Title

A Programming Model for GPU Load Balancing

ISBN

9798400700156

Authors

Osama, Muhammad

Porumbescu, Serban D

Owens, John D

Publication Date

2023-02-25

DOI

10.1145/3572848.3577434

A Programming Model for GPU Load Balancing

Muhammad Osama
mosama@ucdavis.edu
University of California, Davis
Davis, California, USA

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

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

Abstract

We propose a GPU fine-grained load-balancing abstraction that decouples load balancing from work processing and aims to support both static and dynamic schedules with a programmable interface to implement new load-balancing schedules. Prior to our work, the only way to unleash the GPU's potential on irregular problems has been to workload-balance through application-specific, tightly coupled load-balancing techniques.

With our open-source framework for load-balancing, we hope to improve programmers' productivity when developing irregular-parallel algorithms on the GPU, and also improve the overall performance characteristics for such applications by allowing a quick path to experimentation with a variety of existing load-balancing techniques. Consequently, we also hope that by separating the concerns of load-balancing from work processing within our abstraction, managing and extending existing code to future architectures becomes easier.

CCS Concepts: • **Computing methodologies** → *Shared memory algorithms.*

Keywords: load balancing, sparse computation, GPU, scheduling

1 Introduction

Graphical Processing Units (GPUs) excel at and are often designed for regular fine-grained parallel problems, such as General Matrix Multiplication (GEMM). In regular problems like GEMM, neighboring threads have similar or identical workloads and often achieve nearly 100% of peak GPU theoretical performance. What is much more challenging is an application with ample *fine-grained* parallelism but *irregular* parallelism. In such applications, neighboring threads running in a lockstep fashion will have different workloads—perhaps different amounts of work—making an efficient implementation on a highly parallel machine like a GPU a significant challenge.

Consider Sparse-Matrix Vector Multiplication (SpMV), with a sparse matrix $\mathbf{A}$ and a dense vector x as inputs. SpMV computes the output vector $y = \mathbf{A}x$ and is an example of irregular fine-grained parallelism. Unlike in GEMM, the sparse matrix in SpMV can contain irregularity within the rows of the matrix: the rows of the matrix can have different numbers of non-zero entries. A simple mapping of one row to each GPU thread can expose this irregularity, where neighboring threads may be assigned different amounts of non-zeros to process, causing threads within the same warp[1] to wait on threads with large amounts of non-zeros. The imbalance created due to this irregularity—specifically, when the work is not equally distributed among the parallel actors, and consequently, some actors are idle while others do more work—is defined as the load-imbalance problem.

Current implementations solve this load-imbalance problem on GPUs using application-specific load-balancing techniques that aim to evenly distribute the work such that each thread gets the same number of work items to achieve maximum performance (for instance, Merrill and Garland's load-balanced SpMV implementation [20]). These load-balancing techniques are often tightly coupled with the application itself. The load-balancing components within these implementations are both complex and often collectively the most significant contributor to the performance of an application. Our work here generalizes today's application-specific load-balancing algorithms into a clean, modular, powerful abstraction that can be applied to many complex irregular workloads.

In the process of building our abstraction, we identified common load-balancing approaches currently deployed within sparse, irregular applications on GPUs: application-specific frameworks such as GraphIt [5], Gunrock [29], and Graph-BLAST [31]; techniques from low-level CUDA libraries such as ModernGPU [3] and CUB [24]; and other hand-coded implementations of load-balancing algorithms within applications such as SpMV/SpMM [10, 14, 20], triangle counting [13, 16], and breadth-first search [6, 21]. We show that

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577434

[1] A CUDA warp is a collection of 32 threads that execute instructions in lockstep. Threads in a warp are divergent-free, and run in a Single Instruction Multiple Thread (SIMT) fashion.

with a simple, intuitive, powerful abstraction, these load-balancing schedules can be extended to support irregular workloads that are more general than the specific problem for which they were designed. We demonstrate this by using sparse-linear-algebra-based load balancing for data-centric graph traversal kernels.

Writing high-performance load-balancing code is complex, in large part because this code must perform many roles. Among other tasks, it must ingest data from a specific data structure, perform user-defined computation on that data, and schedule that computation in a load-balanced way. The key insight in our abstraction is to separate the concerns between workload mapping (the load-balance task) and work execution (the user-defined computation), where we *map* sparse formats (such as Compressed Sparse Row (CSR)) to simple abstraction components called work **atoms**, **tiles**, and **sets**. These fundamental components are expressed as composable C++ ranges and range-based for loops, and are used to build load-balancing schedules. Programmers can then use these APIs to build load-balanced, high-performance applications and primitives. Expressed in this way, we can reconstruct existing application-dependent load-balancing techniques that address irregularity to be more *general*, *portable*, and *programmable*. The contributions of our work are as follows:

1. We present a novel abstraction for irregular-parallel workloads on GPUs. Our abstraction at a high level allows programmers to develop sparse, irregular-parallel algorithms with minimal code while delivering high performance.
2. We design and implement a set of intuitive APIs, available in our open-source GPU load-balancing framework, built on the proposed abstraction using CUDA-C++ ranges and range-based for loops.
3. We show the ease of implementing new load-balancing schedules by implementing a novel cooperative groups-based load-balancing schedule, described in Section 5.2, which is a generalization of previous thread-, warp-, and block-level load-balancing schedules [30].
4. We provide state-of-the-art SpMV performance as a benchmark with a geomean of speedup of 2.7× for the SuiteSparse Matrix Collection [11] over cuSparse's state-of-the-art implementation using simple heuristics and 3 GPU load-balancing schedules.

2 Design Goals

Our programming model focuses on the broad category of fine-grained nested data parallelism. Load-balancing task-level parallelism requires a different approach and is beyond the scope of this work. This section highlights the design goals of our load-balancing abstraction:

Achieve high performance. First and foremost, the goal of our work is to achieve the high performance of existing-load balancing algorithms for irregular applications. Our abstraction cannot come at the cost of significant overhead or performance degradation. We measure our success in achieving high performance by comparing the performance of our abstraction against the performance of existing hardwired implementations.

A composable and programmable interface. Importantly, we do not want to restrict the user to a library interface that takes control of the larger system. Programmers strongly prefer to adopt new software components that fit into their control structures rather than require them to adopt a new control structure. We want to allow the users to (1) maintain control of GPU kernel boundaries (kernel launches), (2) be able to add new load-balancing algorithms, and (3) compose new load-balanced primitives from existing load-balancing APIs. We measure the programmability of our work by comparing the Lines of Code (LOC) of our abstraction against existing implementations and show composability by implementing a new load-balancing algorithm in terms of our existing APIs.

Extensible to new applications. We aim to decouple and extend application-specific load-balancing techniques to new irregular-parallel domains. Our abstraction seeks to promote the reuse of existing load-balancing techniques for new applications. We use SpMV as a benchmark application implemented using three different load-balancing techniques, some of which were previously used to implement parallel graph analytics kernels [5, 6, 10, 29].

