GPU SEMIRING PRIMITIVES FOR SPARSE NEIGHBORHOOD METHODS

Corey J. Nolet [1 2] **Divye Gala** [1] **Edward Raff** [2 3] **Joe Eaton** [1] **Brad Rees** [1] **John Zedlewski** [1] **Tim Oates** [2]

ABSTRACT

High-performance primitives for mathematical operations on sparse vectors must deal with the challenges of skewed degree distributions and limits on memory consumption that are typically not issues in dense operations. We demonstrate that a sparse semiring primitive can be flexible enough to support a wide range of critical distance measures while maintaining performance and memory efficiency on the GPU. We further show that this primitive is a foundational component for enabling many neighborhood-based information retrieval and machine learning algorithms to accept sparse input. To our knowledge, this is the first work aiming to unify the computation of several critical distance measures on the GPU under a single flexible design paradigm and we hope that it provides a good baseline for future research in this area. Our implementation is fully open source and publicly available as part of the RAFT library of GPU-accelerated machine learning primitives (https://github.com/rapidsai/raft).

1 INTRODUCTION

Many machine learning and information retrieval tasks operate on sparse, high-dimensional vectors. Nearest-neighbor based queries and algorithms in particular are instrumental to many common classification, retrieval, and visualization applications(Schölkopf et al., 2001; Alpay, 2012; Berlinet and Thomas-Agnan, 2011; Smola et al., 2007; Scholkopf and Smola, 2018). As General-purpose GPU computing (GPGPU) has become more popular, the tools for IR and distance computations on GPUs has not kept pace with other tooling on dense representations like image and signal processing that have contiguous access patterns that are easier to code for (Guo et al., 2020). Sparse methods of linear algebra on GPUs have long existed, though they are often specialized and difficult to adapt to new distance measures. This stems from having to account for various hardware and application-specific constraints (Jeon et al., 2020; Guo et al., 2020; Gale et al., 2020; Gray et al., 2017; Bell and Garland, 2008), and assumptions on the distribution of non-zeros in the input and output data (Sedaghati et al., 2015; Mattson et al., 2013). This complexity and specialization has slowed the adoption for sparse data and operations in general purpose tools like PyTorch and Tensorflow.

To develop a more general code base that supports good performance and flexibility for new distance measures on sparse data, we develop an approach leveraging Semirings. Semirings provide a useful paradigm for defining and computing inner product spaces in linear algebra using two operations, as in the MapReduce (Mattson et al., 2013; Emoto et al., 2012) paradigm, where a *product()* function is used to define a mapping between point-wise corresponding elements of vectors and a *sum()* function is used to reduce the products into a scalar. Using semirings to implement algorithms with sparse linear algebra on GPUs is an active area of research (Fender, 2017; Gildemaster et al., 2020; Lettich, 2021) and has been widely studied for helping to consolidate both the representation and execution of operations on graphs and probabilistic graphical models. In this paper, we show that semirings can be used for sparse neighborhood methods in machine learning, extending the benefits to all algorithms capable of using them. We define semirings more formally in subsection 2.2 but use the more general description above to navigate the benefits and related work in the following section.

A common issue for large-scale sparse problems in high-performance single-instruction multiple-data (SIMD) environments, like the GPU, is load balancing in order to keep the processing units constantly moving forward. As we will show in Section 3.1, the imbalanced load and resource requirements for a simple and straightforward naive semiring implementation, capable of computing distances like Manhattan, suffers from large thread divergences within warps, highly uncoalesced global memory accesses, and resource requirements which are unrealistic in many real-world datasets.

In order to integrate into an end-to-end data science or scientific computing workflow, such as in the PyData or RAPIDS (Raschka et al., 2020) ecosystems, an efficient

[1]NVIDIA [2]University of Maryland, Baltimore County [3]Booz Allen Hamilton. Correspondence to: Corey J. Nolet <cjnolet@gmail.com>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.

implementation of a primitive for computing pairwise distances on sparse datasets should ideally preserve as many of the following characteristics as possible. In this paper, we show that our implementation preserves more of the below characteristics than any other known implementation.

1. Maintain uniformity of intra-warp instruction processing.
2. Coalesce both reads from and writes to global memory.
3. Process data inputs without transposition or copying.
4. Use as little memory as necessary.
5. Enable semirings in addition to the simple dot product.

2 SEMIRINGS AND PAIRWISE DISTANCES

We formalize the concepts of semirings and distance measures in this section and describe building blocks required to implement several popular distance measures, often encountered in machine learning applications, into the semiring framework.

In machine learning applications, a distance measure is often performed on two row matrices containing data samples with columns that represent some number of observations, or features. In this paper, we will refer to these two matrices as A and B in upper-case where $A \in \mathbb{R}^{m \times k}$, and $B \in \mathbb{R}^{n \times k}$ and a single vector as a and b in lowercase where $a \in \mathbb{R}^k$ or $b \in \mathbb{R}^k$. As we show in this section, the core of computing pairwise distances between A and B is a matrix multiplication $AB^\top$ in a topological space equipped with an inner product semiring that defines distances between vectors. When this inner product is defined to be the dot product semiring, the topological space defines the standard matrix multiply but we can capture many other core distances in machine learning applications by simply redefining the inner product semiring.

While some distance measures can make use of the simple dot product semiring from matrix-matrix multiplication routines, we show that a more comprehensive package for computing pairwise distances requires more flexibility in terms of the arithmetic operations supported. Further, the explicit transposition of B which is required in routines such as the cuSPARSE *csrgemm()* requires a full copy of B, since no elements can be shared between the original and transposed versions in the CSR data format. This has a negative impact on scalability in memory-constrained environments such as GPUs.

2.1 Distances

Sparse matrix-matrix multiplication with a standard dot product semiring is most performant in cases where only the intersection is needed between pairs of corresponding nonzero columns in each vector. Because a standard multiplication between two terms has an identity of 1 and multiplicative annihilation (e.g. $a_i * 0 = 0$), the dot product semiring between two vectors can be computed efficiently by iterating over the nonzero columns of one vector and only computing the product of the corresponding nonzero columns of the other vector. Many distances can make use of this property, in table 1 we dervice the semi-ring annihilators and expansions (as needed) for 15 distances.

For a distance to define a metric space, it must follow four properties- implication ($d(a,b) = 0 \implies a = b$), positivity ($d(a,b) >= 0$), symmetry ($d(a,b) = d(b,a)$), and the triangle inequality ($d(a,c) \leq d(a,b) + d(b,c)$). Several metrics, including Chebyshev, Manhattan, and Euclidean, are derived from the generalized Minkowski formula $\left(\sum_i^k |a_i - b_i|^p\right)^{1/p}$ where p defines a degree. The absolute value in this equation defines a commutative semiring which requires commutativity in the difference of each vector dimension. Euclidean distance is equivalent to Minkowski with a degree of 2 ($(\sum_i^k |a_i - b_i|^2)^{1/2}$). Because the square of a number is always positive, this equation can be expanded to $(a-b)^p$ for all even degrees and still preserve the absolute value, such as $(a-b)^2 = a^2 - 2\langle a,b\rangle + b^2$ in the case of Euclidean distance. While numerical instabilities can arise from cancellations in these expanded equations, we will show in section 2.2 that the expanded form is often preferred in sparse algebras, when distances can make use of it, because it requires less computations than the exhaustive evaluation over the nonzeros of k. By example, the distances which don't have an expanded form, such as Manhattan (Minkowski with degree 1) and Chebyshev (Minkowski with degree max) distance, are often non-annihilating (e.g. $x*0 = x$) and require computation over the full union of nonzero columns from both vectors in order to preserve commutativity.

