Bit-GraphBLAS: Bit-Level Optimizations of Matrix-Centric Graph Processing on GPU

Jou-An Chen*, Hsin-Hsuan Sung*, Xipeng Shen*, Nathan Tallent†, Kevin Barker†, and Ang Li†
Department of Computer Science, North Carolina State University, Raleigh, NC, USA*
Pacific Northwest National Laboratory, Richland, WA, USA†
*{jchen73, hsung2, xshen5}@ncsu.edu, †{nathan.tallent, kevin.barker, ang.li}@pnnl.gov

***Abstract*—In a general graph data structure like an adjacency matrix, when edges are homogeneous, the connectivity of two nodes can be sufficiently represented using a single bit. This insight has, however, not yet been adequately exploited by the existing matrix-centric graph processing frameworks. This work fills the void by systematically exploring the bit-level representation of graphs and the corresponding optimizations to the graph operations. It proposes a two-level representation named Bit-Block Compressed Sparse Row (B2SR) and presents a series of optimizations to the graph operations on B2SR by leveraging the intrinsics of modern GPUs. Evaluations on NVIDIA Pascal and Volta GPUs show that the optimizations bring up to $40\times$ and $6555\times$ for essential GraphBLAS kernels SpMV and SpGEMM, respectively, making GraphBLAS-based BFS accelerate up to $433\times$, SSSP, PR, and CC up to $35\times$, and TC up to $52\times$.**

I. Introduction

Despite drawing great attention recently [1]–[11], accelerating graph analytics on GPUs remains challenging in that: (i) graphs are often irregular in connectivity, causing warp divergence, memory non-coalescing, and low arithmetic intensity; (ii) graphs are often large, many of which cannot directly fit into the caches, shared memory, or even the DRAM of GPUs. To address these challenges, the matrix-centric approach has been proposed and increasingly employed by GPU-based graph frameworks [3], [4], [12], [13]. Unlike traditional graph-centric frameworks that go through relevant nodes or edges iteratively, this approach employs sparse storage formats—such as compressed sparse column (CSC) or compressed sparse row (CSR)—to represent the adjacency matrix of a graph and then uses highly optimized linear algebra kernels (sparse matrix-vector and matrix-matrix multiplications, SpMV and SpGEMM) on the sparse formats for computation.

Although GraphBLAS has shown improved performance over traditional approaches in handling large graphs, there is still considerable potential to tap into. This work aims to unlock the potential by exploiting bit-level representations and optimizations on GPUs. The underlying observation is that for a large class of graphs (homogeneous graphs), a single bit is sufficient for indicating the adjacency relation of two vertices in a graph (1: adjacent, 0: not adjacent). Given that many graph algorithms center around computations upon adjacency matrices, using a bit-level representation can potentially reduce storage usage and improve computation efficiency.

Bit representations (bitmaps, bitvectors) have been used in vertex-based graph frameworks [14]–[16] for representing frontiers (i.e., active nodes); bit-level optimizations have, however, not yet been systematically explored in matrix-based graph frameworks. Existing GraphBLAS [17], [18] frameworks typically build on existing linear algebra libraries, which offer no bit-level representations of matrices or bit-level implementations of linear algebra functions.

This work answers three key research questions.

***RQ-1**: What storage format should be used for a binary adjacency matrix?*

Unlike the Boolean data structures (frontiers) in graph-centric frameworks, for matrix-based graph frameworks, the binary data structure in focus is the entire adjacency matrix, which sits at the center of the linear algebra operations in GraphBLAS. Systematic studies are needed for manipulating it; simply representing it as a bitmap cannot tap into the full potential of space savings by accommodating many unnecessary zeros; that also causes difficulties for the graph operations to leverage the highly tuned matrix-based libraries.

Based on the properties of adjacency matrices and various tradeoffs, we design a storage format, namely Bit-Block Compressed Sparse Row (B2SR). B2SR is inspired by the Block Compressed Sparse Row (BSR) format [19]. It takes a two-level structure: The upper level is similar to BSR's upper level, using a sparse format to represent the locations of non-zero blocks (or called submatrices); the lower level differs from BSR in that it represents each non-zero block as a dense *bit* matrix—that is, each element in the block becomes one bit in the representation. The representation allows it to efficiently harvest the hardware computation capability at the low level and at the same time leverage the (regional) sparsity at the high level. Balancing the space savings and the indexing overhead lays the foundation for tapping into the potential of sparse binary matrices.

***RQ-2**: How to efficiently compute on the new representation?*

We propose several new algorithms to implement the critical linear algebra kernels (SpMV and SpGEMM) for manipulating sparse matrices represented in B2SR. The design carefully tailors the kernel implementations around the efficient low-level bit-manipulation intrinsics on GPUs. It exploits the new optimizations and hardware-specific capability brought by the latest GPUs. These kernels lay the foundation for efficient manipulations of sparse binary matrices for graph analytics.

***RQ-3**: What are the performance implications?*

We evaluate the B2SR-based SpMV and SpGEMM on 521 binary matrices and five graph algorithms. The benefits are significant. B2SR provides up to $32\times$ space savings. On two generations of GPUs (NVIDIA's Pascal and Volta), we observe $40\times$ and $6555\times$ maximum speedups over the state-of-the-art sparse linear algebra libraries cuSPARSE [20] and GraphBLAST [4]. On graph algorithms, it offers up to $433\times$ acceleration on Breadth-first-search (BFS), $55\times$ on Single-Source-Shortest-Path (SSSP), $28\times$ on PageRank (PR), $69\times$ on Connected Component (CC) algorithms, and $52\times$ on Triangle Counting (TC) algorithm over GraphBLAST [4], a GPU graph processing framework with state-of-the-art performance. As is well known [21]–[23], no sparse format fits all matrices. So despite the effectiveness of B2SR, there are matrices that fit other sparse formats better. We provide a brief discussion and a simple sampling approach to assisting users in applying B2SR.

II. Background and Related Work

Graph programming frameworks are based on either graph-centric abstraction (vertex- or edge-centric) [1], [24]–[28] or matrix abstraction. For the performance advantages and direct leverage of advances in high-performance linear algebra libraries, matrix abstraction has received increasing interest in recent years. GraphBLAS [17], [18] is the mathematical core of matrix-based graph frameworks. It models graph traversal as operations on semi-rings. Frameworks that implement the standard include nvGraph [12], cuGraph [29], SuiteSparse [13], GraphBLAS template library (GBTL) [3], GraphBLAST [4], and so on. Among them, GraphBLAST [4] represents state of the art, achieving high performance on GPU by exploiting input and output sparsity [30] and enhanced load balance by exploiting the memory access patterns of sparse matrix multiplication.

These frameworks are mainly built under the same line of unified graph construct–using CSC or CSR to establish floating-point element space and perform matrix operations with underlying linear algebra libraries. They have not exploited bit-level optimizations. Even though in the sparse linear algebra libraries, bitmaps or bitvectors may be used to index the non-zero elements in a sparse matrix [31]–[33], those libraries fundamentally assume that the sparse matrices are general rather than binary matrices. They hence leave an immense performance potential untapped (as our comparison in Section VI shows).

In graph-centric frameworks (e.g., GraphMat [14], Graphlt [15], SpbLA [16]), there are some Boolean data structures (e.g., frontiers or active nodes), which are sometimes represented in bitmaps or bitvectors. Some works [34]–[36] exploit binary encoding or compression to achieve storage reduction and algorithm acceleration. For matrix-centric graph frameworks, the binary data structure in focus is the entire adjacency matrix, which sits at the center of the linear algebra operations in GraphBLAS—simply representing it as a bitmap leaves not only lots of potential

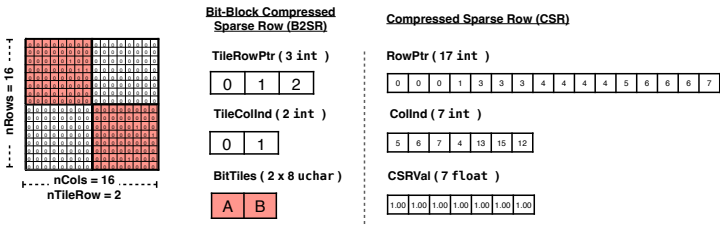


Figure 1: Illustration of the Bit-Block Compressed Sparse Row (B2SR).

for space savings but also causes difficulties for the graph operations to leverage the highly tuned matrix-based libraries. Therefore, systematic studies are needed for manipulating it.

