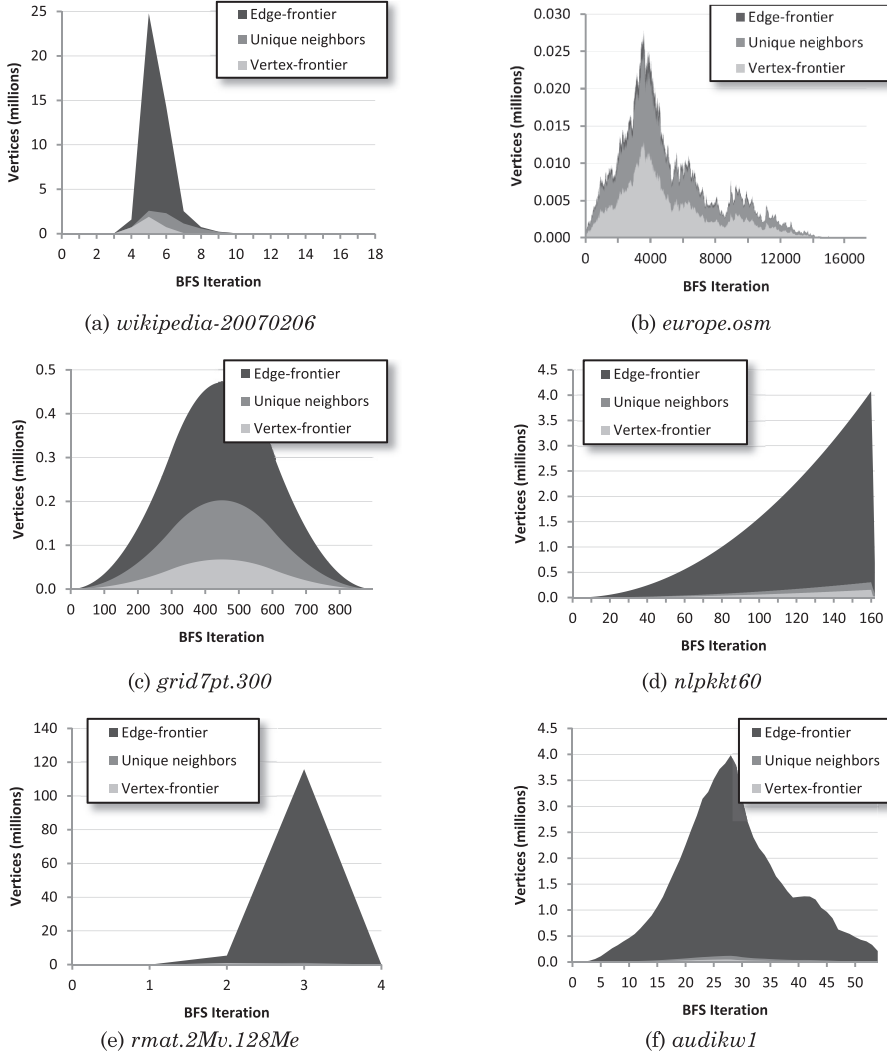


Fig. 4. Sample frontier plots of logical vertex- and edge-frontier sizes during graph traversal.

4.1. Neighbor Gathering in Isolation

This analysis investigates serial and parallel strategies for simply gathering neighbors from adjacency lists. The enlistment of threads for parallel gathering is a form of task scheduling. We evaluate a spectrum of scheduling granularity, from individual tasks (higher scheduling overhead) to blocks of tasks (higher underutilization from partial filling). We show that the serial expansion and warp-centric techniques described by prior work underutilize the GPU for entire genres of sparse graph datasets.

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.³ The gathered neighbors are not enqueued into a global edge frontier.

³For full BFS, we do not perform any preprocessing.

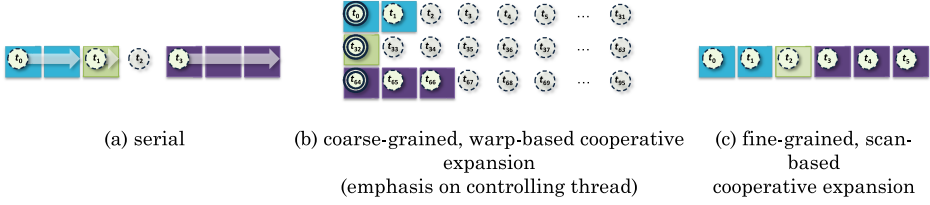


Fig. 5. Alternative strategies for gathering four unexplored adjacency lists having lengths 2, 1, 0, and 3.

Serial Gathering. Each thread obtains its preprocessed row-range bounds and then serially acquires the corresponding neighbors from the column-indices array C . Figure 5(a) illustrates four threads assigned to gather four unexplored adjacency lists having lengths 2, 1, 0, and 3. Graphs having nonuniform 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. Figure 5(b) illustrates three warps gathering the three nonempty adjacency lists in “broadside” parallel fashion, each under the control of a specific thread.

As described in Algorithm 5, 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 this 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 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 underfilled. 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. As presented in Algorithm 6, this approach performs a fine-grained redistribution of gathering workloads using CTA-wide parallel prefix sum. 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 Figure 3 illustrates the cooperative expansion of column-indices offsets into a shared gather vector. As illustrated in Figure 5(c), 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- and warp-based expansion. CTA-wide gathering is similar to warp-based gathering, except that threads vie for control of the entire CTA for strip-mining very large adjacency lists. Then we apply warp-based

ALGORITHM 5 GPU pseudocode for a warp-based, strip-mined neighbor-gathering approach.

