G-Sparse: Compiler-Driven Acceleration for Generalized Sparse Computation for Graph Neural Networks on Modern GPUs

Yue Jin[†]
Ant Group
jinyue.jy@antgroup.com

Chengying Huan
Chinese Academy of Sciences
huanchengying@iscas.ac.cn

Heng Zhang
Chinese Academy of Sciences
zhangheng17@iscas.ac.cn

Yongchao Liu[†]
Ant Group
yongchao.ly@antgroup.com

Shuaiwen Leon Song
University of Sydney
shuaiwen.song@sydney.edu.au

Rui Zhao
Ant Group
oahzior@gmail.com

Yao Zhang[‡]
Microsoft
zhanyao@microsoft.com

Changhua He[‡]
Dipeak Ltd
changhua@gmail.com

Wenguang Chen
Ant Group
yuanben.cwg@antgroup.com

***Abstract*—Graph Neural Network (GNN) learning over non-Euclidean graph data has recently drawn a rapid increase of interest in many domains. Generalized sparse computation is crucial for maximizing the performance of GNN learning, while most recent GNNs primarily focused on optimizing coarse-grained parallelism associated with nodes, edges, and additional feature dimensions. However, efficiently implementing generalized sparse computation is challenging. The performance optimization of generalized sparse computation lacking in-depth architecture-aware design is seldom supported by existing Domain-Specific Languages (DSLs) and is hard to be tuned by experts, which involves substantial trial and error. In this work, we propose `G-Sparse`, a new compiler framework that extends the popular Halide compiler to enable effective acceleration for generalized sparse computations for GNNs through compiler-driven optimizations and auto-tuning. To facilitate generalized sparse computations, `G-Sparse` separates algorithms from schedules and introduces several novel sparse computation optimization techniques for modern GPUs, including two-dimensional shared memory optimizations and efficient cost-driven design space exploration and auto-tuning. Extensive evaluation against highly-optimized state-of-the-art sparse computation kernels and on end-to-end GNN training and inference efficiency has demonstrated that our proposed `G-Sparse` achieves up to a $4.75\times$ speedup over the state-of-the-art sparse kernels, and a training and inference speedup of $1.37\times \sim 2.25\times$ over three popular GNN frameworks including GCN, GraphSAGE, and GAT. The source code of `G-Sparse` is publicly available at https://github.com/TuGraph-family/tugraph-db/tree/master/learn.**

***Index Terms*—domain specific language compiler, GPU, SpMM, SDDMM, graph neural network**

I. Introduction

Graph Neural Network (GNN) is an emerging field in deep learning and have been applied to a wide range of real-world applications, including but not limited to bioinformatics [1], social science [2], recommendation systems [3], and quantum chemistry [4]. Graph-structured representation learning using GNN models is crucial for many real-world graph-related applications, such as node classification [5]–[10], graph isomorphism [11], and diffusion patterns [12]. In contrast to conventional deep learning models like DNNs, which rely heavily on dense matrix operations, GNNs involve message-aware propagation operations that recursively update each vertex's features based on its neighbors. When working with real-world graph datasets, their sparsity and irregularity pose performance and design challenges for recent state-of-the-art frameworks, such as TensorFlow [13], PyTorch [14], and MXNet [15].

Generally, the computational patterns of GNNs can be described using message-passing primitives. For instance, DGL [16] introduces generalized sparse computations, specifically generalized Sampled Dense-Dense Matrix Multiplication (g-SDDMM) and generalized Sparse Matrix-Matrix Multiplication (g-SpMM), to express these message-passing primitives. As illustrated in Figure 1 as an example, the aggregation of features for target vertices from neighboring vertices can be represented as an *SpMM* operation when performing an aggregating sum operation. Similarly, the aggregation of features for target edges can be represented as an *SDDMM* operation between the source vertex features and the target vertex features. These operations are considered generalized because the aggregation type in GNNs can include sum, max, and so on. However, these operations lead to a large overhead of GNN training. The SpMM- and SDDMM-like operations account for more than 60% of the total execution time [17], [18], and become the bottleneck of improving GNN learning speed, motivating us to accelerate them.