2.2 Semirings

A *monoid* is a semigroup containing an associative binary relation, such as addition ($\oplus$), and an identity element ($id_\oplus$). A *semiring* (Ratti and Lin, 1971), denoted $(S, \mathbb{R}, \{\oplus, id_\oplus\}, \{\otimes, id_\otimes\})$, is a tuple endowed with a domain along with additive ($\oplus$) and multiplicative ($\otimes$) monoids where

1. $\oplus$ is commutative, distributive, and has an identity element 0
2. $\otimes$ distributes over $\oplus$

Table 1: Common distances and their semirings. While all distances can be computed with the NAMM (where $id_{\otimes} = 0$), the distances in this table which require it have their $\otimes$ listed. The expansion function and any potential norms are provided for distances that can be computed in the more efficient expanded form.

Distance	Formula	NAMM	Norm	Expansion
Correlation	$1 - \frac{\sum_{i=0}^{k}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=0}^{k} x_i - \bar{x}^2}\sqrt{\sum_{i=0}^{2} y_i - \bar{y}^2}}$		L_1,L_2	$1 - \frac{k\langle x \cdot y\rangle - \|x\|\|y\|}{\sqrt{(k\|x\|_2 - \|x\|^2)(k\|y\|_2 - \|y\|^2)}}$
Cosine	$\frac{\sum_{i=0}^{k} x_i y_i}{\sqrt{\sum_{i=0}^{k} x_i^2}\sqrt{\sum_{i=0}^{k} y_i^2}}$		L_2	$1 - \frac{\langle x \cdot y\rangle}{\|x\|_2^2 \|y\|_2^2}$
Dice-Sorensen	$\frac{2\lvert\sum_{i=0}^{k} x_i y_i\rvert}{(\sum_{i=0}^{k} x)^2 + (\sum_{i=0}^{k} y)^2}$		L_0	$\frac{2\langle x \cdot y\rangle}{\lvert x\rvert^2 + \lvert y\rvert^2}$
Dot Product	$\sum_{i=0}^{k} x_i y_i$			$\langle x \cdot y \rangle$
Euclidean	$\sqrt{\sum_{i=0}^{k} \lvert x_i - y_i\rvert^2}$		L_2	$\|x\|_2^2 - 2\langle x \cdot y\rangle + \|y\|_2^2$
Canberra	$\sum_{i=0}^{k} \frac{\lvert x_i - y_i\rvert}{\lvert x_i\rvert + \lvert y_i\rvert}$	$\{\frac{\lvert x-y\rvert}{\lvert x\rvert + \lvert y\rvert}, 0\}$		
Chebyshev	$\sum_{i=0}^{k} \max(x_i - y_i)$	$\{\max(x - y), 0\}$		
Hamming	$\frac{\sum_{i=0}^{k} x_i \neq y_i}{k}$	$\{x \neq y, 0\}$		
Hellinger	$\frac{1}{\sqrt{2}}\sqrt{\sum_{i=0}^{k} (\sqrt{x_i} - \sqrt{y_i})^2}$			$1 - \sqrt{\langle \sqrt{x} \cdot \sqrt{y} \rangle}$
Jaccard	$\frac{\sum_{i=0}^{k} x_i y_i}{(\sum_{i=0}^{k} x_i^2 + \sum_{i=0}^{k} y_i^2 - \sum_{i=0}^{k} x_i y_i}$		L_0	$1 - \frac{\langle x \cdot y\rangle}{(\|x\| + \|y\| - \langle x \cdot y\rangle)}$
Jensen-Shannon	$\sqrt{\frac{\sum_{i=0}^{k} x_i \log \frac{x_i}{\mu_i} + y_i \log \frac{y_i}{\mu_i}}{2}}$	$\{x \log \frac{x}{\mu} + y \log \frac{y}{\mu}, 0\}$		
KL-Divergence	$\sum_{i=0}^{k} x_i \log(\frac{x_i}{y_i})$			$\langle x \cdot \log \frac{x}{y} \rangle$
Manhattan	$\sum_{i=0}^{k} \lvert x_i - y_i\rvert$	$\{\lvert x - y\rvert, 0\}$		
Minkowski	$(\sum_{i=0}^{k} \lvert x_i - y_i\rvert^p)^{1/p}$	$\{\lvert x - y\rvert^p, 0\}$		
Russel-Rao	$\frac{k - \sum_{i=0}^{2} x_i y_i}{k}$			$\frac{k - \langle x \cdot y\rangle}{k}$

Some formal definitions of semirings require that $id_{\otimes} = 1$. Given two sparse vectors $a, b \in \mathbb{R}^k$, a semiring with $(S, \mathbb{R}, \{\oplus, 0\}, \{\otimes, 1\})$ and $annihilator_{\otimes} = 0$ has the effect of only requiring $\otimes$ be computed on columns that are both nonzero (e.g. $nonzeros(a) \cap nonzeros(b)$). These rules are often relaxed in practice, for example in tropical semirings in Equation 1, which can solve dynamic programming problems such as the Viterbi algorithm. An *annihilator* is an input that will always cause a monoid to evaluate to 0 and the multiplicative annihilator ($annihilator_{\otimes}$) is often assumed to be $id_{\oplus}$. A monoid is non-annihilating when it does not have a defined annihilator. When an expanded form is not possible or efficient, $\otimes$ also must be commutative in metric spaces, and thus must be non-annihilating and $id_{\otimes} = 0$. We refer to this monoid as a *non-annihilating multiplicative monoid* (NAMM).

$$(S, \mathbb{R} \cup \{+\infty\}, \{min, +\infty\}, \{+, 0\}) \tag{1}$$

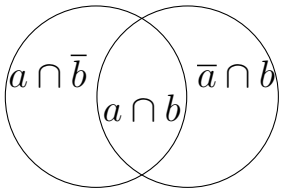
Table 1 uses semirings to construct several commonly used distances common in machine learning and data science applications. When an expanded form is possible, an expansion function can be performed as an element-wise operation on a simple pairwise dot product semiring with arrays of row-vector norms. While most of the expanded form distances can directly use the dot product semiring, KL-divergence directly replaces the $\otimes$ with $a_i \log(a_i / b_i)$ and makes no further assumption of symmetry. A NAMM is required for all unexpanded distance measures where $id_{\otimes} = 0$ and special care must be taken to ensure it is applied to the full union of the non-zero columns of corresponding elements from each pair of vectors.

As mentioned in the previous section, the Euclidean distance can be expanded to $\|A\| - 2\langle AB^{\top} \rangle + \|B\|$. This equation can be decomposed into the sum of individual L2 norms, a matrix product, and an element-wise expansion function executed in parallel over the individual dot prod-

ucts from the matrix product to combine the parts into a single scalar distance. Given vectors A_i, B_j, the expansion function for Euclidean distance can be derived by distributing their squared difference over the exponent to produce $(A_i - B_i) \times (A_i - B_i)$ and further expanding it to $\|A\|_i + 2\langle A_i, B_j \rangle - \|B\|_j$.

The $annihilator_\otimes$ and $id_\otimes$ determine the number of times the $\otimes$ monoid must be applied during the computation of pairwise distances. When $annihilator_\otimes = id_\oplus$, then $\otimes(a_i, 0) = 0$ and $\otimes(0, b_i) = 0$ so $\otimes$ can be applied only to the intersection of columns. When $annihilator_\otimes$ is undefined and $id_\otimes = 0$, then $\otimes$ must be applied exhaustively over the union of columns because $\otimes(a_i, 0) = a_i$ and $\otimes(0, b_i) = b_i$.