Input: Vertex-frontier Q_{vfront} , column-indices array C , and the offset cta_offset for the current tile within Q_{vfront}

Functions: $WarpAny(pred_i)$ returns true if any $pred_i$ is set for any thread t_i within the warp.

```

1  GatherWarp(cta_offset, Q_vfront, C) {
2      volatile shared comm[WARPS][3];
3      {r, r_end} = Q_vfront[cta_offset + thread_id];
4      while (WarpAny(r_end - r)) {
5
6          // vie for control of warp
7          if (r_end - r)
8              comm[warp_id][0] = lane_id;
9
10         // winner describes adjlist
11         if (comm[warp_id][0] == lane_id) {
12             comm[warp_id][1] = r;
13             comm[warp_id][2] = r_end;
14             r = r_end;
15         }
16
17         // strip-mine winner's adjlist
18         r_gather = comm[warp_id][1] + lane_id;
19         r_gather_end = comm[warp_id][2];
20         while (r_gather < r_gather_end) {
21             neighbor = C[r_gather];
22             // do stuff with neighbor...
23             r_gather += WARP_SIZE;
24         }
25     }
26 }
```

ALGORITHM 6 GPU pseudocode for a fine-grained, scan-based neighbor-gathering approach.

Input: Vertex-frontier Q_{vfront} , column-indices array C , and the offset cta_offset for the current tile within Q_{vfront}

Functions: $CtaPrefixSum(val_i)$ performs a CTA-wide prefix sum where each thread t_i is returned the pair $\{\sum_{k=0}^{i-1} val_k, \sum_{k=0}^{CTA_THREADS-1} val_k\}$.
 $CtaBarrier()$ performs a barrier across all threads within the CTA.

```

1  GatherScan(cta_offset, Q_vfront, C) {
2      shared comm[CTA_THREADS];
3      {r, r_end} = Q_vfront[cta_offset + thread_id];
4      // reserve gather offsets
5      {rsv_rank, total} = CtaPrefixSum(r_end - r);
6      // process fine-grained batches of adjlists
7      cta_progress = 0;
8      while ((remain = total - cta_progress) > 0) {
9          // share batch of gather offsets
10         while ((rsv_rank < cta_progress + CTA_THREADS)
11             && (r < r_end))
12             {
13                 comm[rsv_rank - cta_progress] = r;
14                 rsv_rank++;
15                 r++;
16             }
17         CtaBarrier();
18         // gather batch of adjlist(s)
19         if (thread_id < min(remain, CTA_THREADS)) {
20             neighbor = C[comm[thread_id]];
21             // do stuff with neighbor...
22         }
23         cta_progress += CTA_THREADS;
24         CtaBarrier();
25     }
26 }
```

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 scan-based gathering limits imbalance from SIMD lane underutilization. Warp enlistment limits offset-sharing imbalance between threads and, 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 dataset⁴, evaluating these kernels on the logical vertex frontier for every iteration. Figure 6(a) plots the

⁴Throughout our evaluation, the same random source nodes are used for a given dataset on each implementation tested on that dataset (different sets of random source nodes may have been sampled for different datasets).

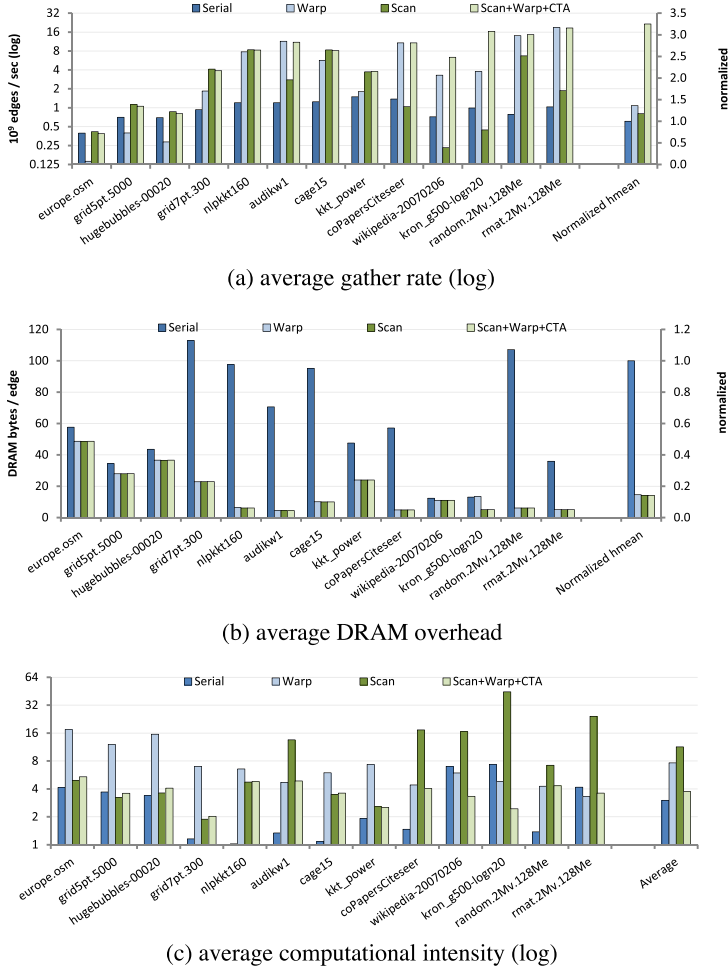


Fig. 6. Neighbor-gathering behavior. Harmonic means are normalized with respect to serial gathering.

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; Figure 6(b) reveals it suffers dramatic overfetch. It plots bytes moved through DRAM per edge, and 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 $\bar{d} \leq 10$. Figure 6(c) reveals 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 and, as a result, a significant number of SIMD lanes go unused during any given cycle.

Figure 6(c) also reveals that scan-based gathering can suffer extreme workload imbalance when only one thread is active within the entire CTA. This phenomenon is reflected in those datasets on the right-hand side 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. Coupling of Gathering and Filtering

Filtering is the other half to neighbor gathering; it entails checking vertex labels to determine which neighbors within the edge frontier have already been visited. This section describes our analyses of filtering workloads, both in isolation and when coupled with neighbor gathering. We reveal that coupling within the same kernel invocation can lead to markedly worse performance than performing them separately.

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 [Agarwal et al. 2010; Scarpazza et al. 2008]. 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 marking.

