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

The large-scale, graph-theoretical analysis of sparse relationships is central to many aspects of computational science. The current trend in microprocessor architecture towards wider data parallelism elevates the importance of graph processing algorithms for GPU and GPU-like processors. As a primitive for graph traversal, breadth-first search (BFS) has direct applications within sparse graph operations. In addition, its behavior is representative of many problems having irregular access patterns and dynamic workloads, including sparse matrix computations utilizing the same underlying data structures. Although recent literature has demonstrated the plausibility of GPU sparse graph traversal, those approaches have not been shown to adequately leverage the bandwidth strengths of the GPU or to be competitive in the absolute with parallel CPU-based implementations.

This report presents a work-efficient $O(|V|+|E|)$ parallelization designed for GPU machine model. Our work addresses the unique challenges of performing parallel adjacency list expansion and managing variable workloads using thousands of hardware-managed threads in the absence of fine-grained synchronization. We present a detailed analysis of simple interactions with a variety of sparse graph data which leads us to somewhat surprising design decisions regarding the amount of data to explicitly move through global memory. Our parallelizations for both single- and multiple-GPU configurations are several factors faster than the state-of-the-art in GPU and CPU BFS. We demonstrate traversal rates in excess of 4.1 billion and 10.3 billion traversed edges per second (TE/s) using single and quad-GPU configurations, respectively.

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. Irregularity is the hallmark of many such algorithms. In this report, we explore the parallelization of one fundamental graph algorithm on GPUs: breadth-first search (BFS). Breadth-first search 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], the emerging Graph500 supercomputer benchmark [3], and will factor into the core challenges of the DARPA Ubiquitous High Performance Computing (UHPC) program [4]. The study of BFS is also relevant to sparse matrix problems that leverage the same underlying data structures.

BFS is representative of many problems for which it is hard to obtain good parallel performance. Parallel processors become notoriously I/O-bound when accessing the problem's large, irregular, and dynamic data structures. Cache optimization is difficult in the general case because data access patterns are determined by the structure of the input graph. In comparison with the serial version, the parallel processing issues of contention, coordination, and load-balancing make it challenging to get significantly better performance when scaling to tens-of-thousands of hardware threads on large-scale machines [5-7].

Algorithms that efficiently utilize tens-of-thousands of hardware thread contexts are increasingly important for single-node problems as well. Microprocessor architecture is evolving toward more aggressive multithreading and wider data parallelism in order to deliver higher throughput while maintaining energy efficiency. Contemporary GPUs are at the leading edge of this trend. They deliver very high throughput on parallel computations, but require large amounts of fine-grained concurrency to do so.

Despite their inherent performance efficiencies, GPU architectures might appear poorly suited for general sparse graph computation. Conventional wisdom posits two important architectural features for facilitating efficient graph algorithms on shared-memory machines [6-8]:

1. Latency tolerance via heavily-overlapped parallelism.
2. Efficient, fine-grained synchronization mechanisms for cooperative workload management and for concurrent data-structure updates.

While GPUs are designed for fine-grained, throughput-oriented computation, they implement a bulk-synchronous, data-parallel programming model that relies upon coarse-grained barrier synchronization for cooperation between threads. GPUs are often perceived as being ill-suited for problems that exhibit dynamic workload expansion and contraction because they do not provide efficient mechanisms for fine-grained atomic read-modify-write operations (e.g., split-phase memory transactions, full-empty bits, or atomic instructions[1]). Without such mechanisms, it is ostensibly difficult to implement parallel algorithms that dynamically

[1] Although GPUs typically provide atomic operations on individual words, their fine-grained usage results in substantial efficiency-loss and serialization of concurrent work.

allocate or relocate data (e.g., BFS, sorting, duplicate removal, search space exploration, etc.). We term such problems as being *allocation-oriented*, the dynamic placement of data being a central aspect of their operation.

This perceived mismatch stems from the GPU machine model which is designed for implementing *stencil-oriented* problems. In a stencil-oriented solution, concurrent threads and their corresponding tasks are statically encoded to produce specific output items. Unfortunately the straightforward application of data-parallel stencils does not often lead to asymptotically-efficient solutions for problems with global task dependences. For problems like BFS and sorting, the only known efficient algorithms require coordination between tasks in order to orchestrate dynamic data movement.

Like our previous work regarding GPU sorting [9], this work espouses the usage of barrier-synchronized prefix-sum as an effective alternative to fine-grained synchronization for allocation-oriented problems. With efficient work management, our implementations show GPUs to be an exceptional architecture for traversing a diversity of large, sparse graphs.

1.1 Contributions

Our work as described in this report makes several important contributions:

Parallelization strategy. We describe a high-performance linear-work BFS parallelization strategy having five important features:

1. *Work-efficient algorithm*. In contrast, nearly all prior GPU efforts implement a stencil-oriented strategy that performs asymptotically quadratic work in the worst case. These approaches require little fine-grained cooperation and synchronization, but their inefficiency renders them unsuitable for traversing graphs having diameters much larger than 5.
2. *Parallel adjacency list expansion*. This is the first presentation of a linear-work GPU implementation to also perform parallel adjacency list expansion. We present a fine-grained approach that is designed to limit all forms of expansion imbalance.
3. *Out-of-core edge-frontier and vertex-frontier queues.* Linear-work strategies for CPUs and other shared-memory, multithreaded architectures typically implement a *vertex-frontier* queue of unexplored vertices in global memory for managing traversal progress. In comparison, we additionally incorporate a global *edge-frontier* queue for expanding all incident neighbors before inspecting any of them. We show the extra queue provides substantially better performance and flexibility.
4. *Heuristics for mitigating concurrent discovery.* Without atomic read-modify-write operations for managing visitation status, parallel BFS implementations are prone to concurrent vertex discovery by multiple threads via different edges. This benign race condition results in extra work and is exacerbated by the wide SMT and SIMD parallelism of the GPU machine model. We introduce two complimentary heuristics for redressing such redundant work: (1) instantaneous warp-culling and (2) localized history-culling.
5. *Use of efficient parallel prefix-sum*. As an algorithmic primitive, prefix-sum can be used to cooperatively compute scatter offsets for each task given its dynamic allocation requirements. Our approach makes heavy use of efficient prefix-sum [10] in place of atomic read-modify-write operations in order to manage workload expansion and contraction.

Multiple-GPU scaling. Recent research has focused on traversal strategies for shared-memory machines that can outfit tens or hundreds of gigabytes of main memory [6], [7], [11], [12]. In order for GPU-based machines to accommodate similarly-sized problems, the problem must be distributed over the physical memories of multiple GPUs. To our knowledge, this is the first presentation of multi-GPU graph traversal. We leverage high-performance GPU sorting [9] in order to batch exchanges of vertex-identifiers between GPUs.

Empirically-driven analyses and design. We present a detailed micro-benchmark study that isolates and analyzes the expansion and contraction aspects of BFS throughout the traversal process. We develop metrics for quantifying redundant expansion overhead and our ability to mitigate its cause, concurrent discovery. In addition, our analyses reveal:

- The serial and warp-centric expansion techniques described by prior work to be inadequate for entire genres of sparse graph datasets.
- The online fusion of neighbor expansion and filtering within the same kernel often yields worse performance than performing them separately.

Excellent absolute and comparative performance. Finally, we show our methods to deliver excellent absolute performance. This is the first presentation of single-processor traversal rates in terms of billions of edges traversed per second (TE/s). We demonstrate rates in excess of 4.1×10^9 and of 10.3×10^9 TE/s for single and quad-GPU configurations, respectively. In comparison to prior work regarding other architectures, our strategies demonstrate:

- 5.8x single-GPU performance advantage over Intel Nehalem-based Xeon CPUs for similarly-parameterized RMAT datasets [7], [11].
- 5.2x quad-GPU performance advantage over parallel implementations for a 128-processor Cray XMT on same-sized uniform-random datasets [13].
- 12x single-GPU performance advantage over Cell Broadband Engine (Cell/BE) for same-sized uniform-random datasets [12].

1.2 Organization of the Report

Section 2 briefly describes the GPU machine and programming models and reviews the parallel breadth-first graph search problem. Section 3 describes the collection of sparse graph datasets that we use for evaluation purposes and characterizes how their intrinsic properties can affect traversal performance. Section 4 presents micro-benchmark performance analyses of the various phases of graph search and the corresponding implications for algorithm design.

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

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

Section 5 describes the details of our single-GPU implementation and overall performance evaluation. Section 6 presents an adaptation for multiple GPUs. Section 7 generalizes our techniques and insights and provides concluding remarks.

2. BACKGROUND

2.1 GPU Machine and Programming Models

The GPU is capable of efficiently executing large quantities of concurrent, ultra-fine-grained tasks. It is often classified as SPMD (single program, multiple data) in that many hardware-scheduled execution contexts, or *threads*, run copies of the same imperative program, or *kernel*.

The typical GPU processor organization entails a collection of cores (*stream multiprocessors*, or SMs), each of which is comprised of homogeneous processing elements (i.e., ALUs). These SM cores employ local SIMD (single instruction, multiple data) techniques in which a single instruction stream is executed by a fixed-size grouping of threads called a *warp*. Each SM uses simultaneous multithreading (SMT) techniques to manage tens of warp contexts per core. Thus the entire processor is host to tens-of-thousands of hardware thread contexts.

Language-level constructs for thread-grouping are provided to facilitate logical problem decomposition in a manner that is convenient for mapping blocks of virtual threads onto physical SM cores. A two-level grouping hierarchy is used for programming a single microprocessor. A *threadblock* (or *CTA*) is a group of threads that will be co-located on the same SM core and share a local memory space. A *grid* is a collection of homogeneous CTAs, encapsulating all of the virtual threads for a given kernel.

GPUs currently serve as co-processors: a program running on a host CPU orchestrates a sequential *stream* of global data flow by repeatedly invoking new kernel instances on the GPU, each of which is initially presented with a consistent view of the results from the previous.

GPU architectures can impose performance pitfalls for data-dependent kernels. These microprocessors are designed for maximum throughput of regularly-structured computation. When the computation becomes dynamic and varied, mismatches with the underlying architecture can result in severe performance penalties. Performance can be significantly 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 [14].

2.2 Breadth First Search

Given a graph G and initial source vertex v_s, our goal is to traverse the vertices of G in breadth-first order starting at v_s. As exploration proceeds outwards from the source vertex along the edges of the graph, each newly-discovered vertex v_i can optionally be labeled with:

- Its distance d_i, the number of edges in the shortest path from v_s to v_i. This labeling establishes the minimum number of hops to each vertex from v_s.
- Its predecessor p_i, the vertex immediately preceding i on the shortest path to it from v_s. This labeling produces a tree of shortest paths rooted at vertex v_s.

Straightforward applications of BFS include: identifying all of the connected components within a graph; testing a graph for bipartiteness; and finding the diameter of tree [15]. More sophisticated problems that incorporate BFS variants include: identifying the reachable set of heap items during garbage collection [16]; belief propagation in statistical inference [17], finding community structure in networks [18], and computing the maximum-flow/minimum-cut for a given graph [19].

Sparse graph representation and data layout. We consider graphs of the form $G = (V, E)$ where the sets V and E are respectively comprised of n vertices and m directed edges. For simplicity, we enumerate the vertices within V as v_0 .. v_{n-1} using integer vertex identifiers. A vertex pair (v_i, v_j) describes a directed edge in the graph from $v_i \rightarrow v_j$. We indicate $A_i = \{v_j \mid (v_i, v_j) \in E\}$ as the set of neighboring vertices incident from vertex v_i. This graph structure is isomorphic to an adjacency matrix **A** whose rows are the adjacency lists A_i.

Unlike dense matrices, the number of edges m in sparse graphs is typically only a constant factor larger than n. There are a variety of sparse matrix representations for efficiently conveying the non-zero entries of the adjacency matrix **A**. We use the well-known compressed sparse row (CSR) representation for storing the graph in memory. As illustrated in Fig. 1, this format is comprised of two arrays: column-indices C and row-offsets R. The column-indices array C is simply the concatenation of the adjacency lists A_i into a single contiguous array of m integers. The row-offsets R array contains $n + 1$ integer offsets, where entry $R[i]$ is the offset into C of the adjacency list A_i for vertex v_i. R has $n + 1$ entries so that we may easily obtain the length of the adjacency list $|A_i| = R[i+1] - R[i]$.

We note that the relationships described by many practical graphs are symmetric. Undirected graphs are a special-case of the general directed-edge representation in which each directed edge (v_i, v_j) in E is accompanied by a corresponding “back edge” (v_j, v_i). Although this property can be exploited in order to halve the memory storage overhead by keeping only the upper (or lower) triangle of **A**, we do not employ any such special storage formats. Many datasets within graph collections such as do not encode graphs this way [20-22]. The storage benefits typically do not outweigh the penalties incurred by higher synchronization overheads, algorithm complexity, and pre-processing overheads. In this work, our discussions of graph sizes and traversal rates are always in the

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)
```

context of directed edges; the edge count *m* is always the sum of the out-degrees of all vertices.

Standard sequential method. As illustrated in Algorithm 1, the standard sequential method for performing BFS proceeds by circulating the vertices of the graph through a FIFO queue [15]. It initializes the distances of all vertices to ∞ and initializes the queue to contain the root vertex v_s. It then iteratively pulls a new vertex from the front of the queue and examines all its neighbors. Any neighbor which is unmarked has its distance set to one more than the distance of the current vertex. (This neighbor can also be labeled with the current vertex as its predecessor.) All newly-marked neighbors are enqueued for later processing. This algorithm performs an asymptotically-linear $O(m+n)$ work. Each vertex is labeled and queued exactly once, and each edge, or neighbor, is traversed exactly once.

The breadth-first nature of the problem implies that each level of the search tree is fully explored before the next. Each BFS level, or iteration, logically identifies an *edge-frontier* and a *vertex-frontier*. The logical edge-frontier is the set of all edges to be traversed during that iteration. Equivalently, it is the concatenation of the neighbors of all vertices marked in the previous iteration. The logical vertex-frontier is the unique subset of such neighbors that are unmarked. They are the unvisited vertices to be enqueued for processing during the next iteration. We can therefore reason about the work performed during each BFS iteration in terms of two phases: an *expansion phase* in which the set of unexplored vertices from the previous iteration are expanded into their adjacency lists, and a *contraction phase* in which this aggregate set of neighbors is reduced to a duplicate-free set of unmarked vertices. As evidenced by the sequential method, these phases are not necessarily disjoint.

Quadratic parallelizations. Quadratic parallelizations create concurrent tasks to inspect every edge or, at a minimum, every vertex in the graph during every iteration. There are two primary variations of the quadratic approach: *vertex-oriented* and *edge-oriented* task decompositions. The vertex-oriented approach maps one task per vertex in which each task serially updates the neighbors of its vertex. The edge-oriented approach assigns one task per edge and concurrently updates all neighbors in the adjacency matrix **A** .

Algorithm 2 presents pseudo-code for the vertex-oriented variant. The distance or predecessor labels for an incident vertex v_j are marked when a task discovers an edge $v_i \rightarrow v_j$ where v_i has been marked and v_j has not. The vertex-oriented methods perform $O(n^2+m)$ work in the worst case: every vertex is inspected upon every iteration, and there may be up to *n* iterations. Similarly, the edge-oriented methods perform $O(mn)$ work in the worst case.

We note that there is a race condition implicit in this algorithm: more than one task may concurrently discover and subsequently mark a vertex v_j. This non-determinism is termed benign because all such contending tasks would apply the same d_j (or p_j).

This method is essentially isomorphic to iterative sparse matrix-vector multiplication (SpMV) in the algebraic semi-ring where the usual (+, ×) operations are replaced with (min, +). This fact makes it easy to adapt generic implementations of SpMV to BFS [23].

This quadratic parallelization strategy has 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. Kernel programs are written such that the specific vertex (or edge) to be inspected by a given thread is statically encoded as a function of the thread's index. Most importantly, each thread's computation is completely independent from that of other threads. Such GPU kernel programs correspond to the inner loop lines 8-11 of Algorithm 2.

Harish *et al.* [24] and Hussein *et al.* [19] describe vertex-oriented versions of this method. Hong *et al.* [25] describe a vectorized version of the vertex-oriented method that is similar to the CSR SpMV approach by Bell and Garland [26]: warps are mapped to vertices instead of threads and the SIMD lanes of the warp are used to *strip mine* the corresponding adjacency lists[2]. This enacts a limited-form of parallel edge-list expansion, effectively replacing the sequential-for loop in line 10 of Algorithm 2 with a parallel-for loop. Deng *et al.* present an edge-oriented implementation [27].

Unfortunately the quadratic strategy is inherently inefficient. Indeed, Hussein *et al.* performed BFS on the GPU largely to avoid the cost of transferring the GPU-resident graph to the CPU just for BFS and then back again [19]. Unless the diameter of a graph is known to be $O(1)$, these algorithms perform asymptotically more work than the sequential algorithm. Moreover, even if the graph diameter were asymptotically constant, the logical vertex-frontier and edge-frontier for a given iteration are typically only a tiny fraction of the respective total numbers of vertices and edges in the graph.

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

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
```

Recently, Hong et al. have argued the superiority of quadratic methods for small-world graphs having low, constant diameter [25], [28]. For only a handful of BFS iterations, they suggest the linear access patterns of referencing the entire vertex set at each iteration will yield better traversal throughput than work-efficient solutions using dynamic task management. By restricting their implementation to datasets having diameter < 256 levels, they can reduce memory traffic by only using 1-byte labels.

