High Performance and Scalable GPU Graph Traversal

Technical Report CS-2011-05
Department of Computer Science, University of Virginia
Aug, 2011

Duane Merrill
Department of Computer Science
University of Virginia

Michael Garland
NVIDIA Research

Andrew Grimshaw
Department of Computer Science
University of Virginia

ABSTRACT

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

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

1. INTRODUCTION

Algorithms for analyzing sparse relationships represented as graphs provide crucial tools in many computational fields ranging from genomics to electronic design automation to social network analysis. In this paper, we explore the parallelization of one fundamental graph algorithm on GPUs: breadth-first search (BFS). BFS is a common building block for more sophisticated graph algorithms, yet is simple enough that we can analyze its behavior in depth. It is also used as a core computational kernel in a number of benchmark suites, including Parboil [1], Rodinia [2], and the emerging Graph500 supercomputer benchmark [3].

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

Despite their high computational throughput, GPUs might appear poorly suited for sparse graph computation. In particular, BFS is representative of a class of algorithms for which it is hard to obtain significantly better performance from parallelization. Optimizing memory usage is non-trivial because memory access patterns are determined by the structure of the input graph. Parallelization further introduces concerns of contention, load imbalance, and underutilization on multithreaded architectures [4–6]. 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 to hide memory latency. The second is fine-grained synchronization, specifically atomic read-modify-write operations. Such algorithms have incorporated atomic mechanisms for coordinating the dynamic placement of data into shared data structures and for arbitrating contended status updates. [5], [7], [8]

Modern GPU architectures provide both. However, atomic serialization is particularly expensive for GPUs in terms of efficiency and performance. In general, mutual exclusion does not scale to thousands of threads. Furthermore, the occurrence of fine-grained and dynamic serialization within the SIMD width is much costlier than between overlapped SMT threads. For example, all SIMD lanes are penalized when only a few experience dynamic serialization.

For machines with wide data parallelism, we argue that prefix sum is often a more suitable approach to data placement [9], [10]. 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 [11] allow us to reorganize sparse and uneven workloads into dense and uniform ones in all phases of graph traversal.

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

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

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

High performance. We demonstrate that our methods deliver excellent performance on a diverse body of real-world graphs. Our implementation achieves traversal rates in excess of 3.3 billion and 8.3 billion traversed edges per second (TE/s) for single and quad-GPU configurations, respectively. To put these numbers 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 [5].

$$\boldsymbol{A} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad \begin{aligned} C &= [0,1,1,2,0,2,3,1,3] \\ R &= [0,2,4,7,9] \end{aligned}$$

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

2. BACKGROUND

Modern NVIDIA GPU processors consist of tens of processor cores, each of which manages on the order of a thousand hardware-scheduled threads. Each processor 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 arbitrarily-bad 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 [12]. In this work, we make extensive use of local prefix sum as a foundation for reorganizing sparse and uneven workloads into dense and uniform ones.

2.1 Breadth First Search

We consider graphs of the form $G = (V, E)$ with a set V of n vertices and a set E of m directed edges. Given a source vertex v_s, our goal is to traverse the vertices of G in breadth-first order starting at v_s. Each newly-discovered vertex v_i will be labeled by (a) its distance d_i from v_s and/or (b) the predecessor vertex p_i immediately preceding it on the shortest path to v_s.

Fundamental uses of BFS include: identifying all of the connected components within a graph; finding the diameter of tree; and testing a graph for bipartiteness [13]. More sophisticated problems incorporating BFS include: identifying the reachable set of heap items during garbage collection [14]; belief propagation in statistical inference [15], finding community structure in networks [16], and computing the maximum-flow/minimum-cut for a given graph [17].

For simplicity, we identify the vertices $v_0 .. v_{n\text{-}1}$ using integer indices. The pair (v_i, v_j) indicates a directed edge in the graph from $v_i \rightarrow v_j$, and the adjacency list $A_i = \{v_j \mid (v_i, v_j) \in E\}$ is the set of neighboring vertices adjacent from vertex v_i. We treat undirected graphs as symmetric directed graphs containing both (v_i, v_j) and (v_j, v_i) for each undirected edge. In this paper, all graph sizes and traversal rates are measured in terms of directed edge counts.

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 represent the graph using an adjacency matrix **A**, whose rows are the adjacency lists A_i. The number of edges within sparse graphs is typically only a constant factor larger than n. We use the well-known compressed sparse row (CSR) sparse matrix format to store the graph in memory consisting of two arrays. As illustrated in Fig. 1, the column-indices array C is formed from the set of the adjacency lists concatenated into a single array of m integers. The row-offsets R array contains $n + 1$ integers, and entry $R[i]$ is the index in C of the adjacency list A_i.

We store graphs in the order they are defined. We do not perform any offline preprocessing 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 describes the standard sequential BFS method for circulating the vertices of the input graph through a FIFO queue that is initialized with v_s [13]. As vertices are dequeued, their neighbors are examined. Unvisited neighbors are labeled with their distance and/or predecessor and are enqueued for later processing. This algorithm performs linear $O(m+n)$ work since each vertex is labeled exactly once and each edge is traversed exactly once.

2.2 Parallel Breadth-First Search

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

Structurally different methods may be more suitable for graphs with very large diameters, e.g., algorithms based on