Challenges: There exist three challenges to optimizing the SpMM- and SDDMM-like operations, including the limitation of current sparse matrix computation optimizations, computational libraries, and Domain-Specific Languages.

Firstly, compared with conventional dense matrix calculations that employ easy-to-use tensor-driven parallelism, sparse matrix operations are more challenging due to their irregular workload, sparse memory access, and limited data reuse. As

[†]Corresponding authors; [‡]The authors contributed to this work when they worked in Ant Group.

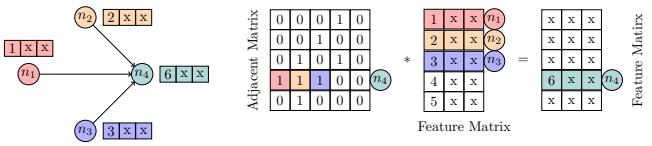


Fig. 1: A g-SpMM operation in GNNs with an aggregate sum. On the left, we aggregate feature vectors of vertex n_1, n_2 and n_3 to the vertex n_4. On the right, we represent the graph as an adjacent matrix so that the features of vertices are computed as an SpMM.

a result, accelerating sparse matrix computation on modern accelerators is essential and challenging for efficient GNN training and inference. To improve the performance of sparse computations in GNN, DGL [16] and FeatGraph [19] attempt to leverage node-centric and edge-centric parallelism. These works bridge GPU-accelerated graph processing and GNN learning by transforming graph data into specific structures and performing scattering and gathering operations along edges and vertices. However, these efforts largely ignore the systematic optimization opportunities presented by the architectural features of the underlying hardware, such as load balancing, shared memory tiling, and register tiling, which have been proven significant for GNN sparse matrix performance. Moreover, their performance is difficult to tune without providing in-depth systematic task scheduling approaches and programming interfaces.

Secondly, although cuSPARSE [20] and Sputnik [21] have considered load balancing and other sparse-specific optimizations, they are mainly targeted for scientific computations and conventional Deep Neural Networks (DNNs) such as Transformers [22] and ResNet-50 [23], and provide very limited support for the operators in GNNs. For instance, in g-SpMM, they only support the case of aggregated summation for source node, and cannot support the case of the edge with more than one feature as input or max as an aggregation function. In particular, even though this operation in GNNs does not require the value array in the CSR sparse matrix, filling the CSR value array with the full assignment value of 1.0 in these existing computational libraries is still compulsory. These redundant computations and memory-accessing operations can bring significant performance degradation to the GNN training procedure.

Finally, existing Domain-Specific Languages (DSLs) such as TVM [24], Halide [25], and Ansor [26] separate algorithms from the schedule and utilize autotuning strategies for efficient automatic code generation, but are not efficient for GNNs. These DSLs mainly target the image processing and natural language processing domain, which commonly utilize interval analysis [27] and provide support for processing regular dense matrices. However, the optimization for sparse computations in the GNN DSL domain is still very limited. One of the reasons is that the DSLs cannot deal with the non-rectangular buffer boundary inference problem in sparse computations. Moreover, they lack sophisticated system optimizations and corresponding autotuning strategies for GNNs and their sparse computations, for example, not supporting sparse-specific optimizations such as row load balancing, adaptive warp shuffle, and hierarchical memory tiling optimizations.

To address these technical challenges, we propose `G-Sparse`, a compiler-driven approach for enabling highly effective acceleration of generalized sparse computations in GNNs running on modern GPUs. `G-Sparse` is a new DSL that extends Halide, and it is capable of efficiently and automatically generating highly optimized code for general sparse kernels. The performance of kernels generated within seconds can surpass the code manually optimized by experts over several weeks. In summary, we make the following contributions:

- We present 2-D shared memory tiling, 1-D register tiling, and row load balancing optimizations for g-SpMM operations in GNNs.
- We propose the adaptive warp shuffle optimization technique for g-SDDMM operations in GNNs.
- We integrate the aforementioned optimization strategies into Halide, a DSL compiler for image-processing tasks.
- We develop a novel DNN-based cost model to predict performance and combine it with two auto-tuning methods (i.e., genetic search and random sampling) for automatically generating highly-optimized sparse kernels tailored for various GNN models.