A union between two sets can be decomposed into an intersection between the two sets, along with the union of the symmetric differences between them. These are shown in Equation 3, where a complement is denoted with a $\overline{bar}$. The nonzero columns of two sparse vectors can be used as sets a and b in this equation and the sparse matrix multiply with an ordinary dot product only requires the application of *product()* across $a \cap b$. The NAMM, however, requires the application of the *product()* across the full union of nonzero columns $a \cup b$.



(2)

$$a \cup b = \{a \cap b\} \cup \{\overline{a} \cap b\} \cup \{a \cap \overline{b}\} \tag{3}$$

A common approach to implementing sparse matrix multiply is to iterate over the nonzeros from b in order to lookup and compute the intersection with the nonzeros from a. This design will also implicitly compute the symmetric difference between either of the two sets of nonzeros, $a \cap \overline{b}$ or $\overline{a} \cap b$, depending on which vector is chosen in the iteration over nonzeros. To compute a full union, the remaining set difference can be computed in a second pass of the matrix multiply by looping over the nonzeros from the vector that remains. We will show in subsection 3.1 that we accomplish this efficiently in our implementation in two passes- one pass to compute the first two terms and another pass to compute the third term. Distances which can be computed with an expansion function only need the first pass while distances which require the NAMM need both. Please refer to subsection A.1 for an example of using semirings to compute the Manhattan distance using the NAMM.

Existing semiring implementations currently require that the $id_\oplus$ be used as $annihilator_\otimes$. For example, the GraphBLAS specification enables the re-interpretation of the zeroth element, but this is necessary to define the identity of the $\oplus$ monoid.

3 GPU-Accelerated Semirings

In this section, we briefly introduce GPU architecture before discussing some naive designs and the inefficiencies that led to the construction of our final design. Our goal was to preserve as many of the ideal design characteristics from section 5 as possible but we found a need to accept trade offs during implementation.

3.1 GPU Architecture

The largest GPUs today contain hundreds of hardware processing cores called streaming multiprocessors (SM) which execute groups of threads called warps. Each warp can process a single instruction at a time in parallel using a paradigm called single-instruction multiple data (SIMD). It's important that threads within a warp minimize conditional branching that will cause the threads to wait for each branch to complete before proceeding. This is called thread divergence, and can severely limit effective parallel execution. On the Volta and Ampere architectures, each SM can track the progress of up to 64 warps concurrently (Tesla, 2018), and rapidly switch between them to fully utilize the SM. Each SM has a set of registers available which allows warps to perform collective operations, such as reductions. Warps can be grouped into blocks and a small amount of memory can be shared across the threads and warps.

Global, or device, memory can be accessed by all of the SMs in the GPU. Accesses to contiguous device memory locations within a warp can be coalesced into a single blocked transaction so long as the accesses are performed in the same operation. In SIMD architectures, uniform patterns can be critical to performance unless latencies from non-uniform processing, such as uncoalesced memory accesses, can be hidden with increased parallelism.

Registers provide the fastest storage, and it's generally preferable to perform reductions and arithmetic as intra-warp collective operations where possible. Intra-block shared memory is also generally preferred over global memory when a problem can be made small enough to benefit. However, contiguous locations of shared memory are partitioned across contiguous banks and any accesses to different addresses in the same bank by the same warp will create a bank conflict and be serialized within the warp, causing the threads to diverge.

3.2 Naive Semi-Ring Full-Union CSR Designs

3.2.1 *Expand-Sort-Contract*

Initial implementations tried to minimize the memory footprint as much as possible by directly computing the output distances from the input CSR format. The CSR format requires columns to be sorted with respect to row and we initially attempted to use a modified variant of the *expand-sort-contract* (Dalton et al., 2015) pattern on the nonzero columns from each pair of row vectors, $a, b \in \mathbb{R}^k$, concatenating the vectors together, sorting them, and applying the $\otimes$ monoid on pairs of duplicate columns to *contract* the sorted array and invoking $\otimes$ with the identity for all other columns. At the row-level of the output matrix, no computations would be able to be reused by subsequent pairs of vectors so we implemented this pattern on the GPU and mapped the nonzero columns and values for each row-vector pair to individual thread-blocks, *expanding* both vectors by concatenating them in shared memory, performing a sort-by-key, and compressing them in parallel. We attempted several efficient sorting algorithms on the GPU including the popular radix sort and bitonic sorting networks and, while the use of shared memory in the sort step enabled coalesced reads from global memory for the nonzero columns and values, the sorting step dominated the performance of the algorithm. Another downside with this particular design is that both vectors need to fit in shared memory, requiring space for $2 * (nonzeros(a) + nonzeros(b))$ elements in order to fit both the columns and corresponding values at the same time. In addition to the need for $n * m$ blocks to be scheduled, the shared memory requirement became a severe limit to scale, which was further compounded by the shared memory size limiting the number of blocks that could be scheduled concurrently on each SM.

Algorithm 1 Semiring on CSR inputs using expand-sort-contract pattern, parallelized across threads in each block.

Input: $A_i, B_j, product\_op, reduce\_op$
Result: $C_{ij} = d(A_i, B_j)$
$smem[0..nnz_{a_{i-1}}] = A_i$
$smem[nnz_{a_i}..nnz_{b_{j-1}}] = B_j$
$sort(smem)$
C_{ij} = reduce(smem, $product\_op$, $reduce\_op$)

3.2.2 *Iterating Sorted Nonzeros*

Since columns will often be sorted within their respective rows in the CSR format, we removed the sort step from algorithm 1 by exhaustively iterating over the non-zeros of each $O(m * n)$ pair of vectors in parallel, one pair per thread, as shown in algorithm 2. We found that even when the neighboring threads processed rows of similar degree, the differing distributions of nonzeros within each row decreased the potential for coalesced global memory accesses and created large thread divergences. Further, the exhaustive nature of this design, while it will guarantee the $\otimes$ monoid is computed on the full union of nonzero columns, will end up performing many unnecessary computations when distances can be computed with the rules of a simple dot product semiring.

Algorithm 2 Semring on CSR inputs. Each thread computes a single dot product.

Input: $A_i, B_j, product\_op, reduce\_op$
Result: $C_{ij} = d(A_i, B_j)$
startA = $indptrA_i$, endA = $indptrA_{i+1}$
startB = $indptrB_j$, endB = $indptrB_{j+1}$
i_{colA} = startA, i_{colB} = startB
while $i_{colA} <$ *endA* —— $i_{colB} <$ *endB* **do**
  colA = $i_{colA} <$ endA ? $indices_{i_{colA}}$: MAX_INT
  colB = i_colB $<$ endB ? $indices_{i_{colB}}$: MAX_INT
  valueA = 0, valueB = 0 **if** *colA* $\leq$ *colB* **then**
    valueA = $valuesA_{i_{colA}++}$
  end
  if *colB* $\leq$ *colA* **then**
    valueB = $valuesB_{i_{colB}++}$
  end
  $v = product\_op(valueA, valueB)$
  $C_{ij} = reduce\_op(C_{ij}, v)$
end

We found marginal gains in performance by coalescing the reads of the vectors from A into shared memory and sharing it across all threads of each thread-block. We attempted to load balance this algorithm by maintaining arrays to look up row information for each column but this increased warp divergence from the overly complicated conditionals required to maintain state across threads and warp boundaries.