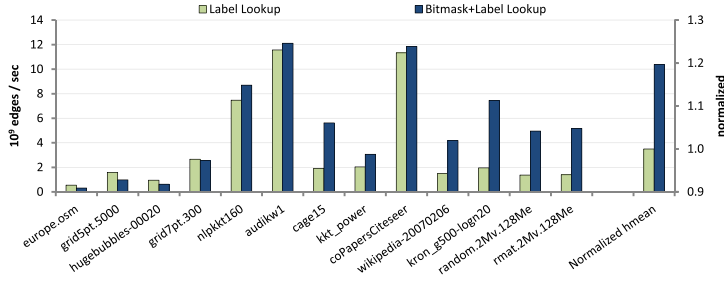
Furthermore, this scheme relies upon capacity and conflict misses to update stale bitmask data within the read-only texture caches. Because the L2 cache is insufficient to contain both the bitmask and the entire dataset, capacity/conflict misses will evict bitmask cache lines, and when they are refetched, whatever was most recently written to global DRAM will replace it.

Similar to the neighbor-gathering analysis, we isolate the filtering workload using a test kernel that consumes the logical edge frontier at each BFS iteration. The filtered neighbors are not enqueued into a global vertex frontier. Figure 7 confirms that the technique can reduce global DRAM overhead and accelerate filtering for GPU architectures that typically provision smaller and more transient last-level caches. The exceptions are those datasets on the left having a hundred or more BFS iterations; the bitmask is less effective for these datasets because texture caches are flushed between kernel invocations. Without coverage, the inspection often requires a second label lookup, which further adds delay to latency-bound BFS iterations. As a result, we skip bitmask lookup for fleeting iterations having edge frontiers smaller than the number of resident threads.

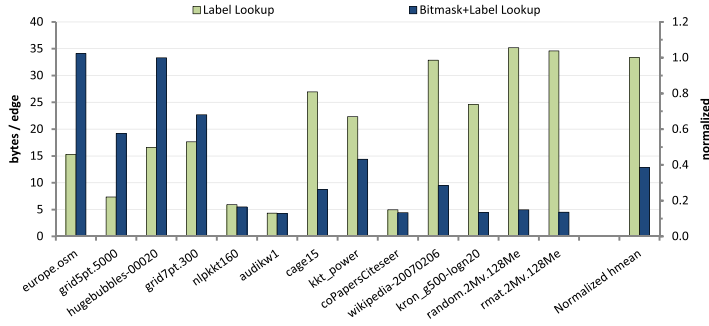
Figure 8 compares the throughputs of lookup versus gathering workloads. We observe that filtering is generally the more expensive of the two, particularly for those datasets on the right-hand side having high average vertex outdegree. The ability for neighbor gathering to coalesce accesses to adjacency lists increases with, whereas accesses for filtering have arbitrary locality.

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).

Figure 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. However, their extra data movement results in net slowdown for the latency-bound datasets on the left-hand side having limited bulk concurrency.



(a) average lookup rate



(b) average DRAM overhead

Fig. 7. Status lookup behavior. Harmonic means are normalized with respect to simple label lookup.

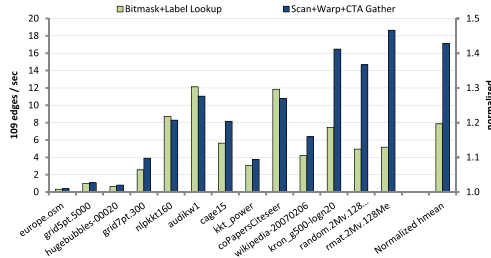


Fig. 8. Comparison of lookup vs. gathering.

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 TLB misses experienced by the neighbor-gathering workload. The column-indices arrays occupy substantial portions of GPU physical memory, and 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.

4.3. Concurrent Discovery

The effectiveness of filtering 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

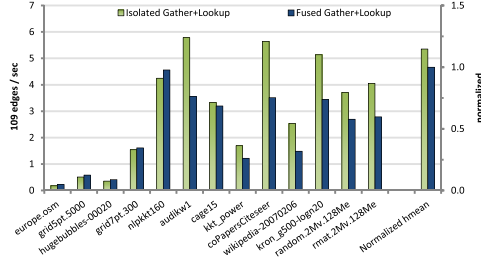
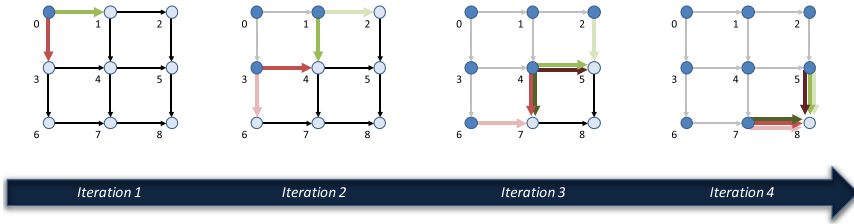


Fig. 9. Comparison of isolated vs. fused lookup and gathering.



BFS Iteration	Actual Vertex-frontier	Actual Edge-frontier
1	0	1,3
2	1,3	2,4,4,6
3	2,4,4,6	5,5,7,5,7,7
4	5,5,7,5,7,7	8,8,8,8,8,8,8

Fig. 10. Example of redundant adjacency list expansion due to concurrent discovery.

discover the same vertices via these duplicates, the corresponding adjacency lists will be expanded multiple times. Without atomic updates to visitation status, we show the SIMD nature of the GPU machine model can introduce a significant amount of redundant work.

Effect on Overall Workload. Prior CPU parallelizations have noted the potential for redundant work, but concluded its manifestation to be negligible [Leiserson and Schardl 2010]. 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 simultaneously inspect same vertex identifier, the SIMD nature of the warp read ensures that all will obtain the same status value and, if unvisited, the adjacency list for this vertex will be expanded for every thread.

Figure 10 demonstrates an acute case of concurrent discovery. In this example, we traverse a small single-source, single-sink lattice using fine-grained cooperative expansion (e.g., Algorithm 6). For each BFS iteration, the cooperative behavior ensures that all neighbors are gathered before any are inspected. No duplicates are culled from the edge frontier because SIMD lookups reveal every neighbor as being unvisited, and actual edge and vertex frontiers diverge from the ideal because no contraction occurs. This is cause for concern; the excess work grows geometrically, only slowing when the frontier exceeds the width of the machine or the graph ceases to expand.