Bit-manipulation primitives have been used in enhancing the performance of deep neural networks (DNNs) on GPUs [37]–[39], ASICs, and FPGAs [40]–[42]. They are about dense binary operations, so they cannot efficiently handle sparse ops as those in graph processing. The irregularity of sparse matrices and accesses makes the issue more complex.

III. Representation: B2SR

This section presents B2SR, the format we have designed for representing an adjacency matrix. Figure 1 illustrates the design. The principle we followed is that the representation should reduce the space cost as much as possible and at the same time facilitate the acceleration of the core graph operations. Drawing on the inspiration of BSR, we create the two-level representation of B2SR. The top-level takes advantage of well-proven effective sparse formats (CSR or CSC) to represent non-empty blocks. The bottom-level treats each non-empty block as a dense block and packs its elements into a bit representation. The combination of sparse formats and bit representation minimizes space usage, while the block-level dense bit format preserves low-level regularity, making efficient computation possible. We will explain the format in detail.

A. Bit-tile Indexing System

Since adjacency matrices are all square, it is natural to have the number of tile rows (*nTileRow*) set as $\frac{nRows+tileDim-1}{tileDim}$, where $tileDim$ is the dimension (or bit-width) of the tile (e.g., 4, 8, 16, 32). The number of non-empty bit-tiles can then be inferred from the nonzeros' coordinates of the sparse matrices. In our implementation, we use cuSPARSE's **`cusparseXcsr2bsrNnz()`** API to obtain the number of non-empty tiles from the CSR format. We utilize indexing arrays to record the coordination of the non-empty byte-aligned bit-tiles. The proposed format comprises three arrays:

- Tile row indices (*TileRowPtr*): an integer array with the size of the number of tile-rows. It records the bit-tiles' row indices. It is an accumulated array with the i-th element recording the sum of total non-empty bit-tiles counting from the first tile row to the (i-1)-th row. Therefore, $TileRowPtr[i+1]-TileRowPtr[i]$ suggests the number of non-empty bit-tiles in the i-th tile-row.

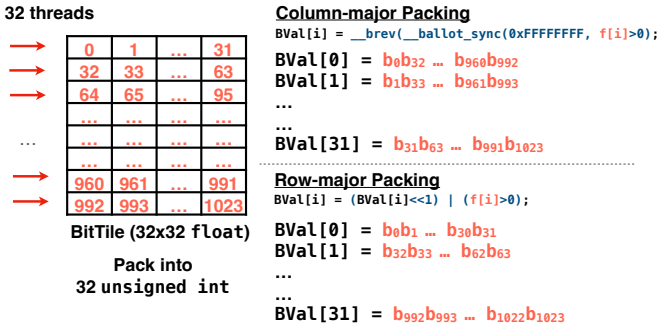


Figure 2: Illustration of column-major and row-major bit packing.

Table I: Binarized packing format.

Tile Size	CSR Storage (at most)	Binarized Packing	Space Saving per Tile
4×4	4×4 `float`	4×1 `unsigned char`	16×
8×8	8×8 `float`	8×1 `unsigned char`	32×
16×16	16×16 `float`	16×1 `unsigned short`	32×
32×32	32×32 `float`	32×32 `unsigned int`	32×

- Tile column indices (*TileColInd*): an integer array with the size equal to the number of non-empty bit-tiles. This array is for recording the tile column indices in the tile coordination system.
- Bit-tiles storage (*BitTiles*): a bit-packing type (unsigned char, unsigned short, unsigned int, or unsigned long long int) array with a size equal to the *tileDim*×*numofTiles* (number of non-empty bit-tiles). It stores the binarized non-empty bit-tiles' layout in the order of their tile column indices.

The proposed format has several merits: (1) It allows simpler transpose of the sparse matrix. By transforming the *TileRowPtr* and *TileColInd* from CSR to CSC, the sparse matrix is transposed. (*BitTiles* do not require transpose since we default the **`mxv()`** and **`mxm()`** algorithm to access the content of a tile always in row-by-row order.) We use cuSPARSE's **`cusparseScsr2csc()`** API to enable this function in our implementation. (2) The format has a better data accessing locality for SpGEMM and SpMV when computing in tile-row by tile-row order; Since the storage format is similar to CSR, we can use existing optimization on CSR-based algorithms to spearhead the linear algebra kernels in use. (3) In *BitTiles*, the format provides storage-saving compared to CSR. The proposed format carries extra zeros than CSR and COO, which store only the nonzeros. Nevertheless, with the binarized packing yielding up to 32× space-saving per tile, most sparse matrices can still benefit from storage compression, especially when configured in small tile size.

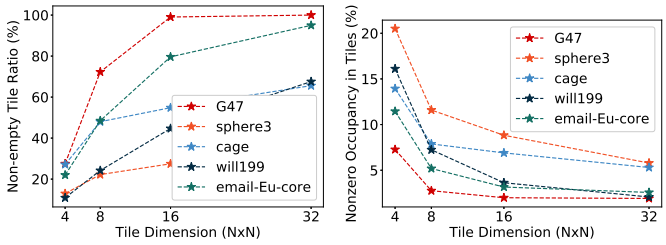
B. Bit Packing

Figure 2 presents the column-major and row-major packing of a bit-tile. We adopt *column-major packing* as default when converting CSR to B2SR. The transpose of B2SR tiles can be achieved by storing the additional row-major layouts. To figure out the proper byte-addressable data types to carry bits in proximity, we explore the packing granularity from 8-bit to 32-bit (reference Table I). Additionally, we use half of the space in an **`unsigned char`** to allow 4-bit (nibble) packing, which further reduces half in one dimension to carry unnecessary zero when the matrix is extremely sparse. The packing result yields four variants of B2SR format: B2SR-4 (for 4×4 tile size), B2SR-8 (for 8×8 tile size), B2SR-16 (for 16×16 tile size), and B2SR-32 (for 32×32 tile size). Generally, the space-saving depends on how the adjacency matrix is initially being stored. State-of-the-art GPU graph frameworks mostly use **`float`** to carry the elements, so our bit-packing can generally provide up to 32× storage savings per square tile; for frameworks that use **`double`** to carry the elements, the bit-packing results in up to 64× savings in space. This indicates an ability to store 32× or 64× larger graphs using the same amount of space. This also brings a potential reduction in required data accessing bandwidth during computation to enable higher throughput.

Bit-packing overhead To transform CSR to B2SR, we parallelize each tile-row's encoding procedure for the large graph. For the 4, 8, 16, or 32 continuous elements in CSR's **`RowPtr`**, we use **`cusparseXcsr2bsrNnz()`** and **`cusparseScsr2bsr()`** to obtain tile-row index and full-precision tiles along a tile-row. Next, column-major or row-major bit-packing kernels are applied to process each tile's encoding. The routine's overall cost is about 3 to 34 ms. In practical applications, a graph can be reused by many users; even within one execution, a graph is often used repeatedly (e.g., for iterative processing). So despite format conversion may be needed, such a one-time cost can be greatly amortized in these conditions.

C. Sampling Profile and Tile Size Configuration

While the proposed method provides storage compression and performance gains in BLAS operators, it is evident that not all graph matrices are suitable for converting to Bit-GraphBLAS binarized format. For example, graph matrices with randomly distributed connections (nonzeros) or relatively dense patterns may not necessarily benefit from this format. When each bit tile does not capture enough nonzeros, we can have many empty bit-rows in tiles after bit packing. Converting the matrices from CSR to B2SR is not ideal since it expands the total amounts of storage. In addition, it adds additional per-thread workloads (OPS) compared to CSR's BLAS kernels in terms of operators. We observe that the tile size selection trade-off can be considerably different in various matrix patterns. Figure 3a shows that when the tile size equals 4×4, there are less than 30% non-empty tiles; when the tile size equals 32×32, non-empty tiles reach more than 80% for some matrices. The reason is that although increasing tile size can decrease the number of non-empty tiles, the reduction is often less than 4× (the times of per tile size increment), causing the ratio of non-empty tiles to increase ultimately. We find that despite the increment, the total B2SR byte size can sometimes decrease because of the



(a) Tile proportion trend. (b) Nonzero occupancy trend.