Regardless, processing every edge or vertex on every iteration can burden the implementation with substantial overhead. Even though we use larger labels, we show that our linear approach holds a notable performance advantage for all of the low-diameter graphs within our benchmark suite.

Work-efficient parallelizations. A work-efficient parallel BFS algorithm should perform $O(n+m)$ work, just as the sequential algorithm does. The key to achieving this is that each iteration of the level-synchronous algorithm should examine only the edges and vertices in that iteration's logical edge and vertex-frontiers, respectively.

Performing work-efficient parallel BFS involves the use of shared queues or other similar concurrent, dynamic data structures in order to track these frontiers. We classify work-efficient BFS strategies by which frontiers are maintained *in-core* versus *out-of-core*. A frontier that is managed in-core is produced and consumed in nearby storage such as registers, cache, or shared scratch memories. It is typically processed in an online fashion, i.e., its state is never wholly realized. On the other hand, a frontier that is managed out-of-core is fully produced in off-chip memory, typically for consumption by the next BFS iteration after a global synchronization step. In order to be level-synchronous, at least one of the two frontiers must be managed out-of-core. Implementations for shared-memory machines typically only queue the vertex-frontier out-of-core, preferring to process the edge-frontier implicitly in-core. The rationale is that the vertex-frontier is smaller, requiring a factor of $\bar{d}$ less explicit global data movement, where $\bar{d}$ is the average out-degree.

Algorithm 3 presents pseudo-code for a work-efficient BFS parallelization constructed around an out-of-core vertex-frontier queue. Each task is mapped to an unexplored vertex v_i in the queue and operates by inspecting the neighbors of v_i serially. The serial expansion of adjacency lists implies an in-core edge-frontier. A given BFS iteration begins with an input queue of vertices to process and produces an output queue of vertices to be processed by the next iteration.

Research has traditionally focused on two aspects of this scheme: (1) improving hardware utilization via the intelligent scheduling of tasks amongst a fixed number of threads without incurring load imbalance; and (2) designing shared data structures that facilitate concurrent updates (i.e, enqueue and dequeue operations) with minimal overhead from contention.

The typical approach for improving utilization is to first reduce the computational granularity to a homogenous task size. Then a task-scheduling runtime can be employed to distribute them evenly across hardware threads. This is done by supplementing the coarse-grained, vertex-oriented tasks with fine-grained edge-oriented tasks. Logically, the sequential-for loop in line 10 of Algorithm 3 is replaced with a parallel-for loop.

We make an important observation that this nested parallel-for loop imposes a subtle decision for the programmer or task-scheduling runtime regarding whether to expand this edge-frontier in-core our out-of-core. The implementation can either: (a) proceed in strict program order by spawning 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.