Facilitate the exploration of optimizations. A key goal of our abstraction is to facilitate the exploration of optimizations for a given application by switching the underlying load-balancing algorithms used to balance the work. We want to encourage our users to experiment with heuristics and new load-balancing techniques to discover what works best for their application needs. We measure the success of this goal by optimizing SpMV's performance response for a large corpus of sparse matrices across several different load-balancing techniques.

Non-Goals

In addition to the above design goals, we also define our non-goals:

Targeting other parallel architectures. Although we believe the lessons learned should apply to other parallel architectures, we explicitly target NVIDIA's CUDA architecture and programming model [23]. Many components of our abstraction leverage CUDA's compute hierarchy of threads, warps and blocks mapped onto the physical streaming multiprocessors, the oversubscription model of assigning more work than the number of processors to fully saturate the

underlying hardware, and CUDA's Cooperative Groups programming model [18], described in Section 5.2, to achieve high performance.

Multi-GPU support. This work focuses on load-imbalance issues for a single GPU and does not consider multi-GPU single-node or multi-node systems, although these are interesting directions for future work.

3 Our Load-Balancing Abstraction

The key insight behind our GPU load balancing abstraction is the *separation of concerns* between the mapping of the work items to processing units and work execution. We divide our abstraction into three key concepts (illustrated in Figure 1), each of which describes a different aspect of an implementation: (1) defining the work; (2) defining the workload balance across GPU threads, warps or blocks; and (3) defining the work execution and computation per thread on the balanced work. This separation allows us to cleanly divide the work between an application developer and a load-balanced-library developer and facilitates the exploration of optimizations by mixing different load-balancing techniques and sparse-irregular algorithms. Sidebar 1 presents a practical example of the motivation for our load balancing abstraction.

Sidebar 1 A practical example of the existing, predominant approach to load-balancing sparse-irregular workloads.

Consider an SpMV implementation on the GPU provided in the open-source CUDA CUB library [24]. CUB implements and maintains the SpMV algorithm presented in the paper by Merrill and Garland [20]. Merge-based SpMV, explained in detail in Section 5.2.1, is a CSR-based, perfectly load-balanced SpMV, where each thread gets an even share of work, and the amount of work is defined by the total number of matrix rows and the total number of non-zeros, summed. In the reference, this highly efficient, state-of-the-art implementation took 1,100 lines of code (LoC) (or 503 LoC of kernel code) across 3 files (not including a 4th file required for a segmented fixup step of an additional 234 LoC). In contrast, the actual computation of SpMV within this reference implementation is expressed within a *single* for-loop and 4–5 LoC! This disparity between the LoC required to map the work items to processing units in a load-balanced way and the LoC required to express the desired computation is the key motivation behind our work. Additionally, the CUB implementation is specifically dedicated to the SpMV algorithm and would require a significant rewrite to apply it to other algorithms, even within the same computing domain. One such example of this exact rewrite is by Yang et al., who extend merge-path load balancing from SpMV to a Sparse-Matrix Dense-Matrix Multiplication (SpMM) implementation [30]. The load-balancing algorithm in both works is the same but applied to different computations, which motivates the need for reuse.

3.1 Input from Sparse Data Structures

We begin with our input data expressed in some form of sparse data structure. Examples of such data structures include, but are not limited to, Compressed Sparse Row (CSR) and Coordinate (COO) formats. The goal of the first stage of our abstraction is to map the input data format to a common data framework and vocabulary that is the input to the next stage. This vocabulary has three simple components that together express the input data:

1. A **work atom**, a single unit of work that is to be scheduled onto the processors (for example, a non-zero element of a sparse matrix). We assume that all work atoms have an equal cost during execution.
2. A **work tile**, a logical entity represented as a set of work atoms (for example, a row of a sparse matrix). Work tiles may have different costs during execution. As we highlighted in the introduction, work is most *logically* parallelized over work tiles but is often most *efficiently* parallelized over work atoms, and mapping between work tiles and work atoms may be expensive and complex.
3. A **tile set**, a set of work tiles that together comprise the entire working problem (for example, a sparse matrix). In our abstraction, the tiles within a tile set must be independent (and thus can run in parallel across multiple processors).

This mapping between sparse formats and atoms/tiles/tile sets is defined by the user. Though we have not implemented all of them, we believe our mapping abstraction here is flexible enough to express a wide variety of existing sparse data formats in the literature [12] in such a way that they are suitable for load balancing in our abstraction's next stage. As well, we have already included several common sparse formats (CSR, CSC, COO) in our load-balancing library implementation so that users can simply select and use them without having to implement them. Given a mapping to atoms/tiles/tile sets, we can next implement a load-balancing algorithm that can parallelize over work atoms or tiles transparently from the computation's perspective.

3.2 Defining Load Balancing

By expressing workloads through an abstraction that captures work at differing levels of granularity (i.e., tile set, atoms, and tiles), we can more easily distribute computation evenly across the GPU's available resources. Given a user-defined input tile set and associated sequences of atoms and tiles, along with a user-selected partitioning algorithm, our load-balancing stage outputs subsequences of atoms and tiles assigned to processor ids (i.e., where atoms or tiles will be processed).

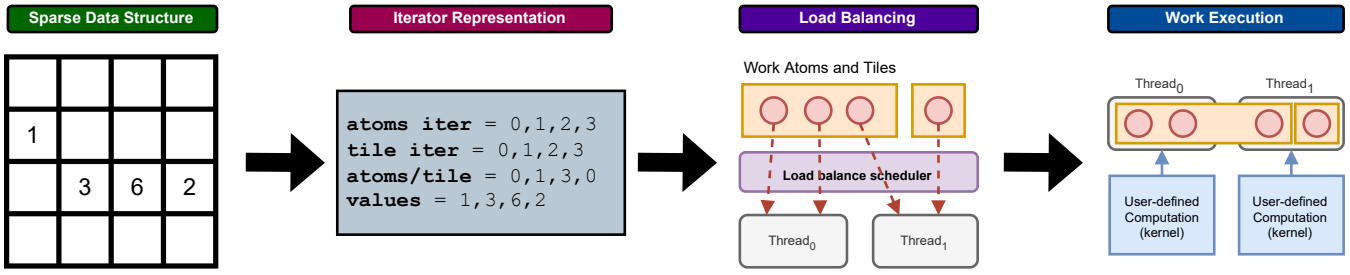

Figure 1. Load balancing as a simple pipeline of the three key concepts of our abstraction: (1) sparse data structures represented as iterators, (2) load-balancing algorithm that partitions the work onto threads, and (3) user-defined computation consuming the balanced work and executing on each thread.

The resulting assignment of subsequences to processor ids is critical to effectively balancing workloads across processing elements and is generally problem- and dataset-specific. The user must specify the necessary sequences. Ideally, an *oracle* would take these sequences and select the most optimal subsequences for every processing element. Finding such an oracle is an open problem and thus we provide the next best thing: the ability for users to choose and experiment from a set of predefined schedules and the ability to implement their own schedules. In general, load-balancing algorithm designers must balance between the cost of scheduling and the benefits from better scheduling. A schedule could be as straightforward as assigning processing elements to tiles with arbitrary numbers of atoms (e.g., rows with an arbitrary number of non-zeros in a sparse matrix) to something more complicated/expensive that takes on a more holistic approach to work (e.g., considering work across multiple rows with a varying number of non-zeros in a sparse matrix).