Figure 3: Effect trends with increment in tile size dimension.

reduction in the number of tiles and indexing arrays. For instance, in matrix *mycielskian12*, we have CSR storage at 3.12 (MB), B2SR-4 at 675.70 (KB), B2SR-8 at 361.46 (KB), B2SR-16 at 358.89 (KB), and B2SR-32 at 429.89 (KB). The total byte size of the format does not monotonically increase as tile size increases. From the other angle, Figure 3b presents the reduction in the average occupancy of nonzeros in the non-empty tiles. The percentage of actual nonzeros in tiles can drop from 20% to less than 5% as the tile dimension differs. If the bit-tile is too large, the computation may waste processing too many tiles; if the bit-tile is too small, the indexing array may carry more unit workloads. Therefore, to fully utilize the double benefits (compression and computation efficiency) of Bit-GraphBLAS, graph users can first identify the potential benefits offline through our sampling profile method. In this way, when tackling extensive graphs or cumbersome, repetitive graph computations, users can experience worthy time and labor cost saving with only one-time format conversion to B2SR. The sampling works as in Algorithm 1. The user first specifies the number of rows to sample as N for $N \leq NumOfRows$ (sampling more rows can accurately capture matrix characteristics but induces more significant overheads). We then have N random indices as the random index set $S \subseteq [0, 1, 2, ..., N]$.

Algorithm 1 Sampling Profile Scheme

```
for i in S do
    for j = RowPtr[i], ..., RowPtr[i+1] do
        for k in {4, 8, 16, 32} do
            ColCounter[k][i][j/k] += 1
        end for
        NnzElement[k][i] = RowPtr[i+1]-RowPtr[i]
        NnzBitRow[k][i] = size of ColCounter[k][i]
    end for
    EstCompressionRate[k] = avg(NnzElement[k][i] / NnzBitRow[k][i])
end for
```

The sampling result provides a rough estimation of the compression rate of Bit-GraphBLAS on B2SR-4 to B2SR-32. Users can select the affordable compression rate and tile size configuration.

IV. Bit Operations and BLAS Kernels Design

This subsection discusses the code patterns and scheme designs that leverage the proposed storage format. Our implementation extensively uses the GPU integer intrinsics for bit-operations and other optimizations for computing efficiency, load balance, and memory performance.

Table II: BMV schemes used in the algorithms.

Scheme	Input Matrix A	Input Vec. B	Output Vec. C
`bmv_bin_bin_bin()`	1-bit	1-bit	1-bit
`bmv_bin_bin_full()`	1-bit	1-bit	32-bit
`bmv_bin_full_full()`	1-bit	32-bit	32-bit
`bmv_bin_bin_bin_masked()`	1-bit	1-bit	1-bit
`bmv_bin_bin_full_masked()`	1-bit	1-bit	32-bit
`bmv_bin_full_full_masked()`	1-bit	32-bit	32-bit

We briefly introduce the GPU bit operation intrinsics that will be used in this work, with respect to the BSTC bit-block abstraction [37]: (1) **`__popc()`**: The population count function is for efficient bit-accumulation across a singe bit-row. CUDA supports population count along a 32-bit unsigned int. Paired with the logical AND operation, it can perform bit-dot-product for two 32-bit bit-rows. (2) **`__shfl_sync()`**: This intrinsic is for exchanging the bit-row across the lanes of a warp. For BMM, it facilitates a faster bit-dot-product between a bit-row and multiple bit-columns. (3) **`__ballot_sync()`**: This warp-vote intrinsic returns a 32-bit unsigned integer whose N-th bit indicates a predicate setting by the N-th thread of a warp (assuming all threads are active). This is essentially equal to transposing a bit-column to a bit-row. Since bits are indexed from right to left in the bit-row packing, the function is equivalent to a 90° clockwise transposition to a bit-row. (4) **`__brev()`**: This intrinsic is used in bit-packing. Together with **`__ballot_sync()`**, this function rotates a bit-column 90° anti-clockwise into a bit-row.

The kernels of matrix-centric graph computing are matrix-vector and matrix-matrix computations. Table II and III list the core schemes we have implemented. They correspond to the different scenarios of the inputs and outputs (binary "1-bit" or full precision "32-bit"). We use two of the schemes to explain our implementations:

Listing 1. As an example scheme of Binarized Sparse Matrix Multiply Vector (BMV), Listing 1 shows the code of **`bmv_bin_bin_full()`** in 32×32 tile size. The function demonstrates the bit multiplication between the binarized bit-tiles and the binarized vector. The result is a full-precision vector. The computation is as follows:

$$A_{i,j}^{(b)} \times b_j^{(b)} = c_i = \texttt{\_\_popc}(A_{i,j}^{(b)} \,\&\, b_j^{(b)})$$

Before computation, the sparse matrix is packed into the hierarchical storage format — the vector is binarized into the column-major order with 32 consecutive elements compacted as an unsigned int. This allows the bit-columns to be fetched according to the same indexing system and enables fast bit-dot-product with each bit-row in the bit-tiles. The **`A`** variable is a bit-row in a tile, and the **`B`** variable is the binarized vector. The output array **`C`** is a vector in full precision. In each warp of the thread block, the number of bit-tiles to be computed is indicated by the *TileRowptr*. In each iteration, the bit-tile and the 32-binarized vector perform bit-matrix-vector-product using bit-wise **`AND`** and **`__popc()`**. Each lane

Table III: BMM schemes used in the algorithms.

Schemes	Input Matrix A	Input Mat. B	Output Value
`bmm_bin_bin_sum()`	1-bit	1-bit	32-bit
`bmm_bin_bin_sum_masked()`	1-bit	1-bit	32-bit

is responsible for the output of a bit-row, whereas the registers being private to each thread accommodates the intermediate result per bit-tile. Finally, the content of the register is stored in the corresponding row of the resulting vector.

Listing 1 BMV. A tile row is computed in a warp.

```
int row_start = rowptr[bx];
int row_end = rowptr[bx+1];
if (row_start != row_end) {
    const unsigned* Asub = &(A[row_start*32]);
    const unsigned* Bsub = &(B[0]);
    T* Csub = &(C[bx*32]);
    register unsigned Cm[1] = {0};
    for (int i=row_start; i<row_end; i++) {
        unsigned r0 = Asub[(i-row_start)*32+laneid];
        unsigned r1 = Bsub[(colind[i])];
        Cm[0] += __popc(r0 & r1);
    }
    Csub[laneid] += (Cm[0]);
}
```

Listing 2. As an example of Binarized Sparse Matrix Multiply Binarized Sparse Matrix (BMM), Listing 2 shows the code of **`bmm_bin_bin_sum()`** for B2SR-32. The *A* and *B* variables are the two bit-vector in tiles of the input sparse matrices. The output **`C`** is a single variable in full precision, summing up the nonzeros (1s) of the resulting bit matrix. In each warp of the thread block, the number of bit-tiles in **`A`**'s tile row is indicated by the *TileRowptr* of **`A`**. In the outer "for" loop, the *TileColInd* of each bit-tile is used to retrieve the corresponding tile rows and bit-tiles of these tile-rows. The inner "for" loop performs the bit-tile-matrix-matrix-multiplication, where **`__shfl_sync()`** is used to retrieve the **`B`**'s bit-vectors in each lane. The temporary result of each bit-vector in **`B`** is accumulated in separate registers for avoiding race conditions. Finally, the content in the 32 registers is summed up. The sum is atomically added to a single destination element in **`C`**.

Allowing efficient full-precision vector load In BMV and BMM, the bit-tiles in a tile-row are handled in a warp of 32 threads, following the warp-consolidation model [43]. By default, each of the thread blocks contains only one warp. So up to 64 thread blocks can be freely scheduled by a single SM scheduler. This works fine when kernels are computing in a binarized vector or matrix. However, in **`bmv_bin_full_full()`**, when engaging a full-precision vector as the multiplier, the one-warp-per-thread-block design impedes the flexibility to preload the common vector portions for the neighboring tile rows into shared memory. Thus, we implement the kernel scheme with 32 warps processing consecutive 32 tile-rows in a thread block. This further enhances the spatial locality of the computation workload. Figure 4 shows the thread mapping for B2SR-32, B2SR-16, B2SR-8, B2SR-4. The execution latency is generally shorter when

Listing 2 BMM. A tile row is computed in a warp.

```
T* Csub = &C[0];
register int Cm[32] = {0};
int sum = 0;
int A_row_start = A_rowptr[bx];
int A_row_end = A_rowptr[bx+1];
const unsigned* Asub = &(A[A_row_start*32]);
for (int i=A_row_start; i<A_row_end; i++) {
    unsigned r0 = Asub[(i-A_row_start)*32+laneid];
    int A_col = A_colind[i];
    int B_row_start = B_rowptr[A_col];
    int B_row_end = B_rowptr[A_col+1];
    const unsigned* Bsub = &(B[B_row_start*32]);
    for (int j=B_row_start; j<B_row_end; j++) {
        unsigned r1 = Bsub[(j-B_row_start)*32
                    +laneid];
        #pragma unroll
        for (int k=0; k<32; k++){
            unsigned r2 = __shfl_sync(0xFFFFFFFF,
                        r1, k);
            Cm[k] += __popc(r0 & r2);
        }
}
```