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)
```

Conventional wisdom holds that it is prudent to throttle the in-core expansion of untraversed edges in order to minimize off-chip memory traffic. For CPU-based architectures, this implies keeping this state resident in nearby cache. For GPU-like architectures with explicitly-managed nearby shared memories, kernels would strive to batch untraversed edges in these scratch areas for subsequent processing. Our micro-benchmark analyses demonstrate that this performance assumption does not always hold true. Queuing the edge-frontier out-of-core can yield better performance despite being less explicitly efficient.

In recent BFS research, Leiserson and Schardl describe an implementation for multiple-socket, multiple-core (SMP + SMT) CPU processors that incorporates a novel multi-set “bag" data structure for efficient set-unioning as an alternative to FIFO queues for tracking the vertex-frontier [11]. Their variation implements concurrent neighbor inspection as described above, using the Cilk++ runtime to manage the fine-grained, edge-oriented tasks in-core.

For the Cray MTA-2, Bader and Madduri developed an algorithm relying on the MTA's support for fine-grained synchronization using full-empty bits for efficient FIFO enqueue operations [6]. They also perform adjacency-list expansion in parallel, relying on the parallelizing compiler and fine-grained thread-scheduling hardware to manage edge-oriented tasks in-core.

Luo *et al.* present an implementation for GPUs that relies upon a hierarchical scheme for enqueuing the vertex-frontier [28]. 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: parallel threads process queued vertices by serially inspecting the neighbors within the corresponding adjacency lists. Their design is influenced by the significant cost of using atomic instructions on modern GPUs. They implement an upward propagation tree of child-queue structures in an effort to mitigate the contention overhead on any given atomically-incremented queue pointer. The global queue in their implementation does not require an explicit dequeue operation; the kernel statically encodes which verteices a given thread will dequeue as a function of its thread rank.

Distributed parallelizations. There are scenarios where it is necessary or desirable to partition the graph structure amongst multiple processor cores. For example, distributed-memory clusters and supercomputers are used for traversal problems on graphs that are too large to fit within the main memory of a single node. 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 [7]. The typical graph-partitioning approach is to assign each processing element a disjoint “slab” of the adjacency matrix **A**. Each processor owns a subset of *V* and the corresponding adjacency lists in *E*.

Algorithm. 4 details the general strategy for implementing work-efficient traversal of distributed graphs. 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. The algorithm uses a set of distributed edge-queues, one for each processor. As adjacency lists are expanded, neighbors are enqueued to the processor that owns them. The synchronization between BFS levels occurs after the expansion phase, meaning that the edge-frontier is collectively realized within these out-of-core queues.

As processors dequeue their vertices during the subsequent iteration, previously-visited vertices are culled while the remaining have their labels marked appropriately and their adjacency lists expanded. This contraction of edges traversed into unexplored vertices is typically implemented on-line and in-core.

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. To construct a depth-ranking, only the expanded neighbor identifier v_j needs to be forwarded its remote processor. However, the predecessor-tree 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.

In recent research, Yoo *et al.* present a variation for implementing distributed BFS on 32K BlueGene/L nodes [5]. As an alternative to the slab partitioning approach, they describe a two-dimensional strategy that decouples vertex ownership (i.e., management of visitation status) from the storage of its corresponding adjacency list. This brings more structure to the overall communication pattern, reducing the number of remote endpoints that a given processor must communicate with from $O(p)$ to $O(\sqrt{p})$, where p is the number of processors in the system.

For shared-memory CPU-based architectures, Xia and Prasanna proposed a variant for multi-socket Intel Nehalem systems that over-partitions the graph. They provision more out-of-core edge-frontier queues than active threads [29]. This reduces the contention at any given queue. By decoupling threads from queues, they can also dynamically adjust the number of active threads in the system. This allows them to minimize the overhead of barrier synchronization between levels by reducing the number of active threads for BFS iterations that expose less concurrency.

Agarwal *et al.* describe an implementation for multi-socket Intel Nehalem systems that implements both out-of-core vertex-frontier and edge-frontier queues for each socket [7]. As a hybrid of Algorithm 3 and Algorithm. 4, a given BFS iteration is composed of two globally-synchronized phases that produce all remote edges before processing any of them. In the first phase, a given socket (1) expands the neighbors of unexplored vertices from its own vertex queue; (2) inspects, culls, and enqueues the unvisited neighbors that it owns; and (3) forwards the remaining neighbors to their remote sockets. Instead of a managing a single contended queue for incoming edges, each socket uses *p*-1 efficient single-producer/single-consumer FastForward queues [30]. In the second phase, incoming neighbors are inspected and enqueued into the socket's vertex queue if not previously marked. Although each Nehalem socket is multi-threaded and processes vertices concurrently, their implementation does not perform parallel adjacency list expansion. Their implementation uses a single, global, atomically-updated bitmask to reduce the overhead of inspecting a given vertex's visitation status.

Scarpazza *et al.* describe a similar hybrid variation for the Cell BE processor architecture where processor cores must communicate via explicit DMA transfers [12]. Instead of synchronizing BFS levels into two phases, their iterations throttle the dequeuing of unexplored vertices. Processor cores perform edge expansion, exchange, and contraction in batches. They use a bitmask for managing visitation status that is partitioned across the processing elements. We note that their implementation relies upon significant graph preprocessing in order to introduce spatial locality into the graph format; they pre-sort the neighbors within each adjacency list into packaged segments owned by each processor core. By also encoding the adjacency list offset and length into each vertex identifier, they avoid a separate sparse lookup and can leverage the processor's DMA engines to effectively perform parallel adjacency list expansion without using parallel threads.

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

3. BENCHMARK SUITE & WORKLOAD CHARACTERIZATION

This section describes the graph datasets that we use within our micro-benchmark analyses as well as within our overall BFS performance evaluations.

3.1 Graph Datasets

Our benchmark suite is composed of the thirteen synthetic and real-world graph datasets listed in Table 1. We generate the square and cubic Poisson lattice graph datasets ourselves. We use the GTgraph synthetic graph generator [31] to construct the *random.2Mv.128Me* and *rmat.2Mv.128Me* datasets. The *wikipedia-20070206* dataset is from the University of Florida Sparse Matrix Collection [21]. The remaining datasets are from the 10th DIMACS Implementation Challenge [20].

We have selected these datasets because of their diversity. The largest connected components within these datasets have diameters spreading five orders of magnitude. Graph diameter is directly proportional to the average search depth, the expected number of BFS iterations needed to explore a graph from a randomly chosen source vertex. Search depth is inversely proportional to the average edge-frontier size for a given BFS iteration. Table 1 is ordered by decreasing average search depth.

One of our goals is to demonstrate good GPU performance for large-diameter graphs. Parallel BFS implementations typically fare relatively worse on graphs having large search depths. Quadratic approaches for large graphs simply cannot be competitive for graph diameters larger than ~5. Even for work-efficient implementations, the overhead of global processor/thread synchronization between BFS iterations can dominate the overall execution.

This collection also varies widely in terms of locality of reference. Graphs having good locality will have the property that nearby adjacency lists will reference row-offsets, labels, and neighbor-adjacency lists that are generally co-located within their respective arrays. The most regularly-structured graphs are the spatially-descriptive square and cubic Poisson lattices; the most incoherent are the Kronecker-product and uniform-random graphs.

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. While these techniques can improve traversal performance, they require at least an asymptotically-linear amount of work, the overhead of which typically cannot be recovered by the savings earned during a single traversal. Our performance evaluations are conducted on the sparse graph data as-is, e.g., some of the datasets encode their adjacency lists in sorted form while others do not.

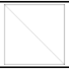
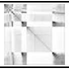
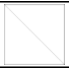
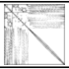
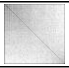
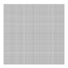
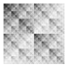
Name	Spy Plot	Description	Vertices (millions)	Edges (millions)	Avg. out-degree $\bar{d}$	Search Depth		Avg. Edge Frontier Size	Ordered Adj. Lists
						μ	(σ/μ)		
europe.osm		Open Street Map road network of Europe (largest strongly-connected component)	50.9	108.1	2.1	19314	14.2%	5,598 (0.01%)	y
grid5pt.5000		5-point Poisson stencil (2D grid lattice)	25.0	125.0	5.0	7500	13.3%	16,663 (0.01%)	n
hugebubbles-00020		Mesh from a synthetic 2D adaptive numerical simulation sequence frame	21.2	63.6	3.0	6151	14.3%	10,336 (0.02%)	y
grid7pt.300		7-point Poisson stencil (3D grid lattice)	27.0	188.5	7.0	679	10.6%	277,473 (0.15%)	n
nlpkkt160		Nonlinear programming matrix for a 3D PDE-constrained optimization problem	8.3	221.2	26.5	142	11.1%	1,556,457 (0.70%)	y
audikw1		Automotive crankshaft model for finite element analysis	0.9	76.7	81.3	62	14.6%	1,241,232 (1.62%)	y
cage15		DNA electrophoresis, transition probabilities between equivalence classes of polymer configurations	5.2	94.0	18.2	37	10.2%	2,534,897 (2.70%)	y
kkt_power		Optimal power flow, nonlinear optimization (KKT)	2.1	13.0	6.3	37	11.9%	355,196 (2.74%)	y
coPapersCiteseer		Citation network: links between co-authors of papers indexed by CiteSeer	0.4	32.1	73.9	26	4.9%	1,257,782 (3.92%)	y
wikipedia-20070206		Links between Wikipedia pages	3.6	45.0	12.6	20	5.3%	2,309,251 (5.13%)	y
kron_g500-logn20		Kronecker graph as per Graph500 (*A*=0.57, *B*=0.19, *C*=0.19, *D*=0.05), small-world, power-law	1.0	100.7	96.0	6	4.8%	16,501,615 (16.4%)	y
random.2Mv.128Me		Uniform random graph in the *G*(*n*, *M*) model	2.0	128.0	64.0	6	0.0%	21,333,333 (16.7%)	n
rmat.2Mv.128Me		Kronecker graph (*A*=0.45, *B*=0.15, *C*=0.15, *D*=0.25), small-world, power-law	2.0	128.0	64.0	6	9.2%	23,272,727 (18.2%)	n

Table 1. Suite of benchmark graphs.

3.2 Plotting Traversal Frontiers

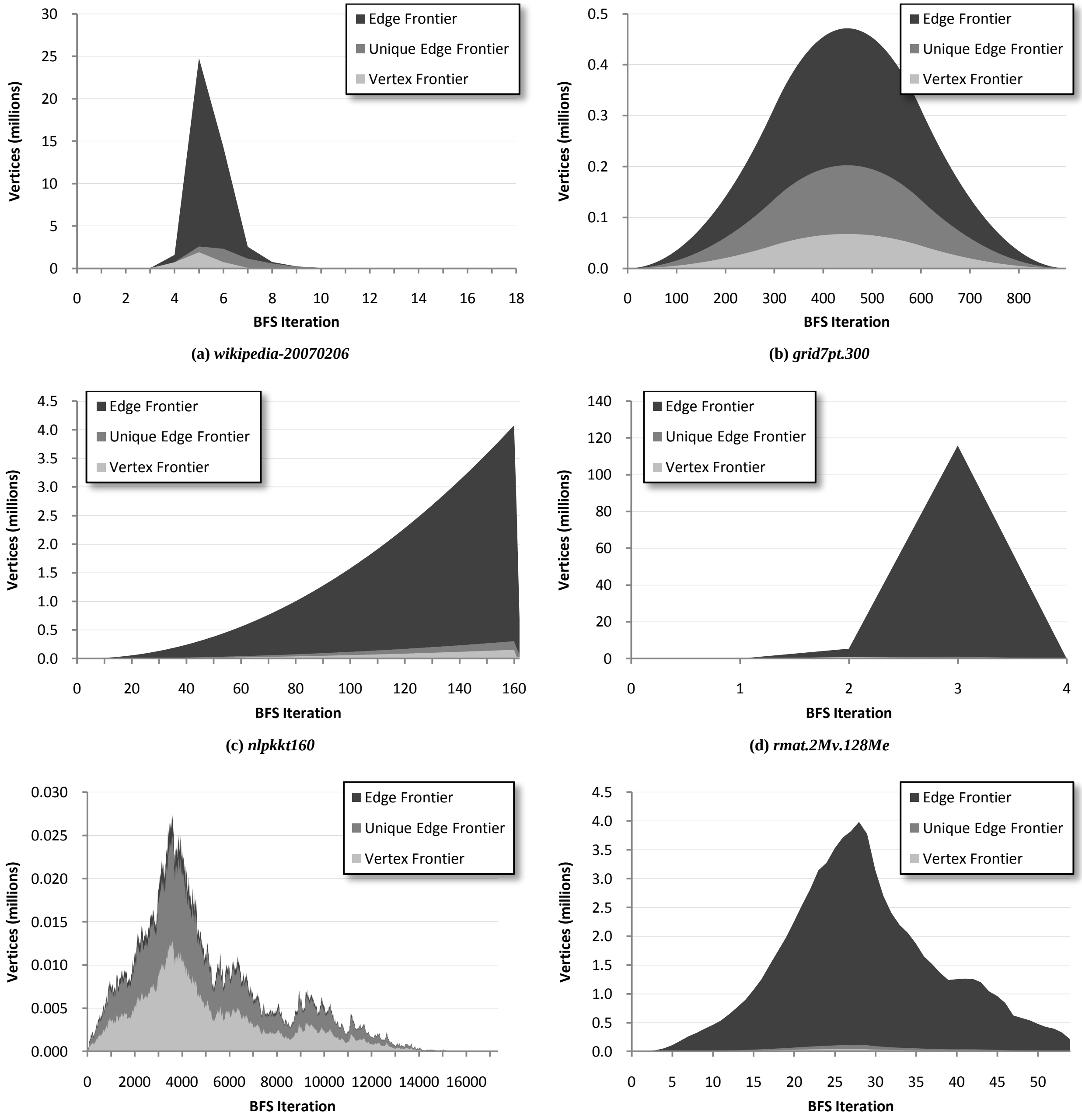
While the spy plot of a given graph's adjacency matrix **A** is useful for illustrating the sparsity pattern and for conveying a sense of locality of reference, it gives us little intuition as to how a traversal will unfold. To help us visualize the rates of workload expansion and contraction, we instead plot the size of the logical edge-frontier as a function of BFS iteration. Alongside the edge-frontier, we also plot (1) the duplicate-free subset of neighbors; and (2) the sub-subset of unique, unvisited vertices (i.e., the vertex-frontier).

Fig. 2 presents example traversal plots for several of the benchmark datasets, each starting from a randomly-chosen source vertex. These plots are implementation-neutral. The logical number of vertices expanded and the number of neighbors visited per iteration are constant properties of the given dataset and starting vertex. Using different source vertices yields similar plots for each dataset, the peaks and valleys simply being time-shifted earlier or later in the traversal process.

These plots enable us to observe how mild or severe the expansion and contraction phases are and how much concurrency they expose. For example, the traversals of the *wikipedia-20070206* and *rmat.2Mv.128Me* dataset are bursty. The bulk of the work is performed in only 1-2 iterations where the concurrency is abundant and the hardware can easily be saturated. Despite the bulk concurrency, the *wikipedia-20070206* dataset has a long tail of light work. This tail contributes heavy global synchronization overhead with little overall progress in terms of the number of edges traversed. The explorations of the other datasets in Fig. 2 proceed in a more sustained manner with less bulk concurrency.

These plots also clearly depict the size-relationships between edge and vertex-frontiers. Many synthetic and real-world graphsthe edge-frontier affor provide two orders-of-magnitude more concurrency for edge-processing work than for vertex-processing work.

This discrepancy has significant consequences for GPU-like architectures that implement larger quantities of processing elements in exchange for higher individual latencies. Care must be taken to redistribute this expansion across processing elements and processor cores. GPU processing elements are comparatively slower than their CPU-based counterparts, and a given BFS iteration can only proceed as fast as its most overloaded processing element.



(a) ***wikipedia-20070206***

(b) ***grid7pt.300***

(c) ***nlpkkt160***

(d) ***rmat.2Mv.128Me***

(e) ***europe.osm***

(f) ***audikw1***

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

Finally, the plots in Fig. 2 also illustrate that the majority of the contraction from edge-frontier down to vertex-frontier can actually be performed using duplicate-removal techniques instead of visitation-status lookup. The implications for BFS contraction are two-fold. First, the sparse lookup of visitation status is one of the most expensive aspects of GPU BFS. This burden can be reduced if local duplicate-removal heuristics can effectively cull many of the redundant neighbors, eliminating sparse reads from out-of-core label/status arrays.

Second, duplicate-removal is also relevant for distributed implementations that partition the graph dataset. The overall amount of networked data movement between processors can be reduced by leveraging the fact that duplicate-removal can be decoupled from status lookup. By first making an effort to remove duplicates from the expansion of non-local neighbors, we can cut down the number of vertex identifiers that must be forwarded to remote queues. We leverage this technique when networking multiple GPUs using a PCI-express interconnect. The benefit of reducing inter-processor traffic is more substantial for multi-GPU traversal because PCI-e provides significantly lower bandwidth than CPU inter-socket interconnection networks such as QPI™ and Hyper-Transport™.

The direct application of this technique is only effective for small numbers of GPU processors. As the number of partitions p increases, the number of duplicates in a given partition correspondingly decreases (assuming a random distribution of edges). In the extreme where $p = |E|$, each processor owns only one edge and there are no duplicates to be locally culled. In order to scale to large numbers of distributed processors, such duplicate-removal techniques would have to be pushed into the hierarchical interconnect. Yoo *et al*. demonstrate a variant of this idea for BlueGene/L using their MPI set-union collective operation [5].

4. MICRO-BENCHMARK ANALYSES

An asymptotically-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 that comprise it: *neighbor-gathering* and *status-lookup*. Although their functions are trivial, the GPU machine model provides interesting challenges for these workloads. In order to develop an intuition for their relative overheads, we further investigate these two activities in the following four analyses:

1. *Isolated neighbor-gathering*. Given a set of unexplored vertices in a vertex-frontier, we present and evaluate alternative strategies for simply loading neighbors from the corresponding adjacency lists. **We show the serial-expansion and warp-centric techniques described by prior work to be inadequate for entire genres of sparse graph datasets**. Alternatively, we present a fine-grained hybrid approach that is designed to limit all forms of expansion imbalance and demonstrate it to be well-suited for all of the datasets within our benchmark suite.
2. *Isolated status-lookup*. Our status-lookup analysis investigates how quickly we can determine which neighbors have been previously visited. While prior work has introduced authoritative bitmasks for managing visitation-status, doing so is not feasible for GPUs. **Instead, we show that even a loosely coherent, best-effort bitmask can significantly accelerate status lookup**.
3. *Coupled gathering and lookup*. Despite the complexity of neighbor-gathering, we reveal status-lookup as the typical limiting factor for GPU BFS. Combining these two workloads within the same kernel reduces memory traffic. **However, we show such coupling can lead to markedly worse performance than performing them separately**.
4. *Concurrent discovery*. Without atomic updates to visitation status, the SIMD nature of the GPU machine model can introduce a significant amount of redundant work due to "concurrent discovery." **We demonstrate alternative duplicate-removal strategies that effectively prevent redundant expansion**.

We performed these analyses using NVIDIA Tesla C2050 GPUs. All of the performance and behavior statistics that we present are derived from measurements taken directly from GPU-based performance counters. (E.g., we do not using CPU-based timers to report throughput.)

For simplicity of presentation, the listings within this section omit several operational details that we consider tuning parameters: (1) the number of CTAs launched per kernel and the number of threads that comprise them; (2) whether CTAs are reused in order to process multiple tiles[3] of items; (3) if reused, whether tiles are evenly distributed across CTAs versus dynamically acquired using coarse-grained atomic operations; (4) how many items each thread processes per tile[4]; and (5) bounds-checks to ensure threads do not read past the end of the input queue.

4.1 Isolated Neighbor Gathering

Our neighbor-gathering analysis is designed to answer the following two questions:

1. *How efficient is the simple vertex-oriented strategy that performs serial adjacency list expansion?* Vertex-oriented expansion traditionally suffers from load imbalance between threads. The varied and coarse task granularity further causes the GPU machine model to suffer from: (1) insufficient concurrency to saturate hardware resources; (2) exacerbated underutilization from load imbalance within warp SIMD lanes; and (3) a diversity of memory access patterns resulting in uncoalesced warp reads and poor bandwidth utilization.
2. *How feasible are strategies for cooperative, parallel adjacency-list expansion*? The enlistment of threads for cooperative expansion is a form of local task redistribution. As with any distribution system, there is a spectrum of scheduling granularity that ranges from individual tasks (higher scheduling overhead) to blocks of tasks (higher underutilization from partial-filling).

We evaluate the spectrum of neighbor-gathering strategies: (1) *serial* gathering; (2) coarse-grained, *warp-based* cooperative gathering; (3) fine-grained, *scan-based* cooperative gathering, (4) a *scan+warp* hybrid; and (5) a *scan+warp+cta* hybrid. We describe these shortly.

We conducted this analysis by performing randomly-sourced traversals of each dataset, invoking these experimental kernels at every BFS iteration. Starting with the traversal's logical vertex-frontier for each iteration, these kernels simply read the row-ranges for the adjacency lists to be expanded and then load the corresponding neighbors into local registers.

For the purposes of this analysis, we preprocess the vertex-frontier before every iteration. Before invoking our kernels,

[3] To avoid further overloading of the term "block", we use *tile* to describe a block of input data that a CTA is designed to process to completion before terminating or obtaining another block of input.

[4] The listings illustrate one dequeued item per thread. However, we can gain greater prefix-sum and gathering efficiency by processing more, e.g., having 128-thread-CTAs that process tiles of 512 items.

Algorithm 5. GPU pseudo-code for a serial neighbor-gathering approach.

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

```
GatherSerial(cta_offset, $Q_{vfront}$, C) {
  {r, r_end} = $Q_{vfront}$[cta_offset + thread_id];
  while (r < r_end) {
    volatile neighbor = C[r];
    r++;
  }
}
```

Algorithm 6. 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 7. 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();
  }
}
```

we first transform each unexplored v_i into the row-range bounds ($R[v_i]$, $R[v_i + 1]$) that describe the location of its adjacency list within the column indices array C. We do this because we wish to evaluate the $O(m)$ gathering of sparse adjacency lists in isolation from the $O(n)$ indirection-lookups within the row-offsets array R.

Serial-gathering. Algorithm 5 presents GPU pseudo-code for the simple serial gathering approach. Similar to Algorithm 3, GPU threads are mapped onto elements of the incoming vertex-frontier queue. Each thread obtains its pre-processed row-range bounds `{r, r_end}` and then serially acquires the corresponding neighbors from the column-indices array C.

Warp-based gathering. Algorithm 6 performs coarse-grained, warp-based cooperative gathering. Each thread begins by obtaining its row-range bounds. If the range is nonzero (i.e., the adjacency list is non-empty), the thread attempts to vie for control of its warp by writing its SIMD lane identifier into a single word shared by all threads of that warp. Because of last-write-wins behavior within the warp, the threads can then read this value to determine 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.

Instead of tasking each thread with its own adjacency list to cooperatively expand, previous warp-centric approaches have assigned only one adjacency-list for the entire warp [22],[23]. While doing so makes the sharing/election aspect of our approach unnecessary, it has two drawbacks: (1) reads from the vertex-frontier queue Q_{vfront} are less efficient because each warp-read obtains only one item instead of a set the size of the warp-width (e.g., 32); and (2) it precludes hybridizing with the following fine-grained approach that maps unexplored vertices to individual threads.

Scan-based gathering. Algorithm 7 performs fine-grained cooperative gathering. Using CTA-wide parallel prefix sum, threads can perfectly pack the gather offsets for 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 neighboring vertex identifiers 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, any imbalance stemming from non-uniformly-sized adjacency lists is suffered in the form of underutilized cycles during the local offset-sharing phase. We can help mitigate imbalance by using a shared `comm` buffer that is several factors larger than the number of threads in the CTA. Regardless, the worst case entails a single thread having more neighbors than the shared `comm` buffer can accommodate, resulting in the idling of all other threads while it serially shares out its gather offsets.

Hybrid scan+warp gathering. A remedy for workload imbalance during gather-offset-sharing is to supplement the fine-grained scan-based cooperation with coarser-grained warp-based cooperation. This approach first applies *warp-based* gathering to acquire portions of adjacency lists greater than or equal to the warp width, and then applies *scan-based* gathering to acquire the remaining "loose ends".

Hybrid scan+warp+cta gathering. While the previous hybrid approach limits the amount of inter-thread load imbalance, opportunities for significant load imbalance between warps are still possible. This is particularly true for graphs with power-law degree distributions where some vertices may have thousands of incident neighbors. We can further mitigate inter-warp workload imbalance by introducing a third granularity of thread-enlistment: the entire CTA. This extended hybrid strategy entails first applying a version of Algorithm 6 that allows threads having adjacency lists larger than `CTA_THREADS` to vie for CTA-control. Once in command of the CTA, the row-range can be broadcast to all threads using barrier-synchronized communication through shared memory in order to subsequently strip-mine the large adjacency list using the width of the entire CTA.

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 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 at their own rate.

Evaluation. For each graph in our benchmark suite, we sample 100 traversals from randomly-chosen starting vertices. Fig. 3a plots the average performances for each strategy in log-scale. The datasets are ordered from left-to-right by decreasing average search depth. For small-world graphs with ample bulk concurrency, the C2050 can achieve sparse neighbor-gathering rates in excess of 16 billion edges/sec. At the other end of the spectrum, a lack of available concurrency and long search depths for road networks limit gathering rates to just under 400M edges/sec.

We begin our discussion of these results by first observing that the hybrid *scan+warp+cta* approach for neighbor-gathering performs on-par-with or better-than the other approaches for every dataset in our collection. It exhibits harmonic mean speedups of 3.3x and 1.5x versus *serial* and *warp-based* gathering, respectively. In determining where the other approaches falter, we can avoid similar shortcomings for other sparse problems beyond breadth-first search.

