Atos: A Task-Parallel GPU Scheduler for Graph Analytics

Yuxin Chen*
University of California, Davis
Davis, California, USA
yxxchen@ucdavis.edu

Benjamin Brock
University of California, Berkeley
Berkeley, California, USA
brock@cs.berkeley.edu

Serban Porumbescu
University of California, Davis
Davis, California, USA
sdporumbescu@ucdavis.edu

Aydın Buluç
Lawrence Berkeley National Laboratory
Berkeley, California, USA
abuluc@lbl.gov

Katherine Yelick
University of California, Berkeley
Berkeley, California, USA
yelick@berkeley.edu

John D. Owens
University of California, Davis
Davis, California, USA
jowens@ece.ucdavis.edu

ABSTRACT

We present Atos, a task-parallel GPU dynamic scheduling framework that is especially targeted at dynamic irregular applications. Compared to the dominant Bulk Synchronous Parallel (BSP) frameworks, Atos exposes additional concurrency by supporting task-parallel formulations of applications with relaxed dependencies, achieving higher GPU utilization, which is particularly significant for problems with concurrency bottlenecks. Atos also offers implicit task-parallel load balancing in addition to data-parallel load balancing, providing users the flexibility to balance between them to achieve optimal performance. Finally, Atos allows users to adapt to different use cases by controlling the kernel strategy and task-parallel granularity. We demonstrate that each of these controls is important in practice.

We evaluate and analyze the performance of Atos vs. BSP on three applications: breadth-first search, PageRank, and graph coloring. Atos implementations achieve geomean speedups of 3.44x, 2.1x, and 2.77x and peak speedups of 12.8x, 3.2x, and 9.08x across three case studies, compared to a state-of-the-art BSP GPU implementation. Beyond simply quantifying the speedup, we extensively analyze the reasons behind each speedup. This deeper understanding allows us to derive general guidelines for how to select the optimal Atos configuration for different applications. Finally, our analysis provides insights for future dynamic scheduling framework designs.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Theory of computation** → *Parallel computing models*; • **Software and its engineering** → Software prototyping.

KEYWORDS

GPU, irregular workloads, task-parallel, asynchrony, speculation, graph algorithms

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08.
https://doi.org/10.1145/3545008.3545056

ACM Reference Format:
Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D. Owens. 2022. Atos: A Task-Parallel GPU Scheduler for Graph Analytics. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545056

1 INTRODUCTION

Bulk-synchronous parallel (BSP) programming [27] is the traditional model for GPU applications. It is a natural fit for statically schedulable, regular problems, such as many dense-matrix, image-analysis, and structured-grid computations. Programming environments like NVIDIA's CUDA and Khronos's SYCL support this relatively simple model, which maps efficiently to the massive fine-grained parallelism on GPUs and can deliver near-peak performance.

However, some important problems are instead irregular, with frequent control flow branches, non-unit-stride memory accesses, variable amounts of work across loop iterations, and dynamically varying degrees of parallelism. Algorithms that operate on graphs or trees or those with recursive formulations often exhibit such irregularity. These more naturally use a task-based programming model. Programming systems like Legion [2], PTask [22], and StarPU [1] use tasking on the CPU to feed GPUs with kernels to mask the latency of communication and keep the GPU busy. In contrast, we consider the problem of very fine-grained tasking where (traditionally) a set of similar application-level tasks are aggregated to form a data-parallel GPU task. This idea is used in state-of-the-art GPU graph libraries like Gunrock [28], where the application-level tasks are vertices or edges. In Gunrock and similar frameworks, each frontier in a graph sweep is launched as a separate GPU kernel in the BSP model. In practice, this may result in insufficient parallelism, uneven finish times, and high kernel launch overhead for small frontiers.

To address the above issues, we present of Atos, a task-scheduling framework for GPUs, that is adaptable to different usage scenarios:

- It supports both expensive and inexpensive frontiers by providing persistent and non-persistent task schedulers. The persistent scheduler is a GPU kernel that runs continuously to minimize launch overhead.
- It allows the user to trade off task and data parallelism by selecting the worker size, which is the number of GPU threads within each worker, and the number of items in each task.

ICPP '22, August 29-September 1, 2022, Bordeaux, France
Yuxin Chen, et al.

- It uses a single shared task queue, which balances load more quickly than a distributed queue, yet is fast enough to keep GPU workers occupied.
- It supports asynchronous execution across frontiers to maximize available parallelism, while mostly preserving cross-frontier ordering and thus minimizing overwork.

We study three graph algorithms on a variety of graph datasets that stress the importance of this adaptability. These algorithms have nested parallelism with outer-loop dependencies, and the ability to relax those dependencies comes at the cost of possible overwork. Thus a second major theme in this paper is this tradeoff between increased parallelism and overwork. Each of our three algorithms explores this tradeoff in a somewhat different manner, which we discuss and analyze in Sections 5 and 6.

Our contributions include:

- Developing a generalized GPU task-parallel framework that explores a broad design space of possible task-parallel implementations;
- A demonstration of the benefits of mixed task and data parallelism for fine-grained parallel applications;
- Identifying relaxed-synchronization applications as a strong candidate for acceleration with a GPU task-parallel framework; and
- A detailed analysis of application performance that highlights the impact of design decisions both within the task scheduler and at the application level.

2 A DYNAMIC, IRREGULAR APPLICATION PATTERN

Atos handles a broad set of applications with fine-grained task and data parallelism, but we choose to focus here on a particularly challenging class of irregular nested loops with the following form:

Listing 1 A program with nested loops, expressed with a frontier abstraction.

```
1 in_frontier = initialize()
2 while (stop condition not met):  // outer loop
3   ---------------CUDA-KERNEL-----------------
4   for (i in in_frontier):  // inner loops
5     for (j = 0 to workload(i).size()):
6       out_frontier.append(f(in_frontier[i], j))
7   -------------------------------------------
8   cudaDeviceSynchronize()
9   in_frontier = out_frontier
```

The inner loops produce data parallelism that may be flattened as in NESL [6] or Gunrock [28] to maximize parallelism and to implement a data-parallel load-balancing technique. For example, if the inner loops iterate over graph vertices and outgoing edges, some pre-analysis may be used to evenly distribute the edges rather than the vertices. Our work also relaxes the outer loop iterations, which will expand opportunities to find parallelism.

Our applications exhibit one or more of the following forms of dynamic, irregular parallelism:

- *The number of tasks varies across outer loop frontiers:* Work is generated dynamically and the number of output tasks produced from each input task is not fixed.
- *The cost of each inner loop task (lines 4–5 in Listing 1) may vary:* The loop bound (`workload(i).size()`) in line 5 is not fixed.
- *Total work may vary:* The outer loop has a loop-carried dependence, so while dynamic generation of tasks will enforce some dependencies, parallel execution of tasks across frontiers can change program behavior, including the total number of tasks processed.

The pattern in Listing 1 is not specific to graph algorithms. Atos could also address a range of problems that also match this pattern, including ray tracing, where reflected/refracted photons are created dynamically when they collide with objects; complex rendering pipelines such as Reyes where recursive stages require dynamic processing; and iterative algorithms with a similar computation pattern to PageRank, such as federated learning algorithms.