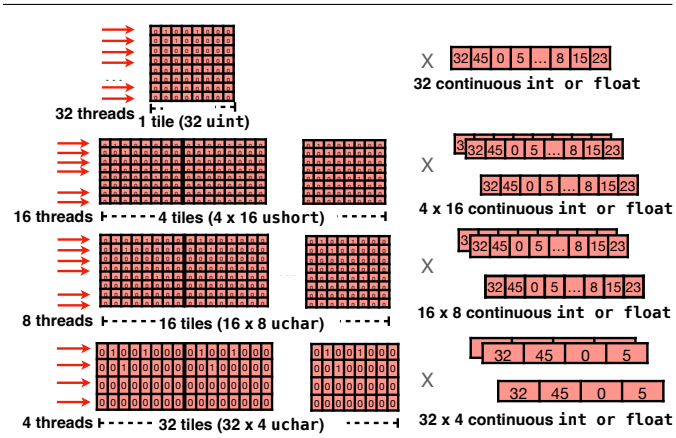


Figure 4: Illustration of **`bmv_bin_full_full()`**.

cooperatively loading 128 bytes (equal to the cache line size of GPUs in our experiment) of binarized tiles from global memory. At the same time, the corresponding number of full-precision subvectors have to be loaded for multiplication. We set the thread block to contain 1024 threads to load the vectors into shared memory before multiplication. Bit-vectors on tile-row with the same column index can share the preload vectors from shared memory.

V. Graph Algorithms

This section explains how graph programs can be implemented upon the core operations.

Matrix-centric graph computing models graph traversals as operations on semirings [44]. As shown in Table IV,

Table IV: Semiring support with BMV and BMM schemes.

Semiring	Domain	Algorithm	Scheme
Boolean	$\{0,1\}$	BFS, diameter, MIS, GC	**`bin-bin-bin`**
Arithmetic	$\mathbf{R}$	LGC, PR, TC	**`bin-full-full`** or **`bin-bin-full`**
Tropical Min-plus	$\mathbf{R} \cup \{+\infty\}$	SSSP, CC	**`bin-full-full`**
Tropical Max-times	$\mathbf{R}$	MIS, GC	**`bin-full-full`**

our implementation can support the key semiring domain operations when performing **vxm()**, **mxv()**, **mxm()**. After the adjacency matrix is in B2SR, it remains binary throughout all operations. The vectors representing the frontier nodes are all in dense format. They can be either binarized for binary semiring or full-precision (float, unsigned, bool, etc.) for the non-binary domains to support a variety of graph algorithms. We also implement efficient masking schemes for both BMV and BMM: **bmv_bin_bin_bin_masked()**, **bmv_bin_bin_full_masked()**, **bmv_bin_full_full_masked()**, and **bmm_bin_bin_sum_masked()**. We next use two graph algorithms to illustrate how to utilize these kernel backends when writing graph algorithms.

Breadth-First-Search (BFS) Breadth-first-search uses boolean semiring. In each iteration, the **vxm()** performs one-degree edge traversal to all the connected vertices. A mask of visited vertices is applied at the end to filter out the visited results. We introduce **bmv_bin_bin_bin_masked()** to enable this. GraphBLAST uses early exit to eliminate the masked element operations in their masked **vxm()**. Yet, a similar strategy does not apply to our case. In our implementation, the consecutive rows in a tile row are operated in the same warp. Early exit causes a performance penalty because of warp divergence. Therefore, in the **bmv_bin_bin_bin_masked()** kernel, the bitmask is applied right before the output store, having bit-wise **AND** with the negation of visited vertex vector (indicates unvisited vertices).

Single-Source Shortest-Path (SSSP) We implement the algorithm with delta-stepping SSSP [8] as in GraphBLAST. SSSP utilizes tropical min-plus semiring. The intermediate vectors are reduced by minimum operation. **bmv_bin_full_full()** maintains the multiplier vector in full-precision, allowing it to carry minimum distance values. To realize the relaxation, we set an extra condition in the **bmv_bin_full_full()** such that the 0s in the adjacency matrix are identified as infinite (∞), indicating unreachable. Only 0s along the diagonal are treated as actual zeros, which we omit their self-connectivity. Within a warp, a thread reduces all non-zero full-precision values along the multiplier vector by **Min()** (for B2SR-32). In B2SR-4, B2SR-8, and B2SR-16, since we use more than one thread to process the values along the multiplier vector, **atomicMin()** is applied to avoid race conditions.

PageRank (PR) PR uses arithmetic semiring. In each iteration, the page rank vector is multiplied by the *column stochastic adjacency matrix*. The *column stochastic adjacency matrix* is the adjacency matrix with each out-vertex connectivity divided by the vertex's out-degree. Since the page rank vector is in full-precision, we use **bmv_bin_full_full()** with an auxiliary vector **v_out_degree** to accommodate each vertex's out-degree. For each 1 on the matrix, the corresponding value on the page rank vector is divided by its out-degree on **v_out_degree**. Eventually, the intermediate vector is summed up with add operation to the output, indicating the weighted sum. Likewise, B2SR-4, B2SR-8, and B2SR-16 requires **atomicAdd()** since more than one thread process the workload along the vector cooperatively.

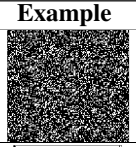
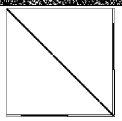
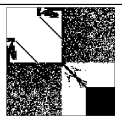
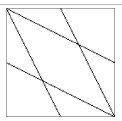
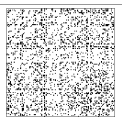
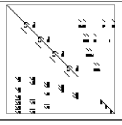
Category	Example	% in Dataset	Description
Dot		36.66%	Contains nonzeros scatter randomly
Diagonal		45.87%	Nonzeros are centralized around diagonal
Block		24.95%	Square or rectangular blocks, countours
Stripe		13.05%	Contain one or more lines in various directions
Road		5.18%	Nonzeros in regular distribution
Hybrid		25.72%	A combination of more than two patterns above

Table V: Matrix pattern category (black indicates nonzeros).

Connected Component (CC) We follow the CC implementation in GraphBLAST, which is based on the FastSV linear-algebraic connected component alogrithm [45], [46]. Similar to SSSP, CC uses tropical min-plus semiring. We adopt **bmv_bin_full_full()** since the frontier vector should be in full-precision. The **mxv()** is achieved by reducing the nonzero full-precision values along the intemediate vector using **Min()** and **atomicMin()**.

Triangle Counting (TC) We implement TC as in GraphBLAST, following Azas and Buluc's [47] and Wolf's [48]. The TC uses arithmetic semiring. It is achieved by multiplying the lower triangle of the adjacency matrix (L) with the transpose of itself (L^T) and then applying (L) as the mask to generate the output matrix. Ultimately, the non-zeros is summed up into one full-precision value; therefore, we fuse the reduction sum kernel with **mxm()** and directly perform **atomicAdd()** to global sum once a bitmap subroutine finishes. Since both input matrices and mask matrix can be sufficiently represented in binary format, so we realize the kernel through **bmm_bin_bin_sum_masked()**.

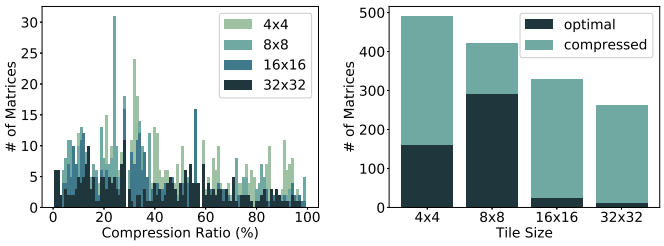
VI. Evaluation

A. Experiment Configuration

Dataset We use all 521 binary square matrices in the SuiteSparse Matrix Collection [13]. The set of matrices contains the number of rows and columns ranging from 2 to 214,005,017 and the number of nonzeros from 2 to 11,588,725,964. To better summarize the similarity between the matrices with higher or lower performance in the evaluation, we further

Table VI: GPU memory and cache hierarchy.