3.3 Defining Work Execution

The final component of our load-balancing abstraction expresses the irregular-parallel computation itself. The previous stage inputs load-imbalanced work and load-balances it; this stage then consumes that load-balanced work by performing computation on it. The scope of what computation can be expressed is extensive, and is only limited by how the load-balanced work, represented as sequences, can be consumed within a CUDA kernel. Since the framework does not assume control of the kernel, anything you can write in a CUDA kernel will also work in our framework. For instance, programmers can express a mathematical operation performed on each atom or each tile of the work, or build cooperative algorithms that not only consume the work assigned to each thread but also combine the results with neighboring threads to implement more complex algorithms such as parallel reduce or scan. Practical examples that we have implemented in our framework (see Section 4.3 and 5.3) using this abstraction include, but are not limited to, sparse-linear algebra kernels, such as Sparse-Matrix and Sparse-Tensor contractions, and data-centric parallel graph algorithms, such as Single-Source Shortest Path (SSSP) and Breadth-First Search (BFS) built on a neighborhood traversal kernel.

We expect typical users of our library will *only* write their own code for *this* stage of the abstraction and use standard data structures and load-balancing schedules that are already part of our library. However, those users can also implement custom data formats and load-balancing schedules.

4 High-Level Framework Implementation

Our GPU load-balancing framework implements the abstraction described in Section 3 using C++17 and CUDA. In our system, programmers use CUDA/C++ to develop irregular-parallel algorithms and implement new load-balancing schedules. Per our design goals of composable APIs, extensibility, and reuse, this and the following section introduce the implementation details of our API, and how it is used to develop new applications that promote the reuse of high-performance load-balancing techniques available within the framework. We also explore a new load-balancing method (Section 5.2) built on CUDA's Cooperative Groups model. Furthermore, we identify how our work can be used to facilitate the exploration of optimizations for a given application such as SpMV.

4.1 Implementing Sparse Data Structures

Our framework translates sparse data structures (e.g., COO, CSR, CSC) into work atoms, work tiles, and tile sets (Section 3.1) using simple C++ iterators. C++ iterators are objects that point to some element in a range of elements and enable iteration through the elements of that range using a set of operators. For example, a `counting_iterator` is an iterator that represents a pointer into a range of sequential values [1]. Our framework requires the user to define three important iterators using C++: (1) an iterator over all work atoms; (2) an iterator over the work tiles; and (3) an iterator over the number of atoms in each work tile. (Our library already supports several common sparse data structures.) Using these iterators, the load-balancing schedule can then determine and distribute load-balanced work across the

```
1  // Simple iterators for atoms and tiles.
2  counting_iterator<int> atoms_iter(0, nnz);
3  counting_iterator<int> tile_iter(0, rows);
4  // Iterator over the atoms within tile i.
5  auto atoms_per_tile = make_transform_iterator(
6    tile_iter,
7    [tile_iter, row_offsets]
8    __host__ __device__(const int& i) {
9      return (row_offsets[tile_iter[i + 1]] -
10             row_offsets[tile_iter[i]]);
11 });
```

Listing 1. Compressed-Sparse Row (CSR) format expressed within our framework using C++17. The CSR format describes a matrix using three arrays: (1) column indices of nonzero values; (2) the extent of rows (row offsets); and (3) the nonzero values. Since the CSR data structure does not contain arrays that point to indices of atoms and tiles (nonzeros and rows), in the listing above we define atom and tile iterators as simple counting iterators from 0 to the total number of nonzeros (*nnz*) and from 0 to the total rows in the matrix (*rows*), respectively (Lines 2–3). The iterator over the atoms-per-work-tile is expressed using a transform iterator, which computes the expression within a provided function for each tile id. For CSR, this is simply the row offset of the current tile subtracted from the offset of the next tile (Lines 5–11).

underlying hardware. Listing 1 shows how our abstraction expresses the commonly used CSR format as a tile set within our framework.

4.2 Implementing Load-Balancing Schedules

Perhaps the most straightforward schedule is scheduling each work tile onto one GPU thread. This approach is common in the literature and practice [3, 10, 21, 26, 30]; although this strategy is ineffective in the presence of significant load imbalance across tiles, we use it here as an example to illustrate how load balancing is defined within our framework.

The inputs are the three iterators from the last stage plus an atom and tile count. The load-balance algorithm developer, then, implements `tiles()` and `atoms()` procedure calls, which return the C++ range of tiles and atoms to be processed by the current thread, effectively creating a map between assigned processor ids and segments of the workload. Listing 2 shows a complete example of the thread-mapped schedule. Although a simple algorithm, it can deliver high performance for well-balanced workloads with coarse-grained parallelism (a small number of atoms per tile), such as multiplying a sparse vector by a dense vector. Furthermore, our abstraction is not limited to only simple scheduling algorithms, as Section 5.2 provides examples of more complex load-balancing algorithms.

```
1  class schedule_t {
2    // Construct a thread-mapped schedule.
3    __host__ __device__
4    schedule_t(atoms_it_t atoms_it,
5      tiles_it_t tiles_it,
6      atoms_it_t atoms_per_tile_it,
7      size_t num_atoms, size_t num_tiles) :
8      m_atoms_it(atoms_it), m_tiles_it(tiles_it),
9      m_atoms_per_tile_it(atoms_per_tile_it),
10     m_num_atoms(num_atoms),
11     m_num_tiles(num_tiles) {}
12   // Range of tiles to process in "this" thread.
13   // Stride by grid dimension.
14   __host__ __device__ auto tiles() {
15     auto begin = m_tiles_it(blockDim.x * blockIdx.x
16                + threadIdx.x);
17     auto end = m_tiles_it(m_num_tiles);
18     return range(begin, end)
19            .step(gridDim.x * blockDim.x);
20   }
21   // Range of atoms to process in "this" thread.
22   __host__ __device__ auto atoms(
23       const std::size_t& tile) {
24     auto begin = m_atoms_per_tile_it[tile];
25     auto end = m_atoms_per_tile_it[tile + 1];
26     return range(begin, end).step(1);
27   }
28 };
29 using schedule_t = thread_mapped_schedule_t;
```

Listing 2. A thread-mapped load-balancing algorithm expressed as C++ ranges, incorporating the atoms and tiles defined as iterators from Listing 1. Each tile is mapped to a thread, where the thread id corresponds to the index of the tile in the tile set. All atoms within a tile are sequentially processed by the thread. After a tile is processed, a thread is mapped to the next tile, obtained by striding the index by the grid size of the kernel.

4.3 Implementing Work Execution

Our framework is designed to explicitly let the user *own* the kernel launch boundary. Owning a CUDA kernel boundary means that the user is responsible for maintaining and configuring launch parameters and implementing the CUDA kernel used to define the application. Although this design decision comes at a cost of convenience and simplicity, it offers significant flexibility in what users can express through our abstraction. This design decision is motivated by the following reasons. (1) Users are not required to add a complex dependency to their existing workflow/libraries, therefore making code maintenance simpler and more scalable as they do not have to rely on our framework to incorporate new CUDA constructs and features. (2) Users are free to express anything and everything CUDA allows within their kernels while consuming our load-balanced C++ ranges. This allows

for versatility in what can be expressed, as the users can now specify multiple load-balanced work domains, range-based for loops, and even fusing multiple computations to build more complex algorithms within a single kernel. (3) Higher-level APIs can be used to build simpler higher-level abstractions that *do* own the kernel boundary and provide simpler APIs at the cost of flexibility.