2.1 Performance Challenges of Dynamic, Irregular Problems

Traditional BSP implementations of such applications usually launch a series of kernels, with each kernel corresponding to a bulk-synchronous step (one iteration of the outer loop). Between each step is a global synchronization barrier. The kernels themselves parallelize over the inner loops, with the list of work passed between iterations forming the "frontier". While optimized BSP implementations of many dynamic irregular applications achieve impressive performance, we identify three performance challenges with this approach.

Small frontier problem The runtime of a kernel is the sum of fixed costs (the cost of the global synchronization barrier plus the kernel launch cost) and the amount of time to process the input frontier. The size of input frontiers across iterations may vary significantly over the lifetime of the program. A small frontier has two performance consequences: (1) The fixed costs dominate the overall processing cost; the GPU is spending a significant amount of time setting up or waiting for computation rather than performing it. (2) A small frontier may not be large enough to fill the GPU with work, leaving processing units idle [24].

Load imbalance Because of the irregular nature of work in the inner loop, efficiently assigning work to GPU threads is critical to achieve the highest performance. Statically computing work assignment is infeasible because work is produced dynamically, so lines 4–6 in Listing 1 are usually implemented with data-parallel load balancing techniques [9, 21, 28] that compute work assignment at runtime. However, the optimal load-balancing technique is problem-dependent and even for a fixed problem may be input-dependent. And the runtime cost of that technique may be high.

Loss of concurrency opportunities The BSP model enforces an ordering between every operation in iteration i with every operation in iteration $i+1$, even if some of those pairs may be independent. We relax this cross-frontier ordering in two ways. One is to consider problems whose typical formulation is as nested loops but whose computation is *amenable to reordering* across BSP iterations. Two is to *speculate* that work in later iterations is not dependent on work in earlier iterations and to repair the misspeculation or retry the computation if it is incorrect.

3 OUR TASK-PARALLEL PROGRAMMING MODEL AND DESIGN SPACE

Our programming model allows implementations of task-parallel formulations of the workloads we discussed in Section 2, with a focus on providing solutions to BSP's three challenges: small-frontier, load-imbalance, and loss of concurrency opportunities. We use the following terminology:

worker: one or a group of GPU threads.

task: one or more pieces of work that are scheduled as a single unit in our system.

application function *ff*(): the code that processes each task.

Listing 2 SPMD code of each thread worker in Atos.

```
1  --------CUDA_KERNEL----------
2  for each worker:
3    while not queue.empty():
4      task = queue.concurrent_pop(task.size())
5      new_tasks = f(task)
6      queue.concurrent_push(new_tasks)
7  ----------------------------
```

At a high level, our model maintains a queue of tasks. Workers fetch a task from the queue, process the task, and add newly generated tasks (if any) to the queue. The program runs until either a stop condition is met or the queue is empty (Listing 2). This programming model addresses the performance challenges from Section 2.1:

- We can implement Listing 2 with a single kernel invocation, avoiding multiple launches of small kernels.
- Task parallelism is implicitly load-balanced because workers can run independent tasks and stay busy even if tasks require different amounts of work.
- Listing 2 has no global synchronization; instead the programmer controls the scheduling of work, allowing more flexible dependencies and thus more opportunities for concurrency.

Managing task dependencies In Atos's programming model, tasks themselves (run inside a worker) are executed synchronously, but different tasks (across workers) are executed asynchronously. For this execution model to work, tasks are only added to the task queue when their dependencies are satisfied. Our current implementation of Atos supports tree-structured task dependency graphs, which is sufficient for the graph applications considered in this paper (and we believe most graph algorithms). Atos can be extended in a straightforward way to DAGs by adding (atomic) counters for each join; the last worker to reach the join would continue the computation beyond the join. This strategy does put a burden on the programmer, but the overall advantage of this approach is to support task-based computations that are generated dynamically (in contrast to Legion [2], StarPU [1], Juggler [4], etc. that must build a static task-dependency graph in advance).

The primary focus of this paper is how to best implement Listing 2 on the GPU. We identify below the four key design decisions in such an implementation.

Relaxing barriers A BSP implementation separates work in each iteration with a global barrier. Can we benefit from relaxing this global barrier constraint?

Worker size We can choose what GPU resources we assign to each worker. What is the worker size that yields the best performance?

Data vs. Task Parallelism We expect to leverage the parallelism between tasks. We can also choose the size of our tasks, and within each task, potentially leverage data parallelism. What is the right balance between task and data parallelism?

Kernel strategy Listing 2 is written in a "persistent" style, implementable with a single kernel call. We could alternatively interchange its outer and inner loops, making one "discrete" kernel call per iteration. Which is best for optimal performance?

3.1 Relaxing Barriers

As discussed in Section 2, many applications have the nested loop structure from Listing 1 and their BSP implementations may lose concurrency opportunities because of the global barrier between the outer loops. In many cases, we *can* remove the barrier and relax the outer loop dependency while still computing the correct result. How? Consider two tasks *A* and *B* that, in a BSP implementation, are ordered: *A* is in an iteration that precedes *B* and thus must run before *B*.

- One possibility is to *speculate* that we can compute *A* and *B* at the same time, or even in the order *B* then *A*, without changing the correctness of the computation. If our speculation is correct, then we expose more concurrency. If our speculation is incorrect, then we must fix it. This fix might be cheap or costly.
- A second possibility is a problem formulation that is robust to computing items out of order. This is also called Dijkstra's don't care non-determinism [11].

In either case, we can relax global barriers and expose additional concurrency; depending on the problem and dataset, we may see significant performance gains. However, relaxing barriers may incur additional costs: the cost of performing incorrectly speculated work, the cost of repairing incorrectly speculated work, and less predictable convergence rates when compared to the BSP counterpart. Overall, if the performance improvements from increased concurrency outweigh these costs, we can deliver performance gains.

Related work: Hassaan et al. [16] studied unordered and ordered versions of several algorithms, demonstrating a tradeoff between parallelism and work efficiency. However, the relaxed barrier formulations we study differ from unordered ones. Consider breadth-first search (BFS). Both Hassaan et al. and we begin with a work-efficient Dijkstra BFS, but they compare to a work-inefficient Bellman-Ford BFS, while we compare to a relaxed (speculative) Dijkstra BFS (Section 5.1). The speculative Dijkstra BFS is more work-efficient than Bellman-Ford BFS. Empirically, speculative Dijkstra's workload is within a small constant factor of that of BSP Dijkstra, which is *#edges* (see Table 4). This is much smaller than Bellman-Ford's workload of *diameter* × *#edges*. Kulkarni et al. [19] studied an abstraction and runtime scheme for workloads with optimistic parallelism, which differ from the relaxing barriers we study in this paper. Their notion of optimistic parallelism assumes many

tasks can run in parallel and stops a task the moment it violates a dependency. In contrast, we allow the computation to commit, even if it violates a dependency, and only fix the mistake afterwards.

3.2 Worker Size

Previous task-parallelism work [1, 7, 8, 22] uses the whole GPU as a single worker. They maintain a task dependency graph on the CPU side and orchestrate the execution by launching a CUDA kernel for each task when dependencies are satisfied. We use *GPU-wide-worker* to describe such an organization. Tasks in those GPU-wide-worker task-parallel frameworks are usually very large to better utilize the entire GPU's resources. This organization is easy to program and has low scheduling overhead.