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*

```
parallel for (i in V) :
  dist[i] := ∞
dist[s] := 0
iteration := 0
do :
  done := true
  parallel for (i in V) :
    if (dist[i] == iteration)
      done := false
      for (offset in R[i] .. R[i+1]-1) :
        j := C[offset]
        dist[j] = iteration + 1
  iteration++
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 method of Ullman and Yannakakis [18]. Such alternatives are beyond the scope of this paper.

Each iteration of a level-synchronous method identifies both an edge and vertex *frontier*. The edge-frontier is the set of all edges to be traversed during that iteration or, equivalently, the set of all A_i where v_i was marked in the previous iteration. The vertex-frontier is the unique subset of such neighbors that are unmarked and which will be labeled and expanded for the next iteration. Each iteration logically expands vertices into an edge-frontier and then contracts them to a vertex-frontier.

Quadratic parallelizations. The simplest parallel BFS algorithms inspect every edge or, at a minimum, every vertex during every iteration. These methods perform a quadratic amount of work. A vertex v_j is marked when a task discovers an edge $v_i \rightarrow v_j$ where v_i has been marked and v_j has not. 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. Each thread's computation is completely independent from that of other threads. Harish *et al.* [19] and Hussein *et al.* [17] describe vertex-oriented versions of this method. Deng *et al.* present an edge-oriented implementation [20].

Hong *et al.* [21] 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 [22]. Rather than threads, warps are mapped to vertices. During neighbor expansion, the SIMD lanes of an entire warp are used to strip-mine[1] the corresponding adjacency list.

These quadratic methods are isomorphic to iterative SpMV in the algebraic semi-ring where the usual (+, ×) operations are replaced with (min, +), and thus can also be realized using generic implementations of SpMV [23].

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

Frontiers may be maintained *in core* or *out of core*. An in-core frontier is processed online and never wholly realized. On the other hand, a frontier that is managed out-of-core is fully produced in off-chip memory for consumption by the next BFS iteration after a global synchronization step.

Implementations typically prefer to manage the vertex-frontier out-of-core. Less global data movement is needed because the average vertex-frontier is smaller by a factor of $\bar{d}$ (average out-degree). As described in Algorithm 3, each BFS iteration maps tasks to unexplored vertices in the input vertex-frontier queue. Their neighbors are inspected and the unvisited ones are placed into the output vertex-frontier queue for the next iteration.

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.

In recent BFS research, Leiserson and Schardl [6] designed an implementation for multi-socket CPU systems that incorporates a novel multi-set data structure for tracking the vertex-frontier. They implement concurrent neighbor

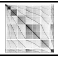
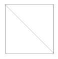
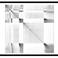
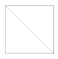
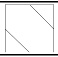
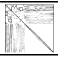
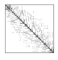
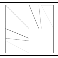
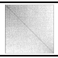
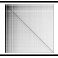
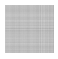
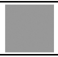
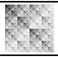
[1] Strip mining entails the sequential processing of parallel batches, where the batch size is typically the number of hardware SIMD vector lanes.

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*, 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

```
parallel for i in V :
  $dist_{proc}[i]$ := ∞
iteration := 0
parallel for (proc in 0 .. processors-1) :
  $inQ_{proc}$ := {}
  $outQ_{proc}$ := {}
  if (proc == Owner(s))
    $inQ_{proc}$.LockedEnqueue(s)
    $dist_{proc}[s]$ := 0
do :
  done := true;
  parallel for (proc in 0 .. processors-1) :
    parallel for (i in $inQ_{proc}$) :
      if ($dist_{proc}[i]$ == ∞)
        done := false
        $dist_{proc}[i]$ := iteration
        for (offset in R[i] .. R[i+1]-1) :
          j := C[offset]
          dest := owner(j)
          $outQ_{dest}$.LockedEnqueue(j)
  parallel for (proc in 0 .. processors-1) :
    $inQ_{proc}$ := $outQ_{proc}$
  iteration++
while (!done)
```

Table 1. 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

inspection, using the Cilk++ runtime to manage the edge-processing tasks in-core.