3.3 Load Balanced Hybrid CSR+COO

While the CSR format enables algorithms to be parallelized over threads for individual rows, we found that using a row index array in coordinate format (COO) for B enabled load balancing, coalescing the loads from each vector from A into shared memory, once per block, and threads of each block parallelizing the application of the semiring over nonzero elements of B. Since the columns in B are assumed to be sorted by their respective row, we use a segmented reduction by key within each warp, bounding the number of potential writes to global memory by the number of active warps over each row of B. Our design extends the work of the COO sparse-matrix dense-vector multiplication described in (Anzt et al., 2020) by storing the vectors from A in dense form in shared memory only when the number of columns are small enough. Our extension enables sparse-

matrix sparse-vector multiplication by storing the vectors in sparse form when their degrees are small enough. We achieve full occupancy on the Volta architecture by trading off the size of the L1 cache to double the amount of shared memory per GPU, allowing each SM to use 96KiB. Since our design uses less than 32 registers, a block size of 32 warps allows two blocks, the full 64 warps, to be scheduled concurrently on each SM.

Algorithm 3 Load-balanced Hybrid CSR+COO SPMV.

Input: $A_i, B, product\_op, reduce\_op$
Result: $C_{ij} = d(A_i, B_j)$
read A_i into shared memory
cur_row=rowidx[ind]
ind = idx of first elem to be processed by this thread
c = product_op(A[ind], x[colidx[ind]])
for $i \leftarrow 1$ **to** nz_per_chunk; **by** $warp\_size$ **do**
  next_row = cur_row + $warp\_size$
  if *next_row != cur_row* —— is_final_iter? **then**
    v = segmented_scan(cur_row, c, product_op)
    if $is\_segment\_leader$? **then**
      atomic_reduce(v, reduce_op)
    end
    c = 0
  end
  cur_row = next_row
  ind += $warp\_size$
  c = product_op(A[ind], x[colidx[ind]])
end

3.3.1 *Two-pass execution*

As described in subsection 2.2, a single execution of this strategy will compute the intersection and symmetric difference $\overline{a} \cap b$ between nonzero columns from each vector a, and b so long as $\otimes$ is applied to all nonzero columns of b. While only a single pass covers distance measures which require only a column intersection (e.g. dot product semiring $(S, \mathbb{R}, \{+, 0\}, \{*, 1\})$), a second pass can compute the remaining symmetric difference required for the full union between non-zero columns by commuting A and B and skipping the application of of $id_{\otimes}$ in B for the second pass.

3.3.2 *Sparsifying the Vector in Shared Memory*

While we found storing the vectors from A in dense form in shared memory to have the highest throughput rate and least amount of thread divergence within each warp, sparse datasets are generally assumed to have high dimensionality and the limited amount of shared memory that can be allocated per SM bounds the size of the vectors that can be stored in it. For example, The $96KiB$ limit per block on Volta allows a max dimensionality of $23K$ with single-precision and the $163KiB$ limit per SM on Ampere allows a max dimensionality of $40K$ with single-precision. Coupling the amount of shared memory to the dimensionality creates a problem for occupancy as it approaches capacity. Both of these architectures limit the maximum block sizes to 1024 threads and max concurrent warps per SM to 64 so anything over $48KB$ of shared memory per block is going to decrease occupancy. For this reason, the maximum dimensionality of dense vectors that can be processed with full occupancy is actually $12K$ and $20K$, respectively.

This boundary becomes too small for many sparse datasets which would instead benefit from coupling the shared memory size to individual row degrees. Inspired by other sparse matrix multiplication implementations on the GPU (Anh et al., 2016; Kunchum, 2017; Liu and Vinter, 2014; Nagasaka et al., 2017), we enhanced the vector insertion and lookup patterns of the COO SPMV design outlined in (Anzt et al., 2020) by building a hash table to store these columns in shared memory. Unlike many other hash table implementations on the GPU (Alcantara et al., 2009; Ashkiani et al., 2018; Alcantara et al., 2012; Pan and Manocha, 2011; Cassee and Wijs, 2017), our implementation builds an independent hash table per thread-block and so many other designs and concurrency patterns that optimize the key distribution and collision-resolution strategies for the GPU are not efficient or cannot be easily ported for our use-case. For this reason, we used a simple hash table with a *Murmur* hash function and linear probing and leave the investigation of a better and more optimized design to future work.

Hash tables have the best performance when the number of entries is less than 50% of the capacity. As the hash table size grows beyond 50% capacity, the collision resolution cycles of linear probing, which are non-uniform, increase the serialization of instructions from warp divergences and also increase the number of transactions from global memory reads of B since they can no longer be coalesced. The hash table strategy decreases the amount of shared memory available, often by a factor of 2, because the nonzeros need to be stored together as key/value pairs to avoid an additional costly lookup to global memory, a side-effect which would only further increase serialized execution from diverging threads. Our hash table strategy allows for a max degree of $3K$ on Volta architectures and $5K$ on Ampere.

Another unfortunate side-effect from the linear-probing collision strategy of our hash table is the increase in lookup times for columns even for elements that aren't in the table. For example, as the hash table approaches capacity, the increase in collisions can cause a lookup to probe through multiple candidates, sometimes hundreds, before finding an element doesn't exist. Bloom filters have been used to implement fast list intersection problems for sparse matrix multiplication problems on the GPU (Zhang et al.,

2020; 2011). As an alternative to the hash table approach, we tried building a bloom filter in shared memory and used a binary search to perform lookups of nonzeros in global memory for positive hits. While we found this technique to yield marginally better performance on the Jensen-Shannon distance in one of our benchmarks, likely because it helped hide some of the compute-bound latencies from the additional arithmetic, we were not able to extract a simple rule from the data shapes or sparsity patterns that would allow us to know, before starting the computation, when it should be used.

3.3.3 *Handling High Degree Columns*

Our hash table implementation shows reasonable performance up to 50% capacity. Rows with degree greater than 50% hash table capacity are partitioned uniformly by their degrees into multiple blocks with subsets of the degrees that can fit into 50% hash table capacity. Using a similar logic to that of blocked sparse techniques, our partitioning strategy does extra work in exchange for scale. Further, this technique requires each thread perform a branching conditional so it can test whether each nonzero column of B is part of the current partition. As we show in section 4, we do find that this strategy can perform well on some datasets when most of the degrees are small enough to fit in the hash table. For example, we found this strategy spent a miniscule amount of time in this step on the Movielens dataset.

3.4 Norms and Expansion Functions

Distances which can be computed in their expanded forms can use the dot product semiring directly and only require a single pass through our SPSV. Computing distances in their expanded form often requires one or more vectors of row norms as well as an *expansion function*, which uses some arithmetic to combine the norm vectors with the individual dot products (refer to Table 1 for examples). Row norms can be computed over CSR matrices using a row-wise reduction on the GPU as each row can be mapped to a single block or warp and the norm computed by a warp-level collective reduction. The reduction primitive necessary for computing these row norms is already part of the GraphBLAS specification.

The actual arithmetic in each expansion function is dependent upon the distance measure, however the kernel to apply the expansion function can be executed embarrassingly parallel using an element-wise primitive, also part of the GraphBLAS specification, to map each entry in the dot product matrix to an individual GPU thread to coalesce the reads and writes.

4 Experiments

We evaluated the runtime performance characteristics and generalization of our approach by benchmarking our semiring strategies against several real-world sparse datasets with different shapes and degree distributions. We also analyze the GPU memory footprint of the cuSPARSE *csrgemm()* and our load-balanced COO SPMV.