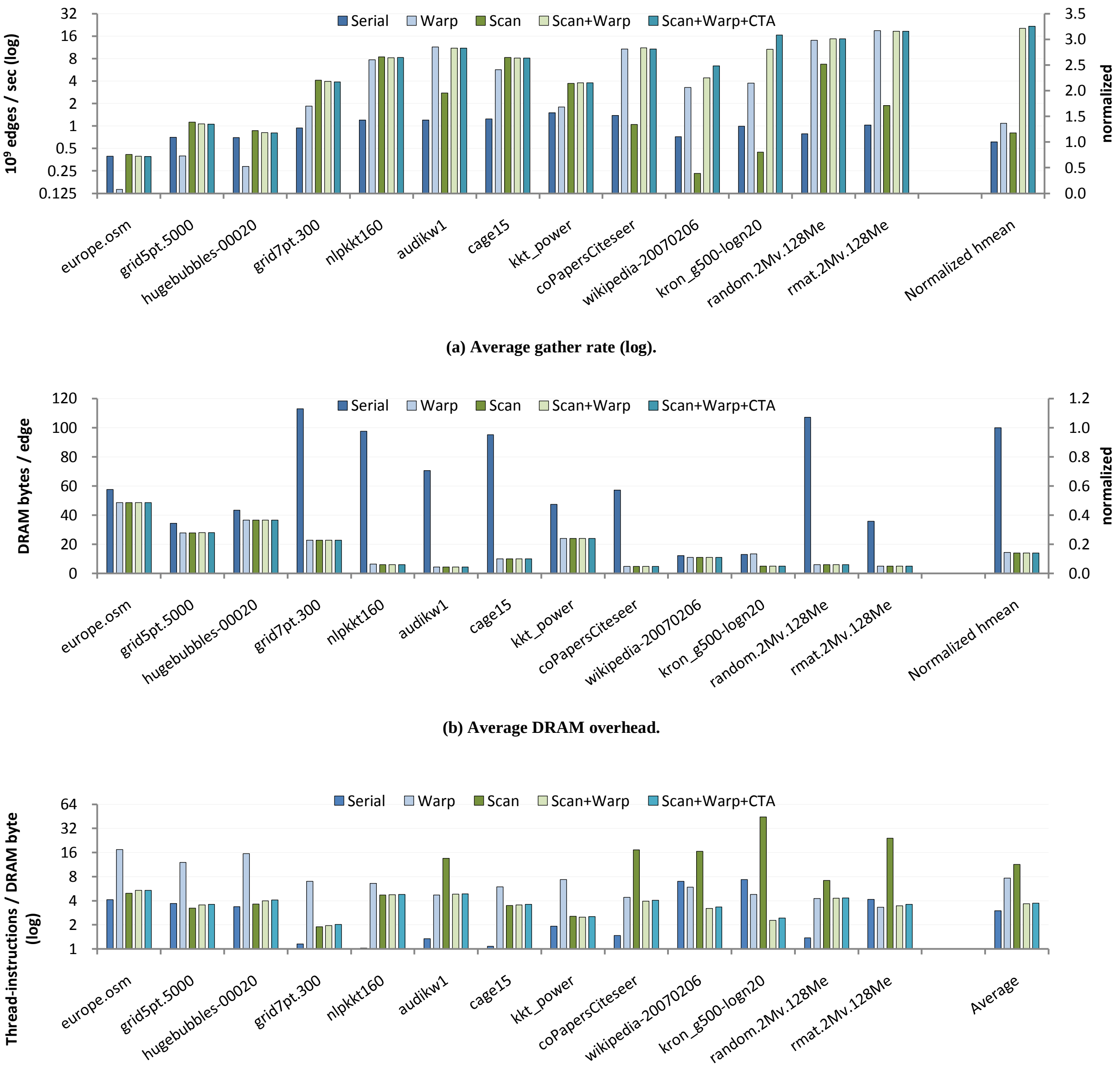
The *serial* approach performs poorly for all but the leftmost two datasets having an average vertex out-degree $\bar{d} \leq 3$. The reason for this is laid bare by Fig. 3b, which plots the average number DRAM bytes accessed per neighbor. 32-bit vertex identifiers imply a lower bound of four bytes per edge. It incurs a dramatic amount of overfetch compared to the other four strategies. This confirms our hypothesis that serialized gathering would suffer uncoalesced memory transactions. As for the leftmost two datasets, no strategy is capable of reliably coalescing accesses to *C* when there are so few neighbors per adjacency list.

On the other hand, the cooperative *warp-based* and *scan-based* approaches suffer from computational inefficiencies. Fig. 3c helps us visualize relative computational inefficiency by plotting computational intensity, the ratio of thread-instructions versus bytes moved through DRAM. We plot the vertical axis using a logarithmic scale; the efficiencies are quite diverse.

The *warp-based* approach performs poorly for the graphs on the left-hand side having average $\bar{d} \leq 10$ Because the average adjacency lists for these graphs are much smaller than the number of threads per warp, a significant number of warp lanes go unused during any given cycle. Gathering throughput suffers because processing elements are busy cycling through threads that are not assigned any work. Fig. 3c counts these unused thread-instructions nonetheless. This observation underscores the need for finer-grained adjacency list expansion.

Unfortunately fine-grained neighbor-gathering alone is not a panacea. As we commented in the description of Algorithm 7, it can suffer from extreme workload imbalance when only one thread is active within the entire CTA. This phenomenon is reflected in Fig. 3c for the datasets on the right-hand size having power-law degree distributions (e.g., *coPapersCiteseer*, *wikipedia-20070206*, *kron_g500-logn20*, etc.). The load imbalance from expanding large adjacency lists leads to substantial dynamic thread instruction counts and corresponding performance degradation.

Although abnormally large instruction counts can be symptomatic of load imbalance, they do not provide direct evidence that it is present. For example, the *warp-based* strategy is uniformly inefficient for the 3D Poisson lattice dataset. To directly capture load imbalance, we introduce the metric of *average CTA duty cycle*. This is the proportion of average versus maximum CTA lifetime. We instrument our



(a) Average gather rate (log).

(b) Average DRAM overhead.

(c) Average computational intensity (log).

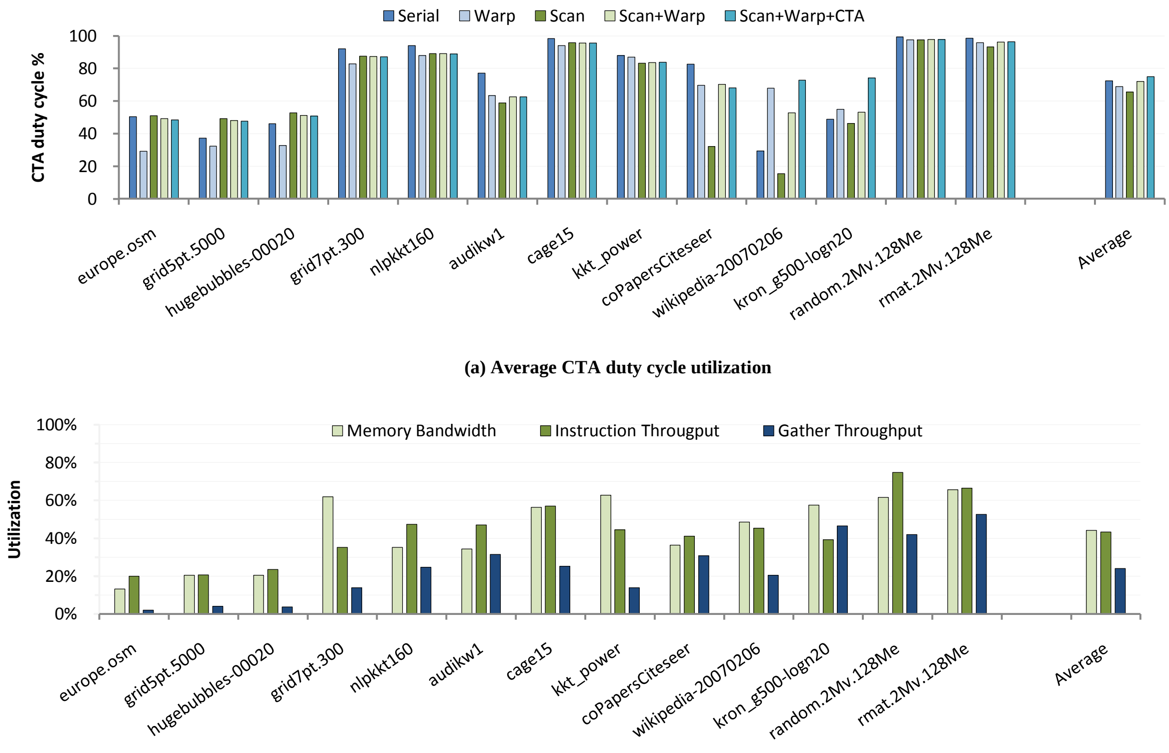
Fig. 3. Neighbor-gathering behavior. Harmonic means are normalized with respect to serial-gathering.

kernels to track the number of cycles for which each CTA is resident on the hardware. A perfectly balanced workload will yield a 100% average duty cycle. This metric is insulated from the overlapped nature of the GPU machine model; fair scheduling implies time progresses equally for all CTAs. By always launching at least as many CTAs as needed to populate every GPU processor core, this metric also captures processor imbalance during workloads too small to saturate the processing elements.

Fig. 4a illustrates the average CTA duty cycle percentages for our neighbor-gathering kernels. The duty cycle for our fine-grained *scan-based* strategy is comparatively much lower for the power-law datasets, confirming our suspicions of load imbalance.

We also observe that the three leftmost graphs exhibit duty cycle statistics of less than 60% for all strategies. With average vertex-frontier sizes less than 4,000 vertices per iteration, these graphs cannot provide fully-resident grids with work for every CTA. The *warp-based* approach underperforms here: its computational inefficiency exacerbates the load imbalance between CTAs with tiles to process and those with none.

Combining the benefits of bulk-enlistment with fine-grained utilization, the hybrid *scan+warp* demonstrates good



(a) Average CTA duty cycle utilization

(b) Throughput utilization

Fig. 4. Utilization for *scan+warp+cta* neighbor-gathering. The C2050 provides a maximum bandwidth of δ_{mem} = 144x10^9 B/sec and a maximum instruction throughput of δ_{inst} = 514x10^9 thread-instructions /sec.

gathering rates across the board. However, the performance opportunities for further including a CTA-sized enlistment granularity are apparent in the *wikipedia-20070206* and *kron_g500-logn20* datasets. These two graphs have a handful of vertices with out-degrees in excess of 128,000 neighbors. As evidenced by improved duty cycles, the *scan+warp+cta* hybrid demonstrates better workload balance and improves gather throughput by 50% for these two datasets.

Finally, Fig. 4b illustrates how well our best strategy *scan+warp+cta* utilizes the underlying hardware. This figure plots the ratios of measured throughputs versus theoretical maximums for (1) DRAM data movement; (2) issued thread-instructions; and (3) neighbors-gathered. We express the maximum gather-throughput for a given graph dataset as

$$\delta_{gather} = (B_{vid}\ \ m\ \ / \ (m+n))\ /\ \delta_{mem}$$

where δ_{mem} is DRAM bandwidth and B_{vid} is the number of bytes per vertex-identifier (in this case four). This discounts the cost of reading the vertex-frontier at each BFS iteration from the theoretically-maximum rate at which the device could read all edges. These metrics facilitate comparisons with future processors and implementations.

4.2 Isolated Status-Lookup

Our status-lookup analysis isolates the work needed to determine which neighbors have been previously visited. Status-lookup and neighbor-gathering and are similar in that both entail $O(m)$ sparse reads. An important distinction is that adjacency lists provide locality for neighbor-gathering by grouping neighbors together. Referencing the visitation-status of these neighbors, however, has arbitrary locality.

This analysis is intended to answer the following question: *Can we decouple visitation-status from vertex labeling in order to reduce bandwidth overhead*? Introducing a bitmask can reduce the size of status data from 32-bits per depth label (or predecessor label) to a single bit per vertex. Prior CPU parallelizations have shown bitmask structures to reduce memory traffic via improved cache coverage [7], [12]. We want to observe whether this holds true for GPU architectures having much smaller and more transient last-level caches. The cost of fetching a cache line is the same whether the miss is for a 32-bit word or a single bit.

Our investigation also differs from CPU approaches in that we avoid protecting the bitmask with atomic operations. The atomic-OR operation is appealing as a convenient mechanism for updating status bits with sequential consistency. Unfortunately atomics are comparatively much less efficient for GPUs. The SIMD aspects of the GPU have severe consequences for datasets with good locality. An entire warp of threads will be serialized if the referenced status bits all reside within the same cache line.

Algorithm 8. GPU pseudo-code for a label-lookup approach to determining visitation status.

Input: Edge-frontier Q_{efront}, label array *L*, and the offset *cta_offset* for the current tile within Q_{efront}.

```
LabelLookup(cta_offset, $Q_{efront}$, L) {
  neighbor = $Q_{efront}$[cta_offset + thread_id];

  // inspect label
  volatile visited = (L[neighbor] != ∞);
}
```

Algorithm 9. GPU pseudo-code for a bitmask-assisted label-lookup approach to determining visitation status.

Input: Edge-frontier Q_{efront}, label array *L*, visitation status bitmask *Mask*, and the offset *cta_offset* for the current tile within Q_{efront}.
Functions: *TexFetch*(*array*, *i*) fetches *array*[*i*] using the texture cache.

```
BitmaskLabelLookup(cta_offset, $Q_{efront}$, L, Mask) {
  neighbor = $Q_{efront}$[cta_offset + thread_id];

  // inspect mask
  mask_byte_offset = neighbor >> 3;
  mask_byte = TexFetch(Mask, mask_byte_offset);
  mask_bit = 1 << (neighbor & 0x7);
  volatile visited;
  if (mask_byte & mask_bit) {

    // seen it
    visited = true;
  } else {

    // unvisited; update mask and inspect label
    Mask[mask_byte_offset] = mask_byte | mask_bit;
    visited = (L[neighbor] != ∞);
  }
}
```

We evaluate two strategies for status-lookup: (1) *label* lookup; and (2) *bitmask-assisted* label lookup. As with the previous analysis, we sample randomly-sourced BFS traversals of each dataset. Kernel invocations for each BFS iteration begin with the traversal's logical edge-frontier in out-of-core memory. Kernels simply read in these vertex identifiers and then obtain the visitation-status for each.

We note two important points regarding this methodology. The first is that lookup overhead is not completely isolated: we also incur the linear streaming overhead of reading the edge-frontier from global memory[5]. Second, starting with a uniform edge-frontier yields the best possible load balancing: lookup tasks are uniformly distributed across all CTAs.

Label lookup. Algorithm 8 presents GPU pseudo-code for determining visitation-status using the labels marked for each vertex. Each thread simply obtains its vertex identifier from the edge-frontier queue Q_{edge} and then references the corresponding label to see if it was previously set.

Bitmask-assisted label lookup. Algorithm 9 introduces a global bitmask *Mask* having *n* bits that caches the visitation-status for each vertex in the graph. Because memory is only byte-addressable, the *Mask* array must be accessed eight bits at a time. After loading its vertex identifier input, each thread uses the GPU texture hardware to obtain the appropriate byte within the *Mask* array. It only accesses the label and updates the enclosing status byte if the corresponding status bit is unset.

We describe this approach as "best-effort" because the bitmask is a conservative approximation of visitation-status. Status bits may appear unset or may be reset for vertices that have been previously visited by concurrent threads. Because we do not serialize updates (e.g., using atomic compare-and-swap), set bits may be "clobbered" due to false-sharing within a single byte.