For the Cray MTA-2, Bader and Madduri [7] describe an implementation using the hardware's full-empty bits for efficient queuing into an out-of-core vertex frontier. They also perform adjacency-list expansion in parallel, relying on the parallelizing compiler and fine-grained thread-scheduling hardware to manage edge-processing tasks in-core.

Luo *et al*. [24] 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 adjacency list expansion and use an upward propagation tree of child-queue structures in an effort to mitigate the contention overhead on any given atomically-incremented queue pointer.

Distributed parallelizations. It is often desirable to partition the graph structure amongst multiple processors, particularly for datasets too large to fit within the physical memory of a single machine. Even for shared-memory SMP platforms, recent research has shown it to be advantageous to partition the graph amongst the different CPU sockets; a given socket will have higher throughput to the specific memory managed by its local DDR channels [5].

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

Yoo *et al.* [25] present a variation for BlueGene/L that implements a two-dimensional partitioning strategy for reducing the number of remote peers each processor must communicate with. Xia and Prasanna [4] propose a variant for multi-socket nodes that provisions more out-of-core edge-frontier queues than active threads, reducing the contention at any given queue and flexibly lowering barrier overhead.

Agarwal *et al.* [5] describe a two-phase implementation for multi-socket systems that implements both out-of-core vertex and edge-frontier queues for each socket. As a hybrid of Algorithm 3 and Algorithm 4, only remote edges are queued out-of-core. Edges that are local are inspected and filtered in-core. After a global synchronization, a second phase is performed to filter edges from remote sockets. Their implementation uses a single, global, atomically-updated

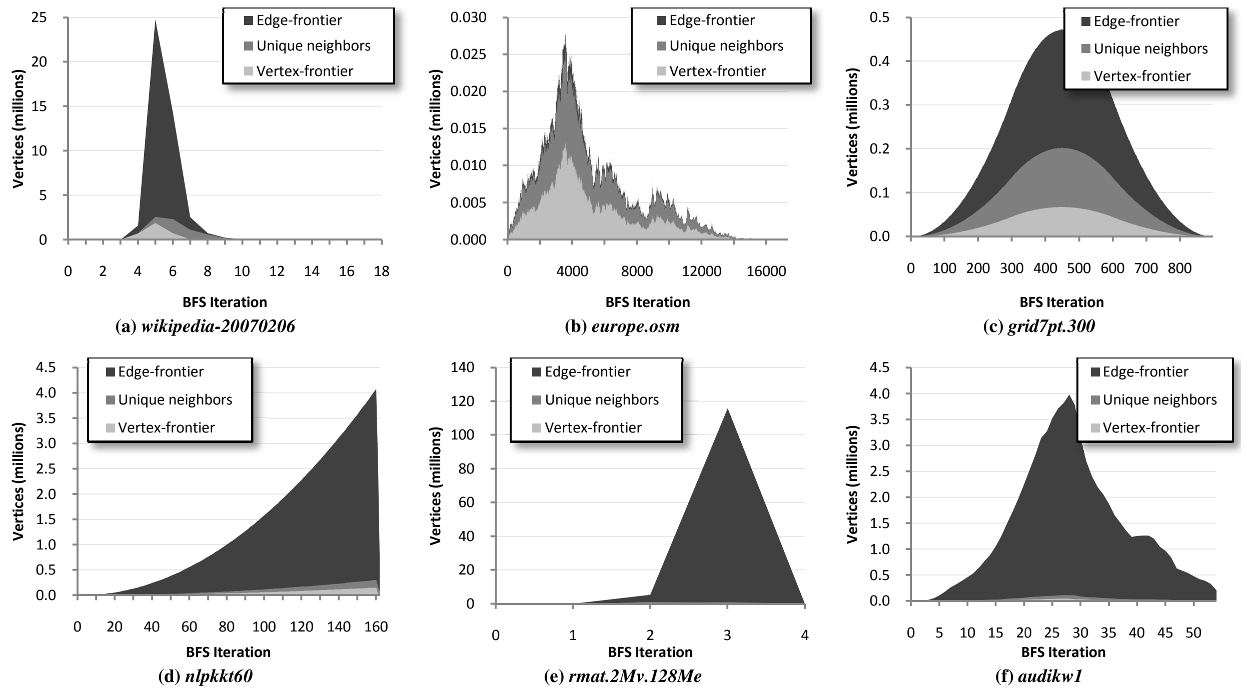


Fig. 2. Sample frontier plots of logical vertex and edge-frontier sizes during graph traversal.

bitmask to reduce the overhead of inspecting a given vertex's visitation status.

Scarpazza *et al.* [26] describe a similar hybrid variation for the Cell BE processor architecture. Instead of separate contraction phase per iteration, processor cores perform edge expansion, exchange, and contraction in batches. DMA engines are used instead of threads to perform parallel adjacency list expansion. Their implementation requires an offline preprocessing step that sorts and encodes adjacency lists into segments packaged by 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.

3. BENCHMARK SUITE

3.1 Graph Datasets

Our benchmark suite is composed of the thirteen graphs listed in Table 1. We generate the square and cubic Poisson lattice graph datasets ourselves. The *random.2Mv.128Me* and *rmat.2Mv.128Me* datasets are constructed using GTgraph [27]. The *wikipedia-20070206* dataset is from the University of Florida Sparse Matrix Collection [28]. The remaining datasets are from the 10th DIMACS Implementation Challenge [29].

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

3.2 Logical Frontier Plots

Although our sparsity plots reveal a diversity of locality, they provide little intuition as to how traversal will unfold. Fig. 2 presents sample *frontier plots* of logical edge and vertex-frontier sizes as functions of BFS iteration. Such plots help visualize workload expansion and contraction, both within and between iterations. The ideal numbers of neighbors expanded and vertices labeled per iteration are constant properties of the given dataset and starting vertex.

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

Finally, Fig. 2 also plots the duplicate-free subset of the edge-frontier. We observe that a simple duplicate-removal pass can perform much of the contraction work from edge-frontier down to vertex-frontier. This has important implications for distributed BFS. The amount of network traffic can be significantly reduced by first removing duplicates from the expansion of remote neighbors.

We note the direct application of this technique does not scale linearly with processors. As p increases, the number of available duplicates in a given partition correspondingly decreases. In the extreme where $p = m$, each processor owns only one edge and there are no duplicates to be locally culled. For large p, such decoupled duplicate-removal techniques should be pushed into the hierarchical interconnect. Yoo *et al.* demonstrate a variant of this idea for BlueGene/L using their MPI set-union collective [25].

Algorithm 5. GPU pseudo-code 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.