GPU	Arch	SMs	Shared/ SM	Shared/ Block	RAM	Memory Bandwidth	L1 Cache Size/SM	L2 Cache Size
GTX1080	Pascal	20	64KB	48KB	8GB	320GB/sec	48KB	2048KB
TITAN V	Volta	80	96KB	96KB	12GB	653GB/sec	96KB	4608KB



(a) Compression rate histogram. (b) Optimal compression tile size.

Figure 5: Compression result of the 521 binary matrices.

classify the matrices into six categories based on their patterns in Table V.

GPU Environments We evaluate the proposed format and computation core functions on two NVIDIA GPU architectures, including Pascal and Volta. With the compute capability 6.0 and 7.0 configured, respectively, how the proposed format adapts to each hardware-specific variance is worth seeing. Table VI shows the configured SM information and memory hierarchy size of the two GPU architectures in the evaluation. We use CUDA version 10.0 across all our evaluations.

Algorithm Parameters The optimization setup of GraphBLAST's algorithms is based on the default running script provided in their GitHub repository [49]. BFS is with early-exit, structure-only, and operand reuse enabled. PR is limited to a maximum iteration of 10. The alpha parameter is set to 0.85, and pdfilon is set to $1e{-}9$. The runtime for all algorithms is measured by the average of 5 runs.

B. Storage Efficiency

B2SR brings significant storage savings for large sparse matrices. We show the compression ratio of the 521 binary graph matrices with respect to the default 32-bit floating-point CSR. The compression ratio thus is defined as: $\frac{B2SR\_size}{CSR\_size}$. A lower value indicates a better compression rate. The compression ratio depends mainly on the nonzero distribution of the binary matrices. Figure 5a shows the compression ratio on the x-axis and the histogram recording the number of matrices using the four B2SR formats on the y-axis. In Figure 5b, the y-axis shows the number of matrices that belong to their: (1) optimal size (colored in blue): the least storage size required among the four B2SR formats of a matrix. (2) compressed size (colored in green): the B2SR format can provide a compression ratio $< 100\%$ for a matrix. For optimal, 162 matrices appear at B2SR-4, 291 matrices at B2SR-8, 26 matrices at B2SR-16, and 12 matrices at B2SR-32. For compressed, 491 matrices can have a compression ratio $< 100\%$ on B2SR-4, 421 on B2SR-8, 329 on B2SR-16, and 263 on B2SR-32.

C. Overview of the Performance Gains

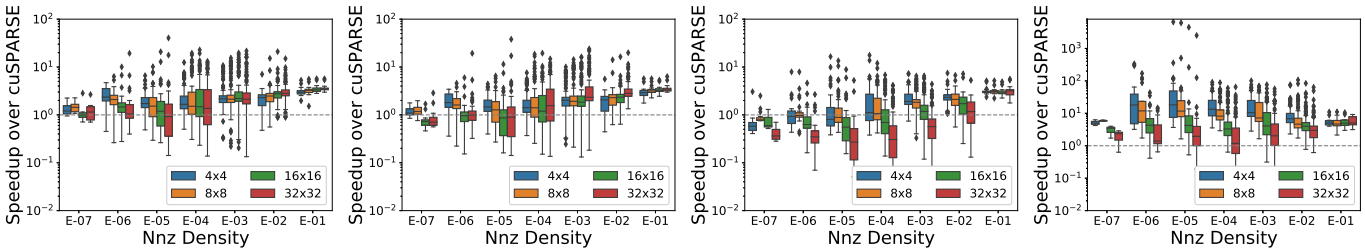
There are multiple factors that contributed to the significant speedups B2SR has achieved. In addition to the gains by using native bit-level intrinsics such as **`__popc()`**, we have observed that more performance is from the reduced memory transactions and enhanced data locality. For example, for the matrix *mycielskian8*, by using B2SR, the number of global memory load transactions reduces by 4× from 6630 to 1826, while the L1 cache hit-rate increases by 24% from 65.63% to 81.83%. We also observed different sweet areas for different B2SR tile sizes—such as, the smaller tile sizes (e.g., 4, 8) draw better L1 hit rates while the larger tile sizes (e.g., 32, 64) favor coalesced memory access—and also the impact from the profiles of individual graphs. In the next two sections, we describe the performance evaluation of the linear algebra kernels (BMV and BMM) and the five graph algorithms, respectively.

D. Linear Algebra Kernels

In this subsection, we evaluate the basic arithmetic cores BMV and BMM in terms of different schemes. We evaluate the speedups of the kernels over cuSPARSE's SpMV (**`cusparseScsrmv()`**) and SpGEMM (**`cusparseScsrgemm()`**) with CSR in 32-bit floating-point nonzero storage. We compare the performance of each kernel scheme on B2SR-4, B2SR-8, B2SR-16, and B2SR-32. In Figure 6 and 7, the y-axis is the average speedups (of 5 runs) over cuSPARSE and the x-axis is the nonzero density of the matrices which is defined as $\frac{\#\_of\_nonzeros}{\#\_of\_elements}$. A higher nonzero density (more to the right in the figure) implies a denser matrix while a lower one (more to the left) implies a sparser matrix.

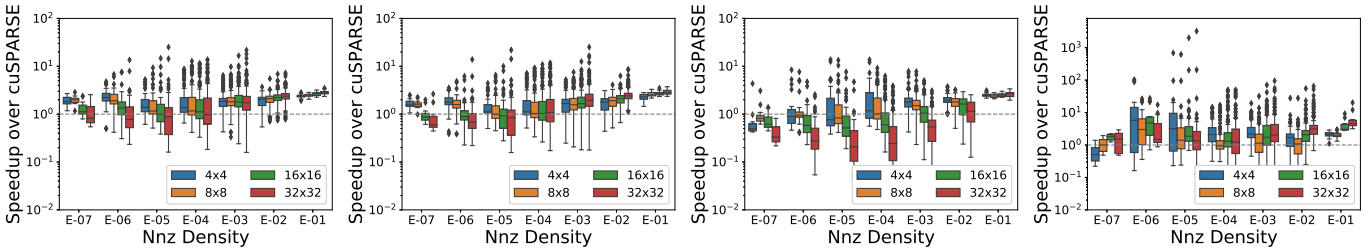
BMV In BMV, we implement three schemes as the basic SpMV kernels (see Table II). We evaluate the performance of **`bmv_bin_bin_bin()`**, **`bmv_bin_bin_full()`**, **`bmv_bin_full_full()`** and compare it with cuSPARSE's **`cusparseScsrmv()`**. **`bmv_bin_bin_bin()`**'s performance is shown in Figure 6a and 7a. Although the arithmetic capability of this scheme is limited to only binary operations, it allows a minimal memory footprint by keeping all value storage in binarized format. On GTX1080, **`bmv_bin_bin_bin()`** achieves an average speedup of 2.41× in B2SR-4, 2.74× in B2SR-8, 2.91× in B2SR-16, and 2.85× in B2SR-32. The max speedup over baseline is 40.47× that appears at matrix *ins2* with B2SR-32. On Titan V, **`bmv_bin_bin_bin()`** achieves an average speedup of 2.04× in B2SR-4, 2.17× in B2SR-8, 2.18× in B2SR-16, and 2.26× in B2SR-32. The max speedup over baseline is 25.16× that appears at matrix *ins2* with B2SR-32.

In **`bmv_bin_bin_full()`**, the vector input is binarized with the same tile dimension of the binary adjacency matrix. Compared to **`bmv_bin_full_full()`**, it requires less vector load bandwidth per matrix-vector multiplication. The inner product of each bit-row is done by bit-wise AND and population count of the resulting bit-row. In the 521 binary matrices, the runtime speedup of **`bmv_bin_bin_full()`** over cuSPARSE's full-precision CSR SpMV is shown in Figure 6b and 7b. On GTX1080, **`bmv_bin_bin_full()`** achieves an average speedup of 2.06× in B2SR-4, 2.36× in B2SR-8, 2.22× in B2SR-16, and 2.97× in B2SR-32. The



(a) **bmv_bin_bin_bin()**. (b) **bmv_bin_bin_full()**. (c) **bmv_bin_full_full()**. (d) **bmm_bin_bin_sum()**.

Figure 6: Arithmetic kernel speedup over CuSPARSE on GTX1080 (Pascal) GPU.



(a) **bmv_bin_bin_bin()**. (b) **bmv_bin_bin_full()**. (c) **bmv_bin_full_full()**. (d) **bmm_bin_bin_sum()**.

Figure 7: Arithmetic kernel speedup over CuSPARSE on Titan V (Volta) GPU.