This scheme also relies upon capacity and conflict misses to update stale bitmask data within the read-only texture cache. The bitmask working set is typically much larger than the individual texture caches, allowing updated data to stream through. We use the texture hardware because it provides lower overhead for incoherent reads and cache misses. The L2/L1 cache hierarchy serializes threads within the same warp that access different cache lines, a common occurrence during sparse traversal.

We must treat the label as the "authoritative" visitation-status because a mask bit may remain unset after a given iteration despite its vertex having been discovered. In comparison, prior work has only used authoritative bitmasks. Agarwal *et al*. implemented atomic updates to a single shared bitmask for SMP CPUs [7] and Scarpazza *et al*. used disjoint bitmasks per parallel processing element for Cell/BE [12]. GPU implementations cannot afford to use fine-grained atomic operations or to manage disjoint graph partitions for tens-of-thousands of threads.

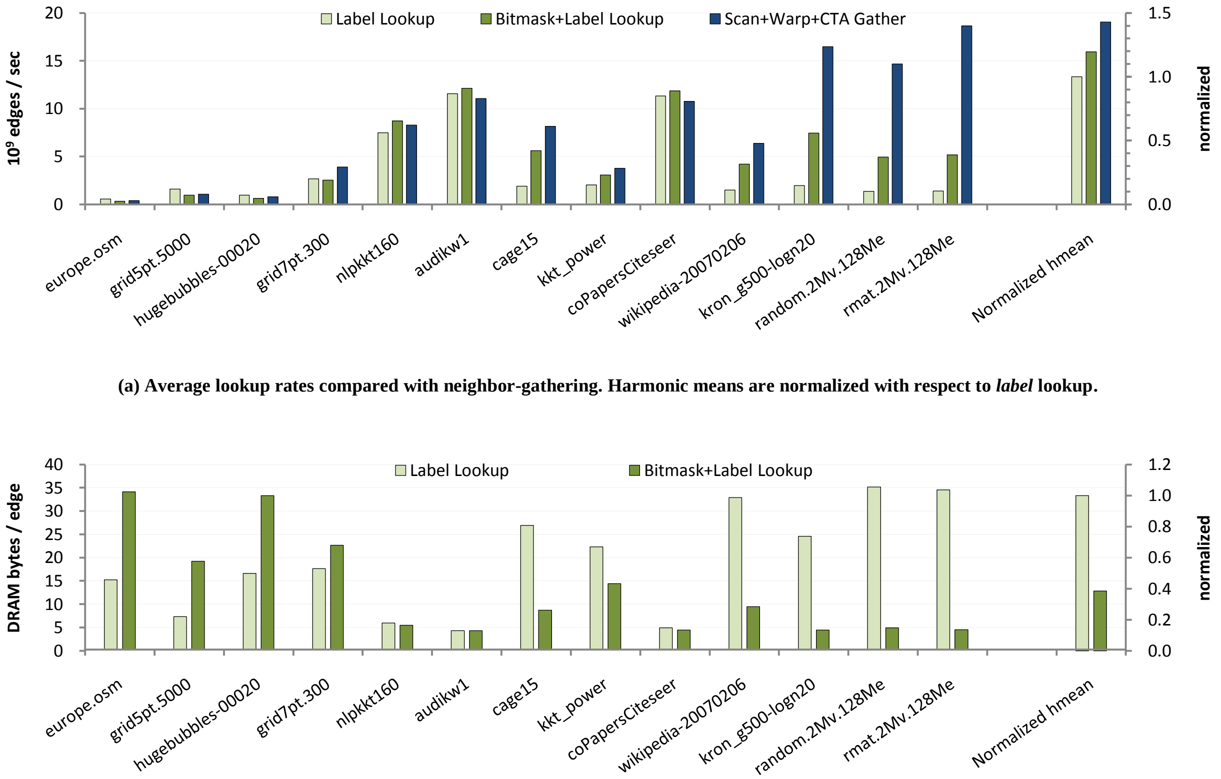
Finally, we note that the bitmask-assisted design is safe. No bit can be set without at least one thread having determined the corresponding vertex to have been previously unvisited.

Evaluation. As with neighbor-gathering, we sample 100 randomly-sourced traversals for each graph dataset. Fig. 5a presents the average performances for both strategies in terms of billions of neighbors inspected per second. For comparison, we also include the gathering rate of our *scan+warp+cta* neighbor-gathering strategy.

We see that the bitmap-assisted strategy significantly improves inspection performance for all but the left-most graphs (*europe.osm*, *grid5pt.5000*, and *hugebuggles-00020*). Despite this, the bitmap-assisted strategy exhibits a 1.2x harmonic mean speedup across all datasets.

With average edge-frontiers having less than 17,000 vertex identifiers, these graphs simply do not provide enough work to saturate the GPU's memory subsystem. Performance is directly tied to latency in the absence of processor saturation. Although the bitmap-assisted strategy demonstrates better bulk-throughput, it suffers longer turn-around latency per neighbor because two global lookups may be required. If we exclude the leftmost three datasets, the bitmap-assisted strategy exhibits a 1.7x harmonic mean speedup.

[5] Such streaming alone can proceed at near-peak bandwidth, 36 billion vertex identifiers per second on the C2050.



(a) Average lookup rates compared with neighbor-gathering. Harmonic means are normalized with respect to *label* lookup.

(b) Average lookup DRAM overheads. Harmonic means are normalized with respect to *label* status-lookup.

Fig. 5. Status-lookup behavior.

The implication is that GPU implementations should skip bitmask inspection for any BFS iteration having an edge-frontier smaller than the number of resident threads. Doing so improves the under-saturated iterations for *all* graphs, including datasets with abundant bulk concurrency in other iterations.

The bitmap-assisted strategy is able to improve performance by reducing the number of sparse DRAM accesses. Fig. 5b illustrates this, plotting the average number of DRAM bytes accessed per neighbor. The harmonic mean traffic reduction across all datasets is 2.6x. (It is 3.2x when excluding the unsaturated leftmost three.)

The overall reduction in DRAM traffic stems from the bitmap's effectiveness at culling vertex identifiers prior to label-inspection. We term this ability *culling-efficiency*. The culling-efficiency of the bitmask is limited by the effects of false-sharing, stale entries in the texture cache, and concurrent-discovery. We further quantify this measure in Section 4.4, showing the bitmap mechanism to identify an average 94% of all neighbors that need to be culled from the edge-frontier.

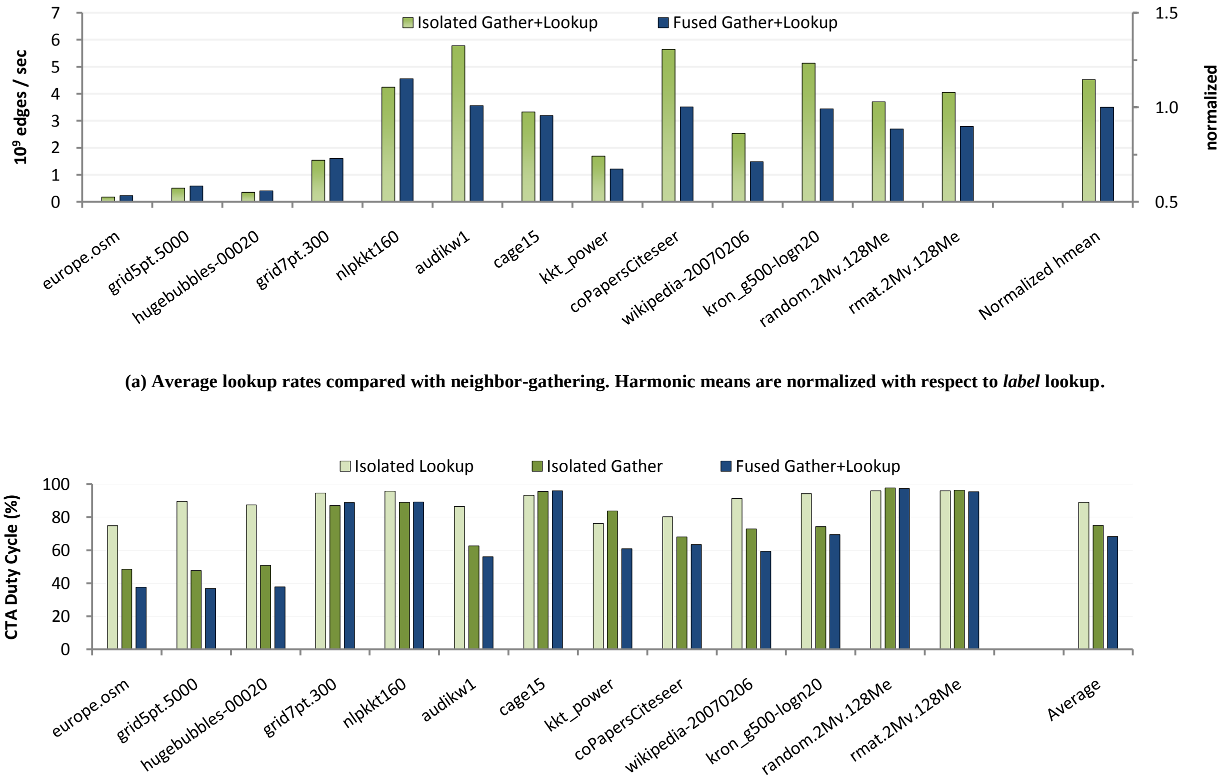
Finally, we compare the throughputs of status-lookup and neighbor-gathering. We observe from Fig. 5a that neighbor-gathering is generally faster. This is particularly true for the datasets on the right-hand side having high average vertex out-degree. The ability for neighbor-gathering to coalesce accesses to adjacency lists increases with $\bar{d}$, whereas status-lookup accesses remain largely uncoalesced.

4.3 Coupling

The previous two analyses investigated neighbor-gathering and status-lookup workloads in isolation. 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. This analysis is intended to answer the following question: *What is the impact of combining these workloads within the same kernel*?

We derive this experiment from our *scan+warp+cta* evaluation of neighbor-gathering. For every neighboring vertex identifier we load (e.g., Algorithm 6 line 21, Algorithm 7 line 24, etc.), we immediately inspect it using Algorithm 9. As with the previous analyses, we sample BFS traversals of each dataset in our benchmark suite. Kernel invocations for each BFS iteration begin with the row-ranges for the traversal's logical vertex-frontier in out-of-core memory. Our fused kernel simply obtains the neighbors within these row-ranges and then reads the visitation-status for each.

We compare this fused kernel with the total isolated gathering and lookup workloads performed separately. The coupled kernel requires $O(m)$ less overall data movement than the other two put together (which effectively reads the edge-frontier twice).



(a) Average lookup rates compared with neighbor-gathering. Harmonic means are normalized with respect to *label* lookup.

(b) Average CTA duty cycle utilization.

Fig. 6. Comparison of isolated versus coupled neighbor-gathering and lookup.

Despite the additional data movement, Fig. 6a shows the separate kernels outperform the fused kernel for the majority of the benchmarks. The average speedup across all datasets is 1.15x. As with the previous two analyses, the speedup is reversed for the graphs on the left-hand side having limited bulk concurrency. The extra data movement results in net slowdown for these latency-bound datasets. The implication is that GPU implementations should consider fused gathering and inspection for any BFS iteration having an edge-frontier smaller than the number of resident threads.

We posit two distinct sources of performance slowdown for the fused strategy: (1) load balancing mismatch between gathering and status workloads; and (2) TLB thrashing. Fig. 6b shows the average CTA duty cycles for the isolated and fused kernels. The duty cycles are much higher for the isolated status-lookup workloads. By starting with a uniform distribution of fine-grained edge-oriented tasks, the status-lookup kernel is better able to spread itself across CTAs for BFS iterations having little bulk concurrency. Fusing the workloads results in poorer overall CTA utilization by forcing neighbors to be expanded and inspected within the same CTA.

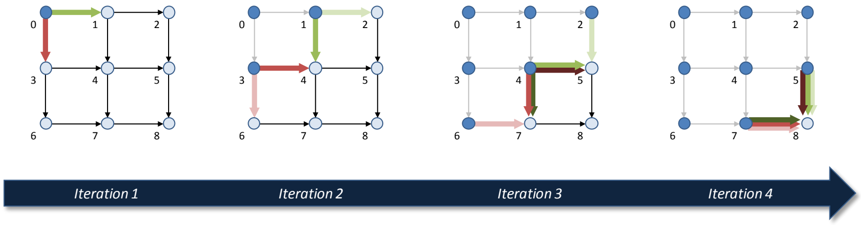
The fused kernel may also suffer from TLB misses experienced by the neighbor-gathering workload. The $O(m)$ column-indices array C is quite large for many datasets, occupying a substantial portion of GPU physical memory. Sparse gathers from it are apt to cause TLB misses. On the other hand, sparse status lookups within the much smaller $O(n)$ bitmask are substantially less likely to cause TLB misses. The fusion of these two workloads inherits the worst aspects of both: TLB misses during uncoalesced status lookups.

4.4 Concurrent Discovery

The BFS contraction phase is responsible for filtering previously-marked and duplicate vertex identifiers from the edge-frontier. Duplicates are representative of different edges incident to the same vertex and can pose a problem for implementations that allow the benign race condition. Multiple threads can concurrently discover the same vertex via these duplicates. Although this does not affect correctness, it does result in the corresponding adjacency list being expanded multiple times. In general, redundant expansion stems from an inability to cull duplicates from the edge frontier.

This analysis is intended to answer the following question: *How much redundant work does status-lookup incur from concurrent discovery?*

Effect on overall workload. Prior CPU implementations that allow the benign race have noted the potential for redundant work, but concluded that its manifestation is negligible [11]. Concurrent discovery is rare due to: (1) a small window of opportunity around status-inspections that are immediately



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

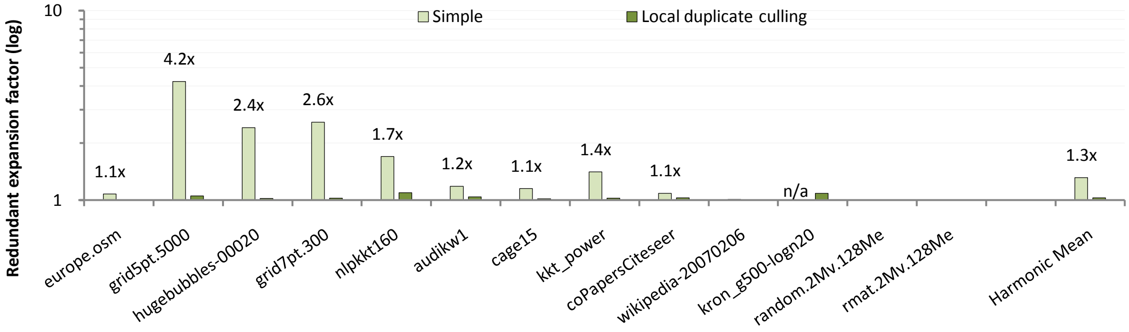


Fig. 8 Redundant work expansion incurred by variants of our *two-phase* BFS implementation. Unlabeled columns are < 1.05x.

followed by status updates; and (2) a relatively low degree of parallelism implemented by CPUs, e.g., ~8 hardware threads.

The GPU machine model, however, is much more vulnerable to concurrent discovery. 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 illustrates 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 7). 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.

While acute, this example is quite representative of real-world lattices and other graphs having good spatial locality. These graphs have a tendency to exhibit nearby duplicates within the edge-frontier. Other dataset properties can also affect the presence of localized duplicates, e.g., multi-edges, sorted adjacency lists, and power-law degree distributions. The concurrent discovery of extremely popular vertices by many threads results in the redundant expansion of very large adjacency lists.