4.1 Datasets

The datasets which we found are often used to benchmark sparse matrix-matrix and matrix-vector implementations on the GPU demonstrate the subtle differences in the objectives between using semirings for sparse neighborhood methods and using sparse linear algebra more generally for things like graph algorithms and eigendecompositions. As an example, one such set of datasets which we found commonly used in papers to benchmark sparse linear algebra implementations (Williams et al., 2007; Bell and Garland, 2008) is composed almost entirely of square connectivities graphs, and these would not provide a useful performance indicator for the objective of creating connectivites graphs from bipartite graphs. For this reason, and the lack of prior research in our objective, we establish a new baseline using datasets that our algorithm would be expected to encounter in practice. Our baseline uses cuSPARSE for all the expanded distance measures, along with the naive CSR full-union semiring implementation as described in section 3.2.2 for the distances which cuSPARSE does not support.

The *MovieLens* (Harper and Konstan, 2015) Large dataset contains ratings given by 283k users for 194k movies. We used a dataset of 70k cells and gene expressions for 26k genes from the human cell atlas (Travaglini et al., 2020) as an example of a single-cell RNA workflow. For natural language processing examples, we benchmarked two different datasets containing TF-IDF vectors for two different use-cases. We used the NY Times Bag of Words dataset(Newman, 2008) for an example of document similarity and n-grams generated from a list of company names from the SEC EDGAR company names database for an example of string matching.

Table 2: Datasets used in experiments

Dataset	Size	Density	Min Deg	Max Deg
Movielens Large	(283K, 194K)	0.05%	0	24K
SEC Edgar	(663K, 858K)	0.0007%	0	51
scRNA	(66K, 26K)	7%	501	9.6K
NY Times BoW	(300K, 102K)	0.2%	0	2K

4.2 Runtime Performance

To get an idea of how each supported distance performed on data of different shapes and degree distributions, we

Table 3: Benchmark Results for all datasets under consideration. All times are in seconds, best result in **bold**. The first italicized set of distances can all be computed as dot products, which are already highly optimized for sparse comparisons today. This easier case we are still competitive, and sometimes faster, than the dot-product based metrics. The Non-trivial set of distances that are not well supported by existing software are below, and our approach dominates amongst all these metrics.

		MovieLens		scRNA		NY Times Bag of Words		SEC Edgar	
	Distance	Baseline	RAFT	Baseline	RAFT	Baseline	RAFT	Baseline	RAFT
Dot Product Based	*Correlation*	130.57	**111.20**	**207.00**	235.00	**257.36**	337.11	134.79	**87.99**
	Cosine	131.39	**110.01**	**206.00**	233.00	**257.73**	334.86	127.63	**87.96**
	Dice	130.52	**110.94**	**206.00**	233.00	**130.35**	335.49	134.36	**88.19**
	Euclidean	131.93	**111.38**	**206.00**	233.00	**258.38**	336.63	134.75	**87.77**
	Hellinger	129.79	**110.82**	**205.00**	232.00	**258.22**	334.80	134.11	**87.83**
	Jaccard	130.51	**110.67**	**206.00**	233.00	**258.24**	336.01	134.55	**87.73**
	Russel-Rao	130.35	**109.68**	**206.00**	232.00	**257.58**	332.93	134.31	**87.94**
Non-Trivial Metrics	Canberra	3014.34	**268.11**	4027.00	**598.00**	4164.98	**819.80**	505.71	**102.79**
	Chebyshev	1621.00	**336.05**	3907.00	**546.00**	2709.30	**1072.35**	253.00	**146.41**
	Hamming	1635.30	**229.59**	3902.00	**481.00**	2724.86	**728.05**	258.27	**97.65**
	Jensen-Shannon	7187.27	**415.12**	4257.00	**1052.00**	10869.32	**1331.37**	1248.83	**142.96**
	KL Divergence	5013.65	**170.06**	4117.00	**409.00**	7099.08	**525.32**	753.56	**87.72**
	Manhattan	1632.05	**227.98**	3904.00	**477.00**	2699.91	**715.78**	254.69	**98.05**
	Minkowski	1632.05	**367.17**	4051.00	**838.00**	5855.79	**1161.31**	646.71	**129.47**

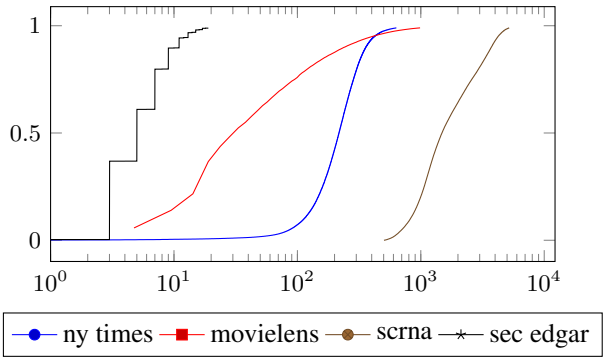


Figure 1: CDFs of Degree Distributions for the datasets used in our benchmark on the interval 0-99%. We can see that 99% of the degrees in the SEC Edgar datasets are ¡10 while 88% of the degrees for Movielens are ¡200. On average scRNA has the largest degrees with 98% of the rows having degree 5k or less. The NY Times dataset has the highest variance, with 99% of the rows having degree less than 1k.

benchmarked all of the supported distances for each of the datasets, even though some of them may provide irrelevant geometries in practice. Benchmarks were performed on a DGX1 containing dual 20-core Intel Xeon ES-2698 CPUs (80 total threads) at 2.20GHZ and a Volta V100 GPU running CUDA 11.0 for both the driver and toolkit. Each benchmark performs a k-nearest neighbors query to test our primitives end-to-end and allow scaling to datasets where the dense pairwise distance matrix may not otherwise fit in the memory of the GPU. We used the brute-force *NearestNeighbors* estimator from RAPIDS cuML for the GPU benchmarks since it makes direct use of our primitive. We used Scikit-learn's corresponding brute-force *NearestNeighbors* estimator as a CPU baseline and configured it to use all the available CPU cores. Each experiment trains the *NearestNeighbors* estimator on the entire dataset and then queries the entire dataset, timing only the query. Compared to the CPU, we observed an average of $28.78\times$ speedup for the dot-product-based distances and $29.17\times$ speedup for the distances which require the non-annihilating product monoid.

From the strategies described in Section 3.1, we benchmarked our best performing approach, the Load-balanced Hybrid COO+CSR SPMV described in subsection 3.3, using the hash table strategy to sparsify the vector in shared memory.

As evidenced in table 3, our implementation consistently outperforms the CPU. We also outperform the baseline,

```
from cuml.neighbors import
↪  NearestNeighbors
nn = NearestNeighbors().fit(X)
dists, inds = nn.kneighbors(X)
from cuml.metrics import
↪  pairwise_distances
dists = pairwise_distances(X,
↪  metric='cosine')
```

Figure 2: Excluding data loading and logging, all the code needed to perform the same GPU accelerated sparse distance calculations done in this paper are contained within these two snippets. Top shows k-NN search, bottom all pairwise distance matrix construction. These are the APIs that most would use.

```
#include
↪  <raft/sparse/distance/coo_spmv.cuh>
#include <raft/sparse/distance/operators.h>

using namespace raft::sparse::distance

distances_config_t<int, float> conf;

// Use conf to set input data arguments...

balanced_coo_pairwise_generalized_spmv(
    out_dists, conf, coo_rows_a,
    AbsDiff(), Sum(), AtomicSum());

balanced_coo_pairwise_generalized_spmv_rev(
    out_dists, conf, coo_rows_b,
    AbsDiff(), Sum(), AtomicSum());
```

Figure 3: The C++ API can be used to construct new semirings. Dot-product-based semirings only need invoke the first function while NAMMs can be constructed by invoking both. While the Python API is part of the RAPIDS cuML project, the C++ API is provided by the RAFT project (http://github.com/rapidsai/raft). RAFT is a header only library that contains fundamental algorithms and primitives for data science, graph, and machine learning applications.

cuSPARSE, for the distances that it supports in two out of the four datasets. In addition to maintaining comparable performance in the remaining two datasets, our design is also flexible enough to provide distances which require the NAMM outlined in subsection 2.2 while using less memory. As mentioned in section 5, it is not uncommon to see different sparse implementations performing better on some datasets than others (Sedaghati et al., 2015) and the flexibility of our implementation, as well as our well-defined set of rules for supporting a wide array of distances, will allow us to continue optimizing our execution strategies to support patterns that we find frequently occurring across different sparse datasets.