As an input to this stage, users consume the load-balanced C++ ranges to implement their computation. This can be done in multiple ways, but one of the most common patterns is a nested range-based for loop that loops over all the assigned tiles and atoms ranges. Listing 3 shows a simple example of a CUDA kernel that implements the SpMV algorithm the using CSR format and thread-mapped load-balancing algorithm described in Listings 1 and 2. In this example, the outer `for` loop within each thread iterates over the assigned rows of the sparse matrix (tiles), and the inner loop sequentially processes the assigned nonzeros (atoms) within each row. In Section 5.3 we implement and discuss more complex kernels and computations.

```
1  // Implements load-balanced SpMV kernel.
2  __global__ void spmv(const size_t rows,
3    const size_t cols, const size_t nnz,
4    const int* offsets, const int* indices,
5    const float* values, const float* x,
6    float* y) {
7    // Configure load-balancing.
8    // Input: iterators defined for CSR format.
9    schedule_t config(
10       atoms_iter, tile_iter,
11       atoms_per_tile_it,
12       nnz, rows);
13   // Consume rows using a range-based for loop.
14   for (auto row : config.tiles()) {
15     type_t sum = 0;
16     // Consume atoms using a range-based for loop.
17     for (auto nz : config.atoms(row))
18       sum += values[nz] * x[indices[nz]];
19     y[row] = sum;
20   }
21 }
22 // Launches SpMV kernel.
23 constexpr size_t blocks = 256;
24 size_t grid = (rows + blocks - 1) / blocks;
25 spmv<<<grid, blocks>>>(rows, cols, nnz,
26   offsets, indices, values, x, y);
```

Listing 3. Sparse-Vector Matrix Multiplication (SpMV) implemented within our load-balancing abstraction using range-based nested `for` loops. The sparse matrix is represented using a CSR-based format, where x is the dense input vector and y is the dense output vector ($y = Ax$). Lines 9–12 use the load-balancing schedule implemented in Listing 2 and the iterators defined in Listing 1 to construct the load-balanced work to be processed. Lines 14 and 17 show the `for` loops within each thread, which iterate over the assigned rows of the sparse matrix and sequentially process the assigned atoms within each row. Line 18 shows the actual computation performed on each work atom (nonzero), and Line 19 writes the result to the dense output vector y.

5 Implementation Details

5.1 Flexible, Composable CUDA-enabled Ranges

The composability of load-balanced primitives and applications using our API is a conscious design choice within our framework supported through the use of CUDA-enabled C++ ranges. Our framework does not *own* the kernel boundary (kernel launch), which forces our APIs to be focused and contained within the kernels. This allows programmers to build and maintain their own kernels while still benefiting from our framework's load-balancing capabilities. This is largely implemented using device-wide C++ functions and classes tagged with CUDA's `__device__` keyword.[2] We implemented and expose several different types of specialized ranges that were particularly useful in implementing load-balanced schedules:

- `step_range`: A range that iterates from `begin` to `end` in steps of `step`. Useful for defining load balancing schedules that require a custom stepping range or process a constant number of work items per thread (which can be defined using `step`).
- `infinite_range`: A range that iterates from `begin` to infinity. Useful for defining load balancing schedules in persistent kernel mode [32], where the kernel persistently runs until all work is consumed or an algorithm has converged.
- `grid_stride_range`: A specialized case of step range that iterates from `begin` to `end` in steps of `step` using the CUDA kernel's grid size. Also supports `block` and `warp` stride variants that iterate in steps of the block or warp size, respectively.

[2]A method decorated with the `__device__` keyword allows the CUDA compiler to generate a device-callable entry point. This allows the code to be called from within kernels [23].

5.2 Implementing Non-Trivial Load-Balancing

As we describe in Section 5.1, we can decouple and express existing load-balancing techniques as a set of C++ ranges. To illustrate the potential of this abstraction, we begin by decoupling and expressing a state-of-the-art load-balancing algorithm known as merge-path [17] previously used for balancing CSR-based SpMV and SpMM [20, 30], and implement three additional load balancing algorithms (warp-, block- and group-mapped), all of which are available in our library for programmers to use. Our new group-mapped algorithm is a tile-per-group-based schedule, where a group is defined

as a collection of threads of any arbitrary size (not limited to a warp or block size). Our group-mapped schedule is a generalization of the tile-per-thread, -warp or -block schedules [5, 21] using CUDA's Cooperative Groups programming model [18].

5.2.1 Merge-path load balancing.

In the language of a sparse matrix, merge-path assumes that each non-zero in the matrix and each new row in the matrix are an equivalent amount of work, then evenly divides nnzs+rows work across the set of worker threads. Each thread then performs a 2-D binary search within the nonzero indices and row offsets of a CSR matrix to find the starting position of the row and nonzero it needs to process. Threads then sequentially process the rows and nonzeros from the starting position until they reach the end of their assigned work [20].

We implement this algorithm as a load-balancing schedule in our abstraction by expressing it in two steps: (1) **Setup:** The initialization step of the C++ schedule class computes the number of work units per thread, conducts a binary search as described above, and stores the starting position of each tile and atom in a thread-local variable. (2) **Ranges:** The second step of the algorithm builds the ranges for each thread to process as "complete" tiles and "partial" tiles [20]. If a thread's atom range lies entirely within one tile, it is "complete", and is processed in a simple nested loop. If a thread's range crosses a tile boundary, the thread processes its work in a separate nested loop.

Because we decouple the load-balancing method (Section 4.2, and above) from work execution (Section 4.3), we can use this merge-path implementation to implement not only SpMV but also any other algorithm whose work can be divided into tiles and atoms, e.g., a graph neighborhood-traversal algorithm used to implement breadth-first search [29]. Just as importantly, the merge-path schedule is now no longer limited to a CSR-based sparse format. Supporting other formats only requires building the necessary slightly more complex iterators that are able to count atoms per tile (the computation that the CSR implementation achieves with the row offsets array in Listing 1).

5.2.2 Warp- and block-level load balancing.

The goal of a warp- or block-level load-balancing schedule is to assign an equal share of tiles to each warp or block, which are then sequentially processed. The work atoms within each tile will be processed in parallel by the available threads within a warp or a block. Each thread strides by the size of the warp or block to process a new work atom until the end of work is reached.

The imbalance across different processing units is left for the hardware scheduler to handle. This scheduler depends on the oversubscription model of CUDA, where the programmer can launch a larger number of warps or blocks than the GPU can physically schedule at any given time. As the processing units finish processing their work, new ones are scheduled from the oversubscribed pool [5, 21].