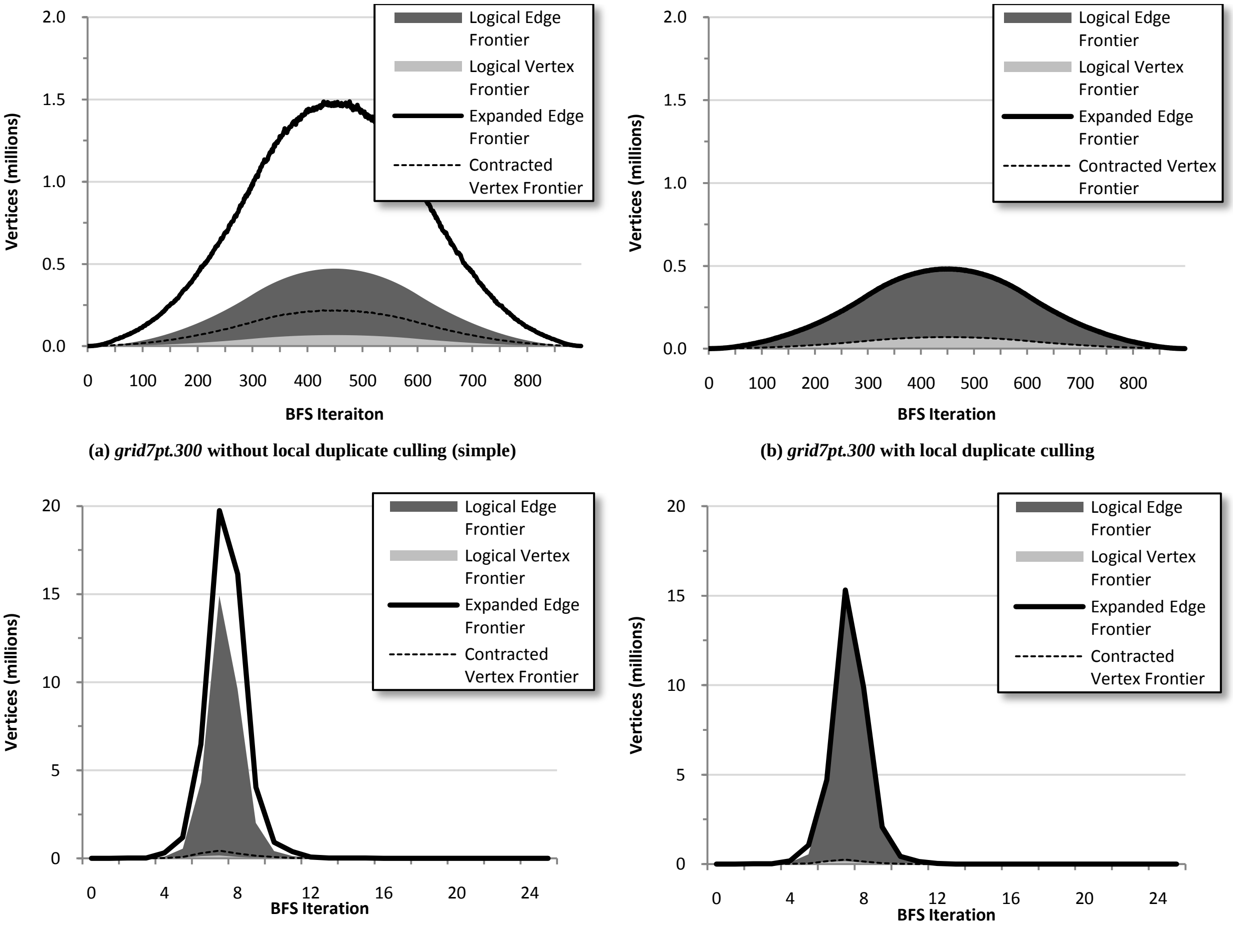
Fig. 8 and Fig. 9 illustrate the serious effects of concurrent discovery upon overall workload. These figures compare two versions of our *two-phase* BFS implementation[6]: (1) a simple version that suffers from concurrent discovery; and (2) our version that mitigates it using two heuristics for local duplicate-culling. Fig. 8 plots the *redundant expansion factor*, the ratio of neighboring vertex identifiers actually expanded versus the total number of edges traversed. Fig. 9 illustrates the workload evolution for several datasets, showing the deltas between the logical and actual frontiers.

For our spatially-descriptive datasets, we observe that the simple version processes several times as many edges than actually traversed. In addition, the simple implementation altogether fails to traverse the *kron_g500-logn20* dataset. Concurrent discovery from a combination of sorted adjacency lists, ultra-popular vertices, and shallow search-depths causes the implementation to run out of physical memory during edge-frontier expansion.

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 examined by more than one thread.

Regarding serial expansion within work-efficient implementations, neither prior work nor our own experiments revealed significant redundant expansion [28]. Any

[6] Briefly, the *two-phase* implementation processes each BFS iteration using separate expansion and contraction kernels. The expansion kernel is based upon our *scan+warp+cta* strategy and wholly produces an out-of-core edge frontier. The contraction kernel is based on our *bitmap-assisted* status-lookup strategy and wholly produces an out-of-core vertex frontier. We provide further description in Section 5.



(a) ***grid7pt.300*** **without local duplicate culling (simple)**

(b) ***grid7pt.300*** **with local duplicate culling**

(c) ***coPapersCiteseer*** **without local duplicate culling (simple)**

(d) ***coPapersCiteseer*** **with local duplicate culling**

Fig. 9. Actual expanded and contracted frontier sizes (with and without local duplicate culling), superimposed over logical frontier sizes.

concurrent discovery from SIMD parallelism during serial inspection is limited to the unlikely event that multiple adjacency lists have the same vertex identifier at the same relative offset. Outside of such instantaneous discovery, the width of opportunity for concurrent discovery is limited because the scope of sequential consistency extends to the entire warp.

Culling Efficiency. We introduce *culling efficiency* as a metric for quantifying our ability to mitigate concurrent discovery. More specifically, it is the ability to discard previously-visited and duplicate vertex identifiers from the edge-frontier during BFS contraction. We measure culling efficiency as the ratio of the number of identifiers actually culled versus the *logical culled set* for a given BFS iteration. The logical culled set is the relative complement of the logical vertex-frontier with respect to the logical edge-frontier. Concurrent discovery occurs when the actual number of culled identifiers is smaller than the logical culled set. Implementations that perform atomic read-modify-write updates to visitation status have 100% culling efficiency. *Culling inefficiency* is the complement of culling efficiency.

We introduce two localized mechanisms for reducing culling inefficiency: (1) *warp-based culling* and (2) *history-based culling*. We describe these techniques shortly. As a continuation of our status-lookup analysis, we perform our analyses in isolation by sampling randomly-sourced traversals of each dataset in our benchmark suite. Kernel invocations for each BFS iteration begin with the traversal's logical edge-frontier in out-of-core memory. Kernels simply read in these vertex identifiers and then determine which should not be allowed into the vertex-frontier. The culling inefficiency is computed from the size-difference between this set and the logical culled set.

Fig. 10 illustrates the progressive application of these mechanisms. We begin with the bitmap heuristic described in Section 4.2 for identifying previously-visited vertices. As a baseline, this establishes an average culling inefficiency of 6.4% across our benchmark suite. The addition of label-lookup (which makes status-lookup safe) improves this average inefficiency to 4.0%.

Without further measure, the culling inefficiency is quite diverse across our benchmark suite. It is as high as 19% for datasets having good spatial locality (*grid5pt.5000*) and as

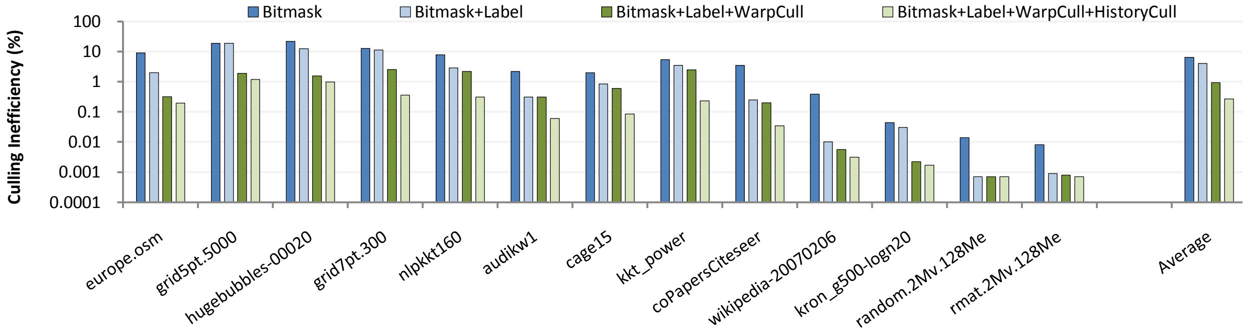


Fig. 10 Culling inefficiency during status-lookup.

Algorithm 10. 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;
}
```

Algorithm 11. GPU pseudo-code for a localized, history-based duplicate-detection heuristic. Not shown, we initialize the *cache* contents to -1 (an invalid vertex-identifier) before the CTA begins tile-processing.

Input: Vertex identifier *neighbor*
Output: True if *neighbor* has conclusively been visited by another thread in the CTA

```
HistoryCull(neighbor) {
  volatile shared cache[2048];
  hash = neighbor & 2047;
  retrieved = cache[hash];
  if (retrieved == neighbor) {
    // seen it
    return true;
  }