4.3 Memory Footprint

The density of the dot product matrix that is returned from the cuSPARSE *csrgemm()* is fully dependent upon the dataset. Because 2 arrays, each of size nnz, are required to represent the cuSPARSE output in CSR format, a density of 50% would require the same amount of space as the full dense pairwise distance matrix. A density of 100% requires 2x the amount of space as the dense pairwise distance matrix. In addition, since the output still needs to be converted to a dense format, this requires an additional allocation of the dense pairwise distance matrix in a space of contiguous memory locations even if the cuSPARSE output was 99.9% dense. We found the density of the cuSPARSE output to be at least 57% on average across the batches for Movielens, 98% for NY Times BoW and was fully dense in scRNA. The SEC Edgar datasets had the highest variance in density from batch-to-batch and were significantly different between n-gram sizes. The unigram and bigram dataset ranged from 5% to 25% output density, for example, while trigrams ranged from 24% to 43%.

This provides further evidence of the subtle but important differences between the types of data we expect to encounter in neighborhood methods, however even more evident is that the matrix resulting from computing the dot product semiring over the square connectivities graphs used in other sparse matrix multiplication research (Williams et al., 2007; Bell and Garland, 2008) is extremely sparse. In addition to the output memory, cuSPARSE required an internal temporary workspace in device memory with anywhere from 300mb to 550mb of additional memory per batch while our dot product semiring required a workspace buffer of size $nnz(B)$ per batch. Strangely, the size of this temporary workspace seemed almost identical even when computed on the square connectivities graphs mentioned above.

5 Related Work

5.1 Sparse matrix multiplication

The task of efficient and performant sparse matrix multiplication is an active area of research, with implementations spanning the spectrum of scientific computing. In high performance computing environments, these solutions are designed around both hardware and software constraints (Jeon et al., 2020; Guo et al., 2020; Gale et al., 2020; Gray et al., 2017; Bell and Garland, 2008), often making use of specialized hardware capabilities and optimizing for specific sparsity patterns, an unfortunate side-effect that can reduce their potential for reuse. What complicates this further are the number of different optimized

variants of sparse matrix multiplication available in open source libraries, each using different concurrency patterns and available memory to provide speedups based on either supported sparse formats or the assumed density of either the inputs or the outputs (Sedaghati et al., 2015; Mattson et al., 2013). We have compareda against the seminal cuSPARSE (Naumov et al., 2010) that is highly optimized for sparse dot product based k-nearest neighbors (Zhou, 2018), and found our approach is faster or competitive in all cases, but is not limited to dot product based measures.

Better able to make use of critical optimizations inherent in their dense counterparts, block compressed sparse formats have become widely popular for representing sparse data (Zachariadis et al., 2020), in part because they can improve load balancing by grouping nonzeros into fixed-sized tiles and scheduling the tiles more uniformly across the processing cores. Enabling sparse formats to be processed more similar to their dense counterparts allows the use of specialized hardware optimizations such as tensor cores. While we do hope to someday support block-sparse formats, it is most often assumed that users will be calling code that invokes our primitive with matrices in the standard compressed sparse row (CSR) format (Williams et al., 2007) and so a conversion would be necessary in order to use a blocked format.

5.2 Semirings

Consolidating seemingly disparate concepts into a lightweight, terse, and abstract set of building-blocks can increase flexibility and promote reuse (Izbicki, 2013). This especially benefits fields which require non-trivial and highly-optimized implementations where the design complexities and costs are high, the basic linear-algebra subroutines (BLAS) API and GPU-accelerated computing being common examples. Semirings provide the efficiency and flexibility to enable algorithms in which the representation and assumptions of the typical BLAS API for dense linear algebra comes up short (Mattson et al., 2013). NIST published a sparse BLAS standard back in 2001 (Duff et al., 2002) and cuSPARSE is one of the most sophisticated implementations of the sparse BLAS standard that has been built on the GPU, however as mentioned above, its multiplication routines fix the inner product to the dot product. GraphBLAS (Davis, 2018) provides a set of primitives, along with an API, for using semiring algebras to implement graph algorithms. The GraphBLAST (Yang et al., 2019) and SuiteSparse (Davis, 2019) libraries provide implementations of the GraphBLAS that also include GPU-accelerated primitives.

The use of semirings in graph theory dates back to the early 1970s (Ratti and Lin, 1971), when "good old-fashioned artificial intelligence", or *Symbolic AI*, was a dominant paradigm in research. Semirings have also been used for some time to implement more modern machine learning methods (Belle and De Raedt, 2020), with the more recent invention of semiring programming attempting to further consolidate these concepts under a single framework and set of symbolic routines. Semirings can be a useful building-block for linear models (Jananthan et al., 2017), probabilistic models, such as Bayesian networks (Wachter et al., 2007) and the use of Tropical semiring in Markov networks (Ilic, 2011). The Tropical semiring is also being used to implement sparse non-negative matrix factorizations (Omanović et al., 2020).

5.3 Neighborhood Methods

Our work is positioned to have an impact on numerous down-stream tasks that often depend on sparse nearest-neighbor retrieval. This includes classic Information Retrieval problems where such methods are still highly competitive or preferred (Mitra and Craswell, 2018; Li, 2016; Soboroff, 2018; Voorhees et al., 2017; Bouthillier et al., 2021). Dimensional reduction approaches like t-SNE (Maaten and Hinton, 2008) and UMAP (McInnes et al., 2018) that lack sparse input support on GPUs without our method (Nolet et al., 2020). ML models based on the kernel trick, such as Guassian Process (Lawrence and Urtasun, 2009) also stand to benefit. The breadth and frequency of nearest neighbor methods on high dimensional data make our work relevant to an especially wide class of practioners.

6 Conclusion

In this paper, we demonstrated a flexible sparse pairwise distance primitive that is able to collectively support, to our knowledge, a larger assortment of widely-used distance measures than any other package on the GPU. We consolidated the design of these distance measures using a couple minor enhancements to the rules of classical semirings, which are traditionally used to implement graph algorithms, and we discussed the impact of our primitive as a core building block of many important neighborhood methods for machine learning and data mining. Finally, we provided a novel implementation as an example of how these semirings can be implemented on the GPU with a lower memory footprint and performance comparable to, or better than, the current state of the art.

References

Dan A Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. 2009. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH Asia 2009*

papers. 1–9.

Dan A Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D Owens, and Nina Amenta. 2012. Building an efficient hash table on the GPU. In *GPU Computing Gems Jade Edition*. Elsevier, 39–53.

Daniel Alpay. 2012. *Reproducing kernel spaces and applications*. Vol. 143. Birkhäuser.

Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. 2016. Balanced Hashing and Efficient GPU Sparse General Matrix-Matrix Multiplication.(2016), 1–12. *Google Scholar Google Scholar Digital Library Digital Library* (2016).

Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. 2020. Load-Balancing Sparse Matrix Vector Product Kernels on GPUs. *ACM Trans. Parallel Comput.* 7, 1, Article 2 (March 2020), 26 pages. https://doi.org/10.1145/3380930

Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 419–429.