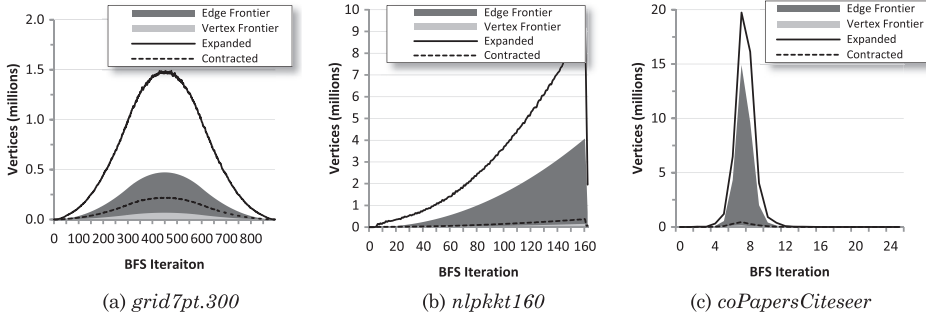


Fig. 11. Actual expanded and contracted queue sizes without local duplicate culling, superimposed over logical frontier sizes.

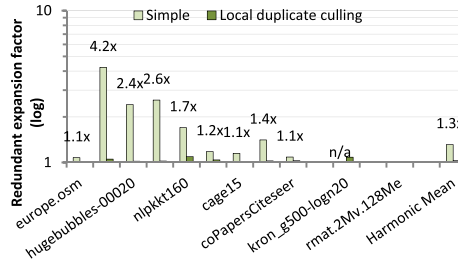


Fig. 12. Redundant work expansion incurred by variants of our two-phase BFS implementation. Unlabeled columns are $< 1.05x$.

We illustrate the effects of redundant expansion on overall workload for several datasets using a simplified version of the *two-phase* BFS implementation described in Section 5. These expansion and contraction kernels make no special effort to curtail concurrent discovery. Figure 11 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 number of edges logically traversed. Figure 12 plots the redundant expansion factors measured for our two-phase implementation, both with and without extra measures to mitigate concurrent discovery. 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 that encodes sorted adjacency lists. The improved locality enables redundant expansion of ultrapopular vertices, ultimately exhausting physical memory when filling the edge queue.

This issue of redundant expansion appears 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 filtering, that is, failure to detect previously visited and duplicate vertex identifiers

ALGORITHM 7 GPU pseudocode for a localized, warp-based duplicate detection heuristic.**Input:** Vertex identifier *neighbor***Output:** True if *neighbor* is a conclusive duplicate within the warp's working set.

```

1  WarpCull(neighbor) {
2      volatile shared scratch[WARPS] [128];
3      hash = neighbor & 127;
4      scratch[warp_id][hash] = neighbor;
5      retrieved = scratch[warp_id][hash];
6      if (retrieved == neighbor) {
7          // vie to be the "unique" item
8          scratch[warp_id][hash] = thread_id;
9          if (scratch[warp_id][hash] != thread_id) {
10             // someone else is unique
11             return true;
12         }
13     }
14     return false;
15 }

```

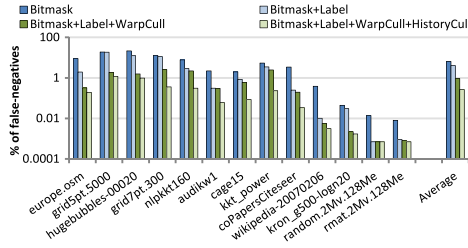


Fig. 13. Percentages of false negatives incurred by status lookup strategies.

within the edge frontier. Atomic read-modify-write updates to visitation status yield zero false negatives, so, as alternatives, we introduce two localized mechanisms for reducing false negatives: (1) *warp culling* and (2) *history culling*.

Warp Culling. Algorithm 7 describes a simple heuristic that 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 this 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.

Figure 13 illustrates the progressive application of lookup mechanisms. The bitmask heuristic alone incurs an average false-negative rate of 6.4% across our benchmark suite, but the addition of label lookup (which makes filtering 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 to reduce the overall redundant expansion factor to less than 1.05x for every graph in our benchmark suite.

5. SINGLE-GPU PARALLELIZATIONS

A complete solution must couple expansion and contraction activities. In this section, we evaluate the design space of coupling alternatives.

- (1) *Expand-contract*. A single kernel consumes the current vertex frontier and produces the vertex frontier for the next BFS iteration.
- (2) *Contract-expand*. This is the converse. A single kernel contracts the current edge frontier, expanding unvisited vertices into the edge frontier for the next iteration.
- (3) *Two-phase*. A given BFS iteration is processed by two kernels that separately implement out-of-core expansion and contraction.
- (4) *Hybrid*. This implementation invokes the *contract-expand* kernel for small, fleeting BFS iterations, otherwise the two-phase kernels.

We describe and evaluate BFS kernels for each strategy. We show the hybrid approach to be on par with or better than the other three for every dataset in our benchmark suite.

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 4.2. 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.

A CTA performs the following steps when processing a tile of input from the incoming vertex-frontier queue.

- (1) Threads perform local warp culling and history culling to determine whether their dequeued vertex is a duplicate.
- (2) If still valid, the corresponding row range is loaded from the row-offsets array R .
- (3) Threads perform coarse-grained, CTA-based neighbor gathering. Large adjacency lists are cooperatively strip-mined from the column-indices array C at the full width of the CTA. These strips of neighbors are filtered in core and the unvisited vertices are enqueued into the output queue as described shortly.
- (4) Threads perform fine-grained, scan-based neighbor gathering. These batches of neighbors are filtered and enqueued into the output queue as described shortly.

For each strip or batch of gathered neighbors:

- (i) threads perform status lookup to invalidate the vast majority of previously visited and duplicate neighbors;
- (ii) threads with a valid neighbor n_i update the corresponding label;
- (iii) threads then perform a CTA-wide prefix sum where each contributes a 1 if n_i is valid, 0 otherwise, this provides each thread with the scatter offset for n_i and the total count of all valid neighbors;

- (iv) $thread_0$ obtains the base enqueue offset for valid neighbors by performing an atomic add operation on a global queue counter using the total valid count, and the returned value is shared to all other threads in the CTA; and
- (v) finally, all valid n_i are written to the global output queue. The enqueue index for n_i is the sum of the base enqueue offset and the scatter offset.