  // update it (best effort)
  cache[hash] = neighbor;
  return false;
}
```

low as 0.0007% for large random graphs (*random.2Mv.128Me*). Because redundant expansion is compounded from one BFS iteration to the next, even small inefficiencies can lead to a sizeable amount of extra work. For example, the 3.5% inefficiency for simple traversal of *kkt_power* translates into a 40% redundant overhead.

Warp-based culling. Localized duplicates within the warp are the most egregious source of redundant expansion for spatially-descriptive datasets. These graphs exhibit low average vertex out-degree and a high frequency of reconvergent exploration. This results in concentrations of duplicates within the edge-frontier that are co-located enough to be inspected by the same warp.

Algorithm 10 presents a heuristic for detecting the presence of duplicate vertex-identifiers within the scope of the warp's immediate working set. Using shared-memory per warp, each thread hashes in the vertex-identifier it is currently inspecting. If a collision occurs and a different value is extracted, nothing can be determined regarding duplicate status. Threads that extract the same value will undergo a last-write-wins protocol. Threads that lose this arbitration can safely classify their vertex-identifiers as duplicates to be culled.

Fig. 10 illustrates the effectiveness of warp-based culling for the spatially descriptive graphs (left-hand side). This technique further reduces culling inefficiency during status-lookup by a factor of ten for these datasets.

History-based culling. Aside from SIMD effects, redundant expansion stems from the temporal window for concurrent discovery caused by the benign race. Without write-coherent L1 caches, GPUs have a longer duration between when the first discovery of an unvisited neighbor is made and the time by which other threads on the processor can view the updated status. It is important to minimize this window when traversing graphs having non-uniform degree distributions. Extremely popular vertices are likely to be discovered via by many edges throughout a given BFS iteration, and the cost of expanding them is orders-of-magnitude larger than average.

Algorithm 11 describes the use of a simple CTA-wide software cache within local shared-memory for maintaining a history of recently-inspected vertex-identifiers. If a given thread observes its vertex identifier to have been previously recorded, it can classify that vertex-identifier as safe for culling.

This cache complements the instantaneous coverage provided by the warp-based culling strategy by capturing an extended window of the edge-frontier in space and time. Fig. 10 illustrates the effectiveness of history-based culling for the remainder of datasets having high culling inefficiency (middle-third). This strategy further reduces inefficiency by a factor of five for these datasets. As shown in Fig. 8, 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 IMPLEMENTATION

The previous section analyzed neighbor-gathering and status-lookup workloads in isolation. A complete BFS implementation must couple these two activities. Expanded neighbors must be culled, the remaining neighbors subsequently expanded, and so on. This section explores the following four options for coupling:

1. *Expand-contract*. A single kernel consumes the vertex-frontier for the current BFS iteration and produces the vertex-frontier for the next. Adjacency lists are expanded and filtered online and in-core.
2. *Contract-expand*. The converse. A single kernel filters out previously-visited and duplicate vertex identifiers from the current edge-frontier. The adjacency lists of the surviving vertices are copied out into the edge-frontier for the next iteration.
3. *Two-phase.* A given BFS iteration is processed by two separate kernels that respectively implement expansion and contraction phases. The first consumes the edge-frontier and produces the vertex-frontier. The second does the opposite.
4. *Hybrid*. If the edge-frontier for a given BFS iteration contains more neighboring vertex identifiers than resident threads, we invoke the *two-phase* implementation for that iteration. Otherwise we invoke the *contract-expand* implementation.

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 Decoupled Host Control

A straightforward design would be for the host program to invoke one or more kernels per BFS iteration, blocking until complete. The host could then check the size of the output queue on the GPU and terminate if empty. However, the high latency of small PCI-e transfers between CPU and GPU can quickly dominate overall execution time, particularly for traversals having large search depths.

As an alternative, we design the host program to issue multiple outstanding BFS kernel invocations. The host simply monitors a memory-mapped flag that is set by the kernel if the input queue is empty for the current iteration. All other control data is maintained on the GPU. We find a single outstanding kernel invocation to be sufficient flow control. This decoupling allows us to overlap kernel execution with PCI-e status traffic.

5.2 Expand-contract

The *expand-contract* kernel is loosely based upon the fused gather-lookup benchmark kernel from Section 4.3. This kernel requires $O(2n)$ global storage for input and output vertex-frontier queues. The roles of these two arrays are reversed for alternating BFS iterations.

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

1. Respective to the current tile offset, each thread loads its vertex identifier v_i from the global input queue.
2. Threads then perform local warp-culling and history-culling to determine if v_i is a duplicate.
3. If still valid, the corresponding row-range is loaded from the row-offsets array *R*.
4. 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 and enqueued into the output queue as described below.
5. Threads perform fine-grained, scan-based neighbor-gathering. Gather-offsets are packed into shared memory and CTA-wide batches of small adjacency lists are gathered from *C*. These batches of neighbors are filtered and enqueued into the output queue as described below.

For each strip or batch processed:

i. Every thread gathers a neighboring vertex identifier n_i.

ii. Threads perform status-lookup to invalidate the vast majority of previously-visited and duplicate n_i. If the incoming frontier is larger than the number of resident threads, we use bitmap-assisted lookup, otherwise only label lookup.

iii. Threads with a valid n_i update the corresponding label.

iv. 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.

v. $Thread_0$ obtains the base enqueue offset for valid neighbors within the strip by performing an atomic-add operation on a global queue counter using the strip's total valid count. The returned value is shared to all other threads in the CTA.

vi. 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.

A traversal using the *expand-contract* kernel requires $O(5n + 2m)$ 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.

The *expand-contract* kernel performs three local prefix sums. One is computed for expansion during scan-based gathering. The other two are used for compaction into the global queue during strip and batch processing, respectively. This has performance implications for latency-bound BFS iterations. Although GPU cores can efficiently overlap concurrent prefix sums from different CTAs, the turnaround time for each can be relatively long.

Finally, we note that we do not perform local duplicate-culling immediately after status-lookup. Rather, we delay this work until the start of the next iteration. This greatly improves the effectiveness of warp-culling, enough so to offset the extra overhead of moving any localized duplicates through the global vertex frontier. The density of duplicates per warp is extremely low at this point simply because there are very few valid neighbors after status-lookup. The next iteration has a much better opportunity to cull any duplicates after compacting the remaining neighbors into the global queue. As an alternative, a fourth prefix sum could be used for local compaction after status-lookup, but would further increase tile-processing latency.

5.3 Contract-expand

When labeling distances, the *contract-expand* kernel requires $O(2m)$ global storage for the input and output edge-frontier queues. Variants that label predecessors, however, require $O(4m)$ storage in order to track both origin and destination identifiers within the edge-frontier. In this case we provision a second pair of “shadow” queues for storing the corresponding parents. As with the previous strategy, the input/output roles are reversed for alternating BFS iterations.

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

1. Respective to the current tile offset, each thread loads its neighbor vertex-identifier n_i from the global input queue. If labeling predecessors, each thread also loads the edge’s origin p_i from the parent shadow queue.
2. Threads progressively test their n_i for validity using (i) status-lookup; (ii) warp-based duplicate culling; and (iii) history-based duplicate culling. If the incoming frontier is larger than the number of resident threads, we use bitmap-assisted lookup, otherwise only label lookup.
3. Provided n_i is valid, the corresponding label is updated with the current iteration (or parent).
4. Threads obtain the corresponding row-range from the row-offsets array R for valid n_i and compute its length $|A_i|$.
5. Threads then perform two concurrent CTA-wide prefix sums. If $|A_i|$ is greater than *WARP_SIZE*, it is contributed to the first prefix sum for computing the base scatter offset for coarse-grained warp and CTA neighbor-gathering. Otherwise it is contributed to the second prefix sum for computing the local scratch reservation offset for fine-grained scan-based gathering.
6. $Thread_0$ 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.
7. Threads then perform two phases of coarse-grained gathering analogous to Algorithm 6: first CTA-based, then warp-based. 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. If labeling predecessors, commandeering threads also share their n_i as the predecessor, which is enqueued by all enlisted threads to the outgoing parent shadow queue.
8. Finally, threads perform fine-grained scan-based gathering. This procedure is a variant of Algorithm 7 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.

The *contract-expand* implementation requires $O(3n+4m)$ explicit global data movement for graph traversal[7]. All m edges will be streamed through global memory three times: into registers from C, out to the edge-frontier queue, and back in again the next iteration. The bitmask, label, and row-offset traffic remain the same.

Despite having a much larger traffic workload, c*ontract-expand* strategy is often better suited for processing small frontiers in under-saturated conditions. This strategy has two distinct advantages in these scenarios. The first is much lower tile-processing latency. It only requires local two prefix sums for expansion, and both can be overlapped to further reduce latency.

The second advantage is better bulk concurrency. Because the edge-frontier is typically much larger, fewer resident CTAs sit idle during small BFS iterations. We observed this during our duty-cycle analysis in Section 4.3: status-lookup was more efficient than neighbor-gathering because it had the wider edge-frontier as input. Furthermore, better concurrency helps absorb the extra traffic workload with negligible cost to the under-saturated memory subsystem.

5.4 Two-phase

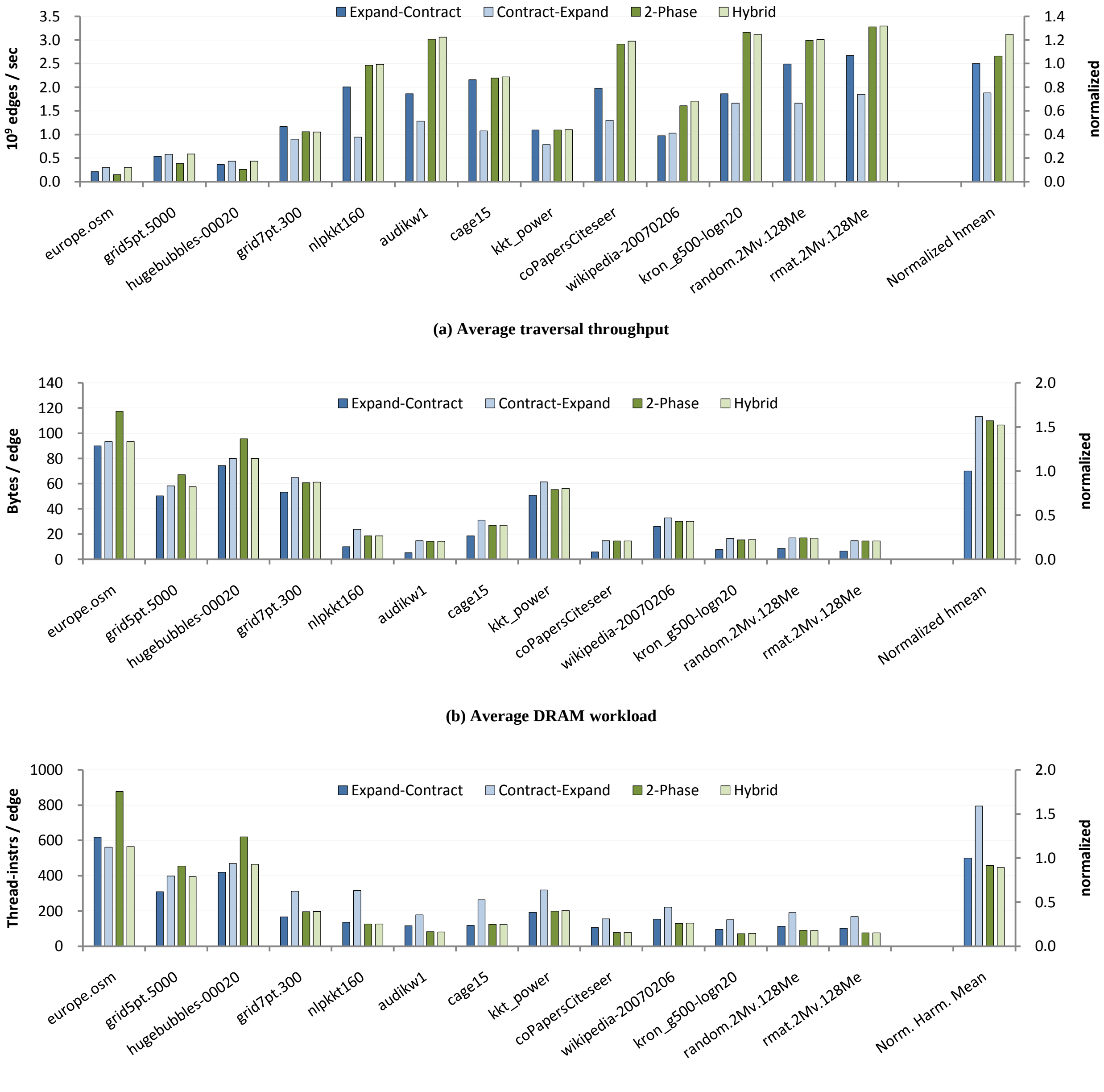
The *two-phase* implementation isolates the contraction and expansion aspects of *contract-expand* into separate kernels. These kernels require $O(n+m)$ global storage for vertex and edge-frontier queues[8].

The contraction kernel begins with the edge-frontier queue as input. Threads filter previously-visited and duplicate neighbors by performing steps 1-3 of the *contract-expand* kernel. The remaining valid neighbors are enqueued into the outgoing vertex-frontier by performing the batch-processing steps *iv-vii* of the expand-contract-kernel.

The expansion kernel begins by loading vertex identifiers from the global input queue. Threads then expand the corresponding adjacency lists into the output edge-frontier queue by performing steps 4-8 of the *contract-expand* kernel.

[7] $O(3n+6m)$ traffic when labeling predecessors.

[8] $O(n+2m)$ global storage when labeling predecessors.



(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.

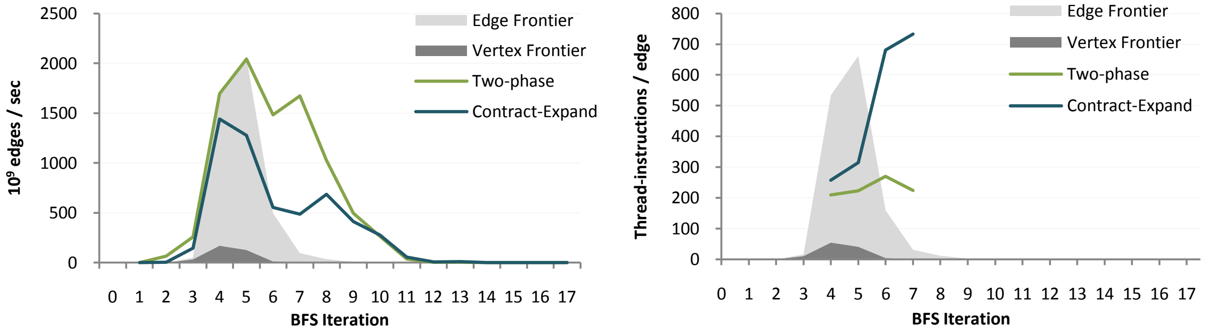
Together, the two kernels require $O(5n+4m)$ explicit global data movement. The memory workload builds upon that of contract contract-expand, but additionally streams n vertices into and out of the global vertex-frontier queue.

5.5 Hybrid

Our hybrid approach combines the relative strengths of the *compact-expand* and *two-phase* approaches: low-latency turnaround for small frontiers and high-efficiency throughput for large frontiers. It inherits the $O(2m)$ global storage requirement from the former and the $O(5n+4m)$ explicit global data movement from the latter.

5.6 Evaluation

Our performance analyses are constructed from 100 randomly-sourced traversals for each dataset. Traversal rates are measured using CPU-based timers in order to capture driver and control overhead. Fig. 11a plots average traversal throughput. As anticipated, the *compact-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. While the *expand-contract* implementation has no outright



(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.

Graph Dataset	CPU Sequential†	CPU Parallel	NVIDIA Tesla C2050 (*hybrid*)				NVIDIA GeForce GTX480 (*hybrid*)			
			Label Distance		Label Predecessor		Label Distance		Label Predecessor	
	10^9 TE/s	10^9 TE/s	10^9 TE/s	Speedup	10^9 TE/s	Speedup	10^9 TE/s	Speedup	10^9 TE/s	Speedup
europe.osm	0.029		0.31	11x	0.31	11x	0.38	13x	0.38	13x
grid5pt.5000	0.081		0.60	7.3x	0.57	7.0x	0.73	9.0x	0.70	8.6x
hugebubbles-00020	0.029		0.43	15x	0.42	15x	0.52	18x	0.51	18x
grid7pt.300	0.038	0.12††	1.1	28x	0.97	26x	1.3	33x	1.2	31x
nlpkkt160	0.26	0.47††	2.5	9.6x	2.1	8.3x	3.2	12x	2.8	11x
audikw1	0.65		3.0	4.6x	2.5	4.0x	3.7	5.7x	3.2	4.9x
cage15	0.13	0.23††	2.2	18x	1.9	15x	2.8	22x	2.4	19x
kkt_power	0.047	0.11††	1.1	23x	1.0	21x	1.3	28x	1.2	26x
coPapersCiteseer	0.50		3.0	5.9x	2.5	5.0x	3.7	7.5x	3.1	6.2x
wikipedia-20070206	0.065	0.19††	1.6	25x	1.4	22x	2.0	31x	2.0	31x
kron_g500-logn20	0.24		3.1	13x	2.5	11x	3.9	16x	3.1	13x
random.2Mv.128Me	0.10	0.50†††	3.0	29x	2.4	23x	3.7	36x	n/a	n/a
rmat.2Mv.128Me	0.15	0.70†††	3.3	22x	2.6	18x	4.1	27x	n/a	n/a

Table 2. Single-processor performance comparison. GPU speedup is in regards to the single-threaded CPU-based implementation. †3.4GHz Core i7 2600K. †† Cited throughput for 2.5 GHz Core i7 4-core, distance-labeling [9]. ††† Cited throughput for 2.7 GHz Xeon X5570 8-core, predecessor labeling [7].

shortcomings, the *hybrid* approach meets or exceeds its performance for every dataset within the benchmark suite.

Fig. 11b illustrates the average DRAM overheads. Compared to the *expand-contract* implementation, the other strategies are explicitly designed to stream three times as much edge data. The actual DRAM savings, however, are substantially less. The ratio is only as high as 2x for the datasets on the right with high $\bar{d}$. The advantage is all but lost in the overhead of over-fetch for the graphs having small $\bar{d}$.

Fig. 11c plots the average computational overheads. We note that the *compact-expand* implementation averages nearly 50% more thread-instructions for graphs having large $\bar{d}$. Using *wikipedia-20070206* as an example, Fig. 12 shows how this behavior is related to workload compaction. The cull-set between edge and vertex-frontiers becomes substantial during BFS iterations 4-6. Fig. 12a shows that *compact-expand* throughput drops significantly during these iterations compared to *two-phase* traversal. Fig. 12b reveals a corresponding increase in dynamic thread-instruction overhead. This is indicative of SIMD underutilization. As neighbors are invalidated by status-lookup and local duplicate removal, the number of valid vertex identifiers within the warp becomes very sparse. Cooperative neighbor-gathering becomes much less efficient as a result.