II. Background

A. Sparse Matrix Computations in GNNs

GNNs represent an emerging field in deep learning that differs from traditional machine learning techniques. While conventional machine learning primarily focuses on dense matrix computations for image and video analysis, GNNs predominantly operate on irregular and sparse graph datasets, necessitating not only dense matrix computations but also large-scale sparse matrix computations.

The computational patterns of GNNs can be characterized using message-passing primitives. According to their operations over edges and vertices, we categorize them as g-SDDMM and g-SpMM, the same as DGL [16], which are widely used for GNN learning. Specifically, g-SpMM calculates the target node representation by aggregating the messages from node and neighbor features, and its inbound edge features. In contrast, g-SDDMM computes the edge representation by gathering its original features, and the features of its incident nodes.

Given a graph $G(V, E)$ with a node set V and an edge set E, let X_u represent the feature (or embedding) of node $u \in V$, and $Y_{e_{u,v}}$ represent the feature (or embedding) of edge $e_{u,v} \in E$, where u is the source node and v is the destination node.

Definition 1 (g-SDDMM). We define the output edge embedding $Z_{e_{u,v}}$ of generalized sampled dense-dense matrix multiplication as

$$Z_{e_{u,v}} = lhs \oplus rhs \tag{1}$$

where $\oplus$ is a binary operator that could be *add*, *sub*, *mul*, *div*, or *dot*, etc. And inputs lhs and rhs can be any of X_u, X_v and $Y_{e_{u,v}}$.

Definition 2 (g-SpMM). The generalized sparse matrix-matrix multiplication computes the embedding of the output node Z_v as

$$Z_v = \Phi_{u \in N(v)}(X_u \oplus Y_{e_{u,v}}) \tag{2}$$

where $N(v)$ is the set of inbound neighbors of v, Φ is a reduce operator over $N(v)$, and $\oplus$ is also a binary operator but differs slightly from that in Eq. (1). The operation of Φ can be sum, mean, max or min.

B. GPU Architecture

During GNN training and inference, the role of GPUs is as accelerators to complement available CPUs by offloading their portions of the workload. These accelerators follow a single instruction multiple data (SIMD) model, and represent a class of hardware enabling massive thread-level parallelism. Using NVIDIA generations as an example, a GPU consists of a number of *streaming multiprocessors* (SMs), each of which includes a total of thousands of Compute Unified Device Architecture (CUDA) cores. For ease of programming, these thread lanes are organized in the form of *warp*, *block*, and *grid*. Commonly, a warp of 32 threads executes the same instruction in parallel on consecutive data.

a) Shared and Global Memory in GPUs: The internal bandwidth of GPU cores accessing global memory is high-speed, e.g., up to 720GB/s for NVIDIA Tesla P100. The shared memory resides on the GPU chip, and each block corresponds to a shared memory unit. Shared memory units are much faster (approximately 1.7TB/s) to access than global memory, and they can be explicitly managed for on-chip data caching and maintaining data locality. This results in fewer global memory transactions and overall performance improvement.

b) Coalesced Access to Global Memory: The efficiency of global memory access can be measured by calculating the ratio of the transactions actual used to the transactions issued. On CUDA-capable GPU architectures, the concurrent global memory accesses of threads within a warp coalesce into multiple transactions equivalent to the number of 32-byte transactions. For example, in the reducing operations for g-SpMM and the dot reducing operations for g-SDDMM with node or edge parallelism, the threads of a warp only load one element (4 bytes as a floating-point element) once for accessing the nonzero elements in each row of the sparse matrix. However, this results in 28 bytes being wasted in a single 32-byte transaction, leading to a global memory load efficiency of only 1/8.

C. Tensor Compiler

To accelerate the training and inference of deep learning models, compiler-based DSLs are widely exploited to efficiently enhance the computational processes of algorithms and offer a useful abstraction of scheduling optimizations such as tiling, vectorization, and thread binding. As a result, optimization code can be rapidly implemented and verified. Frameworks like TVM [24], Halide [25], and other similar compiler frameworks [26], [28]–[31] have demonstrated strong performance in accelerating DNNs and image analysis applications. FeatGraph [19] is the only framework to employ TVM IR to implement generalized sparse computations in GNNs, achieving performance comparable to that of cuSPARSE. However, these works only study coarse-grained parallelism at the level of node, edge, and feature dimensions, which significantly limits the attainable performance of GNNs.