5.2.3 Group-level load balancing.

Group-level load balancing generalizes warp- and block-level schedules. Instead of requiring that group sizes are the size of a warp or block, as above, this method leverages CUDA's Cooperative Groups (CG) programming model [18] to allow programmer-specified dynamically sized groups of arbitrary size. Within these groups, the CG model permits detailed control of the group's synchronization behaviors as well as simple parallel group-level collectives such as reduce or scan. We leverage this powerful tool to implement a generalized group-level load balancing schedule, effectively giving us the warp- and block-level schedules above for free when the group size equals that of a warp or a block.

Our schedule assigns work tiles to a group, and each group looks at its equal share of tiles and computes the number of atoms for each tile and stores it in a scratchpad memory (CUDA's *shared memory*). The group then performs a parallel prefix-sum, a widely used parallel algorithm that inputs an array and produces a new array where the element at any position is a sum of all previous elements [4]. We use this prefix-sum array for two purposes: (1) the last element of a prefix-sum array indicates the aggregated number of work atoms that a group has to process, and (2) the position of each sum in the prefix-sum array corresponds to the work tile to which those atoms belong. The setup phase of the schedule builds the prefix-sum arrays per group in the scratchpad memory, and the ranged-loop of the schedule returns the atom to process in each thread. The corresponding tile, if needed, is obtained by a simple `get_tile(atom_id)` operation, which executes a binary search within the prefix-sum array to find the tile corresponding to the atom being processed.

Relying on the CG model for this load-balancing schedule has a unique advantage of configuring the group size (effectively software constructs that directly map onto the hardware) per the shape of the problem and the underlying hardware architecture. For example, targeting GPUs where the warp size is not 32 threads (AMD's GPU architecture supports a warp size of 64 [2]) is now possible with a simple compile-time constant, or configuring the group size to perfectly align with the structure of the problem.

5.3 Application Space

Our work definition (Section 3.1), composable APIs (Section 5.1), and multiple sophisticated, high-performance load-balancing schedules (Section 5.2) together provide for a versatile and extensible framework with plenty of room for application-specific optimizations. In Listing 3 we already demonstrated how to implement the SpMV algorithm using

```
// ... Inside the CUDA kernel.
// Loop over all the assigned rows.
for (auto row : config.tiles()) {
  // Loop over all the columns of Matrix B.
  for (auto col : range(size_t(0), B.cols)
    .stride(size_t(1))) { /// < New Loop
    float sum = 0;
    // Loop over all the assigned nonzeros.
    for (auto nz : config.atoms(row))
        sum += values[nz] * B(nz, col);
    // Output the sum to Matrix-C.
    C(row, col) = sum;
  }
}
```

Listing 4. A simple loop wrapped around SpMV introduced in Listing 3 allows us to represent the slightly more complex SpMM load-balanced computation.

our framework. A simple and natural extension is to implement Sparse-Matrix Matrix Multiplication (SpMM). Listing 4 shows the minor change necessary, which adds another loop over the columns of the **B** matrix around the existing code from Listing 3 to implement SpMM. This implementation could also be extended to support Gustavson's General Sparse Matrix-Matrix Multiplication (SpGEMM), using two kernels and an allocation stage; the first kernel would compute the size of the output rows used to allocate the memory for the output sparse matrix and the second kernel would perform the multiply-accumulation.

Beyond sparse linear algebra, we can use our framework to address applications in other domains. Listing 5 implements the graph primitive Single-Source Shortest Path (SSSP) using our group-level load-balancing schedule. SSSP's performance on GPUs is largely gated by good load balancing [5, 29], but if the programmer chooses a load-balancing schedule from our library, the details of load balancing are completely hidden. Moreover, the same schedules that were used in one application domain (e.g., sparse linear algebra) are easily reusable in this different application domain.

6 Evaluation

We aim to show that our framework, built on our load balancing abstraction, enables both high performance and better programmability for sparse-irregular problems. Our evaluation below uses our SpMV implementation as a benchmark against state-of-the-art implementations provided within NVIDIA's (open-source) CUB library and production (closed-source) cuSparse library. We considered (and implemented) several additional applications for evaluation, including SSSP, BFS, and SpMM. We found they led to similar high-level conclusions. Thus our evaluation here focuses on SpMV. Our test corpus consists of approximately the *entire* SuiteSparse Matrix Collection [11] with a broad scope of sparse matrices

```
// ... Inside the CUDA kernel.
// Loop over all the assigned edges to process.
for (auto edge : config.atoms()) {
  auto source = config.get_tile(edge);
  // G is the graph data structure
  auto neighbor = G.get_neighbor(source, edge);
  auto weight = G.get_edge_weight(edge);
  float source_dist = dist[source];
  float neighbor_dist = source_dist + weight;
  // Check if the destination node has been
  // claimed as someone's child.
  float recover_distance =
    atomicMin(&(dist[neighbor]), neighbor_dist);
  // Add the neighbor to the frontier.
  if (neighbor_dist < recover_distance)
    out_frontier[neighbor] = true;
}

// ... Outside the CUDA kernel.
// Loop until the frontier is empty.
```

Listing 5. The parallel single-source shortest path (SSSP) graph primitive expressed using our load-balanced schedule.

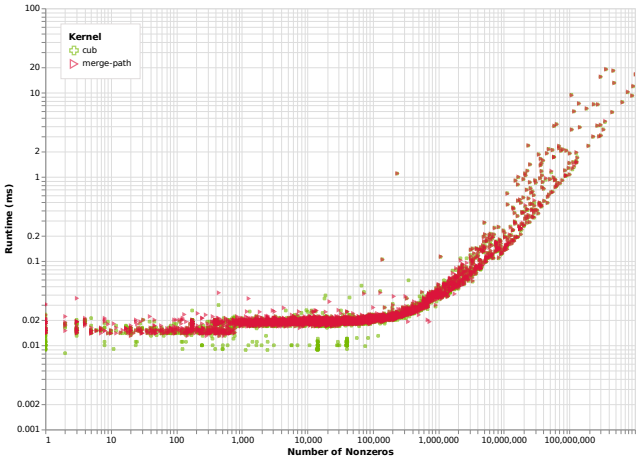

Figure 2. SpMV runtime comparison: our merge-path SpMV implementation vs. CUB across all SuiteSparse datasets. Our runtimes almost perfectly match CUB's for all datasets. The small number of datasets where CUB is faster is due to a simple heuristic that CUB uses for single-column sparse matrices (i.e., a sparse vector).

from many different high-performance computing domains. We ran all experiments on a Ubuntu 20.04 LTS-based workstation with an NVIDIA Tesla V100 GPU and CUDA 11.7.

6.1 Performance Overhead

Our first and foremost goal is to ensure that the elements within our abstraction do not add any additional performance overhead to the existing load balancing techniques and algorithms developed using them. To verify this, we

compare the runtime performance of our SpMV implementation using the merge-path schedule to the implementation provided by NVIDIA's CUB library [24] (also used for Merrill and Garland's merge-path SpMV paper [20]) on the SuiteSparse collection. As previously mentioned, and in contrast to our design, CUB contains a hardwired implementation of the merge-path scheduling algorithms and does not decouple workload balancing from the actual SpMV computation. CUB's approach is not reusable for any other irregular parallel problem without significant changes to the implementation.