However, the GPU-wide-worker scheme is a poor match when task dependencies require finer management. Consider the following extreme example: Task A and Task B both contain 10,000 data items, and only a single data item in B depends on a single item in A. A GPU-wide-worker implementation must wait for A to complete before beginning work on B, even though most of the data items can be processed independently and concurrently. We use the term *false dependency* to describe the situation when a data item has all its dependencies satisfied, but cannot be processed because another data item in the same task has unresolved dependencies. One can reduce such false dependencies by decomposing a large task into many smaller tasks to expose more parallelism. As a result, workers should be smaller, matching the size of tasks and allowing full utilization of the GPU's resources. This approach has motivated a number of recent task-parallelism frameworks [4, 23, 26, 29], which use workers sized as either warps or Cooperative Thread Arrays (CTAs). The resulting additional complexity in scheduling many smaller workers motivates also moving scheduling decisions from the CPU to the GPU.

Most task-parallelism frameworks only provide one worker size. Our framework provides thread-, warp-, and CTA-sized workers, to support tasks of different size and different synchronization requirements. The only prior work that uses multiple granularities is Whippletree [23]. Whippletree's thread and warp worker sizes are primarily a programming model concept and suffer from synchronization penalties at the implementation level. In Whippletree's implementation, threads are still synchronized within entire CTAs, suffering from false dependencies if tasks require finer synchronization than at CTA granularity.

For graph analytics frameworks in particular, data-parallel bulk-synchronous execution models are by far the most common on GPUs because of their high GPU utilization and effective use of data-parallel load-balancing techniques (e.g. Gunrock [28], cuGraph [12], Medusa [30], SIMD-X [20], GraphBLAST [10]). Current multi-GPU task-based asynchronous graph libraries—Groute [5], Lux [18], and Galois [17]—use a data-based bulk-synchronous model for the computation kernels launched on each GPU.

3.3 Balance Between Data and Task Parallelism

Many parallel applications have work items that require different amounts of processing. Traditional BSP applications address this challenge with explicitly coded data-parallel load balancing techniques. We describe two different approaches in the context of Listing 1: One widely used technique is load balancing search [9], which dynamically computes the prefix-sum of `workload(i).size()` for $i \in$ `in_frontier`, then flattens the two for-loops into one big array and redistributes the work in the array to each CUDA thread (see Baxter [3] for details). Another popular data-parallel load balancing technique separates the work in `in_frontier` into different buckets based on `workload(i).size()` and launches a separate kernel with the best processing strategy for that size for each bucket [21].

Task parallelism is a natural fit for these irregular applications. Workers in our framework do not directly synchronize with each other. One of the primary advantages of this lack of coordination is that it allows workers to attend to any available work items as soon as they become available ("implicit task-parallel load balancing").

Task parallelism and data parallelism are not exclusive—individual tasks of sufficient size may also exploit data parallelism in their execution. Thus we consider a continuous spectrum with pure task-parallel and pure data-parallel load balancing at the extremes, and expect that the optimal trade-off will be application-dependent. Our framework supports two worker sizes larger than a thread (warp and CTA) and offers the programmer the ability to exploit data parallelism within each warp-sized or CTA-sized task. In the framework, workers operate on tasks asynchronously, but an individual worker itself is executed synchronously. Therefore, we can use a worker's capacity as a parameter to control the trade-off between data and task parallelism. Given a fixed number of threads available, increasing a worker's capacity reduces the total number of workers available for a given application. At the same time, an increase in worker capacity results in more opportunities to perform data-parallel load balancing within each worker. In the extreme, setting a worker's capacity to the entire GPU leaves no room for task parallelism and is equivalent to the BSP model. We found that data-parallel load balancing inside the worker combined with task-parallel load balancing provided by Atos results in better overall load balancing (Section 6). We are not aware of any previous work that combines these two types of load balancing.

3.4 Kernel Strategy

Traditional GPU kernels divide a variable amount of input work into fixed-size CTAs and launch a kernel over a CTA count proportional to the amount of input work. Persistent kernels [14] decouple the relationship between data size and launched CTAs. A persistent kernel launches only enough CTAs to fill the GPU. These CTAs remain resident for the entire kernel and run a loop that maps naturally to the task-parallelism model in Listing 2.

Advantages of persistent kernels Persistent kernels reduce kernel launch overhead and CPU/GPU communication. This is particularly significant when many small kernels are required. The persistent kernel approach reduces CPU involvement in favor of programmer-written GPU logic within the persistent kernel.

Disadvantages of persistent kernels GPU workers in the persistent kernel concurrently pop from a shared queue; this requires atomic operations to ensure exclusive pops. Persistent kernels have higher register usage than discrete kernels (requiring extra registers to maintain the queue loop).

Atos: A Task-Parallel GPU Scheduler for Graph Analytics ICPP '22, August 29-September 1, 2022, Bordeaux, France

Intuitively, if a discrete-kernel application suffers from large kernel overhead (exhibits the small frontier problem), a persistent kernel may be preferred. Otherwise, it may be better and simpler to choose a discrete kernel. Previous task-parallelism frameworks either use discrete kernels [1, 7, 22] or persistent kernels [4, 8, 23, 25, 26, 29], but none of them provide both and/or expose that decision to the programmer. In Section 6.5, we discuss the results of our experiments with respect to this choice.

4 FRAMEWORK API

Listing 3 Atos framework APIs

```
template<typename T, typename COUNTER_T>
struct Queues {
    __host__ void init (COUNTER_T capacity, int num_queues, int iteration);

    template<typename F1, typename F2, typename... Args>
    __host__ void launchThread (bool ifPersist, int numBlock,
    int numThread, int sharedMem_size, F1 f1, F2 f2, Args... arg);

    template<typename F1, typename F2, typename... Args>
    __host__ void launchWarp (bool ifPersist, int numBlock,
    int numThread, int sharedMem_size, F1 f1, F2 f2, Args... arg);

    template<int FETCH_SIZE, typename F1, typename F2, typename... Args>
    __host__ void launchCTA (bool ifPersist, int numBlock,
    int numThread, int sharedMem_size, F1 f1, F2 f2, Args... arg);
}
```

With the discussion from Section 3 in mind, we introduce the Atos API shown in Listing 3. launch* API functions are used to launch workers who repeatedly pop tasks from the work queue; each worker then applies function f1 to the task popped. When the worker fails to pop, it runs function f2 (default noop) instead. Under the persistent-kernel mode, numThread × numBlock cannot exceed the maximum number of threads that can concurrently reside on the GPU based on the application's register and shared memory usage. By default, these are set to the maximum allowed values, but can be overridden by users. As discussed in Section 3, when we execute launchCTA, the choice of numThread will determine the tradeoff between task and data parallelism. Argument FETCH_SIZE defines how many data items constitute a task to be popped from the queue by a worker. Given numThread, the FETCH_SIZE should be set accordingly. Increasing (decreasing) FETCH_SIZE will reduce (increase) the number of available tasks for other workers but increase (reduce) local data parallelism.

To illustrate the use of the framework API, we use the example of speculative BFS with warp granularity. In Listing 4, we create a BFS class and define and allocate memory for its relevant variables. The code defines a SIMD function BFSWarp, in which all threads in a warp participate, collectively iterating over all neighbors of node and updating their depth values. If the depth of a neighbor is improved, the neighbor is pushed into the queue. Lastly, we pass BFSWarp() and its arguments onto launchWarp and invoke it.