In contrast, our `G-Sparse` compiler integrates sparse-specific optimizations like 2-D shared memory tiling and introduces a novel NN-based cost model, enabling auto-tuning for GNN performance and achieving much better results than the other tensor compilers mentioned above.

TABLE I: List of notations used in the paper

Notation	Description
$\mathbf{A}$	The sparse matrix.
$\mathbf{A}.rowPtr$	The row entry array in $\mathbf{A}$.
$\mathbf{A}.colIdx$	The column indices array in $\mathbf{A}$.
$\mathbf{A}.values$	The value array in $\mathbf{A}$.
$balancedIdx$	The row index array for $\mathbf{A}.rowPtr$.
m	The number of rows in $\mathbf{A}$.
n	The number of columns in $\mathbf{A}$.
nnz	The number of nonzero elements in $\mathbf{A}$.
$nnzPerRow$	Non-zero elements for each row in $\mathbf{A}$.
$\mathbf{U}$	The input dense matrix.
$\mathbf{O}$	The output dense matrix.
$kMTile$	The tiling size in the row dimension.
$kNTile$	The tiling size in the column dimension.
$kNnzTile$	the tiling size in each $nnzPerRow$.

TABLE II: Ablation study. The 2-D shared memory tiling and row balancing optimizations show the greatest impact on performance for g-SpMM. Whether to use warp shuffles is critical for g-SDDMM.

g-SpMM			
Feature Length	8	64	512
2-D Shared Memory Tiling	63.17%	67.54%	60.14%
Row Load Balancing	53.90%	58.01%	57.03%
1-D Stride Register Tiling	100.71%	82.80%	89.61%
g-SDDMM			
Feature Length	8	64	512
Adaptive Warp Shuffles	74.16%	44.52%	20.53%

III. System Design

A. Overview

Figure 2 illustrates the workflow pipeline of the `G-Sparse` DSL compiler. In contrast to conventional GNN system optimization, we leverage compiler techniques to enhance the performance of g-SpMM and g-SDMM operations and enable autotuning. This includes optimizing GPU memory access, workload balancing, and SIMD-aware execution. In the following sections, we discuss g-SpMM and g-SDDMM optimizations and their corresponding schedules using our extended `G-Sparse` DSL.

For g-SpMM, we introduce the buffer-bound inference technique to enable 2-D shared memory optimization, the buffer binding index expression to facilitate row load balancing optimization, and the 1-D stride register tiling to optimize data reuse at the register level. As demonstrated in Table II, these

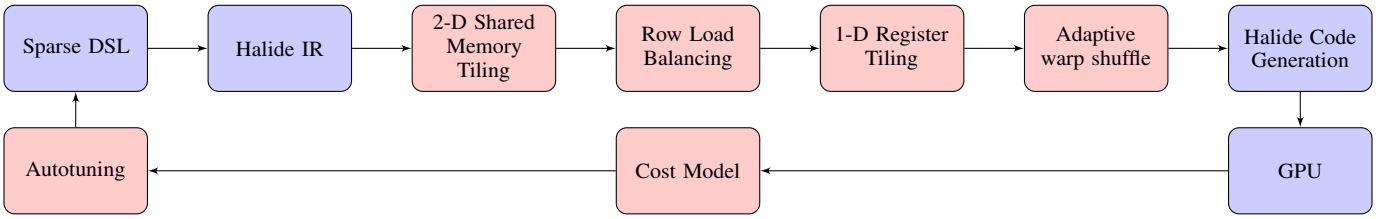


Fig. 2: `G-Sparse` compilation pipeline. Section III-B describes 2-D shared memory tiling, row land balancing, and 1-D register tiling. Section III-C describes adaptive warp shuffle. Section III-D describes the NN-based cost model and autotuning.