Table VII: SpMV-based graph algorithm performance on GTX1080 (Pascal) GPU.

Matrix	Algorithm runtime (ms)	BFS			SSSP			PR			CC		
		GBlst	Ours	Speedup	GBlst	Ours	Speedup	GBlst	Ours	Speedup	GBlst	Ours	Speedup
delaunay_n14	algorithm	3.73	1.09	**3**×	8.35	1.99	**4**×	1.04	0.13	**8**×	1.89	0.61	**3**×
	kernel	2.43	0.35	**7**×	5.11	1.08	**5**×	0.51	0.03	**14**×	1.02	0.14	**7**×
se	algorithm	2.06	0.54	**4**×	4.75	1.31	**4**×	1.22	0.17	**7**×	0.72	0.20	**4**×
	kernel	1.68	0.15	**12**×	2.75	0.80	**3**×	0.56	0.03	**20**×	0.27	0.06	**5**×
debr	algorithm	5.85	2.27	**3**×	28.88	15.64	**2**×	14.39	4.20	**3**×	10.96	3.39	**3**×
	kernel	2.74	1.83	**2**×	14.64	11.42	**1**×	4.51	0.52	**9**×	1.91	1.05	**2**×
ash292	algorithm	1.45	0.02	**61**×	2.97	0.51	**6**×	0.98	0.12	**8**×	0.65	0.17	**4**×
	kernel	0.83	0.01	**152**×	2.03	0.17	**12**×	0.46	0.01	**35**×	0.24	0.03	**9**×
netz4504_dual	algorithm	2.33	0.02	**98**×	5.32	0.88	**6**×	0.97	0.06	**16**×	0.73	0.21	**4**×
	kernel	1.39	0.01	**221**×	3.30	0.28	**12**×	0.49	0.01	**34**×	0.28	0.03	**9**×
minnesota	algorithm	6.22	0.08	**79**×	10.48	2.66	**4**×	0.99	0.06	**17**×	0.94	0.26	**4**×
	kernel	4.67	0.02	**243**×	6.36	0.83	**8**×	0.47	0.01	**33**×	0.38	0.04	**9**×
jagmesh6	algorithm	7.35	0.02	**318**×	11.50	2.32	**5**×	0.97	0.07	**15**×	0.92	0.26	**3**×
	kernel	5.74	0.01	**1025**×	7.38	0.92	**8**×	0.47	0.02	**30**×	0.40	0.04	**10**×
uk	algorithm	10.15	0.22	**46**×	15.59	2.97	**5**×	1.00	0.07	**15**×	3.15	0.93	**3**×
	kernel	8.29	0.05	**165**×	9.04	1.19	**8**×	0.48	0.02	**26**×	1.73	0.29	**6**×
whitaker3_dual	algorithm	25.27	0.06	**433**×	32.69	11.80	**3**×	1.07	0.12	**9**×	1.93	0.67	**3**×
	kernel	21.67	0.02	**1414**×	19.54	6.17	**3**×	0.49	0.02	**25**×	0.85	0.11	**8**×
rajat07	algorithm	4.08	0.03	**160**×	9.82	2.12	**5**×	1.11	0.12	**9**×	1.13	0.36	**3**×
	kernel	2.20	0.01	**250**×	6.05	1.15	**5**×	0.52	0.03	**16**×	0.50	0.09	**6**×
3dtube	algorithm	28.28	1.37	**21**×	8.48	6.15	**1**×	2.54	0.30	**8**×	1.38	0.49	**3**×
	kernel	25.83	0.83	**31**×	6.66	5.54	**1**×	1.71	0.12	**15**×	0.82	0.28	**3**×
Erdos02	algorithm	0.32	0.11	**3**×	1.15	0.13	**9**×	1.09	0.20	**5**×	0.54	0.12	**5**×
	kernel	0.18	0.05	**4**×	0.65	0.06	**11**×	0.53	0.15	**4**×	0.17	0.03	**6**×
mycielskian9	algorithm	0.23	0.06	**4**×	0.81	0.07	**12**×	0.95	0.07	**14**×	0.46	0.10	**5**×
	kernel	0.10	0.01	**7**×	0.44	0.03	**17**×	0.46	0.02	**23**×	0.14	0.02	**7**×
EX3	algorithm	0.13	0.13	**4**×	1.43	0.25	**6**×	0.94	0.07	**12**×	0.50	0.09	**5**×
	kernel	0.34	0.05	**7**×	0.82	0.08	**11**×	0.46	0.04	**13**×	0.15	0.03	**6**×
net25	algorithm	0.18	0.18	**3**×	2.21	0.26	**8**×	1.40	0.19	**7**×	0.87	0.23	**4**×
	kernel	0.38	0.10	**4**×	1.44	0.15	**10**×	0.83	0.13	**6**×	0.44	0.08	**5**×
mycielskian10	algorithm	0.09	0.06	**4**×	0.85	0.08	**10**×	0.99	0.08	**12**×	0.47	0.12	**4**×
	kernel	0.10	0.02	**6**×	0.47	0.04	**11**×	0.49	0.04	**13**×	0.15	0.03	**5**×

max speedup over baseline is 38× that appears at matrix *ins2* with B2SR-32. On Titan V, **bmv_bin_bin_full()** achieves an average speedup of 1.72× in B2SR-4, 1.84× in B2SR-8, 1.92× in B2SR-16, and 2.32× in B2SR-32. The max speedup over baseline is 25× that appears at matrix *vsp_c-30_data_data* with B2SR-32.

For **bmv_bin_full_full()**, the performance is present in Figure 6c and 7c. Unlike **bmv_bin_bin_full()**, the average performance gain decreases when enlarged the tile size. On GTX1080, it achieves an average speedup of 2.06× in B2SR-4, 1.92× in B2SR-8, 1.43× in B2SR-16, and

Table VIII: SpMV-based graph algorithm performance on Titan V (Volta) GPU.

Matrix	Algorithm runtime (ms)	BFS			SSSP			PR			CC		
		GBlst	Ours	Speedup	GBlst	Ours	Speedup	GBlst	Ours	Speedup	GBlst	Ours	Speedup
delaunay_n14	algorithm	5.17	1.75	**3**×	10.86	1.99	**5**×	1.43	0.17	**8**×	3.09	0.67	**5**×
	kernel	3.29	0.41	**8**×	6.38	0.85	**7**×	0.65	0.04	**15**×	1.67	0.14	**12**×
se	algorithm	2.59	0.63	**4**×	5.80	1.24	**5**×	1.45	0.22	**7**×	0.97	0.24	**4**×
	kernel	2.06	0.17	**12**×	3.38	0.64	**5**×	1.45	0.22	**7**×	0.35	0.04	**9**×
debr	algorithm	5.32	1.26	**4**×	20.17	7.86	**3**×	8.36	5.36	**2**×	7.37	1.63	**5**×
	kernel	2.46	0.78	**3**×	9.59	5.33	**2**×	2.21	0.19	**12**×	0.98	0.41	**2**×
ash292	algorithm	1.91	0.03	**64**×	4.08	0.81	**5**×	1.32	0.07	**19**×	0.84	0.23	**4**×
	kernel	1.09	0.01	**152**×	2.36	0.22	**11**×	0.62	0.01	**43**×	0.32	0.03	**10**×
netz4504_dual	algorithm	3.22	0.03	**115**×	7.38	1.37	**5**×	1.40	0.08	**19**×	0.87	0.27	**3**×
	kernel	1.89	0.01	**264**×	4.23	0.36	**12**×	0.57	0.02	**33**×	0.36	0.04	**9**×
minnesota	algorithm	8.60	0.09	**96**×	14.48	3.15	**5**×	1.33	0.09	**15**×	1.20	0.36	**3**×
	kernel	6.34	0.02	**282**×	8.36	0.83	**10**×	0.62	0.02	**32**×	0.48	0.05	**9**×
jagmesh6	algorithm	9.82	0.03	**349**×	8.36	0.83	**10**×	1.33	0.07	**20**×	1.09	0.35	**3**×
	kernel	6.69	0.01	**824**×	9.07	0.95	**10**×	0.58	0.02	**33**×	0.51	0.05	**10**×
uk	algorithm	13.39	0.25	**53**×	20.57	4.61	**4**×	1.31	0.09	**15**×	0.51	0.05	**10**×
	kernel	11.85	0.07	**175**×	11.43	1.29	**9**×	0.63	0.02	**31**×	3.26	0.26	**13**×
whitaker3_dual	algorithm	30.97	0.08	**414**×	40.67	10.11	**4**×	1.33	0.16	**8**×	2.59	0.64	**4**×
	kernel	26.14	0.02	**1344**×	23.82	4.46	**5**×	0.64	0.02	**33**×	1.08	0.12	**10**×
rajat07	algorithm	5.20	0.03	**165**×	11.96	2.37	**5**×	1.34	0.16	**8**×	1.46	0.41	**4**×
	kernel	3.13	0.01	**339**×	6.96	1.01	**7**×	0.62	0.04	**18**×	0.61	0.08	**8**×
3dtube	algorithm	17.65	1.01	**18**×	7.52	5.72	**1**×	2.08	0.34	**6**×	1.21	0.38	**3**×
	kernel	15.13	0.39	**39**×	5.06	4.89	**1**×	1.14	0.08	**15**×	0.61	0.13	**5**×
Erdos02	algorithm	0.43	0.13	**3**×	1.53	0.17	**9**×	1.43	0.26	**6**×	0.62	0.16	**4**×
	kernel	0.26	0.06	**4**×	0.84	0.07	**11**×	0.69	0.18	**4**×	0.22	0.04	**6**×
mycielskian9	algorithm	0.29	0.07	**4**×	1.08	0.11	**10**×	1.29	0.08	**17**×	0.59	0.14	**4**×
	kernel	0.14	0.02	**9**×	0.58	0.04	**16**×	0.60	0.03	**25**×	0.19	0.03	**7**×
EX3	algorithm	0.72	0.21	**3**×	1.68	0.24	**7**×	1.09	0.10	**10**×	0.56	0.13	**4**×
	kernel	0.46	0.07	**7**×	0.97	0.09	**11**×	0.57	0.05	**12**×	0.21	0.03	**8**×
net25	algorithm	0.69	0.23	**3**×	2.27	0.29	**8**×	1.36	0.25	**5**×	0.93	0.32	**3**×
	kernel	0.46	0.13	**4**×	1.29	0.13	**10**×	0.67	0.16	**4**×	0.39	0.11	**4**×
mycielskian10	algorithm	0.31	0.07	**4**×	1.13	0.14	**8**×	1.31	0.09	**14**×	0.56	0.14	**4**×
	kernel	0.14	0.02	**6**×	0.56	0.05	**10**×	0.64	0.04	**15**×	0.19	0.04	**4**×