Figure 2 plots the number of nonzeros (i.e., the total work) vs. runtime for our work vs. CUB's implementation. Our implementation has minimal performance overhead when using our abstraction: a geomean slowdown of 2.5% vs. CUB, with 92% of datasets achieving at least 90% of CUB's performance. Figure 2 shows our implementation almost perfectly matches CUB for all datasets, except for some datasets with fewer than 100,000 nonzeros. Upon further investigation, we identify that CUB uses a simple heuristic to launch a thread-mapped SpMV kernel where the number of columns of a given input matrix equals 1 (i.e., a sparse vector). Unlike our more general implementation, CUB's simple (but specialized) thread-mapped SpMV kernel has no load-balancing overhead for a perfectly balanced workload such as SpVV computation.

6.2 Improved Performance Response

We also compare our work to NVIDIA's vendor library for sparse computations, cuSparse. Figure 3 shows the performance response of our SpMV implementation using each of our scheduling algorithms individually vs. cuSparse's state-of-the-art implementation. Switching between any of our implementations requires very little code change; in the case of merge-path and thread-mapped, we need only update a single C++ enum (identifier) to select the desired load-balancing schedule.

We then combine our scheduling algorithms into one implementation for SpMV (Figure 4), demonstrating noticeable performance improvements over cuSparse. This is primarily possible due to our ability to quickly experiment with different heuristic schemes with a variety of available load-balancing schedules. Here, we use merge-path unless either the number of rows or columns are less than the threshold α and the nonzeros of a given matrix are less than threshold β (we choose $\alpha = 500$ and $\beta = 10000$ for SuiteSparse). In this case, we use thread-mapped or group-mapped load balancing instead of merge-path. Our system shows a peak performance speedup of 39× and a geomean performance speedup of 2.7× vs. cuSparse.

Our framework not only allows programmers to express computations efficiently and simply (i.e., without worrying about the load-balancing algorithms), but also quickly optimize a given application using a range of scheduling algorithms, both with minor code changes.

Load Balancing Algorithm	NVIDIA/CUB	Our Work
Merge-Path	503	36
Thread-Mapped	22	21
Group-Mapped	N/A	30
Warp-Mapped	N/A	30 (free)
Block-Mapped	N/A	30 (free)

Table 1. Lines of code (LoC) comparison for NVIDIA's CUB library versus our work for SpMV application implemented using merge-path, thread-mapped and group-mapped (warp- and block-mapped use the *exact* same code for group-mapped) load balancing algorithms. We report only non-commented lines of code, formatted using the `clang-format` tool with the Chromium style guide [15], that contributes to the kernel implementation.

6.3 Lines of Code (LOC)

We are able to achieve these performance gains with minimal code complexity. Table 1 shows lines of code (LOC) for our framework when compared to the state-of-the-art open-source implementation of merge-path and thread-mapped within NVIDIA's CUB library. We deliver the same performance results as highlighted in the previous sections with 14× and 1× fewer lines of code for merge-path and thread-mapped scheduling algorithms, respectively. Using our merge-path implementation only requires ~15 additional LoC to the trivial thread-mapped schedule.

Furthermore, we extend the same SpMV computation to our novel *group-mapped* load balancing schedule (that can also be specialized to perform *block-* and *warp-mapped* load balancing) within the same 30 LoC.

7 Related Work

Load balancing is the key to achieving high performance on GPUs for sparse, irregular parallel problems. Several high-performance computing applications deploy sophisticated load balancing algorithms on the GPUs. For instance, high-performance sparse-matrix vector multiplication (SpMV) leverages merge-path [20] (discussed in detail in this paper) or a nonzero splitting algorithm, which partitions the number of non-zeros in a sparse-matrix evenly across the number of threads [3, 9, 26]. Sparse-matrix matrix multiplication (SpMM) and sparse matricized tensor times Khatri-Rao product (SpMTTKRP) use binning and bundling algorithms [14, 22, 30], which attempt to bin like-length work together such that they are processed together.

While some applications actively perform work to load-balance a given input, others store the input in more efficient, already-load-balanced/-partitioned formats. These include the F-COO format (a variant of coordinate format)