5 THREE CASE STUDIES

In this paper we select three classic nested loop problems in the domain of graph computation for in-depth study. We choose them because their implementations are well-studied in the BSP model so we can be confident that our results are meaningful. Also, when run on particular datasets, their BSP implementations exhibit one or more of the challenges described in Section 2.1.

Listing 4 Atos-based relaxed BFS (worker size: warp)

```
struct BFS {
  int num_nodes;
  int num_edges;
  CSR *csr;
  int *depth;
  Queues worklists;

  BFS(CSR _csr, int capacity, int num_queues) {
    csr = &_csr;
    worklists.init(capacity, num_queues);
    cudaMalloc(&depth, sizeof(int) * num_nodes);
  }

  void BFSStartWarp(int numBlock, int numThread) {
    worklists.launchWarp(1, numBlock, numThread, 0, BFSWarp(), *this);
  }
};

template<typename VertexId = int, typename SizeT = int>
class BFSWarp {
 public:
  __device__ void operator()(VertexId node, BFS bfs) {
    VertexId depth = bfs.depth[node];
    SizeT node_offset = bfs.csr.get_neighbor_list_start(node);
    SizeT neighborlen = bfs.csr.get_neighbor_list_length(node);
    for (int item = LANE; item < neighborlen; item = item + 32) {
      VertexId neighbor = bfs.csr.get_neighbor(node_offset + item);
      VertexId old_depth = atomicMin(bfs.depth + neighbor, depth + 1);
      if (old_depth > depth + 1) bfs.worklists.push_warp(neighbor);
    }
    __syncwarp();
  }
};
```

5.1 Breadth-First Search

The pseudocodes of BSP BFS and relaxed-barrier BFS ("Speculative BFS") are shown in Algorithms 1 and 2. Algorithm 1 shows that BSP BFS is exactly Dijkstra's algorithm; BSP BFS advances the outer iteration only when all vertices in the frontier are traversed; thus strict breadth-first ordering is preserved, ensuring every vertex will first be reached via its optimal path.

In contrast, in Algorithm 2, Speculative BFS relaxes the barrier constraints of the outer loop of BSP BFS. Speculative BFS has a single dynamic frontier, and many independent CUDA workers asynchronously push and pop vertices to the frontier. Consequently, vertices at different distances can be processed simultaneously. This raises the possibility of extra work, since out-of-order iteration may require visiting vertices multiple times in order to find the shortest path.

Despite this difference, we stress that both algorithms are based on a similar vertex traversal. Speculative BFS is more precisely seen as Dijkstra with relaxed barrier constraints, and differs from SSSP algorithms such as Bellman-Ford.

5.2 PageRank

PageRank computes the importance (rank) of nodes in a graph. PageRank has a "push" and a "pull" formulation. Both BSP and relaxed-barrier PageRank (asynchronous PageRank) use push and their pseudocodes are shown in Algorithm 3 and 4. The algorithm begins with an initial rank and residue value per node. In Algorithm 3, BSP PageRank is computed by two kernels that are repeatedly called in an outer loop until all vertices converge. The first kernel pushes a fraction of the residue of each vertex on the frontier to its neighbors. The second kernel aggregates all vertices with residue $> \epsilon$ and adds them to the frontier. Having two

frontier to its neighbors. The second kernel aggregates all vertices with residue > ε and adds them to the frontier. Having two separate kernels allows threads to be remapped for more flexible load balancing.

Asynchronous PageRank (Algorithm 4) removes the global synchronization and fuses the two kernels from the BSP implementation together. In asynchronous PageRank, CUDA workers pop vertices from the queue asynchronously, push their residue to their neighbors, then exclusively reserve check_size number of vertices and push those vertices onto the queue if their residue ≥ ε.

PageRank is an iterative unordered algorithm as it is essentially a random walk. In theory, the BSP global barrier is not necessary or beneficial. In our experiments, BSP PageRank does not converge faster than asynchronous PageRank.

5.3 Graph Coloring

Graph coloring is usually implemented using either an independent-set-based approach or a greedy approach. In this paper, *both* the BSP and relaxed barrier graph coloring use a speculative greedy graph coloring algorithm [13] shown in Algorithms 5 and 6. In Algorithm 5, the BSP implementation uses a double buffer and consists of two kernels that are called in the outer loop until all vertices are appropriately colored. The first kernel assigns colors to vertices from in_frontier while avoiding color conflicts with their neighbors. The second kernel aggregates each vertex that, after the first kernel's color assignment, still has a color conflict with a neighbor, and adds it to out_frontier. The speculative part of this algorithm is in the first kernel: it allows a vertex to be colored using the outdated color information from its neighbors and if the vertex color assignment fails (and is detected in the second kernel), that vertex is re-added to the frontier for recoloring.

In contrast, in Algorithm 6, the relaxed barrier version fuses the two kernels by an uberkernel [15]; we use the sign of the vertex ID to distinguish between the color assignment task and the conflict detection task. Specifically, if we pop a vertex with positive ID, we perform color assignment (lines 8–11 of Algorithm 6, identical to lines 11–14 of Algorithm 5), and if we pop a vertex with negative ID, we perform conflict detection (lines 16–18 of Algorithm 6, identical to lines 21–23 of Algorithm 5). Thus, Algorithm 6 is almost identical to Algorithm 5; the only difference is that Algorithm 6 does not enforce a global barrier between the color assignment and conflict detection kernels.

Algorithm 1 Bulk Synchronous BFS

```
# Initialization
Graph G
in_frontier = [source]
for vertex in G:
    vertex.dist = MAX_UINT32
# Start BFS
while not in_frontier.empty():
    out_frontier = []
    ------------CUDA_KERNEL-------------
    for vertex in in_frontier:
        for neighbor in vertex.neighbors:
            neighbor_dist=atomicMin(&neighbor.dist, vertex.dist+1)
            if vertex.dist + 1 < neighbor_dist:
                out_frontier.append(neighbor)
    ------------------------------------
    cudaDeviceSynchronize()
    in_frontier = out_frontier
```

Algorithm 2 Speculative BFS

```
#Same initialization as Algorithm 1
#Start BFS
------------CUDA_KERNEL-------------
while not frontier.empty():
    vertex = frontier.pop()
    for neighbor in vertex.neighbors:
        neighber_dist=atomicMin(&neighbor.dist, vertex.dist+1)
        if vertex.dist + 1 < neighbor_dist:
            frontier.append(neighbor)
------------------------------------
```

Algorithm 3 Bulk Synchronous PageRank

```
# Initialization
Graph G
rank[G.total_vertices] = {1-lambda}
residue[G.total_vertices] = {0}
for vertex in G.all_vertices():
    atomicAdd(residue+vertex, (1-lambda)/lambda*vertex.neighborLen)
    frontier.append(vertex)
# Start PageRank
while not frontier.empty():
    ------------CUDA_KERNEL-------------
    for vertex in frontier:
        res = atomicExch(residue+vertex,0)
        rank[vertex] = rank[vertex]+res
        res = res*lambda/vertex.neighborLen
        for neighbor in vertex.neighbors:
            atomicAdd(residue+neighbor, res)
    ------------------------------------
    cudaDeviceSynchronize()
    frontier = []
    ------------CUDA_KERNEL-------------
    for vertex in G.all_vertices():
        if residue[vertex] > epsilon:
            frontier.append(vertex)
    ------------------------------------
    cudaDeviceSynchronize()
```

Algorithm 4 Asynchronous PageRank

```
#Same initialization as Algorithm 3
#Start PageRank
------------CUDA_KERNEL-------------
while not frontier.empty():
    vertex = frontier.pop()
    res = atomicExch(residue+vertex,0)
    atomicAdd(rank+vertex, res)
    res = res*lambda/vertex.neighborLen
    for neighbor in vertex.neighbors:
        atomicAdd(residue+neighbor, res)
    check_start = atomicAdd(check, Check_Size)
    for check_id in check_start+Range[0,Check_Size):
        if residue[check_id%G.total_vertices] > epsilon:
            frontier.append(check_id%G.total_vertices)
------------------------------------
```

The above three graph algorithms show different approaches to managing asynchrony that can all be addressed within the Atos framework: (1) PageRank: a robust asynchronous algorithm; (2) BFS: benign speculation at the cost of extra work; (3) Graph Coloring: explicit correction for misspeculation.