```
GatherWarp(cta_offset, $Q_{vfront}$, C) {
  volatile shared comm[WARPS][3];
  {r, r_end} = $Q_{vfront}$[cta_offset + thread_id];
  while (WarpAny(r_end – r)) {

    // vie for control of warp
    if (r_end – r)
      comm[warp_id][0] = lane_id;

    // winner describes adjlist
    if (comm[warp_id][0] == lane_id) {
      comm[warp_id][1] = r;
      comm[warp_id][2] = r_end;
      r = r_end;
    }

    // strip-mine winner's adjlist
    r_gather = comm[warp_id][1] + lane_id;
    r_gather_end = comm[warp_id][2];
    while (r_gather < r_gather_end) {
      volatile neighbor = C[r_gather];
      r_gather += WARP_SIZE;
    }
  }
}
```

Algorithm 6. GPU pseudo-code 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.

```
GatherScan(cta_offset, $Q_{vfront}$, C) {
  shared comm[CTA_THREADS];
  {r, r_end} = $Q_{vfront}$[cta_offset + thread_id];
  // reserve gather offsets
  {rsv_rank, total} = CtaPrefixSum(r_end – r);
  // process fine-grained batches of adjlists
  cta_progress = 0;
  while ((remain = total - cta_progress) > 0) {
    // share batch of gather offsets
    while((rsv_rank < cta_progress + CTA_THREADS)
        && (r < r_end))
    {
        comm[rsv_rank – cta_progress] = r;
        rsv_rank++;
        r++;
    }
    CtaBarrier();
    // gather batch of adjlist(s)
    if (thread_id < Min(remain, CTA_THREADS) {
      volatile neighbor = C[comm[thread_id]];
    }
    cta_progress += CTA_THREADS;
    CtaBarrier();
  }
}
```

4. MICRO-BENCHMARK ANALYSES

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

4.1 Isolated Neighbor Gathering

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

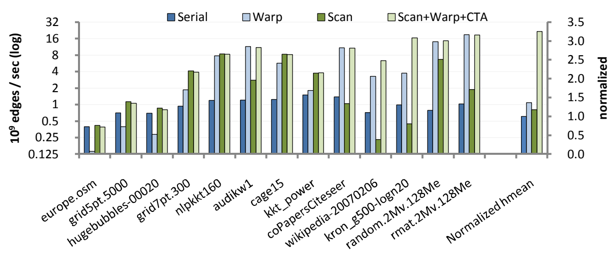
Serial-gathering. Each thread obtains its preprocessed row-range bounds and then serially acquires the corresponding neighbors from the column-indices array C.

Coarse-grained, warp-based gathering. Threads enlist the entire warp to assist in gathering. As described in Algorithm 5, each thread attempts to vie for control of its warp by writing its thread-identifier into a single word shared by all threads of that warp. Only one write will succeed, thus determining which is allowed to subsequently enlist the warp as a whole to read its corresponding neighbors. This process repeats for every warp until its threads have all had their adjacent neighbors gathered.

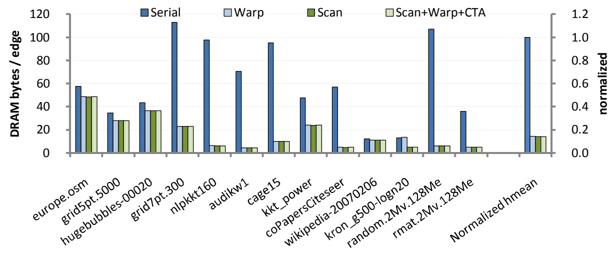
Fine-grained, scan-based gathering. Algorithm 6 illustrates fine-grained gathering using CTA-wide parallel prefix sum. Threads use the reservation from the prefix sum to perfectly pack segments of gather offsets for the neighbors within their adjacency lists into a single buffer that is shared by the entire CTA. When this buffer is full, the entire CTA can then gather the referenced neighbors from the column-indices array C. Perfect packing ensures that no SIMD lanes are unutilized during global reads from C. This process repeats until all threads have had their adjacent neighbors gathered.

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

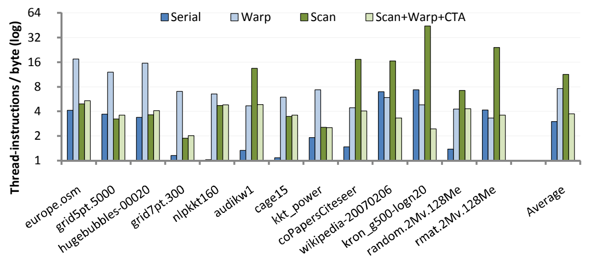
Scan+warp+CTA gathering. We can mitigate this imbalance by supplementing fine-grained *scan-based* expansion with coarser CTA-based and warp-based expansion. We first apply a CTA-wide version of *warp-based* gathering. This allows threads with very large adjacency lists



(a) Average gather rate (log)

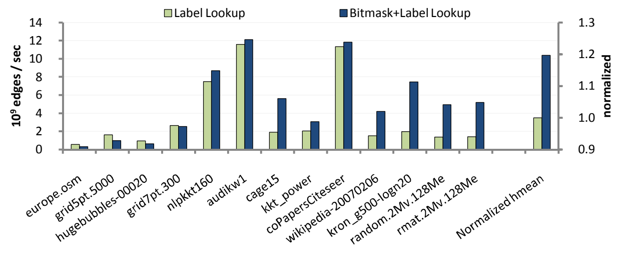


(b) Average DRAM overhead

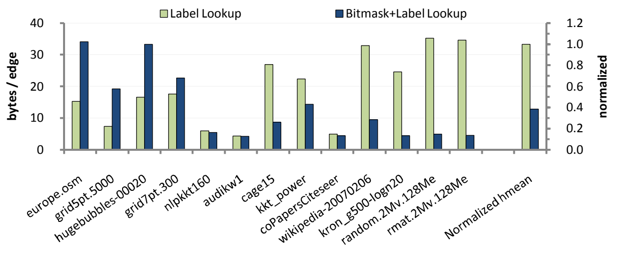


(c) Average computational intensity (log)

Fig. 4. Neighbor-gathering behavior. Harmonic means are normalized with respect to serial-gathering.



(a) Average lookup rate



(b) Average DRAM overhead

Fig. 3 Status-lookup behavior. Harmonic means are normalized with respect to simple label-lookup.

to vie for control of the entire CTA, the winner broadcasting its row-range to all threads. Any large adjacency lists are strip-mined using the width of the entire CTA. Then we apply *warp-based* gathering to acquire portions of adjacency lists greater than or equal to the warp width. Finally we perform *scan-based* gathering to acquire the remaining "loose ends".

This hybrid strategy limits all forms of load imbalance from adjacency list expansion. Fine-grained scan-based distribution limits imbalance from SIMD lane underutilization. Warp enlistment limits offset-sharing imbalance between threads. CTA enlistment limits imbalance between warps. And finally, any imbalance between CTAs can be limited by oversubscribing GPU cores with an abundance of CTAs and/or implementing coarse-grained tile-stealing mechanisms for CTAs to dequeue tiles[2] at their own rate.

Analysis. We performed 100 randomly-sourced traversals of each dataset, evaluating these kernels on the logical vertex-frontier for every iteration. Fig. 4a plots the average edge-processing throughputs for each strategy in log-scale. The datasets are ordered from left-to-right by decreasing average search depth.

The *serial* approach performs poorly for the majority of datasets. Fig. 4b reveals it suffers from dramatic over-fetch. It plots bytes moved through DRAM per edge. The arbitrary references from each thread within the warp result in terrible coalescing for SIMD load instructions.