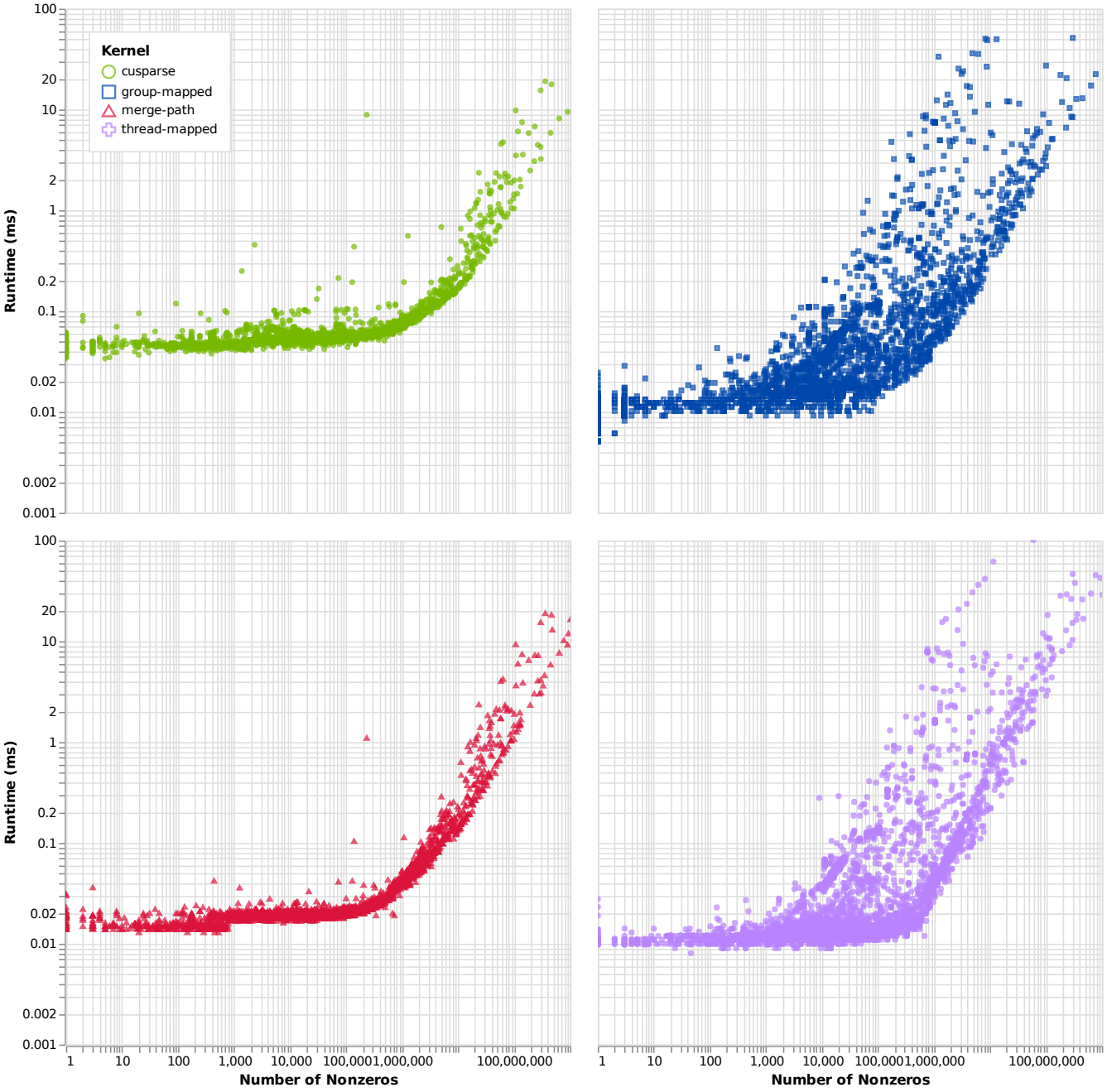

Figure 3. Complete performance landscape of SpMV across all SuiteSparse datasets using 3 load balancing schedules vs. NVIDIA's cuSparse library. This performance comparison highlights the impact of different approaches to load-balancing SpMV for a given dataset and number of nonzero entries within each dataset. Later in Figure 4 we use this insight to select the fastest schedule for an improved overall performance. Additionally, our 3 different SpMV implementations are made possible with very little code change.

used for SpMTTKRP and Sparse-Tensor Tensor Multiplication (SpTTM), where each thread gets the same number of nonzeros to process [19].

Many of the above GPU load-balancing algorithms, along with other novel techniques, were first described in the graph analytics domain. Davidson et al. and Merrill and Garland were the first to present Warp, Block-level and Thread-Warp-CTA dynamic load balancing techniques for Single-Source

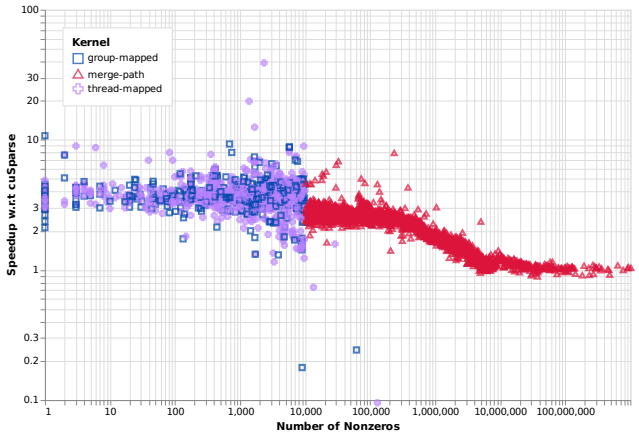

Figure 4. Speedup of our framework's SpMV vs. cuSparse's SpMV across SuiteSparse using a heuristic (Section 6.2) to choose the appropriate load-balancing schedule.

Shortest Path (SSSP) and Breadth-First Search (BFS) respectively [10, 21]. Logarithmic Radix Binning (LRB) is a particularly effective technique for binning work based on a logarithmic work estimate, used for the Triangle Counting graph algorithm and more [13, 16]. Gunrock, GraphIT, and GraphBLAST are graph analytics libraries that implement several different graph algorithms such as BFS, SSSP, PageRank, Graph Coloring, and more, built on these previously mentioned load-balancing techniques [5, 29, 31]. Although many of these are effective load balancing techniques with high-performance implementations, they all tightly couple workload scheduling with the application itself. Our framework is designed to separate these two concerns, allowing the application to be independent of the load-balancing algorithm, and therefore be expressed simply. Our approach also allows these previously proposed techniques to be implemented within our framework, and be used for applications beyond those originally targeted.

Relatively few GPU works target generalized load balancing for irregular workloads. Most of these are focused on providing a singular, dynamic load-balancing solution centered on task parallelism, often using a GPU queue-based data structure. Cederman and Tsigas proposed a task-based approach to load balancing an octree partitioning workload using lock-free and lock-based methods [7]. Two Tzeng works provide task-management frameworks that implement load balancing of tasks using a single monolithic task queue and distributed queues with task stealing and donation [27, 28]. CUIRRE, a framework for load balancing and characterizing irregular applications on GPUs, also uses a task-pool approach [33], and more recently, Atos, a task-parallel GPU dynamic scheduling framework, targets asynchronous algorithms [8]. All of these works deploy either a centralized or a distributed queue-like data structure on the GPUs, each making design decisions on how the queue is to be partitioned and updated. Except for the most recent Atos work, most earlier works focus on a coarse-grained parallelism approach of effectively distributing tasks to the GPU. Our work takes advantage of more modern GPU architectures, which are more effectively utilized by a fine-grained parallelism approach (parallelizing over work atoms instead of work tiles). Unlike our abstraction, these aforementioned works also rely on a singular load-balancing solution, whereas our abstraction flexibly adapts to many different load-balancing techniques, static and dynamic, and allows for new schedules to be implemented within our framework.

8 Conclusion

In this paper, we present a programming model for GPU load balancing for sparse irregular parallel problems. Our model is built on the idea of separation of concerns between workload mapping and work execution. In the future, we are interested in expanding our model to a multi-GPU environment, and implementing load-balancing schedules that span across the GPU boundary covering multiple devices and nodes for massive parallel problems. Our current work focuses solely on load balancing, but we also identify locality to be another key factor for high performance. We are interested in identifying an orthogonal model that builds an abstraction for caching and locality into our existing load-balancing framework.

Acknowledgments

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-3-0007 and the National Science Foundation under Contract No. OAC-1740333. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). We would like to acknowledge Michael Garland and Duane Merrill from NVIDIA for their guidance on the framework. We would also like to acknowledge Toluwanimi Odemuyiwa, Jonathan Wapman, Matthew Drescher and Muhammad Awad for research discussions and feedback on the work. We also acknowledge the support of AMD, Inc. (Jalal Mahmud and AMD Research) in the form of travel funding, which enables us to attend the conference to present this work.

References

[1] 2017. *ISO International Standard ISO/IEC 14882:2017(E) - Programming Language C++*. Technical Report. International Organization for Standardization (ISO). https://isocpp.org/std/the-standard.

[2] Advanced Micro Devices, Inc. 2022. HIP Programming Guide v5.2. (June 2022). https://docs.amd.com/bundle/HIP-Programming-Guide-v5.2/page/Introduction.html

[3] Sean Baxter. 2013–2016. Moderngpu: Patterns and Behaviors for GPU Computing. (2013–2016). http://moderngpu.github.io/moderngpu.

[4] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University.