6 EXPERIMENTS AND ANALYSIS

All experiments in this paper are run on a Linux workstation with 2.20 GHz Intel(R) hyper-threaded E5-2698 v4 Xeon(R) CPUs, 128 GB of main memory, and an NVIDIA V100 GPU with 32 GB on-board memory. All programs were compiled with NVIDIA's nvcc compiler (version 11.1.168) with the -O3 flag and gcc 9.3.0 with the -O3 flag. All results ignore transfer time and are averaged over 20 runs.

Algorithm 5 Bulk Synchronous speculative Graph Coloring

```
# Initialization
Graph G
in_frontier = G.all_vertices()
for vertex in in_frontier:
    vertex.color = -1
# Start graph coloring
while not in_frontier.empty():
  # Assign each vertex a color different from neighbors'
  ---------------------- CUDA_KERNEL ----------------------
  for vertex in in_frontier:
    forbidden[G.max_degree] = {0}
    for neighbor in vertex.neighbors:
      forbidden[neighbor.color] = 1
    vertex.color = find_min_color(forbidden[color] == false)
  ----------------------------------------------------------
  cudaDeviceSynchronize()
  # Collect vertices whose colors collide with neighbors'
  out_frontier = []
  ---------------------- CUDA_KERNEL ----------------------
  for vertex in in_frontier:
    for neighbor in vertex.neighbors:
      if vertex.color == neighbor.color:
        out_frontier.append(vertex)
  ----------------------------------------------------------
  cudaDeviceSynchronize()
  in_frontier = out_frontier
```

Algorithm 6 Asynchronous speculative Graph Coloring

```
#Same initialization as Algorithm 5
#Start graph coloring
---------------------- CUDA_KERNEL ----------------------
while not frontier.empty():
  # Assign each vertex a color different from neighbors'
  vertex = frontier.pop()
  if vertex > 0:
    forbidden[G.max_degree] = {0}
    for neighbor in vertex.neighbors:
      forbidden[neighbor.color] = 1
    vertex.color = find_min_color(forbidden[color] == false)
    frontier.append(-1*vertex)
  # push vertices whose colors collide with neighbors'
  else if vertex < 0:
    vertex = -1 * vertex
    for neighbor in vertex.neighbors:
      if vertex.color == neighbor.color:
        frontier.append(vertex)
----------------------------------------------------------
```

6.1 Experimental Overview

We evaluate the design principles discussed in Section 3 using three implementation variants based on a combination of Atos configurations. (1) "persist-32" utilizes persistent kernels with *warp-sized* workers. It has no data-parallel load balancing within a worker, instead only using implicit task-parallel load balancing. (2) "persist-worker_size-FETCH_SIZE" utilizes persistent kernels with *CTA-sized* workers. (3) "discrete-worker_size-FETCH_SIZE" utilizes discrete kernels and *CTA-sized* workers. Both CTA variants use load balancing search [21] (a data-parallel load balancing technique) inside workers in conjunction with implicit task-parallel load balancing.

For BFS and PageRank, we compare the performance of our implementations to Gunrock [28], a state-of-the-art single-GPU BSP-based graph framework which extensively uses data-parallel load-balancing techniques. For Graph Coloring, Gunrock's independent-set graph coloring algorithm is not comparable, so we faithfully implemented a BSP graph coloring using the same speculative greedy graph coloring algorithm. Our BSP implementation uses Gunrock's bucket-based data-parallel load balancing method [28], described in Section 3.3.

Table 1: Runtime with speedup comparing to BSP in the paraphrase of BSP and three Atos implementations. Graph types are s (scale-free) and m (mesh-like). Runtime unit: ms

Application: BFS (BSP=Gunrock)

Dataset	BSP	persist warp	persist CTA	discrete CTA
soc-LiveJournal1^s	15.3	22.3 (x0.68)	12.4 (x1.23)	10.7 (x1.42)
hollywood_2009^s	9.26	12.2 (x0.75)	6.23 (x1.48)	4.56 (x2.02)
indochina_2004^s	13.2	15.6 (x0.84)	8.03 (x1.65)	7.42 (x1.79)
road_usa^m	604	327 (x1.84)	46.9 (x12.8)	174 (x3.46)
roadNet_ca^m	55.9	39.6 (x1.41)	4.35 (x12.8)	15.5 (x3.58)

Application: PageRank (BSP=Gunrock)

Dataset	BSP	persist warp	persist CTA	discrete CTA
soc-LiveJournal1^s	262	156 (x1.68)	113 (x2.31)	116 (x2.25)
hollywood_2009^s	87.1	80.0 (x1.08)	68.5 (x1.27)	72.4 (x1.20)
indochina_2004^s	159	84.7 (x1.88)	52.6 (x3.02)	49.6 (x3.20)
road_usa^m	221	169 (x1.30)	121 (x1.81)	112 (x1.95)
roadNet_ca^m	20.5	16.2 (x1.26)	10.1 (x2.03)	8.28 (x2.47)

Application: Graph Coloring

Dataset	BSP	persist warp	persist CTA	discrete warp
soc-LiveJournal1^s	96.5	20.4 (x4.71)	36.1 (x2.67)	63.2 (x1.52)
hollywood_2009^s	77.9	31.9 (x2.4)	59.3 (x1.31)	274 (x0.28)
indochina_2004^s	673	74.1 (x9.08)	184 (x3.65)	2073 (x0.32)
road_usa^m	38.2	51.4 (x0.74)	19.3 (x1.97)	81.9 (x0.46)
roadNet_ca^m	9.11	4.18 (x2.18)	3.52 (x2.58)	12.0 (x0.75)

Table 2: Summary of datasets used in our experiments. Graph types are s (scale-free) and m (mesh-like)

Dataset	Vertices	Edges	Diam.	Max. indeg.	Max. outdeg.	Avg. degree
soc-LiveJournal1^s	4.8M	68M	20	13,905	20,292	14
hollywood_2009^s	1.1M	11M	11	11,467	11,467	105
indochina_2004^s	7.4M	191M	26	256,425	6,984	8
road_usa^m	23.9M	57M	6,809	9	9	2
roadNet_ca^m	1.9M	5M	849	12	12	2

We run the three case studies on three scale-free and two mesh-like datasets (Table 2) and summarize the runtime (speedup) results for four implementations in Table 1.

6.2 Performance Challenges in Three Study Cases

Each of our study cases embodies a subset of the BSP performance challenges discussed in Section 2.1, which informs our choice of design decisions in the Atos framework. We summarize the BSP performance challenges in Table 3.