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- 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 both 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.

A CTA performs the following steps when processing a tile of input from the incoming edge-frontier queue.

- (1) Threads progressively test their neighbor vertex identifier n_i for validity using: (i) status lookup; (ii) warp-based duplicate culling; and (iii) history-based duplicate culling.
- (2) Threads update labels for valid n_i and obtain the corresponding row ranges from R .
- (3) Threads then perform two concurrent CTA-wide prefix sums: the first for computing enqueue offsets for coarse-grained warp and CTA neighbor gathering, and the second for fine-grained scan-based gathering. $|A_i|$ is contributed to the first prefix sum if greater than $WARP_SIZE$, otherwise to the second.
- (4) $Thread_0$ obtains a base enqueue offset for valid neighbors within the entire tile by performing an atomic add operation on a global queue counter using the combined totals of the two prefix sums. The returned value is then shared to all other threads in the CTA.
- (5) Threads then perform coarse-grained CTA and warp-based gathering. When a thread commandeers its CTA or warp, it also communicates the base scatter offset for n_i to its peers. After gathering neighbors from C , enlisted threads enqueue them to the global output queue. The enqueue index for each thread is the sum of the base enqueue offset, the shared scatter offset, and thread rank.
- (6) Finally, threads perform fine-grained scan-based gathering. This procedure is a variant of Algorithm 6 with the prefix sum being hoisted out and performed earlier in step 4. After gathering packed neighbors from C , threads enqueue them to the global output. The enqueue index is the sum of the base enqueue offset, the coarse-grained total, the CTA progress, and thread rank.

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 in the next iteration. The bitmask, label, and row-offset traffic remain the same as for expand-contract.

Despite a much larger queuing workload, the contract-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 and, by operating on the larger edge frontier, the contract-expand kernel also enjoys better bulk concurrency in which fewer resident CTAs sit idle.

5.3. Two-Phase (Out-of-Core Vertex and Edge Queues)

Our two-phase implementation isolates the expansion and contraction workloads into separate kernels. Our microbenchmark analyses suggest this design for better overall bulk throughput. 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 described before, 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 then 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 ones. If the edge queue for a given BFS iteration contains more vertex identifiers than resident threads, we invoke the two-phase implementation for this 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, Figure 14 plots average traversal throughput across 100 randomly sourced traversals of each dataset. 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 version 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, but 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 overfetch traffic for those graphs having small \bar{d} ; they use large memory transactions to retrieve small adjacency lists.

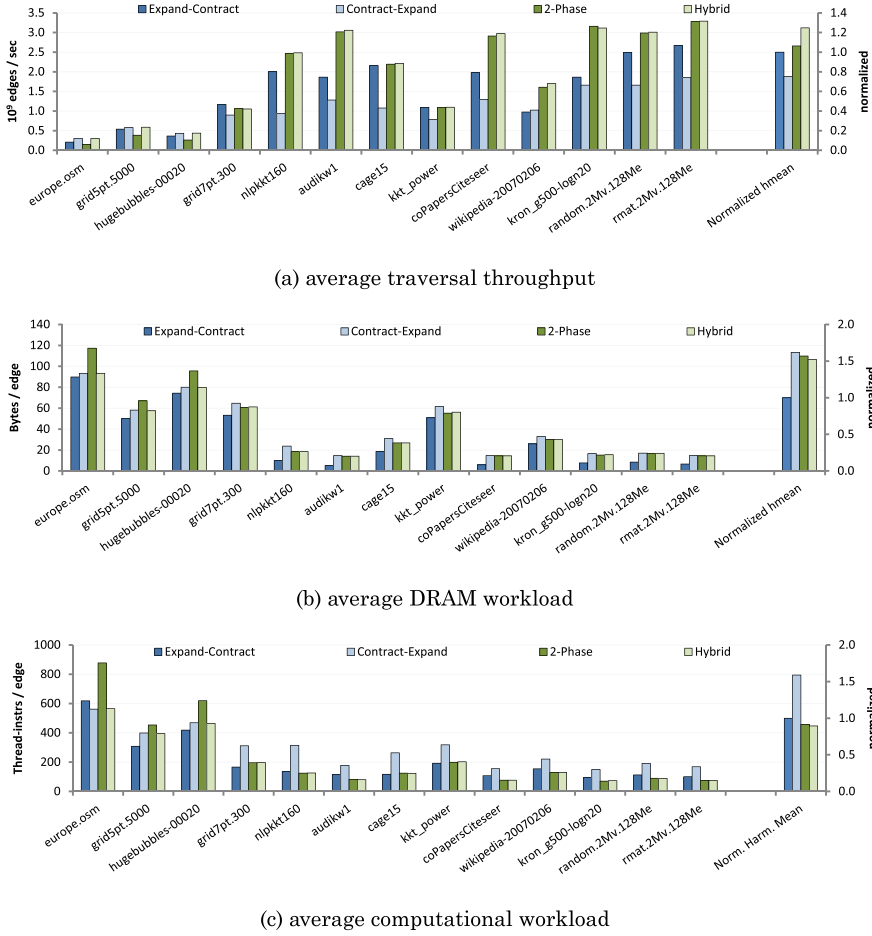


Fig. 14. BFS traversal performance and workloads. Harmonic means are normalized with respect to the expand-contract implementation.

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. Figure 15 illustrates using a sample traversal of *wikipedia-20070206*. We observe a correlation between large contraction workloads during iterations 4–6 and significantly elevated dynamic thread instruction counts. 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 II compares the traversal throughput of the classic sequential BFS algorithm with both our hybrid strategy as well as with prior single-socket parallelizations for both GPU and multicore CPU architectures.