0.92× in B2SR-32. The most significant speedup happens at *vsp_south31_slptsk* with B2SR-4, yielding 18×. On Titan V, it achieves an average speedup of 1.88× in B2SR-4, 1.71× in B2SR-8, 1.27× in B2SR-16, and 0.81× in B2SR-32. The most significant speedup happens at matrix *ins2* with B2SR-4, with 14× acceleration.

BMM In BMM, we implement **`bmm_bin_bin_sum()`** to support the SpGEMM kernels (see Table III). Figure 6d and 7d presents the performance of the BMM kernel compared to cuSPARSE's **`cusparseScsrgemm()`**. On GTX1080, it achieves an average speedup of 33.96× in B2SR-4, 27.84× in B2SR-8, 17.81× in B2SR-16, and 10.22× in B2SR-32. The maximum speedup is 6555× that happens at matrix *ins2* with B2SR-4. On Titan V, the performance gain is moderate compared to GTX1080. We accounts the reason for cuSPARSE's APIs have better performance gain on Volta than Pascal, while our implementation perform similar or evenly slightly poor on Volta than on Pascal. **`bmm_bin_bin_sum()`** on Titan V achieves an average speedup of 5.34× in B2SR-4, 3.65× in B2SR-8, 9.03× in B2SR-16, and 12.25× in B2SR-32. Interestingly, the most significant speedup is 3243× also happens at matrix *ins2* but with B2SR-32 instead of B2SR-4 as on GTX1080.

E. Graph Algorithms

We compare the implemented B2SR-based Bit-GraphBLAS algorithms with GraphBLAST [4], a state-of-the-art GPU-based GraphBLAS framework. GraphBLAST switches between sparse and dense vector/matrix computation depending on sparsity degree across algorithm interactions with optimized CUDA kernels. For the GraphBLAST configuration, we use the same environment (e.g., CUDA Runtime 9.1) as indicated on the Github repository [49]. The graph algorithms are implemented following the GraphBLAS convention. For iteration-based algorithms, such as BFS, SSSP, PR, and CC, each iteration contains a frontier vector that performs a matrix-vector multiplication with the adjacency matrix and several element-wise scalar operations. They are used to update the frontier vector for indicating neighbor aggregation in each iteration, through the mathematical semi-ring operation. The number of iterations depends on when the algorithm is converged at runtime.

Table IX: SpGEMM-based graph algorithm performance on GTX1080 (Pascal) and Titan V (Volta) GPU.

Matrix	TC runtime (ms) on Pascal			TC runtime (ms) on Volta		
	GBlst	Ours	Speedup	GBlst	Ours	Speedup
delaunay_n14	0.14	0.06	**2**×	0.11	0.13	**1**×
se	0.14	0.03	**4**×	0.11	0.02	**5**×
debr	3.51	0.26	**13**×	1.07	0.09	**12**×
sstmodel	0.51	0.03	**20**×	0.66	0.03	**21**×
jagmesh2	0.29	0.01	**22**×	0.10	0.02	**4**×
lock2232	0.51	0.03	**19**×	0.62	0.03	**22**×
ramage02	12.59	0.53	**24**×	3.96	0.37	**11**×
s4dkt3m2	3.83	0.15	**26**×	1.05	0.06	**18**×
opt1	7.03	0.27	**26**×	2.30	0.21	**11**×
trdheim	3.84	0.08	**49**×	1.29	0.06	**23**×
3dtube	151.89	2.91	**52**×	79.49	2.95	**27**×
mycielskian12	22.47	4.63	**5**×	12.51	4.47	**3**×
Erdos02	7.37	0.31	**23**×	3.62	0.57	**6**×
mycielskian9	0.48	0.08	**6**×	0.35	0.08	**4**×
mycielskian13	93.21	20.37	**5**×	48.87	18.87	**3**×
vsp_c-60_data_cti_cs4	139.41	9.43	**15**×	72.32	5.19	**14**×

Since matrix-vector multiplication is the major performance concern per iteration (>80% of the workload), in Bit-GraphBLAS, our major goal is to improve the efficiency of SpMV through our B2SR based BMV kernel. Table VII and VIII list the algorithm and kernel execution latency in ms for the proposed Bit-GraphBLAS with respect to GraphBLAST. As can be seen, under both conditions, Bit-GraphBLAS achieves considerable speedups through B2SR and the strong bit computation capability of modern GPUs.

For Bit-GraphBLAS, in the 521 binary matrices dataset, the patterns with better performance fall into three main categories: **diagonal**, **block**, and **stripe** (reference the classification in Table V). The performance of matrices from the three categories are shown in Table VII and VIII. In the subset of matrices, *delaunay_n14*, *se*, and *debr* are **stripe** patterns; *Erdos02*, *mycielskian9*, *EX3*, *net25*, and *mycielskian10* are **block** patterns; *ash292*, *netz4504_dual*, *minnesota*, *jagmesh6*, *uk*, *whitaker3_dual*, *rajat07*, *3dtube* are **diagonal** patterns. In the algorithm evaluation, **BFS** relies on the kernel **`bmv_bin_bin_bin_masked()`** with boolean semiring. On Pascal, **diagonal** pattern matrices can achieve up to 433× acceleration in the whole algorithm and 1414× in kernel; On Volta, it achieves up to 349× speedup to GraphBLAST in algorithm and 1344× in kernel. Matrices in **stripe** and **block** generally perform a moderate speedup from 2× to 7× on both GPU architectures. Other three SpMV-based algorithms (**SSSP**, **PR**, and **CC**) are fulfilled by **`bmv_bin_full_full()`** with relaxation and extension (details are described in Section V). They mainly achieve a acceleration over GraphBLAST with no more than 20× algorithm-wise and 40× kernel-wise.

In Table IX, we demonstrate the performance improvement of TC algorithm, which is essentially a one-time execution of the **`bmm_bin_sum_masked()`** kernel. On both Pascal and Volta, we still have **diagonal** patterns with the best performance. It can achieve up to 52× on Pascal and 27× on Volta GPUs.