The *warp-based* approach performs poorly for the graphs on the left-hand side having $\bar{d} \leq 10$. Fig. 4c reveals that it is computationally inefficient for these datasets. It plots a log scale of computational intensity, the ratio of thread-instructions versus bytes moved through DRAM. The average adjacency lists for these graphs are much smaller than the number of threads per warp. As a result, a significant number of SIMD lanes go unused during any given cycle.

Fig. 4c also reveals that that *scan-based* gathering can suffer from extreme workload imbalance when only one thread is active within the entire CTA. This phenomenon is reflected in the datasets on the right-hand size having skewed degree distributions. The load imbalance from expanding large adjacency lists leads to increased instruction counts and corresponding performance degradation.

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

4.2 Coupling of Gathering and Lookup

Status-lookup is the other half to neighbor-gathering; it entails determining which neighbors within the edge-frontier have already been visited. This section describes our analyses of status-lookup 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.

[2] We term *tile* to describe a block of input data that a CTA is designed to process to completion before terminating or obtaining more work.

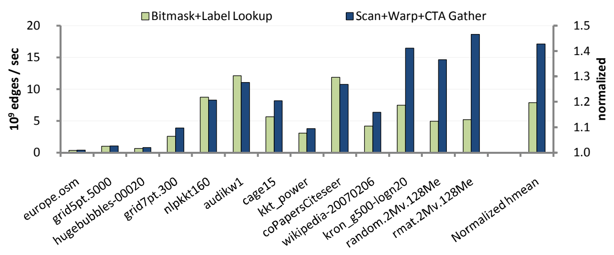
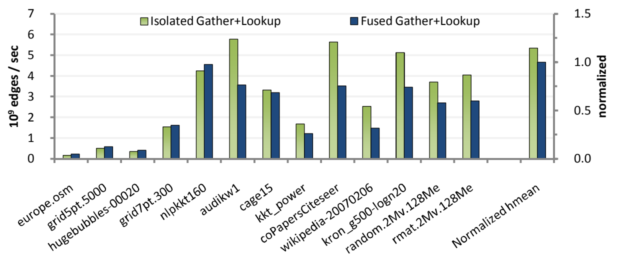
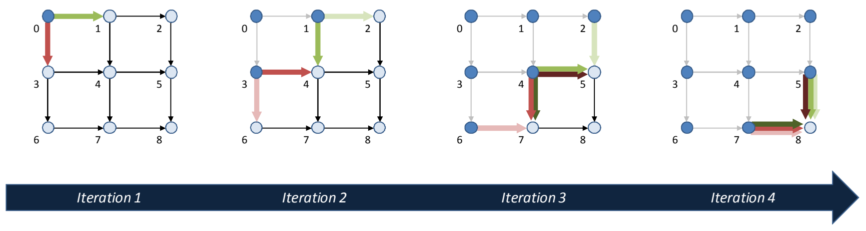


Fig. 5. Comparison of lookup vs. gathering.





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

Fig. 7. Example of redundant adjacency list expansion due to concurrent discovery

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 [5], [26]. 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 perform a second read to check the corresponding label to ensure the vertex is safe for marking. This scheme relies upon capacity and conflict misses to update stale bitmask data within the read-only texture caches.

Similar to the neighbor-gathering analysis, we isolate the status-lookup workload using a test-kernel that consumes the logical edge-frontier at each BFS iteration. Despite having much smaller and more transient last-level caches, Fig. 3 confirms the technique can reduce global DRAM overhead and accelerate status-lookup for GPU architectures as well. The exceptions are the 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.

Fig. 5 compares the throughputs of lookup versus gathering workloads. We observe that status-lookup is generally the more expensive of the two. This is particularly true for the datasets on the right-hand side having high average vertex out-degree. The ability for neighbor-gathering to coalesce accesses to adjacency lists increases with $\bar{d}$, whereas accesses for status-lookup 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).

Fig. 6 compares this fused kernel with the aggregate throughput of the isolated gathering and lookup workloads performed separately. Despite the additional data movement, the separate kernels outperform the fused kernel for the majority of the benchmarks. Their extra data movement results in net slowdown, however, for the latency-bound datasets on the left-hand side having limited bulk concurrency. The implication is that fused approaches are preferable for fleeting BFS iterations having edge-frontiers smaller than the number of resident threads.

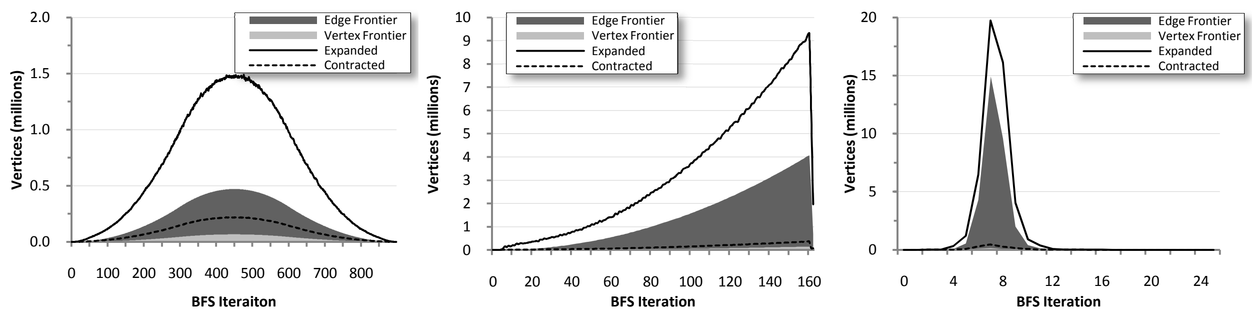
The fused kernel likely suffers from TLB misses experienced by the neighbor-gathering workload. The column-indices arrays occupy substantial portions of GPU physical memory. Sparse gathers from them are apt to cause TLB misses. The fusion of these two workloads inherits the worst aspects of both: TLB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

Fig. 6. Comparison of isolated vs. fused lookup and gathering.

4.3 Concurrent Discovery

Duplicate vertex identifiers within the edge-frontier are representative of different edges incident to the same vertex. This can pose a problem for implementations that allow the benign race condition. Adjacency lists will be expanded multiple times when multiple threads concurrently discover the same vertices via these duplicates. 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 [6]. Concurrent discovery on



(a) ***grid7pt.300*** **(b)** ***nlpkkt160*** **(c)** ***coPapersCiteseer***

Fig. 8. Actual expanded and contracted queue sizes without local duplicate culling, superimposed over logical frontier sizes.

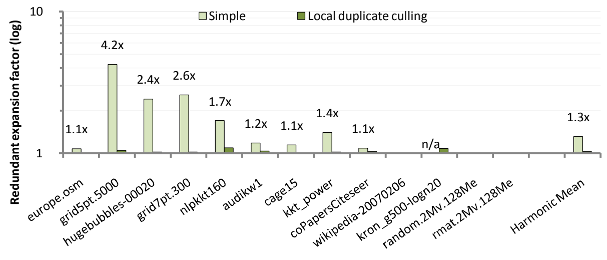


Fig. 9 Redundant work expansion incurred by variants of our *two-phase* BFS implementation. Unlabeled columns are < 1.05x.

CPU platforms is rare due to a combination of relatively low parallelism (~8 hardware threads) and coherent L1 caches that provide only a small window of opportunity around status-inspections that are immediately followed by status updates.

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

Fig. 7 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. The actual edge and vertex-frontiers diverge from 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.

We measure the effects of redundant expansion upon overall workload 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. For several sample traversals, Fig. 8 illustrates compounded redundancy by plotting 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.

Algorithm 7. GPU pseudo-code 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.