Distance vs. Predecessor Labeling. Although the sequential algorithm has identical performance for both distance- and predecessor-labeling versions, the throughput of

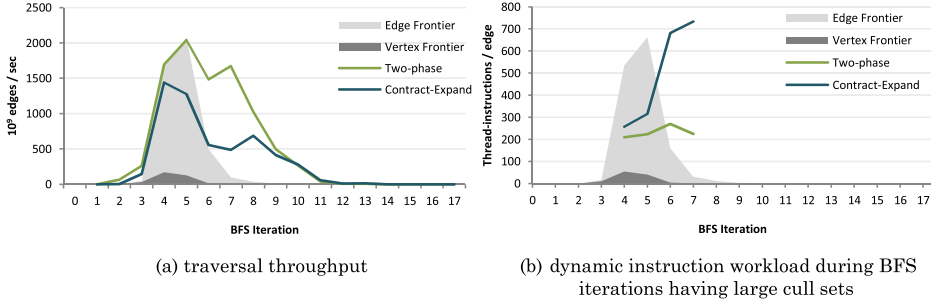


Fig. 15. Sample *wikipedia-20070206* traversal behavior. Plots are superimposed over the shape of the logical edge and vertex frontiers.

Table II. Average Single-Socket Graph Traversal Rates (10^9 TE/s)

Graph Dataset	Sequential CPU*	Parallel CPU (work-efficient)		GPU (work-inefficient)	GPU (work-efficient hybrid strategy)			
		Distance [29] * (speedup vs seq. CPU)	Predecessor [3] **		Distance*** (speedup vs seq. CPU)		Predecessor*** (speedup vs seq. CPU)	
europe.osm	0.03	0.10 (3.6x)		0.00014 (0.0x)	0.31 (11x)		0.31 (11x)	
grid5pt.5000	0.08	0.24 (3.0x)		0.00078 (0.0x)	0.6 (7.4x)		0.57 (7.0x)	
hugebubbles-00020	0.03	0.12 (4.0x)		0.00061 (0.0x)	0.43 (15x)		0.42 (15x)	
grid7pt.300	0.04	0.18 (4.9x)		0.012 (0.3x)	1.1 (29x)		0.97 (26x)	
nlpkt160	0.26	1.18 (4.6x)		0.21 (0.8x)	2.5 (9.7x)		2.1 (8.2x)	
audikw1	0.65	2.32 (3.6x)		1.2 (1.8x)	3.0 (4.6x)		2.5 (3.9x)	
cage15	0.12	0.46 (3.7x)		0.50 (4.0x)	2.2 (18x)		1.9 (15x)	
kkt_power	0.05	0.22 (4.7x)		0.18 (3.9x)	1.1 (23x)		1.0 (21x)	
coPapersCiteseer	0.50	2.04 (4.1x)		2.2 (4.4x)	3.0 (6.0x)		2.5 (5.0x)	
wikipedia-20070206	0.06	0.37 (5.8x)		0.39 (6.1x)	1.6 (25x)		1.4 (22x)	
kron_g500-logn20	0.24	1.15 (4.8x)		1.5 (6.1x)	3.1 (13x)		2.5 (10x)	
random.2Mv.128Me	0.10	0.31 (3.0x)	0.50	1.2 (11.6x)	3.0 (29x)		2.4 (23x)	
rmat.2Mv.128Me	0.15	0.70 (4.7x)	0.70	1.3 (8.6x)	3.3 (22x)		2.6 (17x)	

* Intel 4-core 3.4 GHz Core i7 2600K with two-way hyperthreading per core

** Intel 8-core 2.7 GHz Xeon X5570 with two-way hyperthreading per core

*** NVIDIA 14-core 1.15 GHz Tesla C2050

parallel predecessor labeling is typically lower because parent identifiers must be communicated between processing elements (whereas the distance level is implicit for a given iteration). The hybrid performance disparity between the two is largely dependent upon average vertex degree \bar{d} , where smaller \bar{d} incurs larger DRAM over fetch that reduces the relative significance of added parent queue traffic. For example, the performance impact of exchanging parent vertices is negligible for *europe.osm*, yet 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. [2011a], 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. [2010]. In the absence of their hand-tuned implementation, we compared ours against

the specific collections of 6-pt lattice datasets⁵ and DIMACS road network datasets⁶ 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 so, rather than constructing our own CPU parallelizations, we choose to compare against prior implementations on a per-socket basis (the construction of high-performance CPU parallelizations is outside the scope of this work). Table II cites the single-socket CPU traversal rates of the recent Cilk-based parallel BFS implementation by Leiserson and Schardl [2010] on our Core i7 Sandybridge, for which our hybrid method achieves a 3x harmonic mean speedup on a NVIDIA C2050⁷. We also cite the performance reported by Agarwal et al. [2010] for datasets common to our experimental corpus, for which we achieve an average speedup of 5.2x.

In general, our GPU performance goal was to achieve a 4–8x speedup with respect to the sequential CPU implementation. As listed in Table II, our C2050 traversal rates meet or exceed this factor for all benchmark datasets (a majority exceed 12x speedup). At the extreme, our average *wikipedia-20070206* traversal rates outperform those of the sequential CPU version by 25x, that is, three to six CPU equivalents. We also note that our methods perform equally well for both large- and small-diameter graphs alike, and our hybrid strategy provides traversal speedups of an order of magnitude for both the *europa.osm* and *kron.g500-logn20* datasets.

Performance vs. Connectivity. Figure 17 presents C2050 traversal performance for synthetic uniform-random and RMAT of datasets having different average degree \bar{d} . Each plot is averaged from 100 randomly sourced traversals. As expected, we achieve our maximum traversal rates of 3.5B and 3.6B TE/s for the most connected ($\bar{d} = 256$) uniform-random and RMAT datasets, respectively. The minimum rates plotted are 710M and 982M TE/s for uniform-random and RMAT datasets having $\bar{d} = 8$ and 256M edges. Performance incurs a dropoff at $n = 8$ million vertices when the bitmask exceeds the 768KB L2 cache size.

Performance Portability. Figure 16 presents traversal performance for our experimental corpus on the last three generations of NVIDIA Tesla GPUs: *GT200* (Tesla C1060), *Fermi* (Tesla C2050), and *Kepler* (Tesla K40C). We observe positive performance scaling for all datasets from one architecture to the next: an average of 2.8x from GT200 to Fermi, and an average of 1.3x from Fermi to Kepler. We also note that the datasets towards the right exhibit better performance scaling for each new processor. Successive NVIDIA architectures have increased core counts, thus can take advantage of the shallower diameters and greater intrinsic parallelism within these 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 6.6GB/s. Under the assumption that GPUs send and receive equal amounts of traffic, the rate

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

⁶New York, Florida, USA-East, and USA-West datasets from the 9th DIMACS Challenge corpus [DIMACS 2011].