It is noteworthy that the same matrix can often find lower GraphBLAST runtime on Volta than on Pascal. A similar effect can be found in both kernel and algorithm evaluations. For example, the *3dtube* runs 151.89 (ms) on Pascal but only 79.49 (ms) on Volta. Nevertheless, Bit-GraphBLAS can have a larger or similar runtime on Volta than on Pascal GPUs. We have a 0.04 (ms) increase in runtime for the *3dtube* case. We attribute the effect to that Volta has updated the warp execution model and eliminated implicit warp synchronous. This poses a little performance slowdown of binary intrinsics like **`__shfl_sync()`** and **`__ballot_sync()`** compared to the non-synchronizing **`__shfl()`** and **`__ballot()`** in Pascal GPUs.

VII. Discussion

Limitations of Bit-GraphBLAS: As Bit-GraphBLAS relies on the bit-operation and bit-data, it only directly applies to homogeneous graphs (i.e., the adjacency matrix is a binary matrix). Throughout the SuiteSparse data collection, we have observed that ∼20% of graphs are homogeneous that can directly benefit from Bit-GraphBLAS. These graphs cover a wide range of domains including mathematics, power-grid, physics, electronics, material science, economics, thermal, fluid dynamics, etc. Additionally, as the weights for many heterogeneous graphs can be expressed by integers or fixed-points (e.g., through normalization), similar to the recent effort decomposing a quantized-neural-network into several concurrent binary-neural-networks for acceleration [39], Bit-GraphBLAS can also be extended to support heterogeneous graphs with short bit-width. We set this as future work.

Platform Portability: Although in the evaluation we showcase Bit-GraphBLAS on NVIDIA GPUs due to hardware availability, the bit intrinsics that Bit-GraphBLAS relies on, such as __popc(), __shfl(), __ballot(), and __brev() are also available (despite using different names) in other GPUs and CPUs (like AMD's GPU and X86 CPUs). Therefore, we did not see significant challenges in supporting Bit-GraphBLAS on alternative hardware platforms (e.g., AMD GPUs through HIPIFY [50]).

VIII. Conclusion

We present Bit-GraphBLAS, a linear algebra-based graph framework that utilizes a Bit-Block Compressed Sparse Row (B2SR) format and bit manipulation primitives on GPUs to enable dense bit-operations on bit tiles within large sparse adjacency matrices. We explore different tile size configurations from 4×4 to 32×32, and suitable bit-packing types. We implement BMV and BMM schemes to support parse kernel operations and demonstrate their effectiveness on graph algorithms by reducing memory footprint. The result shows significant performance gain over full-CSR-based GPU graph frameworks. In sum, the novel storage format and algorithms compress the graphs' storage and accelerate the linear algebra kernels SpMV and SpGEMM through finer-grained bit-wise parallelism.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work was partially supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. The evaluation platforms were supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The work was additionally supported by NSF under Grants CNS-1717425, CCF-1703487, CCF-2028850. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

REFERENCES

[1] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," in *Proc. of HPDC'19*, 2014, pp. 239–252.

[2] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Symp. of SIGPLAN'16*, 2016, pp. 1–12.

[3] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan, "Gbtl-cuda: Graph algorithms and primitives for gpus," in *Symp. of IPDPSW'16*. IEEE, 2016, pp. 912–920.

[4] C. Yang, A. Buluç, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *ACM Trans. Math. Softw.*, vol. 48, no. 1, feb 2022. [Online]. Available: https://doi.org/10.1145/3466795

[5] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. of SC'12*. IEEE, 2012, pp. 1–10.

[6] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on gpus," in *Proc. of SC'15*, 2015, pp. 1–12.

[7] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 117–128, 2012.

[8] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel gpu methods for single-source shortest paths," in *IPDPS'14*. IEEE, 2014, pp. 349–359.

[9] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient pagerank and spmv computation on amd gpus," in *ICPP'10*. IEEE, 2010, pp. 81–89.

[10] J. Soman, K. Kishore, and P. Narayanan, "A fast gpu algorithm for graph connectivity," in *Symp. of IPDPSW'10*. IEEE, 2010, pp. 1–8.

[11] M. Bisson and M. Fatica, "High performance exact triangle counting on gpus," *TPDS'19*, vol. 28, no. 12, pp. 3501–3510, 2017.

[12] NVIDIA, "nvgraph," 2021. [Online]. Available: https://developer.nvidia.com/nvgraph

[13] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," vol. 45, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3322125

[14] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," 2015.

[15] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276491

[16] E. Orachev, M. Karpenko, A. Khoroshev, and S. Grigorev, "Spbla: The library of gpgpu-powered sparse boolean linear algebra operations," in *IPDPSW'21*. IEEE, 2021, pp. 272–275.

[17] J. Kepner, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *Proc. of HPEC'16*. IEEE, 2016, pp. 1–9.

[18] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira, "Graphblas c api: Ideas for future versions of the specification," in *HPEC'17*. IEEE, 2017, pp. 1–6.

[19] E.-J. Im and K. A. Yelick, "Optimizing sparse matrix vector multiplication on smp." in *PPSC*. Citeseer, 1999.

[20] NVIDIA, "cusparse," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[21] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for spmv-based applications," in *Symp. of IPDPS'18*. IEEE, 2018, pp. 950–959.

[22] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proc. of SIGPLAN'18*, 2018, pp. 94–108.

[23] W. Zhou, Y. Zhao, X. Shen, and W. Chen, "Enabling runtime spmv format selection through an overhead conscious method," *TPDS'19*, vol. 31, no. 1, pp. 80–93, 2019.

[24] Z. Fu, M. Personick, and B. Thompson, "Mapgraph: A high level api for fast development of high performance graph analytics on gpus," in *GRADES-NDA'18*, 2014, pp. 1–6.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of SIGMOD'10*, 2010, pp. 135–146.

[26] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *CSUR'15*, vol. 48, no. 2, pp. 1–39, 2015.

[27] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. of SOSP'13*, 2013, pp. 456–471.

[28] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[29] NVIDIA, "cugraph," 2021. [Online]. Available: https://github.com/rapidsai/cugraph/blob/main/README.md

[30] C. Yang, A. Buluç, and J. D. Owens, "Implementing push-pull efficiently in graphblas," in *ICPP'18*, 2018.

[31] A. Buluc, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 721–733.

[32] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S.-h. Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes," in *Proc. of SC'13*, 2013, pp. 1–12.

[33] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix–matrix multiplication with gpu tensor cores," *Computers & Electrical Engineering*, vol. 88, p. 106848, 2020.

[34] G. Li and W. Rao, "Compression-aware graph computation," in *Proc. of UbiComp'16*, 2016, pp. 1295–1302.

[35] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler, "Log (graph) a near-optimal high-performance graph representation," in *PACT'18*, 2018, pp. 1–13.

[36] N. R. Brisaboa, S. Ladra, and G. Navarro, "k 2-trees for compact web graph representation," in *SPIRE'09*. Springer, 2009, pp. 18–30.

[37] A. Li, T. Geng, T. Wang, M. Herbordt, S. L. Song, and K. Barker, "Bstc: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets," in *Proc. of SC'19*. ACM, 2019.

[38] A. Li and S. Su, "Accelerating binarized neural networks via bit-tensor-cores in turing gpus," *TPDS'19*, vol. 32, no. 7, pp. 1878–1891, 2020.

[39] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, "Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores," *arXiv preprint arXiv:2106.12169*, 2021.

[40] S. Ghodrati, H. Sharma, C. Young, N. S. Kim, and H. Esmaeilzadeh, "Bit-parallel vector composability for neural acceleration," in *DAC'20*. IEEE, 2020, pp. 1–6.

[41] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA'18*. IEEE, 2018, pp. 764–775.

[42] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ISCA'18*. IEEE, 2018, pp. 383–396.

[43] A. Li, W. Liu, L. Wang, K. Barker, and S. L. Song, "Warp-consolidation: A novel execution model for gpus," in *Proc. of SC'18*, 2018, pp. 53–64.

[44] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[45] Y. Zhang, A. Azad, and A. Buluç, "Parallel algorithms for finding connected components using linear algebra," *JPDC'20*, vol. 144, pp. 14–27, 2020.

[46] Y. Zhang, A. Azad, and Z. Hu, "Fastsv: A distributed-memory connected component algorithm with fast convergence," in *Proc. of PP'20*. SIAM, 2020, pp. 46–57.

[47] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," ser. IPDPSW '15, 2015, p. 804–811.

[48] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *HPEC'17*. IEEE, 2017, pp. 1–7.

[49] C. Yang, "Graphblast," https://github.com/gunrock/graphblast, 2020.

[50] AMD, "Hip porting guide," 2022. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-porting-guide.html