Load Imbalance Problem: All three algorithms involve iterating over a vertex's neighbor list, thus variance in the vertices' degrees leads to load imbalance. This issue is much more severe on scale-free datasets where vertex degree variance is high. In contrast, mesh-like graphs have low maximum degree, and hence low degree variance (see Table 2).

Small Frontier Problem: In Figures 1, 2, and 3, we plot throughput against timeline for each BSP implementation of the three algorithms. Low throughput over a long duration of time indicates the presence of the small frontier problem.

BFS: In Figure 1, Gunrock has high throughput on scale-free datasets, and thus does not have the small frontier problem. This is because scale-free datasets have low diameter (leading to a small number of BSP iterations) and high average degree (leading to a large amount of work per iteration). In contrast, Gunrock on

Table 3: Summary of performance challenges for each case study.

	BFS	PageRank	Graph Coloring
Scale-Free	Load Imbalance	Load Imbalance	Load Imbalance + Small Frontier
Mesh-Like	Small Frontier	None	None

Table 4: Upper: Workload ratio of three Atos implementations relative to Gunrock's implementations for BFS and PageRank. A workload ratio of n means that our implementation does n times as much work as Gunrock. Lower: the workload ratio of four implementations relative to the input graph's total vertex count for graph coloring.

Dataset	Application: BFS			Application: PageRank		
	persist warp	persist CTA	discrete CTA	persist warp	persist CTA	discrete CTA
soc-LiveJournal1^s	1.43	1.06	1.01	0.73	0.72	0.72
hollywood_2009^s	2.26	1.19	1.07	1.08	1.18	0.9
indochina_2004^s	1.28	1.00	1.00	0.76	0.73	0.75
road_usa^m	3.56	1.05	1.04	0.79	0.79	0.92
roadNet_ca^m	2.05	1.02	1.04	1.18	1.11	0.97

Dataset	Application: Graph Coloring			
	BSP	persist warp	persist CTA	discrete warp
soc-LiveJournal1^s	1.17	1.00	1.74	2.78
hollywood_2009^s	3.31	1.15	5.24	37.34
indochina_2004^s	1.96	1.04	4.45	16.97
road_usa^m	1.22	1.00	1.46	1.41
roadNet_ca^m	2.55	1.00	1.74	2.44

mesh-like datasets *does exhibit* the small frontier problem, because these datasets have high diameters and small average degree; consequently, there is a large number of iterations, with little work per iteration, leading to low throughput over many iterations.

PageRank: In Figure 2, for Gunrock PageRank, both scale-free and mesh-like datasets do not exhibit the small frontier problem, as they have high throughput over most of the execution time and converge in fewer than 35 iterations (though Indochina-2004 exhibits a long flat tail in the latter half of execution).

Graph Coloring: In Figure 3, for BSP graph coloring, scale free datasets have low throughput for more than 70% of execution time, and thus have the small frontier problem. Mesh-like datasets terminate in fewer than 40 iterations, and have short tails, and thus do not have the small frontier problem. This is because on scale-free datasets, the high-degree vertices will need to be recolored many times, leading to a large number of iterations, during which the frontier contains a few high-degree vertices with color conflicts. In contrast, mesh-like datasets have low average degree, and are less likely to have color conflicts.

6.3 Relaxing Barriers

As discussed in Sections 2–3, relaxing barriers exposes more concurrency, giving higher throughput and shorter execution time. However, relaxing barriers may result in extra work. If the performance improvement from increased concurrency outweighs the cost of extra work, we obtain a net performance gain.

There are two key factors influencing this tradeoff. First, we find that in the presence of a small frontier problem, the increase in concurrency from relaxing barriers is always more significant than the cost of extra work. Second, on naturally unordered algorithms such

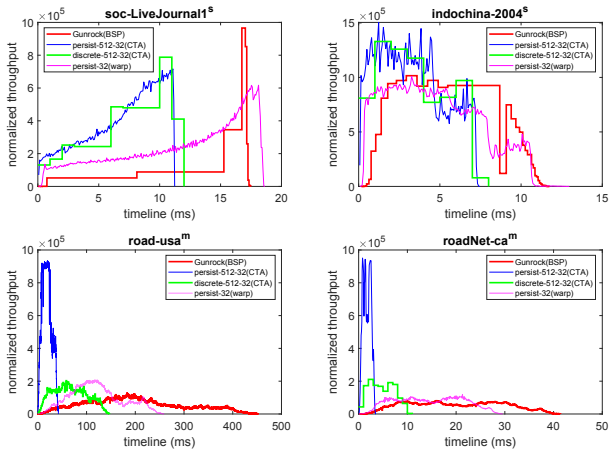

Figure 1: Normalized throughput vs. time on BFS. The top charts are scale-free; bottom charts are mesh-like.

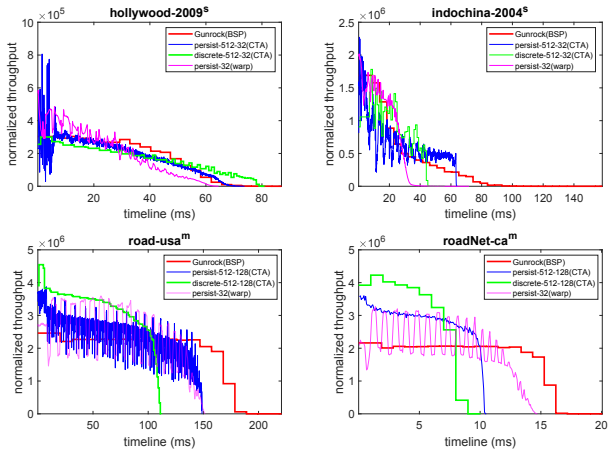

Figure 2: Normalized throughput vs. time on PageRank.

as PageRank, one can always relax the barrier: although the barrier gives the BSP implementation a more predictable convergence rate, the barrier generally does not make the convergence faster.

Table 4 summarizes the extra work for three study cases. Figures 1, 2 and 3 plot the throughput of four implementations (BSP + three Atos variants) of three study cases against timeline for four datasets. Notably, these plots show the *normalized throughput*, which is the measured throughput divided by the overwork factors in Table 4. This gives a fair measure of overall performance, as it incorporates both the benefits of improved concurrency (higher absolute throughput) and the cost of extra work. Essentially, normalized throughput measures "useful" throughput rather than raw absolute throughput. We provide detailed analysis below for each application.

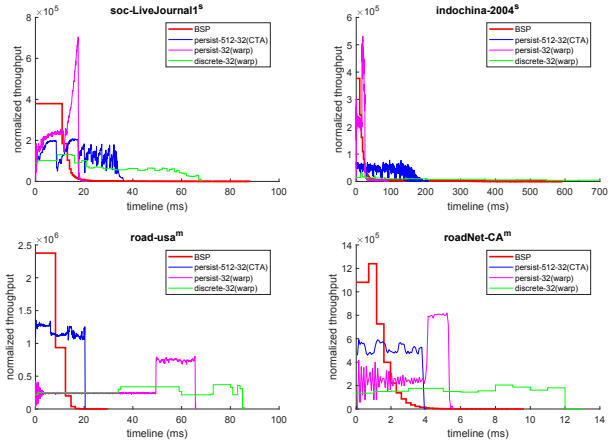

Figure 3: Normalized throughput vs. time on graph coloring.