⁷On a normalized per-socket basis, we found the execution of this implementation on our CPU hardware to exceed the performance of Chhugani et al. [2012].

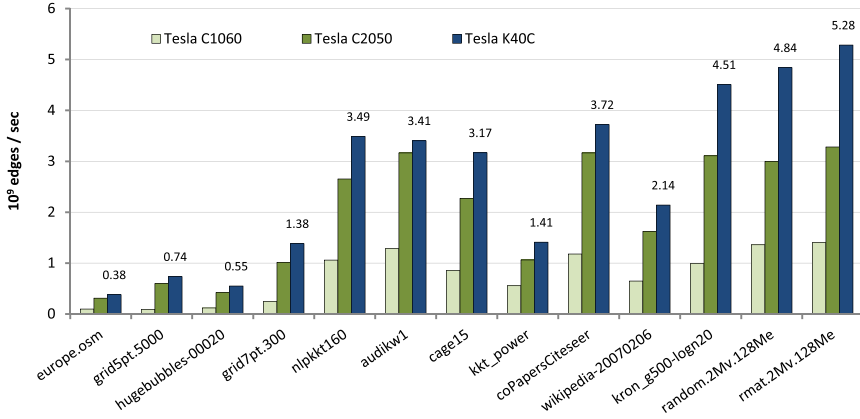


Fig. 16. Performance portability across generations of NVIDIA microarchitecture.

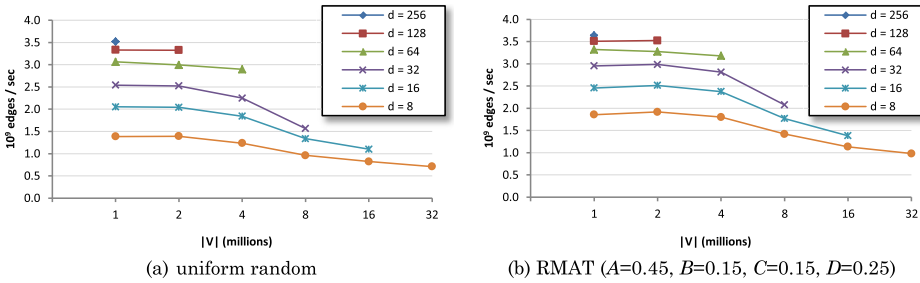


Fig. 17. NVIDIA Tesla C2050 traversal throughput.

at which each GPU can be fed with remote work is conservatively bound by 825×10^6 neighbors/s, where neighbors are 4-byte identifiers. This rate is halved for predecessor-labeling variants.

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 (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, which is particularly useful for graph datasets having concentrations of popular vertices. For example, RMAT datasets encode the most popular vertices with the largest adjacency lists near the beginning of R and C . Alternatives that divide such data into contiguous slabs can be detrimental for small systems, since: (a) an equal share of vertices would overburden the first GPU with an abundance of edges; or (b) an equal share of edges will leave the first GPU underutilized because it owns fewer vertices most of which are likely to be filtered remotely. However, this method of partitioning progressively loses any inherent locality as the number of GPUs increases.

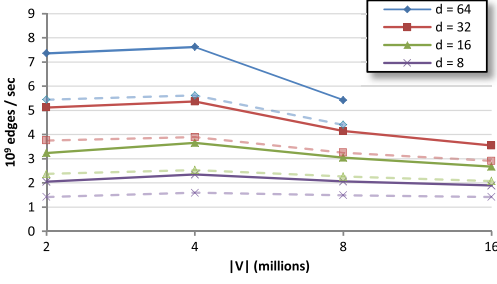


Fig. 18. Multi-GPU sensitivity to graph size and average outdegree \bar{d} for uniform-random graphs using four C2050 processors. Dashed lines indicate predecessor labeling variants.

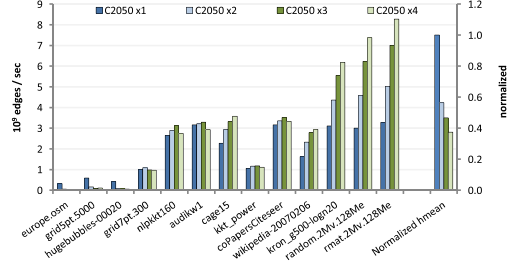


Fig. 19. Average multi-GPU traversal rates. Harmonic means are normalized with respect to the single-GPU configuration.

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 Q_{vertex_i} into an edge queue Q_{edge_i} .
- (2) Invoke a fused filter+partition operation for each GPU_i that sorts neighbors within Q_{edge_i} by ownership into $pbins$. Vertex identifiers undergo opportunistic local duplicate culling and bitmask filtering during the partitioning process. This partitioning implementation is analogous to the three-kernel radix-sorting pass [Merrill and Grimshaw 2011].
- (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 Q_{vertex_i} for the next BFS iteration.

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

6.2. Evaluation

Figure 19 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, allowing to circumvent the PCI-e cap of 825×10^6 edges/s per GPU. With four GPUs, we demonstrate traversal rates of 7.4 and 8.3 billion edges/s 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 *rmatrix.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.

Figure 18 further illustrates the impact of opportunistic duplicate culling for uniform-random graphs up to 500M edges and varying outdegree \bar{d} , where increasing \bar{d} yields significantly better performance. Other than a slight performance drop at $n = 8$ million vertices when the bitmask exceeds the L2 cache size, graph size has little impact on traversal throughput.

To our knowledge, these are the fastest traversal rates demonstrated by a single-node machine. The work by Agarwal et al. [2010] is representative of the state-of-the-art in CPU parallelizations, demonstrating up to 1.3 billion edges/s for both uniform-random and RMAT datasets using four 8-core Intel Nehalem-based XEON CPUs. However, we note that the host memory on such systems can further accommodate datasets having tens of billions of edges.