```
WarpCull(neighbor) {
  volatile shared scratch[WARPS][128];
  hash = neighbor & 127;
  scratch[warp_id][hash] = neighbor;
  retrieved = scratch[warp_id][hash];
  if (retrieved == neighbor) {
    // vie to be the "unique" item
    scratch[warp_id][hash] = thread_id;
    if (scratch[warp_id][hash] != thread_id) {
      // someone else is unique
      return true;
    }
  }
  return false;
}
```

We define the *redundant expansion factor* as the ratio of neighbors actually enqueued versus the number of edges logically traversed. Fig. 9 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 which encodes sorted adjacency lists. The improved locality enables the redundant expansion of ultra-popular vertices, ultimately exhausting physical memory when filling the edge queue.

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

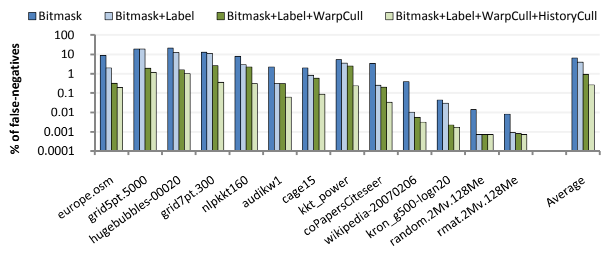


Fig. 10 Percentages of false-negatives incurred by status-lookup strategies.

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

Warp culling. Algorithm 7 describes this heuristic for preventing 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 then write their thread-identifier into the same hash location. Only one write will succeed. Threads that subsequently retrieve a different thread-identifier can safely classify their neighbors as duplicates to be culled.

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

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

Fig. 10 illustrates the progressive application of lookup mechanisms. The bitmask heuristic alone incurs an average false-negative rate of 6.4% across our benchmark suite. The addition of label-lookup (which makes status-lookup safe) improves this to 4.0%. Without further measure, the compounding nature of redundant expansion allows even small percentages to accrue sizeable amounts of extra work. For example, a false-negative rate of 3.5% for traversing *kkt_power* results in a 40% redundant expansion overhead.

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

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*. 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 *expand-contract* kernel 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 if 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 below.
4. Threads perform fine-grained, scan-based neighbor-gathering. These batches of neighbors are filtered and enqueued into the output queue as described below.

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. The returned value is shared to all other threads in the CTA.

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.

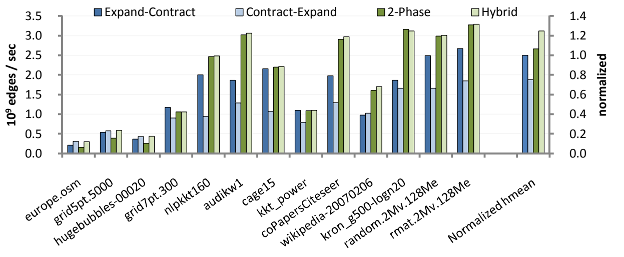
Each CTA performs two or more local prefix-sums per tile. One is used for allocating room for gather offsets during scan-based gathering. We also need prefix sums to compute global enqueue offsets for every strip or batch of gathered neighbors. Although GPU cores can efficiently overlap concurrent prefix 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)

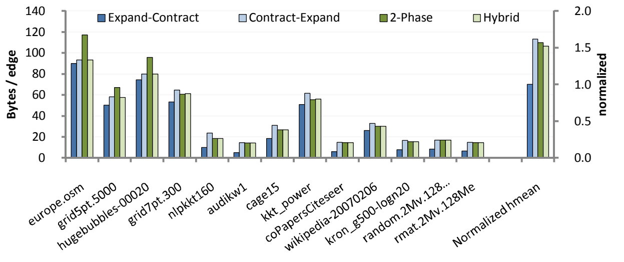
Our *contract-expand* kernel filters previously-visited and duplicate neighbors from the current edge queue. The adjacency lists of the surviving vertices are then expanded and copied out into the edge queue for the next iteration.

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

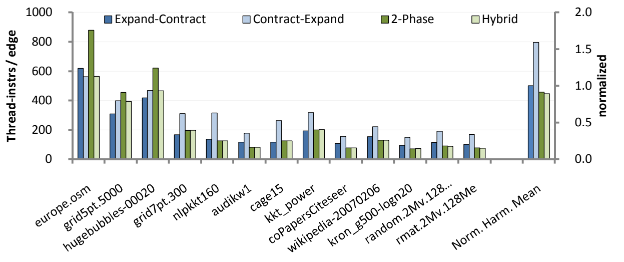
1. Threads progressively test their neighbor vertex identifier n_i for validity using (i) status-lookup; (ii) warp-based duplicate culling; and (iii) history-based duplicate culling.
2. Threads update labels for valid n_i and obtain the corresponding row-ranges from R.
3. Threads then perform two concurrent CTA-wide prefix sums: the first for computing enqueue offsets for coarse-grained warp and CTA neighbor-gathering, 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 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.



(a) Average traversal throughput



(b) Average DRAM workload



(c) Average computational workload

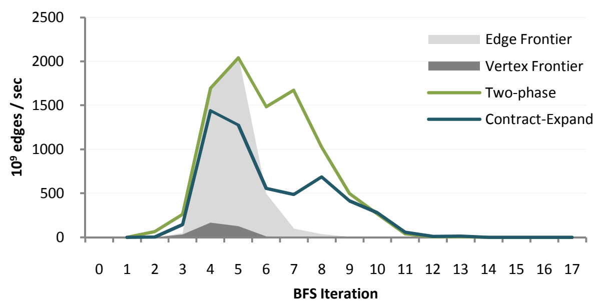
Fig. 11 BFS traversal performance and workloads. Harmonic means are normalized with respect to the *expand-contract* implementation.

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

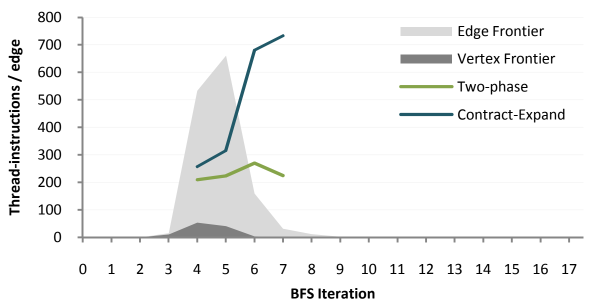
Despite a much larger queuing workload, the *contract-expand* strategy is often better suited for processing small, fleeting BFS iterations. It incurs lower latency because CTAs only perform local two prefix sums per block. We overlap these prefix-sums to further reduce latency. By operating on the larger 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 micro-benchmark analyses suggest this design for better overall bulk throughput. The expansion kernel employs the



(a) traversal throughput



(b) Dynamic instruction workload during BFS iterations having large cull-sets

Fig. 12. Sample *wikipedia-20070206* traversal behavior. Plots are superimposed over the shape of the logical edge and vertex-frontiers.

scan+warp+cta gathering strategy to obtain the neighbors of vertices from the input vertex queue. As with the *contract-expand* implementation above, it performs two overlapped local prefix-sums to compute scatter offsets for the expanded neighbors into the global edge queue.

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

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

5.4 Hybrid

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

5.5 Evaluation

Our performance analyses are constructed from 100 randomly-sourced traversals of each dataset. Fig. 11 plots average traversal throughput. As anticipated, the *contract-expand* approach excels at traversing the latency-bound datasets on the left and the *two-phase* implementation efficiently leverages the bulk-concurrency exposed by the datasets on the right. Although the *expand-contract* approach is serviceable, the *hybrid* approach meets or exceeds its performance for every dataset.

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

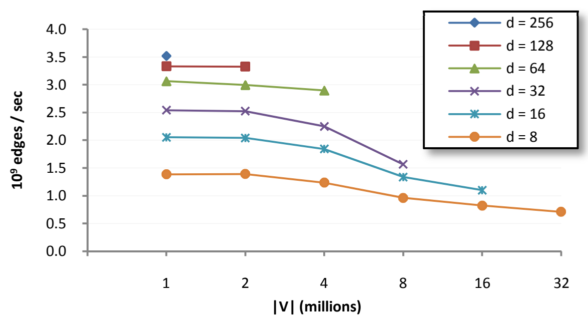
Graph Dataset	CPU Sequential†	CPU Parallel	NVIDIA Tesla C2050 (*hybrid*)			
			Label Distance		Label Predecessor	
	10^9 TE/s	10^9 TE/s	10^9 TE/s	Speedup	10^9 TE/s	Speedup
europe.osm	0.029		0.31	11x	0.31	11x
grid5pt.5000	0.081		0.60	7.3x	0.57	7.0x
hugebubbles-00020	0.029		0.43	15x	0.42	15x
grid7pt.300	0.038	0.12††	1.1	28x	0.97	26x
nlpkkt160	0.26	0.47††	2.5	9.6x	2.1	8.3x
audikw1	0.65		3.0	4.6x	2.5	4.0x
cage15	0.13	0.23††	2.2	18x	1.9	15x