BFS: Figure 1 shows that for the two mesh-like datasets, all 3 Atos implementations achieve considerably higher normalized throughput than Gunrock. Why? Table 3 shows that Gunrock on mesh-like datasets has a severe small frontier problem. Therefore, the increase in concurrency in the 3 Atos implementations offers a significant performance advantage. Table 4 indicates that the persistent-warp implementation generates 3.5x extra work vs. Gunrock, but despite this extra work, Atos's normalized throughput is still significantly higher than Gunrock. Scale-free graphs, on the other hand, exhibit more parallelism and do not suffer from the small-frontier problem. Atos's fastest implementations are still faster than Gunrock's, but not nearly as much as for the mesh networks.

On all BFS experiments, Atos's CTA implementations are faster than its warp ones. Atos's CTA implementations use a combination of task-parallel and data-parallel load balancing techniques (see Section 6.4 for details), and thus have better load balancing than its warp implementations, which only use task-parallel load balancing. This leads to higher GPU utilization and hence higher absolute throughput. Second, CTA implementations produce less extra work than warp (see Table 4). Due to better load balancing in CTA, the workload of each worker has lower variance. If a worker receives too much work, there will be a long delay before the vertices' updated depths are visible to other workers; this increases the likelihood that downstream vertices are first reached via other sub-optimal paths, which leads to extra work.

PageRank: Unlike BFS, PageRank is naturally unordered, as it satisfies Dijkstra's don't care non-determinism [11]. Therefore, relaxing the barrier in the outer loop does not generate any misspeculations and hence results in no wasted work. In fact, Table 4 shows that the Atos implementations perform *less* work than Gunrock in general. This is because the BSP barrier forces each vertex to be processed at most once per iteration. By relaxing this barrier, the Atos implementations can update certain important vertices (e.g., vertices with high centrality) more frequently than other vertices, thus leading to more efficient propagation of rank.

Figure 2 shows that all three Atos implementations compact the workload and process it with higher normalized throughput (persist-CTA has a higher profiling cost). Though PageRank does not suffer from the small frontier problem, the three Atos implementations nonetheless have superior performance over Gunrock, because relaxing the barrier increases concurrency. In addition, relaxing the barrier lowered the overall workload in practice, even though in theory it may lead to a more unpredictable convergence rate.

Graph Coloring: Unlike BFS and PageRank, all graph coloring implementations (including BSP) use a speculative approach (greedy graph coloring) and thus all have extra work. Table 4 summarizes the multiplicative factor of extra work, which is defined as a ratio vs. the number of vertices in the graph (the lowest possible workload). Atos's persist warp has the least extra work; on some datasets, the extra work is less than 1%, which means after the first color assignment, only 1% of vertices have a color conflict and must be recolored. Atos's discrete warp has the most extra work (on hollywood-2009, 37.34x). The extra work is due to the combination of two factors:

1. Conflicts tend to arise when neighboring vertices are colored concurrently: From Section 5.3, given a vertex, the algorithm first checks its neighbors' colors, then assigns a color to the vertex that does not conflict with its neighbors. The color assignment is speculative because it is done using possibly outdated color information from the vertex's neighbors. When neighboring vertices are colored simultaneously, they read outdated colors from each other, leading to conflicts and recoloring.

2. Consecutive vertices on the work queue are likely to be neighbors: On many if not most graphs, the vertex ID is semantically meaningful: vertices whose vertex ID are numerically close are more likely to be neighbors. At the beginning of graph coloring, all vertices are initially inserted onto the work queue in order of vertex ID.

Since consecutive vertices on the work queue tend to be assigned colors concurrently, the above implies a high likelihood of color conflicts. We verify that the large amount of extra work is indeed due to semantically meaningful vertex IDs: running the exact same experiment with randomly permuted vertex IDs, the amount of extra work drops to less than 1.5x for all four implementations on all datasets. ID permutation leads to the following runtime improvements (in ms) on scale-free datasets:

Impl.	soc-LiveJournal1	hollywood	indochina
discrete-warp	63 → 31	274 → 26	2073 → 222
persist-CTA	36 → 21	59 → 28	184 → 50
BSP	96 → 89	77 → 61	673 → 485

The BSP implementation has a more modest improvement because BSP's thread-warp-CTA load balancing scheme [21] already divides each bucket into three individually-load-balanced subbuckets, reducing inter-bucket conflicts. Persist-warp has little change as there is almost no extra work even before permutation. Notably, after permutation, all three Atos variants are faster than BSP implementation on scale-free datasets.

Comparing persist-warp and persist-CTA: persist-CTA has better load balancing, allowing for more (potentially adjacent) vertices to be colored simultaneously, resulting in more extra work than persist-warp. We verify this from Table 4. Roughly speaking, the amount of extra work for persist-CTA is more significant on

scale-free graphs, as a vertex can have a large number of neigl bors, leading to more potential conflicts. Therefore, persist-CT outperforms persist-warp on mesh-like graphs, where the increase concurrency and better load-balancing outweigh the cost of waste work; conversely, on scale-free datasets, persist-CTA is slower tha persist-warp, because the cost of extra work is too high (see Tables and 4).

Comparing persist-warp and discrete-warp: Discrete-war has more extra work than persist-warp, hurting its performanc for two reasons.

(1) The scheduling policies of discrete and persistent kernel are different. When kernels are launched from the CPU (discret kernel strategy), the kernel launched earlier always has a highe scheduling (hardware) priority than the kernel launched later. This effectively causes vertices to always be colored in roughly the same order as their initial ordering (by vertex ID, which causes many conflicts). In contrast, the persistent kernel only incurs one kernel launch and warps within it are scheduled by the hardware scheduler, whose decisions are much less ordered by vertex ID. Thus persist-warp has fewer coloring conflicts caused by adjacencies and hence less overwork.

(2) Discrete-warp has lower register usage than persistent-warp (72 vs. 42), so persist-warp only achieves 43% occupancy per SM and discrete-warp achieves 62%. Therefore the discrete-wrap assigns colors to more vertices simultaneously, leading to a greater likelihood of conflicts than persist-warp. Unlike our other applications, in graph coloring, the cost of extra work largely reduces the benefit of increased concurrency. On scale-free datasets (without random permutation), our highest performance is achieved with lower concurrency and less overwork (persist-warp variant), which achieves a lower absolute throughput but a higher normalized throughput (and hence higher performance overall).

6.4 Worker Size and Trade-off between Task- and Data-Parallelism Load Balancing

As discussed in Section 3, Atos enables the user to trade off between task-parallelism and data-parallelism load balancing by adjusting the worker size and FETCH_SIZE. Atos's persist-CTA, like persist-warp, uses a persistent kernel to exploit task parallelism, but now the task-parallel work units are fewer and larger (the size of a CTA) and we can leverage more data parallelism within a CTA. In most cases, persist-CTA outperforms persist-warp with both higher normalized/absolute throughput, except for the graph coloring on scale-free datasets, where it achieves only higher absolute throughput. Figure 4 illustrates this tradeoff for BFS and PageRank on soc-LiveJournal (scale-free) and road_usa (mesh-like). We exclude graph coloring because it can only be run with one CTA size, due to high register usage (72) and high shared memory usage (46 KB).