[5] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2021. Compiling Graph Applications for GPUs with GraphIt. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2021)*. 248–261. https://doi.org/10.1109/CGO51591.2021.9370321

[6] F. Busato and N. Bombieri. 2015. BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 26, 7 (July 2015), 1826–1838. https://doi.org/10.1109/TPDS.2014.2330597

[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] 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 *Proceedings of the International Conference on Parallel Processing (ICPP 2022)*. https://doi.org/10.1145/3545008.3545056 arXiv:2112.00132

[9] Steven Dalton, Sean Baxter, Duane Merrill, Luke Olson, and Michael Garland. 2015. Optimizing Sparse Matrix Operations on GPUs Using Merge Path. In *IEEE International Parallel and Distributed Processing Symposium*. 407–416. https://doi.org/10.1109/IPDPS.2015.98

[10] 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

[11] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[12] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw.* 43, 4, Article 30 (Jan. 2017), 49 pages. https://doi.org/10.1145/3017994

[13] James Fox, Alok Tripathy, and Oded Green. 2019. Improving Scheduling for Irregular Applications with Logarithmic Radix Binning. *2019 IEEE High Performance Extreme Computing Conference* (2019), 1–7. https://doi.org/10.1109/HPEC.2019.8916333

[14] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU Kernels for Deep Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. Article 17, 14 pages. https://doi.org/10.1109/SC41405.2020.00021

[15] Google. 2023. Chromium C++ style guide. (2023). https://chromium.googlesource.com/chromium/src/+/HEAD/styleguide/c++/c++.md

[16] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A. Bader. 2018. Logarithmic Radix Binning and Vectorized Triangle Counting. In *IEEE High Performance Extreme Computing Conference (HPEC 2018)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547581

[17] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. 331–340. https://doi.org/10.1145/2304576.2304621

[18] Mark Harris and Kyrylo Perelygin. 2017. Cooperative Groups: Flexible CUDA Thread Programming. (2017). https://developer.nvidia.com/blog/cooperative-groups/

[19] Bangtian Liu, Chengyao Wen, Anand D. Sarwate, and Maryam Mehri Dehnavi. 2017. A Unified Optimization Approach for Sparse Tensor Operations on GPUs. *IEEE International Conference on Cluster Computing* (Sept. 2017), 47–57. https://doi.org/10.1109/CLUSTER.2017.75

[20] Duane Merrill and Michael Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. 678–689. https://doi.org/10.1109/SC.2016.57

[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] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P. Sadayappan. 2019. Load-Balanced Sparse MTTKRP on GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*. 123–133. https://doi.org/10.1109/IPDPS.2019.00023

[23] NVIDIA Corporation. 2007–2022. CUDA C++ Programming Guide. (Dec. 2007–2022). https://docs.nvidia.com/cuda/ PG-02829-001_v12.0.

[24] NVIDIA Corporation. 2023. CUB: Cooperative primitives for CUDA C++. (2023). https://nvlabs.github.io/cub/.

[25] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. 2022. A Programming Model for GPU Load Balancing. (Dec. 2022). https://doi.org/10.5281/zenodo.7465053

[26] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing (ICS 2017)*. ACM, 13:1–13:11. https://doi.org/10.1145/3079079.3079086

[27] 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

[28] 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

[29] 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

[30] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Proceedings of the 24th International European Conference on Parallel and Distributed Computing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). 672–687. https://doi.org/10.1007/978-3-319-96983-1_48

[31] Carl Yang, Aydın Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Software* 48, 1, Article 1 (Feb. 2022), 51 pages. https://doi.org/10.1145/3466795

[32] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, and Satoshi Matsuoka. 2022. Persistent Kernels for Iterative Memory-bound GPU Applications. *CoRR* (April 2022). arXiv:2204.02064

[33] Tao Zhang, Wei Shu, and Min-You Wu. 2014. CUIRRE: An open-source library for load balancing and characterizing irregular applications on GPUs. *J. Parallel and Distrib. Comput.* 74, 10 (Oct. 2014), 2951–2966. https://doi.org/10.1016/j.jpdc.2014.07.004

A Artifact Description

We provide the source code of our load-balancing framework called `loops` and our testing harness for evaluating the results provided within this paper.

A.1 Requirements

1. **Operating System** Ubuntu 18.04, 20.04, Windows.
2. **Hardware** NVIDIA GPU (Volta microarchitecture or newer).
3. **Software** CUDA 11.7 or above and cmake 3.20.1 or newer.
4. **Compilation** NVCC (comes with CUDA), g++ and gcc, msvc with support for C++14 standard.
5. **Output** Comma-separated values (CSV) files that are used to generate the graphs in Section 6.
6. **Disk space** 886 GB to store the entire SuiteSparse Matrix Collection [11] compressed and uncompressed. Can be reduced significantly by running the tests on only a subset of the dataset.
7. **Code License** Apache 2.0.

A.2 How to Access

The main repository is hosted on GitHub: https://github.com/gunrock/loops. Our framework is also available as a Zenodo archive: https://doi.org/10.5281/zenodo.7465053 [25]. Detailed and well-formatted instructions are available within the README markdown file in the repositories, and a summary is available below.

A.3 Getting Started

Before building `loops`, make sure you have the CUDA Toolkit and cmake installed on your system, and exported in `PATH` of your system. Other external dependencies such as `thrust`, `cub`, etc. are automatically fetched using cmake.

```
cd loops
mkdir build && cd build
cmake -DCMAKE_CUDA_ARCHITECTURES=70 ..
make -j$(nproc)
```

A.3.1 Sanity Check. Run the following command in the cmake's `build` directory:

```
bin/loops.spmv.merge_path \
 -m ../datasets/chesapeake/chesapeake.mtx \
 --validate -v
# Expected Output
# Elapsed (ms):   0.063328
# Matrix:         chesapeake.mtx
# Dimensions:     39 x 39 (340)
# Errors:         0
```

A.4 Reproducing Results

We provide the following instructions to regenerate the results presented in this paper.

1. In the run script, update `DATASET_DIR` to point to the path of all the downloaded datasets (set to the path of the directory containing the `MM` directory; inside `MM` are subdirectories with `.mtx` files): `scripts/run.sh`.
   - You may change the path to `DATASET_FILES_NAME` containing the list of all the datasets (default points to `suitesparse.txt` file in the `datasets` directory).
2. Fire up the complete run using `run.sh` found in the scripts directory, `cd scripts && ./run.sh`. Note one complete run can take up to 3 days (the run goes over the entire SuiteSparse matrix collection dataset four times with four different algorithms; the main bottleneck is loading files from disk).
   - Warning: Some runs on the matrices are expected to fail as they are not in proper MatrixMarket Format although labeled as `.mtx`. These matrices and the ones that do not fit on the GPU will result in runtime exceptions or type overflow and can be safely ignored.
3. To run `N` number of datasets, simply adjust the stop condition here (default set to 10): `run.sh#L22`, or remove this if-condition entirely to run on all available `.mtx` files: `run.sh#L22-L26`.

Additionally, we provide pre-generated results (in the form of CSV files) to create the plots from Section 6 without needing to run all the experiments. These pre-generated results are available under the `docs` directory of the repository.

A.5 Expected Output and Plots

The expected output from the above runs are csv files in the same directory as the `run.sh`. These can replace the existing csv files within `docs/data`, and a python jupyter notebook can be used to evaluate the results. The python notebook includes instructions on generating plots. See a sample output of one of the csv files below:

```
kernel,dataset,rows,cols,nnzs,elapsed
merge-path,144,144649,144649,2148786,0.07202
merge-path,08blocks,300,300,592,0.0170898
merge-path,1138_bus,1138,1138,4054,0.0200195
```