kkt_power	0.047	0.11††	1.1	23x	1.0	21x
coPapersCiteseer	0.50		3.0	5.9x	2.5	5.0x
wikipedia-20070206	0.065	0.19††	1.6	25x	1.4	22x
kron_g500-logn20	0.24		3.1	13x	2.5	11x
random.2Mv.128Me	0.10	0.50†††	3.0	29x	2.4	23x
rmat.2Mv.128Me	0.15	0.70†††	3.3	22x	2.6	18x

Table 2. Single-socket performance comparison. GPU speedup is in regard to sequential CPU performance. †3.4GHz Core i7 2600K. †† 2.5 GHz Core i7 4-core, distance-labeling [6]. ††† 2.7 GHz Xeon X5570 8-core, predecessor labeling [5].

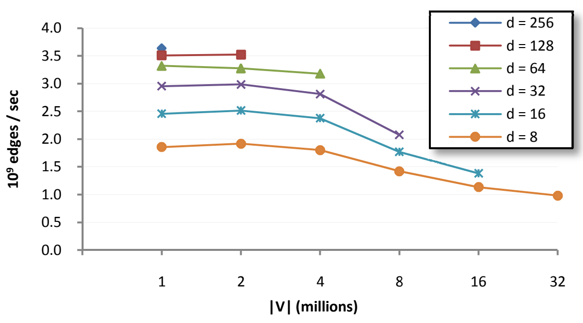
The *contract-expand* implementation performs poorly for graphs having large $\bar{d}$. This behavior is related to a lack of explicit workload compaction before neighbor gathering. Fig. 12 illustrates this using a sample traversal of *wikipedia-20070206*. We observe a correlation between large contraction 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.

Table 2 compares *hybrid* traversal performance for distance and predecessor labeling variants. The performance difference between variants is largely dependent upon $\bar{d}$. Smaller $\bar{d}$ incurs larger DRAM over-fetch which reduces the relative significance of added parent queue traffic. For example, the performance impact of exchanging parent vertices is negligible for *europe.osm*, yet is as high as 19% for *rmat.2Mv.128Me*.

When contrasting CPU and GPU architectures, we attempt to hedge in favor of CPU performance. We compare our GPU traversal performance with the sequential method and then assume a hypothetical CPU parallelization with perfect linear scaling per core. We note that the recent single-socket CPU results by Leiserson *et al.* and Agarwal *et al.* referenced by Table 2 have not quite managed such scaling. Furthermore, our sequential implementation for a state-of-the-art 3.4GHz Intel Core i7 2600K (Sandybridge) exceeds their



(a) Uniform random



(b) RMAT (*A*=0.45, *B*=0.15, *C*=0.15, *D*=0.25)

Fig. 13. NVIDIA Tesla C2050 traversal throughput.

single-threaded results despite having fewer memory channels. [5], [6]

Assuming 4x scaling across all four 2600K CPU cores, our C2050 traversal rates would outperform the CPU for all benchmark datasets. In addition, the majority of our graph traversal rates exceed 12x speedup, the perfect scaling of three such CPUs. At the extreme, our average *wikipedia-20070206* traversal rates outperform the sequential CPU version by 25x, eight CPU equivalents. We also note that our methods perform well for large and small-diameter graphs alike. Comparing with sequential CPU traversals of *europe.osm* and *kron_g500-logn20*, our *hybrid* strategy provides an order-of-magnitude speedup for both.

Fig. 13 further presents C2050 traversal performance for synthetic uniform-random and RMAT datasets having up to 256 million edges. Each plotted rate is averaged from 100 randomly-sourced traversals. Our maximum traversal rates of 3.5B and 3.6B TE/s occur with $\bar{d}$ = 256 for uniform-random and RMAT datasets having 256M edges, 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 drop-off at n=8 million vertices when the bitmask exceeds the 768KB L2 cache size.

We evaluated the quadratic implementation provided by Hong *et al.* [21] on our benchmark datasets. At best, it achieved an average 2.1x slowdown for *kron_g500-logn20*. At worst, a 2,300x slowdown for *europe.osm*. For *wikipedia-20070206*, a 4.1x slowdown.

We use a previous-generation NVIDIA GTX280 to compare our implementation with the results reported by Luo *et al.* for their linear parallelization [24]. We achieve 4.1x and 1.7x harmonic mean speedups for the referenced 6-pt grid lattices and DIMACS road network datasets, respectively.

6. MULTI-GPU PARALLELIZATION

Communication between GPUs is simplified by a unified virtual address space in which pointers can transparently reference data residing within remote GPUs. PCI-express 2.0 provides each GPU with an external bidirectional bandwidth of 6.6 GB/s. Under the assumption that GPUs send and receive equal amounts of traffic, the rate at which each GPU can be fed with remote work is conservatively bound by 825x10^6 neighbors / sec, 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 an (m/p)-element C_i and (n/p)-element R_i and $Labels_i$ arrays. Because the system is small, we can provision each GPU with its own full-sized n-bit best-effort bitmask.

We stripe ownership of V across the domain of vertex identifiers. Striping provides good probability of an even distribution of adjacency list sizes across GPUs. This 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: (a) an equal share of vertices would overburden first GPU with an abundance of edges; or (b) an equal share of edges leaves the first GPU underutilized because it owns fewer vertices, most of which are apt to be filtered remotely. However, this method of partitioning progressively loses any inherent locality as the number of GPUs increases.

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

1. Invoke the *expansion* kernel on each GPU$_i$, transforming the vertex queue $Qvertex_i$ into an edge queue $Qedge_i$.
2. Invoke a fused *filter+partition* operation for each GPU$_i$ that sorts neighbors within Q*edge$_i$* by ownership into p bins. Vertex identifiers undergo opportunistic local duplicate culling and bitmask filtering during the partitioning process. This partitioning implementation is analogous to the three-kernel radix-sorting pass described by Merrill and Grimshaw [30].
3. Barrier across all GPUs. The sorting must be completed on all GPUs before any can access their bins on remote peers. The host program uses this opportunity to terminate traversal if all bins are empty on all GPUs.
4. Invoke p-1 *contraction* kernels on each GPU$_i$ to stream and filter the incoming neighbors from its peers. Kernel invocation simply uses remote pointers that reference the appropriate peer bins. This assembles each vertex queue $Qvertex_i$ for the next BFS iteration.

The implementation requires ($2m$+n)/p storage for queue arrays per GPU: two edge queues for pre and post-sorted

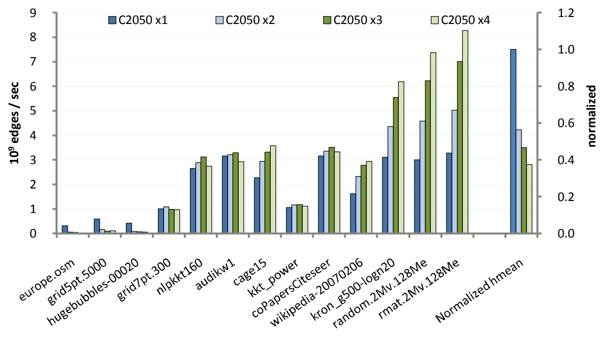


Fig. 14. Average multi-GPU traversal rates. Harmonic means are normalized with respect to the single GPU configuration.

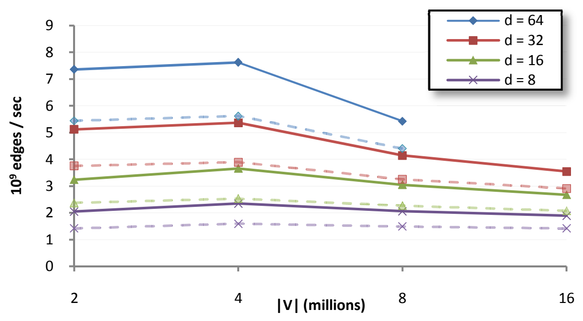


Fig. 15. Multi-GPU sensitivity to graph size and average out-degree $\bar{d}$ for uniform random graphs using four C2050 processors. Dashed lines indicate predecessor labeling variants.

neighbors and a third vertex queue to avoid another global synchronization after Step 4.