6.5 Kernel Strategy

From Section 3, the chief advantage of the persistent kernel is removing the overhead associated with kernel invocation, which is most significant for fine-grained tasks that involve many small kernel launches. Based on the performance results in Table 1 and

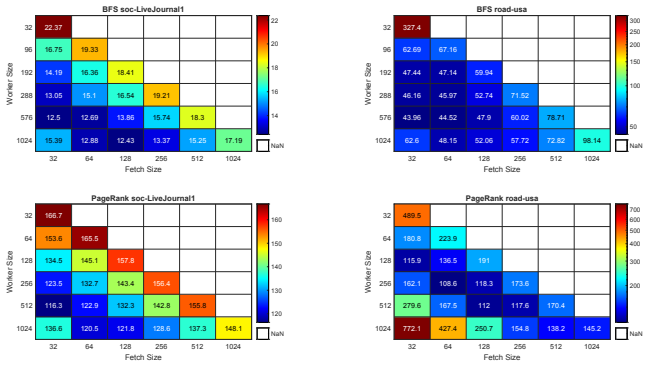

Figure 4: Runtime (ms) heatmap plotted with different worker size and fetch size for BFS and PageRank on soc-LiveJournal^{s} and road_usa^{m}. Note only the lower triangle is valid.

Figure 1, the performance gap between persistent kernel and discrete kernel is particularly large for BFS on mesh-like graphs, which require many small kernel launches due to the high diameter and small workload per iteration. Graph coloring on indochina-2004 also shows a large kernel launch overhead. Using a random permutation of vertex IDs (see Section 6.3), Atos's persistent variant is 4.3x faster than its discrete variant.

7 CONCLUSION

In this paper, we present our task-parallel GPU dynamic scheduling framework, Atos, and analyze its performance across numerous design parameters on three case studies. Our analysis provides the following guidelines on what applications are suitable to run in a capable task-parallel framework, as well as what Atos configurations to use, given an application's characteristics:

(1) If the dynamic application either exhibits the small frontier problem or has load imbalance, Atos will have a performance advantage.
(2) If the application exhibits the small frontier problem, it should be run with a persistent kernel.
(3) If the application exhibits load imbalance, it should be run with both task- and data-parallelism load balancing in tandem to achieve better performance. For different applications, the optimal tradeoff point varies.
(4) By relaxing the outer loop dependency in the application, Atos increases concurrency at the cost of extra work due to mis-speculation, or less predictable convergence rates. The optimal tradeoff between the increased concurrency and additional cost is application-dependent. When an application is naturally unordered (e.g., PageRank) or has the small frontier problem (e.g., BFS on mesh-like datasets and graph coloring on scale-free datasets), the increased concurrency usually outweighs the cost. Conversely, on problems such as BFS on scale-free graphs or graph coloring on mesh-like graphs, the cost of extra work can hurt performance. The best way to reduce extra work is application-dependent and

may include better load balancing (e.g., BFS) or reducing concurrency (e.g., graph coloring).

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation (NSF) under projects CCF-1823034, CCF-1823037, and OAC-1740333; by the Department of Defense Advanced Research Projects Agency (DARPA) under projects HR0011-18-3-0007 and FA8650-18-2-7835; by an NVIDIA gift and hardware donations; and by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. 2010. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. IEEE, 291–298. https://doi.org/10.1109/ICPADS.2010.129

[2] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11. https://doi.org/10.1109/SC.2012.71

[3] Sean Baxter. 2013. moderngpu: Load-Balancing Search. https://moderngpu.github.io/loadbalance.html. Accessed: 2022-07-29.

[4] Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Laxmi N. Bhuyan. 2018. Juggler: A Dependence-Aware Task-Based Execution Framework for GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 54–67. https://doi.org/10.1145/3178487.3178492

[5] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. 235–248. https://doi.org/10.1145/3018743.3018756

[6] Guy E. Blelloch and Gary W. Sabot. 1990. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *J. Parallel and Distrib. Comput.* 8, 2 (Feb. 1990), 119–134. https://doi.org/10.1016/0743-7315(90)90087-6

[7] Daniel Cederman and Philippas Tsigas. 2008. On Dynamic Load-Balancing on Graphics Processors. In *Graphics Hardware (GH '08)*. 57–64. https://doi.org/10.2312/EGGH/EGGH08/057-064

[8] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R Gao. 2010. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2010)*. IEEE. https://doi.org/10.1109/IPDPS.2010.5470413

[9] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single Source Shortest Paths. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*. 349–359. https://doi.org/10.1109/IPDPS.2014.45

[10] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)* 45, 4 (2019), 1–25. https://doi.org/10.1145/3322125

[11] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Pearson.

[12] Alex Fender, Brad Rees, and Joe Eaton. 2022. RAPIDS cuGraph. In *Massive Graph Analytics*. Chapman and Hall/CRC, Chapter 17, 483–493. https://doi.org/10.1201/9781003033707-22

[13] Assefaw Hadish Gebremedhin and Fredrik Manne. 2000. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience* 12, 12 (Nov. 2000), 1131–1146. https://doi.org/10.1002/1096-9128(200010)12:12<1131::AID-CPE528>3.0.CO;2-2

[14] Kshitij Gupta, Jeff Stuart, and John D. Owens. 2012. A Study of Persistent Threads Style GPU Programming for GPGPU Workloads. In *Proceedings of Innovative Parallel Computing (InPar '12)*. https://doi.org/10.1109/InPar.2012.6339596

[15] Shawn Hargreaves. 2004. Generating Shaders from HLSL Fragments. In *ShaderX3: Advanced Rendering with DirectX and OpenGL*. Chapter 7.3, 555–568.

[16] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. Unordered: a Comparison of Parallelism and Work-efficiency in Irregular Algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. 3–12. https://doi.org/10.1145/1941553.1941557

[17] Vishwesh Jatala, Roshan Dathathri, Gurbinder Gill, Loc Hoang, V. Krishna Nandivada, and Keshav Pingali. 2020. A Study of Graph Analytics for Massive Datasets on Distributed Multi-GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 84–94. https://doi.org/10.1109/IPDPS47924.2020.00019

[18] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A Distributed Multi-GPU System for Fast Graph Processing. *Proc. VLDB Endow.* 11, 3 (Nov. 2017), 297–310. https://doi.org/10.14778/3157794.3157799

[19] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 211–222. https://doi.org/10.1145/1250734.1250759

[20] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 411–428. https://www.usenix.org/conference/atc19/presentation/liu-hang

[21] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. 117–128. https://doi.org/10.1145/2145816.2145832

[22] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 233–248. https://doi.org/10.1145/2043556.2043579

[23] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU. *ACM Transactions on Graphics* 33, 6, Article 228 (Nov. 2014), 11 pages. https://doi.org/10.1145/2661229.2661250

[24] Gunrock team. 2017. Throughput vs. Frontier Size. https://gunrock.github.io/docs/#/analysis/frontier_size. Accessed: 2022-07-29.

[25] Stanley Tzeng, Brandon Lloyd, and John D. Owens. 2012. A GPU Task-Parallel Model with Dependency Resolution. *IEEE Computer* 45, 8 (Aug. 2012), 34–41. https://doi.org/10.1109/MC.2012.255

[26] Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of High Performance Graphics (HPG '10)*. 29–37. https://doi.org/10.2312/EGGH/HPG10/029-037

[27] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[28] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49. https://doi.org/10.1145/3108140

[29] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. 221–234. https://doi.org/10.1145/3018743.3018754

[30] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552. https://doi.org/10.1109/TPDS.2013.111