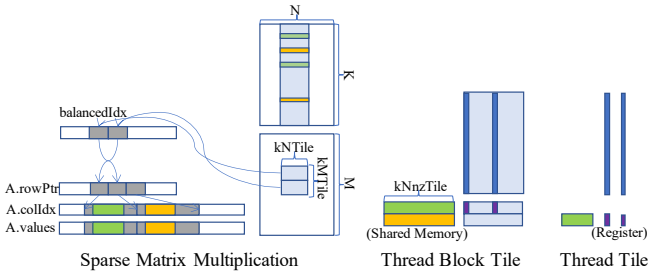


Fig. 3: `G-Sparse` optimizations of the g-SpMM operation.

optimizations, which have not been employed by previous DSLs, are crucial for enhancing the performance of sparse computations. We adopt DGL's g-SpMM interface, which employs the CSR format for the adjacent sparse matrix in g-SpMM.

For g-SDDMM, we apply the warp shuffle optimization [32] and our autotuning strategy with a more extensive search space to achieve better performance. Warp shuffle functions utilize registers, rather than shared or global memory, for thread communication within a warp and have been used in sparse linear algebra acceleration [18], [33]. We adopt DGL's g-SDDMM interface, which employs the COO format for the sparse matrix in g-SDDMM.

Lastly, we introduce our autotuning approach that incorporates the genetic search algorithm with an extensive search space. This enables the automatic search for optimal results without human intervention. Table I presents the notations used in our system implementations.

B. G-SpMM Optimizations

We implement g-SpMM algorithms in Halide DSL and introduce 2-D shared memory tiling, row load balancing, and 1-D register tiling optimizations. As an illustration, Figure 4 shows the SpMM DSL and Figure 5 shows the CUDA code generated by the DSL.

a) 2-D Shared Memory Tiling with Buffer Bound Inference: The `G-Sparse` compiler aims to optimize memory hierarchy tiling for g-SpMM computations. As shown in Figure 3, we reuse the column index array and values array of the sparse matrix in the GPU shared memory to prevent accessing global memory every time when computing the same row of the output matrix. Additionally, preloading the sparse matrix with the GPU coalesced threads into the shared memory allows us to avoid wasting global memory transactions if we

```
// SpMM DSL Description.
Buffer<int> rowPtr, colIdx, values, input;
Func spmm("spmm"), out("out");
Var vm, vn;
Expr nnzPerRow = rowPtr(vm + 1) - rowPtr(vm);
RDom k(0, nnzPerRow);
Expr col = colIdx(rowPtr(vm) + k);
spmm(vn, vm) = 0.0f;
spmm(vn, vm) += input(vn, col) * values(rowPtr(vm) + k);
out(vn, vm) = spmm(vn, vm);
// Schedules.
Buffer<int> balancedIdx;
vm.bind_buffer(balancedIdx);
Var mi, mo, nii, ni, no;
RVar ko, ki;
out.split(vn, no, ni, kRegisterTileSize *
  ↪  kNTileSize).split(ni, nii, ni, kNTileSize);
out.split(vm, mo, mi, kMTileSize);
spmm.compute_at(out, mo);
spmm.update().reorder(mi, k);
spmm.update().split(k, ko, ki, kNnzTileSize)
out.gpu_blocks(mo, no).gpu_threads(ni).gpu_threads(mi);
colIdx.compute_at(spmm,
  ↪  ko).store_in(MemoryType::GPUShared).gpu_threads(_0);
values.compute_at(spmm,
  ↪  ko).store_in(MemoryType::GPUShared).gpu_threads(_0);
```

Fig. 4: The SpMM DSL description.

were to serially read the sparse matrix values in one row, as mentioned in Section II-B0b. This improves performance.

The `G-Sparse` compiler implements this optimization by extending the Halide DSL. Halide employs interval analysis to infer the loop extent and allocation size, recursively inferring from the output loop extent and allocation size back to each function. However, Halide can only analyze the bounds of a variable and a parameter, not buffer access. When Halide encounters a situation it cannot infer, given the row index is i, it pessimistically assumes that the bound is infinite, which is

$$bounds(A.rowPtr(i)) = [-inf, +inf]$$

. As a result, the bound analysis of the non-rectangular buffer `A.colIdx` cannot proceed, and in the end, Halide can only perform the most pessimistic analysis and then load the entire `A.colIdx` before computing the `kNnzTile`. Although this approach can make the global memory load of `A.colIdx` efficient (Section II-B0b), it will cause repeated calculations and loading, and the entire nnz elements are often too large for the GPU's shared memory. For instance, REDDIT has approximately 114 million non-zero elements, while the shared memory size of the NVIDIA GPU V100 is only 96K.