7. CONCLUSION

This article 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 [Hong et al. 2011a, 2011b], they suffer 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 emphasizes several general themes for implementing sparse and dynamic problems for the GPU machine model. The computation of a 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 the 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, in that the data movement from reorganization may be insignificant in comparison to the actual overfetch 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.

ACKNOWLEDGMENTS

We would like to thank NVIDIA for supplying additional GPUs and Sungpack Hong of Stanford University for providing us with the code for their BFS implementation.

REFERENCES

- Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 2010. Scalable graph exploration on multicore processors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. 1–11.

- David A. Bader, Guojing Cong, and John Feo. 2005. On the architectural requirements for efficient execution of graph algorithms. In *Proceedings of the International Conference on Parallel Processing (ICPP'05)*. 547–556.
- David A. Bader and Kamesh Madduri. 2006a. Designing multithreaded algorithms for breadth-first search and ST-connectivity on the Cray MTA-2. In *Proceedings of the International Conference on Parallel Processing (ICPP'06)*. 523–530.
- David A. Bader and Kamesh Madduri. 2006b. GTgraph: A synthetic graph generator suite. <http://www.cse.psu.edu/~madduri/software/GTgraph>.
- Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, 12:1–12:10.
- Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM Press, New York, 18:1–18:11.
- Guy E. Blelloch. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput.* 38, 11, 1526–1538.
- Guy E. Blelloch. 1990. Prefix sums and their applications. In *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Francisco, 35–60.
- Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. 1990. Scan primitives for vector computers. In *Proceedings of the ACM/IEEE Conference on Supercomputing (Supercomputing'90)*. IEEE Computer Society Press, 666–675.
- Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. 44–54.
- Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. 2012. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, 13:1–13:12.
- Fabio Checconi and Fabrizio Petrini. 2014. Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*. 425–434.
- C. J. Cheney. 1970. A nonrecursive list compacting algorithm. *Comm. ACM* 13, 11, 677–678.
- Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep K. Dubey. 2012. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Proceedings of the 26th IEEE International Parallel Distributed Processing Symposium (IPDPS'12)*. 2012. 378–389.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*, 2nd Ed. MIT Press.
- Tim Davis and Yifan Hu. University of florida sparse matrix collection. <http://www.cise.ufl.edu/research/sparse/matrices/>.
- Yangdong Deng, Bo David Wang, and Shuai Mu. 2009. Taming irregular EDA applications on GPUs. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'09)*. ACM Press, New York, 539–546.
- DIMACS. 2011. 9th DIMACS implementation challenge. <http://www.dis.uniroma1.it/~challenge9/download.shtml>.
- DIMACS. 2012. 10th DIMACS implementation challenge. <http://www.cc.gatech.edu/dimacs10/index.shtml>.
- Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS'08)*. ACM Press, New York, 205–213.
- Michael Garland. 2008. Sparse matrix computations on manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference (DAC'08)*. ACM Press, New York, 2–6.
- Joseph Gonzalez, Yucheng Low, and Carlos Guestrin. 2009. Residual splash for optimally parallelizing belief propagation. In *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS'09)*. 177–184.
- Graph List. 2011. The graph 500 list. <http://www.graph500.org/>.
- Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC'07)*. 197–208.
- W. Daniel Hillis and Guy L. Steele. 1986. Data parallel algorithms. *Comm. ACM* 29, 12, 1170–1183.
- Takaaki Hiragushi and Daisuke Takahashi. 2013. Efficient hybrid breadth-first search on GPUs. In *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel*

- Processing (ICA3PP'13)*, Rocco Aversa, Joanna Kołodziej, Jun Zhang, Flora Amato, and Giancarlo Fortino, Eds. Lecture Notes in Computer Science, vol. 8286. Springer, 40–50.
- Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011a. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM Press, New York, 267–276.
- Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011b. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 78–88.
- Mohamed Hussein, Amitabh Varshney, and Larry Davis. 2007. On implementing graph cuts on CUDA. In *Proceedings of the 1st Workshop on General Purpose Processing on Graphics Processing Units (GPGPU'07)*.
- Charles E. Leiserson and Tao B. Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*. ACM Press, New York, 303–314.
- Lijuan Luo, Martin Wong, and Wen-Mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference (DAC'10)*. ACM Press, New York, 52–55.
- Duane Merrill. 2011. Back40 computing: Fast and efficient software primitives for GPU computing. <http://code.google.com/p/back40computing/>.
- Duane Merrill and Andrew Grimshaw. 2009. Parallel scan for stream architectures. Tech. rep. CS2009-14, Department of Computer Science, University of Virginia.
- Duane Merrill and Andrew Grimshaw. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Process. Lett.* 21, 2, 245–272.
- Mark Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Phys. Rev. E* 69, 2.
- John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. 2008. GPU computing. *Proc. IEEE* 96, 5, 879–899.
- Nadathur Satish, Changkyu Kim, Jatin Chhugani, and Pradeep Dubey. 2012. Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, 14:1–14:11.
- Daniele P. Scarpazza, Oreste Villa, and Fabrizio Petrini. 2008. Efficient breadth-first search on the cell/be processor. *IEEE Trans. Parallel Distrib. Syst.* 19, 10, 1381–1395.
- Shubhabrata Sengupta, Mark Harris, and Michael Garland. 2008. Efficient parallel scan algorithms for GPUs. <https://research.nvidia.com/sites/default/files/publications/nvr-2008-003.pdf>.
- John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Tech. rep. IMPACT-12-01, Center for Reliable and High-Performance Computing. <http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf>.
- Jeffery Ullman and Mihalis Yannakakis. 1990. High-probability parallel transitive closure algorithms. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'90)*. 200–209.
- Yinglong Xia and Viktor K. Prasanna. 2009. Topologically adaptive parallel breadth-first search on multicore processors. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing and Systems (PDCS'09)*.
- Andy Yoo, Edmond Chow, Keith Henderson, William Mclendon, Bruce Hendrickson, and Umit Catalyurek. 2005. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'05)*. 25.

Received June 2013; revised November 2014; accepted November 2014