Table 2 compares distance and predecessor labeling variants of our *hybrid* implementation for both NVIDIA Tesla C2050 and GeForce GTX480 GPUs[9]. The performance difference between labeling variants is highly dependent upon the average out-degree and the corresponding effect it has on DRAM over-fetch. For example, the extra overhead for exchanging parent vertices is negligible for *europe.osm* and is as high as 19% for *rmat.2Mv.128Me*.

In comparing our implementations with other architectures, we note that CPU parallelizations have never been shown to scale linearly with respect to the trivial sequential implementation. Table 2 references state-of-the-art single-socket CPU parallelizations by Leiserson *et al.* and Agarwal *et al.* Our own single-threaded implementation for a state-of-the-art 3.4GHz Intel Core i7 2600K (Sandybridge) is congruent with their single-threaded results. [7], [11]

[9] The GTX480 provisions half as much global memory as the C2050, but has 22% faster processor and memory clocks.

We compare our performance with CPUs as if they could achieve perfect linear scaling per core. Even conceding a hypothetical 4x scaling across all four 2600K CPU cores, our single-C2050 traversal rates would outperform the CPU for all benchmark datasets. In addition, the majority of our single-GTX480 graph traversal rates exceed 16x speedup, the perfect scaling of four such CPUs. At the extreme, our average *wikipedia-20070206* traversal rates outperform the sequential CPU version by 25x and 31x for the C2050 and GTX480, respectively.

Using the C2050, we also evaluated the quadratic-work, distance-labeling implementation provided by Hong *et al.* [25]. Their parallelization is unable to yield comparable traversal rates for any of our datasets. The best and worst traversal rates were 1.5 billion TE/s (2.1x slowdown) for *kron_g500-logn20* and 14 thousand TE/s (2,300x slowdown) for *europe.osm*, respectively. The average *wikipedia-20070206* traversal rate was 392 million TE/s (4.1x slowdown).

Finally, we 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.

These results underscore the importance of workload reorganization for the GPU machine model. Quadratic parallelizations that avoid workload management altogether are simply not competitive. In addition, we show that explicit task compaction is equally as important as efficient task expansion for overall performance.

6. MULTI-GPU IMPLEMENTATION

This section describes our parallelization strategy for multiple GPUs networked by PCI-express links. We can traverse graphs having up to 500 million edges by partitioning the data across the physical memories of up to four Tesla C2050 GPUs. Communication between GPUs is simplified by a unified virtual address space in which pointers can transparently reference data residing within memory attached to the local GPU, remote GPUs, or within pinned pages on the host.

PCI-e 2.0 provides each GPU with an external bidirectional bandwidth of 6.6 GB/s. Thus the rate at which each GPU can simultaneously feed and be fed remote work is conservatively bound by 825 million vertex identifiers per second (four bytes each). This rate is halved for predecessor-labeling variants that must also exchange parent vertices. Remote work dominates local work for systems having more than two GPUs. The ability for the system to substantially exceed this traversal rate per GPU will require the culling of duplicate neighbors before they are forwarded to their owners.

6.1 Design

We implement a simple one-dimensional partitioning of the graph into equally-sized, disjoint subsets of *V*. For a system of *p* GPUs, we initialize each processor p_i with arrays R_i, *Labels*$_\text{i}$, and C_i of size $O(n/p)$ and $O(m/p)$, respectively. Because the system is small, we can provision each GPU with its own full-sized $O(n)$ best-effort bitmask.

We stripe ownership of *V* across the *R* and *Labels* arrays. This is particularly useful for graph datasets having concentrations of popular vertices. For example, Kronecker datasets encode the most popular vertices with the largest adjacency lists near the beginning of *R* and *C*. Dividing 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. Striping provides good probability of an even distribution of adjacency list sizes across all GPUs. However, this method of partitioning progressively reduces any inherent locality as the number of GPUs increases.

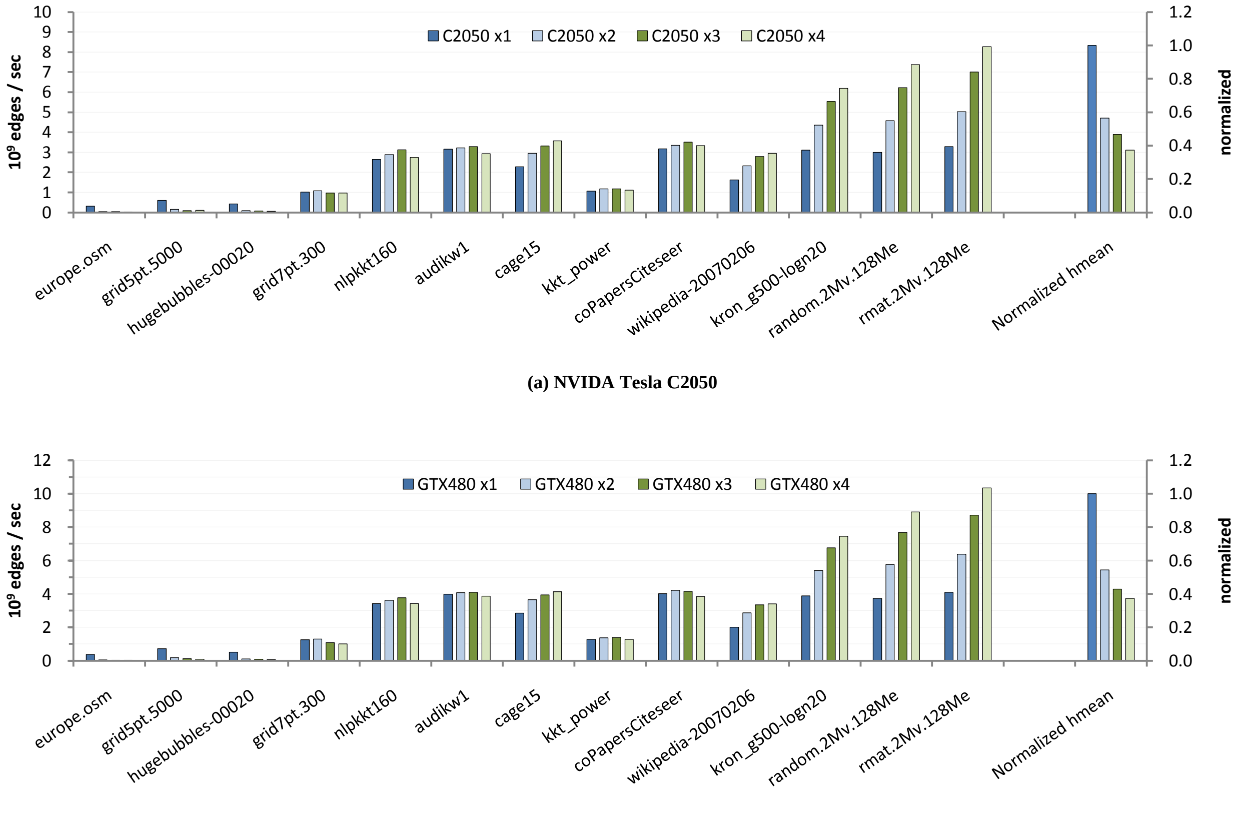
Graph traversal proceeds in level-synchronous fashion. Unlike our single-GPU implementation, we cannot decouple the host program from synchronization. Barriers across all GPUs have much higher latencies because they are actively coordinated by the host.

The host program orchestrates BFS iterations in the following manner:

1. Invoke the single-GPU expansion kernel on each GPU$_i$, transforming the vertex-frontier queue Qv_i into an edge-frontier queue Qe_i.
2. Invoke a fused contraction-partition operation for each GPU$_i$ that sorts neighbors within Qe_i by ownership into *p* bins. Vertex identifiers undergo local duplicate culling and bitmask filtering during the partitioning process.

   This partitioning implementation is analogous to the three-kernel radix-sorting pass we describe in [9]. The “upsweep” kernels compute per-CTA histograms of the numbers of neighbors owned by each GPU. The “spine” kernels perform prefix sums over these histograms. The “downsweep” kernels reorganize the neighbors into bins based upon the offsets computed by the spine kernels. After prefix sum, the histogram for CTA$_0$ on each GPU$_i$ contains the offsets that bin the edge-frontier Qe_i by peer GPU.
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 to stream and filter the incoming neighbors from remote processors. This assembles the local vertex-frontier queue Qv_i on each GPU for the next BFS iteration. The unified address space allows us to directly use the single-GPU contraction kernel. We simply specify input queue pointers that reference the appropriate bin within the sorted edge-frontier on the remote peer.

The implementation requires $O(2m+n)$ storage for queue arrays. We require two edge-frontier queues per GPU (for pre- and post-sorted neighbors) and a separate vertex-frontier queue for triple-buffering in order to avoid a second global synchronization after Step 4.



(a) NVIDA Tesla C2050

(b) NVIDIA GeForce GTX480

Fig. 13. Average multi-GPU traversal rates. Harmonic means are normalized with respect to the single GPU configuration.

6.2 Evaluation

Our test hardware consists of four Tesla C2050 GPUs, four GeForce GTX-480 GPUs, and an EVGA Classified 4-way SLI motherboard[10]. The structure of the PCI-e interconnect follows a binary tree having four leaves (GPUs) linked by three interior switches (an Intel X58 I/O hub and two NF200 multiplexors). Inter-GPU communication is point-to-point; it is not routed through the host CPU. Each link implements all sixteen electrical PCI-e lanes.

Synchronization overhead, long communication latencies, and all-to-all communication patterns have historically limited the scalability of distributed BFS. Fig. 13 demonstrates the same for our parallelization. Our configuration specifically suffers from high GPU synchronization cost. We experience net slowdown for datasets having average search depth > 100.

We do yield notable speedups for the three rightmost datasets having small search depth and large average out-degree. For example, we demonstrate speedups of 1.5x, 2.1x, and 2.5x when traversing *rmat.2Mv.128Me* using two, three, and four GPUs, respectively. As expected, this strong-scaling is not linear. Adding more GPUs reduces the percentage of duplicates per processor. Overall PCI-e traffic increases with fewer duplicates to cull.

The opportunistic duplicate culling within the sorting pass is quite effective for these small-diameter graphs. It allows us to significantly exceed the optimistic cap of 825 million edges/sec per GPU from the PCI-e interconnect. Using four GTX480 processors, we demonstrate traversal rates of 8.9 and 10.3 billion edges/sec for the uniform-random and RMAT datasets respectively.

Fig. 14 further illustrates the impact of opportunistic duplicate culling for uniform random graphs up to 500M edges and varying out out-degree $\bar{d}$. We observe a slight performance drop-off at n=8 million vertices when the bitmask exceeds the 768KB L2 cache size. Otherwise graph size has little impact upon traversal throughput when out-degree is kept constant. Stepping up the average out-degree, however, yields significantly better performance by reducing the relative PCI-e communication overhead. Compared to single-GPU traversal, the relative performance differential between distance and predecessor labeling is much higher due to a doubling of PCI-e overhead.

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

[10] The SLI-adapter is not used. All inter-GPU traffic transpires over PCI-express.

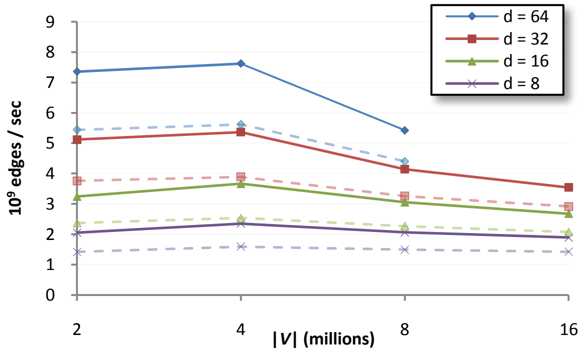


Fig. 14. Multi-GPU sensitivity to graph size and average out-degree $\bar{d}$ for uniform random graphs using four C2050 processors. Dashed lines indicate predecessor labeling variants.

Intel Nehalem-based XEON CPUs [7]. However, we note that the host memory on such systems can further accommodate datasets having tens of billions of edges.

7. CONCLUSION

This report presents a linear-work parallelization of breadth-first search for GPU processors and reveals the GPU to be a superior architecture for sparse graph traversal. We measure single-processor traversal rates in terms of billions of edges per second. Our implementation is designed for a diversity of degree distributions and diameters. Despite a difference of five orders-of-magnitude in diameter between road-network and power-law RMAT graphs, our implementation compares favorably with CPU-based implementations equally well for both.

Our BFS performance is predicated on efficient, dynamic workload management. 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. We place specific emphasis on reorganizing sparse and uneven workloads into dense and uniform ones in all phases of graph traversal. Unlike parallelizations for other architectures, we use efficient prefix sum as a foundation for workload management. This allows us to perform fine-grained work redistribution in a manner that is congruous with the machine model.

Quadratic methods avoid dynamic workload management altogether. Recent publications have claimed these methods to be superior for low-diameter and small-world graphs. We have not observed this to be the case for our benchmark graphs, which have average search depths as low as 6.

Although we demonstrate GPUs to be adroit at graph traversal, we also remark that the programming model does not necessarily facilitate concise and elegant code. The imperative style of encoding local cooperation within a single SPMD program requires the programmer to manage considerable complexity. Constructing a tuned implementation of CTA-wide prefix sum requires several hundred lines of code. In comparison, the appeal of atomic operations is that they can be expressed in a single statement. We suggest that platform vendors provide developers with APIs for local prefix sum, implemented either in software or hardware. This would greatly simplify the programming effort needed to implement high-performance solutions involving dynamic workload management.

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

- Atomic read-modify-write mechanisms operate by serializing fine-grained contention. This makes them unfriendly to the bulk-synchronous and SIMD models of parallelism. As a high-throughput alternative, prefix-sum is often better suited for coordinating the placement of items within shared data structures.
- In contrast to coarse-grained CPU multithreading, GPU kernels cannot afford to have individual threads streaming through unrelated sections of data. The SIMD aspects of load and store transactions suggest that GPU threads should cooperatively enlist each other for data movement tasks.
- Prior GPU work regarding dynamic problems has primarily focused on the expansion aspects of dynamic workloads, i.e., how best to enqueue data into a shared structure. We show that explicit workload contraction is equally important and that we can gain substantially better utilization by actively compacting and reassigning tasks.
- Task redistribution can appear much more expensive than it actually is. In practice we find that any additional data movement is typically dwarfed by the preexisting memory traffic (which is magnified by sparse access patterns and corresponding over-fetch). In addition, the extra instruction overhead is often absorbed without penalty because the processor is memory-bound.
- Online task processing does not always produce the best results. Alternatively, a global redistribution of fine-grained tasks can significantly improve performance. By provisioning separate vertex and edge-frontiers, our design (1) exposes more parallelism to saturate the hardware during expensive status-lookup phases; (2) minimizes the impact of load-imbalance during irregular neighbor-gathering phases; and (3) insulates lookup phases from the TLB thrashing of gather phases.
- It is useful to provide separate implementations for saturating versus fleeting workloads. Without enough bulk concurrency, the performance benefits of work redistribution and other heuristics do not outweigh the extra overhead. Hybrid approaches can leverage a shorter code-path for retiring underutilized phases as quickly as possible.

8. 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] W. Harrod, Ed., "DARPA-SN-09-46 Broad Agency Announcement (BAA) Ubiquitous High Performance Computing (UHPC)."Defense Advanced Research Projects Agency (DARPA) Transformational Convergence Technology Office (TCTO), 02-Mar-2010.

[5] 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.

[6] 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.

[7] 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.

[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] 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.

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

[11] 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, p. 303–314.

[12] 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, 2008.

[13] D. Mizell and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy, 2009, pp. 1-9.

[14] 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, 2008.

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

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

[17] 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.

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

[19] 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.

[20] "10th DIMACS Implementation Challenge."[Online]. Available: http://www.cc.gatech.edu/dimacs10/index.shtml. [Accessed: 11-Jul-2011].

[21] 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].

[22] "Stanford Large Network Dataset Collection."[Online]. Available: http://snap.stanford.edu/data/. [Accessed: 11-Jul-2011].

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

[24] 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, p. 197–208.

[25] 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, p. 267–276.

[26] 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, p. 18:1–18:11.

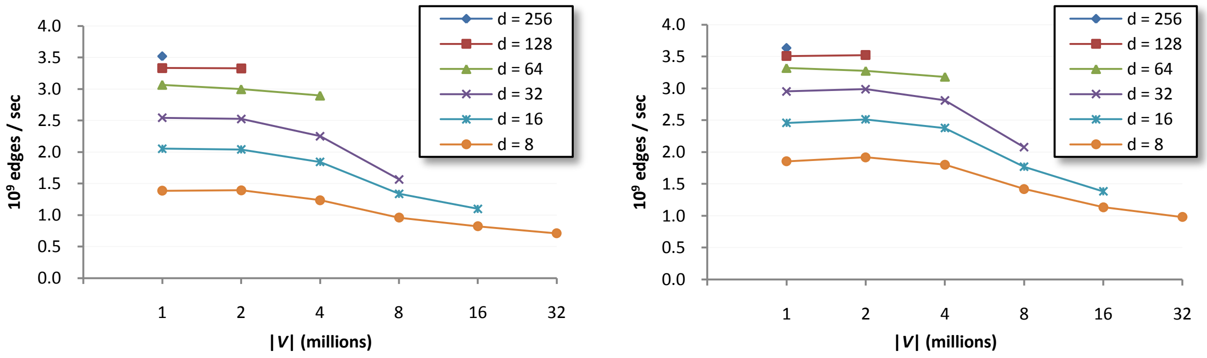
[27] Y. (Steve) Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, New York, NY, USA, 2009, p. 539–546.

[28] 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, p. 52–55.

[29] 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.

[30] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2008, p. 43–52.

[31] 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].



(a) Uniform random

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

Fig. 15. NVIDIA Tesla C2050 traversal throughput.

APPENDIX

This appendix presents C2050 single-GPU 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.