6.2 Evaluation

Fig. 14 presents traversal throughput as we scale up the number of GPUs. We experience net slowdown for datasets on the left having average search depth > 100. The cost of global synchronization between BFS iterations is much higher across multiple GPUs.

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

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

Fig. 15 further illustrates the impact of opportunistic duplicate culling for uniform random graphs up to 500M edges and varying out out-degree $\bar{d}$. 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 upon traversal throughput.

To our knowledge, these are the fastest traversal rates demonstrated by a single-node machine. The work by Agarwal et al. is representative of the state-of-the-art in CPU parallelizations, demonstrating up to 1.3 billion edges/sec for both uniform-random and RMAT datasets using four 8-core Intel Nehalem-based XEON CPUs [5]. However, we note that the host memory on such systems can further accommodate datasets having tens of billions of edges.

7. CONCLUSION

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

Furthermore, our implementation performs an asymptotically optimal amount of work. While quadratic-work methods might be acceptable in certain very narrow regimes [21], [31], they suffer from high overhead and did not prove effective on even the lowest diameter graphs in our experimental corpus. Our linear-work method compares very favorably to state-of-the-art multicore implementations across our entire range of benchmarks, which spans five orders of magnitude in graph diameter.

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

- Prefix-sum can serve as an effective alternative to atomic read-modify-write mechanisms for coordinating the placement of items within shared data structures by many parallel threads.
- In contrast to coarse-grained parallelism common on multicore processors, GPU kernels cannot afford to have individual threads streaming through unrelated sections of data. Groups of GPU threads should cooperatively assist each other for data movement tasks.
- Fusing heterogeneous tasks does not always produce the best results. Global redistribution and compaction of fine-grained tasks can significantly improve performance when the alternative would allow significant load imbalance or underutilization.
- The relative I/O contribution from global task redistribution can be less costly than anticipated. The data movement from reorganization may be insignificant in comparison to the actual over-fetch traffic from existing sparse memory accesses.
- It is useful to provide separate implementations for saturating versus fleeting workloads. Hybrid approaches can leverage a shorter code-path for retiring underutilized phases as quickly as possible.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

[1] "Parboil Benchmark suite." [Online]. Available: http://impact.crhc.illinois.edu/parboil.php. [Accessed: 11-Jul-2011].

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

[3] "The Graph 500 List." [Online]. Available: http://www.graph500.org/. [Accessed: 11-Jul-2011].

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

[5] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable Graph Exploration on Multicore Processors," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, USA, 2010, pp. 1-11.

[6] C. E. Leiserson and T. B. Schardl, "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*, New York, NY, USA, 2010, pp. 303–314.

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

[8] D. A. Bader, G. Cong, and J. Feo, "On the Architectural Requirements for Efficient Execution of Graph Algorithms," in *2005 International Conference on Parallel Processing (ICPP'05)*, Oslo, Norway, pp. 547-556.

[9] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170-1183, Dec. 1986.

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

[11] D. Merrill and A. Grimshaw, *Parallel Scan for Stream Architectures*. Department of Computer Science, University of Virginia, 2009.

[12] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, May. 2008.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second. Cambridge, MA: MIT Press, 2001.

[14] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, pp. 677–678, Nov. 1970.

[15] J. Gonzalez, Y. Low, and C. Guestrin, "Residual Splash for Optimally Parallelizing Belief Propagation.," *Journal of Machine Learning Research - Proceedings Track*, vol. 5, pp. 177-184, 2009.

[16] M. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, Feb. 2004.

[17] M. Hussein, A. Varshney, and L. Davis, "On Implementing Graph Cuts on CUDA," in *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, 2007.

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

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

[20] Y. (Steve) Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, New York, NY, USA, 2009, pp. 539–546.

[21] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, New York, NY, USA, 2011, pp. 267–276.

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

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

[24] L. Luo, M. Wong, and W.-mei Hwu, "An effective GPU implementation of breadth-first search," in *Proceedings of the 47th Design Automation Conference*, New York, NY, USA, 2010, pp. 52–55.

[25] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L," in *ACM/IEEE SC 2005 Conference (SC'05)*, Seattle, WA, USA, pp. 25-25.

[26] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient Breadth-First Search on the Cell/BE Processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1381-1395, Oct. 2008.

[27] K. Madduri and D. A. Bader, "GTgraph: A suite of synthetic random graph generators." [Online]. Available: https://sdm.lbl.gov/~kamesh/software/GTgraph/. [Accessed: 11-Jul-2011].

[28] T. Davis and Y. Hu, "University of Florida Sparse Matrix Collection." [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/. [Accessed: 11-Jul-2011].

[29] "10th DIMACS Implementation Challenge." [Online]. Available: http://www.cc.gatech.edu/dimacs10/index.shtml. [Accessed: 11-Jul-2011].

[30] D. Merrill and A. Grimshaw, "High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Processing Letters*, vol. 21, no. 2, pp. 245-272, 2011.

[31] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration for Multi-Core CPU and GPU," presented at the Parallel Architectures and Compilation Techniques, New York, NY, USA, 2011, p. to appear.