Nathan Bell and Michael Garland. 2008. *Efficient sparse matrix-vector multiplication on CUDA*. Technical Report. Citeseer.

Vaishak Belle and Luc De Raedt. 2020. Semiring programming: A semantic framework for generalized sum product problems. *International Journal of Approximate Reasoning* 126 (2020), 181–201.

Alain Berlinet and Christine Thomas-Agnan. 2011. *Reproducing kernel Hilbert spaces in probability and statistics*. Springer Science & Business Media.

Xavier Bouthillier, Pierre Delaunay, Mirko Bronzi, Assya Trofimov, Brennan Nichyporuk, Justin Szeto, Naz Sepah, Edward Raff, Kanika Madan, Vikram Voleti, Samira Ebrahimi Kahou, Vincent Michalski, Dmitriy Serdyuk, Tal Arbel, Chris Pal, Gaël Varoquaux, and Pascal Vincent. 2021. Accounting for Variance in Machine Learning Benchmarks. In *Machine Learning and Systems (MLSys)*. arXiv:2103.03098 http://arxiv.org/abs/2103.03098

Nathan Cassee and Anton Wijs. 2017. Analysing the performance of GPU hash tables for state space exploration. *arXiv preprint arXiv:1712.09494* (2017).

Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 1–20.

Timothy A Davis. 2018. *Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra*. Technical Report. 24 pages.

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.

Iain S Duff, Michael A Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)* 28, 2 (2002), 239–267.

Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. 2012. Filter-embedding semiring fusion for programming with MapReduce. *Formal Aspects of Computing* 24, 4 (2012), 623–645.

Alexandre Fender. 2017. *Parallel solutions for large-scale eigenvalue problems arising in graph analytics*. Ph.D. Dissertation. Université Paris-Saclay.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. *arXiv preprint arXiv:2006.10901* (2020).

Brandon Gildemaster, Prerana Ghalsasi, and Sanjay Rajopadhye. 2020. A Tropical Semiring Multiple Matrix-Product Library on GPUs: (not just) a step towards RNA-RNA Interaction Computations. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 160–169. https://doi.org/10.1109/IPDPSW50202.2020.00037

Scott Gray, Alec Radford, and Diederik P Kingma. 2017. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224* 3 (2017).

Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse dnn models without hardware-support via tile-wise sparsity. *arXiv preprint arXiv:2008.13006* (2020).

F Maxwell Harper and Joseph A Konstan. 2015. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)* 5, 4 (2015), 1–19.

Velimir M Ilic. 2011. Entropy semiring forward-backward algorithm for HMM entropy computation. *arXiv preprint arXiv:1108.0347* (2011).

Michael Izbicki. 2013. Algebraic classifiers: a generic approach to fast cross-validation, online training, and parallel training. In *International Conference on Machine Learning*. PMLR, 648–656.

Hayden Jananthan, Suna Kim, and Jeremy Kepner. 2017. Linear systems over join-blank algebras. In *2017 IEEE MIT Undergraduate Research Technology Conference (URTC)*. IEEE, 1–4.

Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. 2020. BiQGEMM: matrix multiplication with lookup table for binary-coding-based quantized DNNs. *arXiv preprint arXiv:2005.09904* (2020).

Rakshith Kunchum. 2017. *On improving sparse matrix-matrix multiplication on gpus*. Ph.D. Dissertation. The Ohio State University.

Neil D Lawrence and Raquel Urtasun. 2009. Non-linear matrix factorization with Gaussian processes. In *Proceedings of the 26th annual international conference on machine learning*. 601–608.

Richard Lettich. 2021. *GALATIC: GPU Accelerated Sparse Matrix Multiplication over Arbitrary Semirings (GALATIC) v1. 0*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

Hang Li. 2016. Does IR Need Deep Learning ? IR and DL. *Keynote speech at SIGIR 2016 Neu-IR workshop* (2016).

Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 370–381.

Laurens Van Der Maaten and Geoffrey Hinton. 2008. Visualizing Data using t-SNE. *Journal of Machine Learning Research* 9 (2008), 2579–2605.

Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, et al. 2013. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–2.

Leland McInnes, John Healy, and James Melville. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv* (2018). arXiv:1802.03426 http://arxiv.org/abs/1802.03426

Bhaskar Mitra and Nick Craswell. 2018. An Introduction to Neural Information Retrieval t. *Foundations and Trends® in Information Retrieval* 13, 1 (2018), 1–126. https://doi.org/10.1561/1500000061

Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 101–110.

M Naumov, LS Chien, P Vandermersch, and U Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

David Newman. 2008. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

Corey J Nolet, Victor Lafargue, Edward Raff, Thejaswi Nanditale, Tim Oates, John Zedlewski, and Joshua Patterson. 2020. Bringing UMAP Closer to the Speed of Light with GPU Acceleration. *arXiv preprint arXiv:2008.00325* (2020).

Amra Omanović, Hilal Kazan, Polona Oblak, and Tomaž Curk. 2020. Data embedding and prediction by sparse tropical matrix factorization. arXiv:2012.05210 [cs.LG]

Jia Pan and Dinesh Manocha. 2011. Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*. 211–220.

Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information* 11, 4 (2020), 193.

JS Ratti and Y-F Lin. 1971. The graphs of semirings. II. *Proc. Amer. Math. Soc.* 30, 3 (1971), 473–478.

Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola. 2001. A generalized representer theorem. In *International conference on computational learning theory*. Springer, 416–426.

Bernhard Scholkopf and Alexander J Smola. 2018. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. Adaptive Computation and Machine Learning series.

Naser Sedaghati, Arash Ashari, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Characterizing dataset dependence for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 2nd workshop on parallel programming for analytics applications*. 17–24.

Alex Smola, Arthur Gretton, Le Song, and Bernhard Schölkopf. 2007. A Hilbert space embedding for distributions. In *International Conference on Algorithmic Learning Theory*. Springer, 13–31.

Ian Soboroff. 2018. Meta-Analysis for Retrieval Experiments Involving Multiple Test Collections. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM '18)*. ACM, New York, NY, USA, 713–722. https://doi.org/10.1145/3269206.3271719

Nvidia Tesla. 2018. V100 GPU architecture. *Online verfügbar unter http://images. nvidia. com/content/volta-architecture/pdf/volta-architecture-whitepaper. pdf, zuletzt geprüft am* 21 (2018).

Kyle J Travaglini, Ahmad N Nabhan, Lolita Penland, Rahul Sinha, Astrid Gillich, Rene V Sit, Stephen Chang, Stephanie D Conley, Yasuo Mori, Jun Seita, et al. 2020. A molecular cell atlas of the human lung from single-cell RNA sequencing. *Nature* 587, 7835 (2020), 619–625.

Ellen M Voorhees, Daniel Samarov, and Ian Soboroff. 2017. Using Replicates in Information Retrieval Evaluation. *ACM Trans. Inf. Syst.* 36, 2 (aug 2017). https://doi.org/10.1145/3086701

Michael Wachter, Rolf Haenni, and Marc Pouly. 2007. Optimizing inference in Bayesian networks and semiring valuation algebras. In *Mexican International Conference on Artificial Intelligence*. Springer, 236–247.

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.

Carl Yang, Aydin Buluc, and John D Owens. 2019. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *arXiv preprint arXiv:1908.01407* (2019).

Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers & Electrical Engineering* 88 (2020), 106848.

Fan Zhang, Di Wu, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. 2011. Fast lists intersection with bloom filter using graphics processing units. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. 825–826.

Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

Brady Beida Zhou. 2018. *GPU accelerated k-nearest neighbor kernel for sparse feature datasets*. Ph.D. Dissertation.

A APPENDIX

A.1 Deriving Distances With Semirings

All of the distances in this paper can be categorized into one of two groups- those which can be computed using the dot product and vector norms and those which cannot. The non-annhiliating multiplicative monoid (NAMM) is used for the latter group, which requires exhaustive computation over the union of non-zeros from each input. The following example derives the semiring for the Manhattan distance, demonstrating why the dot-product cannot be used.

Let vector $a = [1, 0, 1]$ and $b = [0, 1, 0]$

We can compute the L1 distance between these two vectors by taking the sum of the absolute value of their differences:

$$\sum(|a - b|) = \tag{4}$$

$$\sum([|1 - 0|, |0 - 1|, |1 - 0|]) = \tag{5}$$

$$\sum([1, 1, 1]) = 3 \tag{6}$$

Semiring standards such as GraphBLAS, for example, often make use of the detail that the multiplicative annihilator is equal to the additive identity. If we follow this detail in our example, we end up with the following result (if any side is 0, the arithmetic evaluates to 0):

$$\sum(|a - b|) = \tag{7}$$

$$\sum([|1 - 0|, |0 - 1|, |1 - 0|]) = \tag{8}$$

$$\sum([0, 0, 0]) = 0 \tag{9}$$

What we need here instead is for the multiplicative identity to be non-annihilating, such that it equals the additive identity, so that the difference in our example behaves like an XOR, evaluating to the other side when either side is zero and evaluating to 0 only in the case where both sides have the same value. For example,

$|1 - 0| = 1$ $|0 - 1| = 1$ $|0 - 0| = 0$ $|1 - 1| = 0$

Now let's perform a sparse-matrix sparse-vector multiply where $A = [[1, 0, 1]]$ and $b = [0, 1, 1]$

We can parallelize this by evaluating the semiring of b over each row vector of A independently, iterating through the nonzero columns from each vector in A and fetching or looking up the corresponding column from b (if it is nonzero). With the standard dot-product semiring, which annihilates multiplicatively over the additive identity, we only need to consider the intersection of columns where both sides are nonzero– column 3 in this example.

Removing the multiplicative annihilator results in the need to consider the union of non-zero columns, and so all columns need to be considered in this example. However if only the nonzero columns in the vectors of A are visited, the nonzero columns in b, which are zero in A, will be missed.

Recall that we can decompose a full union across all nonzero columns into a union of the symmetric difference between nonzero columns of A and b (that is, all columns which are nonzero in A and zero in b), the intersection between nonzero columns of A and b (where both sides are nonzero), and the symmetric difference between the nonzero columns of b and A (that is, all columns which are nonzero in b and zero in A).

A spmv will often compute the intersection between the nonzero columns of A and b and the symmetric difference between nonzero columns of A and b will be computed only as a side-effect. In order to compute the union between the nonzero columns of A and b, the symmetric difference between the nonzero columns of b and A still needs to be computed. We compute this with a second pass of the spmv by flipping the inputs to the spmv and ignoring the intersecting columns in the second pass.

B ARTIFACT APPENDIX

B.1 Abstract

High-performance primitives for mathematical operations on sparse vectors must deal with the challenges of skewed degree distributions and limits on memory consumption that are typically not issues in dense operations. We demonstrate that a sparse semiring primitive can be flexible enough to support a wide range of critical distance measures while maintaining performance and memory efficiency on the GPU. We further show that this primitive is a foundational component for enabling many neighborhood-based information retrieval and machine learning algorithms to accept sparse input. To our knowledge, this is the first work aiming to unify the computation of several critical distance measures on the GPU under a single flexible design paradigm and we hope that it provides a good baseline for future research in this area. Our implementation is fully open source and publicly available at https://github.com/rapidsai/raft.

B.2 Artifact check-list (meta-information)

- **Algorithm: sparse matrix-vector multiplication, pairwise distance**
- **Program: rapids, cuml, raft**
- **Compilation: cmake, python**
- **Binary: source build**

- **Data set: movielens, ny times bow, sec edgar, scrna**
- **Run-time environment: linux, 64-bit, x86_64**
- **Hardware: gpu, dgx1, v100, volta**
- **Metrics: End-to-end runtime performance**
- **Output: Runtimes printed to screen**
- **Experiments: Scripts provided to load data and execute algorithm on datasets with various distance measures**
- **How much disk space required (approximately)?: 20gb**
- **How much time is needed to prepare workflow (approximately)?: 1-2 hours**
- **How much time is needed to complete experiments (approximately)?: 2 days**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Yes**
- **Data licenses (if publicly available)?: Yes**

B.3 Description

B.3.1 How delivered

The artifact is delivered as a public github repository containing the datasets, Python benchmarking scripts, and instructions to build the source code.

The instructions for building the artifacts, along with the detailed specifications of the system used to produce the paper results, are located at
https://github.com/cjnolet/sparse_neighborhood_semiring_paper

B.3.2 Hardware dependencies

An Nvidia DGX1 was used to produce the results in the paper. Similar results can be produced with any Nvidia GPU which is capable of running Nvidia RAPIDS (Pascal architecture or newer and CUDA 11.0+).

B.3.3 Software dependencies

These benchmarks were executed using a custom branch of RAPIDS cuML version 0.19 and CUDA toolkit 11.0. All dependencies should be installed with Anaconda and instructions are provided in the documentation to install them.

B.3.4 Data sets

MovieLense 20M Ratings:
https://files.grouplens.org/datasets/movielens/ml-20m.zip

NY Times Bag of Words:
https://archive.ics.uci.edu/ml/machine-learning-databases/bag-of-words/docword.nytimes.txt.gz

SEC EDGAR Company Names:
https://www.kaggle.com/dattapiy/sec-edgar-companies-list

scRNA 70k Lung Cell:
https://rapids-single-cell-examples.s3.us-east-2.amazonaws.com/krasnow_hlca_10x.sparse.h5ad

B.4 Installation

After installing the software dependencies in an Anaconda environment by following the instructions provided in the Github repository, the cuML source code needs to be built and installed from the branch-0.19 branch.

1. Create an Anaconda environment with the necessary dependencies:
   https://github.com/rapidsai/cuml/blob/branch-0.19/BUILD.md#setting-up-your-build-environment.
2. Check out the code for both the baseline and our novel implementation outlined here:
   https://github.com/cjnolet/sparse_neighborhood_semiring_paper
3. Build from source using the build instructions at
   https://github.com/rapidsai/cuml/blob/branch-0.19/BUILD.md#installing-from-source.

B.5 Experiment workflow

1. Clone the repository containing the benchmark scripts and instructions:
   https://github.com/cjnolet/sparse_neighborhood_semiring_paper
2. Download all of the datasets and place them in a directory *datasets* in the root of the repository.
3. Clone and build the two provided cuml branches for the baseline and the optimized versions, installing only one at a time.
4. Execute the scripts in the *scripts* directory for both the baseline and optimized versions of cuML.
5. Execute the scripts in the *scripts* directory for the CPU benchmarks.

B.6 Evaluation and expected result

- All of the GPU benchmarks should be consistently faster than the CPU benchmarks.
- For *MovieLens*, the optimized version should be faster than the baseline for all distances.
- For *SEC Edgar*, the optimized version should be faster than the baseline for all distances.
- for *NY Times*, the optimized version should be faster than the baseline for all of the non-trivial distances. For the dot-product based distances, the baseline should be faster than the optimized version.
- for *scRNA*, the optimized version should be faster than the baseline for all of the non-trivial distances. For the dot-product based distances, the baseline shoul dbe faster than the optimized version.