To address this issue, we introduce non-rectangular buffer bound inference to determine the bound and allocation size

```
// g-SpMM pseudo code with the sum aggregation of vertex feature matrix U.
// m equals to the node size, n equals to feature length 32.
for (vm, 0, m) {
 nnz = rowPtr(vm+1) - rowPtr(vm);
 for (vn, 0, n) {
  float sum = 0.0f
  for (k, 0, nnz) {
   col = colIdx(rowPtr(vm) + k);
   sum += U(vn, col);
  }
  out(vn, vm) = sum;
 }
}

// shared memory tiling, register tiling and row balancing.
gpu_block<CUDA> (m.block_id_x, 0, m/32) {
 allocate shmTile[int32*4096] in GPUShared
 gpu_thread<CUDA> (.thread_id_y, 0, 32) { // kMTile equals to 32
  gpu_thread<CUDA> {.thread_id_x, 0, 16} { // kNTile equals to 16
   allocate regTile[float32*2] // kRegisterTileSize equals to 2
   produce sum {
    // 1. row index vm is replaced by balancedIdx[vm] to do row balancing.
    let rstart = rowPtr[balancedIdx[m.block_id_x*32 + .thread_id_y]]
    let nnz = rowPtr[balancedIdx[m.block_id_x*32 + .thread_id_y + 1]]
     - rstart
    regTile[0] = 0.0f
    regTile[1] = 0.0f
    for (ro, 0, (nnz + 127)/128) {
     // 2. shared memory tiling.
     // kNnzTile euqals to 8*16(8*kNTile)
     // the shared memory size equals to kNnzTile*kMTile
     for (rio, 0, 8) {
      if (rio*16 + .thread_id_x < min(nnz - ro*128, 128)) {
       shmTile[.thread_id_y*128 + rio*16 + .thread_id_x] =
        colIdx[ro*128 + rio*16 + .thread_id_x + rstart]
      }
     }
     gpu_thread_barrier(0)
     for (k.ri, 0, min(nnz - ro*128, 128)) {
      // 3. register 1-D stride tiling.
      regTile[0] += U[shmTile[.thread_id_y*128 + k.ri]*U.stride.1
       + .thread_id_x]
      regTile[1] += U[shmTile[.thread_id_y*128 + k.ri]*U.stride.1
       + .thread_id_x + 16]
     }
    }
   }
   consume sum {
    out[balancedIdx[m.block_id_x*32 + .thread_id_y]*out.stride.1
     + .thread_id_x] = regTile[0]
    out[balancedIdx[m.block_id_x*32 + .thread_id_y]*out.stride.1
     + .thread_id_x + 16] = regTile[1]
   }
  }
 }
}
```

Fig. 5: The generated SpMM pseudo-CUDA code.

for `A.colIdx`. Since the access to `A.colIdx` is an integer scalar value, we assert that for every access of the same index, the bound is a single point, which is a scalar integer value `A.rowPtr(i)`. Concerning `A.colIdx`, the loop bounds and allocation size are determined by the expression $A.rowPtr(i)+reduceDomain$, where $reduceDomain$ represents a reduction dimension that iterates from 0 to the number of non-zero elements minus one. At this point, for every row, `A.colIdx` has the bounds:

$$[min(A.rowPtr(i) + reduceDomain),\\ max(A.rowPtr(i) + reduceDomain)]$$

Algorithm 1 2-D shared memory optimization algorithm.

```
1: start ← A.rowPtr(i)
2: reduceDomain ← [0, A.rowPtr(i + 1) - A.rowPtr(i))]
3: Split reduceDomain to ro and ri with factor kNnzTile
4: mextent ← kMTile, kextent ← kNnzTile
5: Allocate 2-D shared memory colIdxTile[mextent, kextent]
6: for ro ← 0 to roextent do
7:     Binding mi and ri to GPU threads while loading A.colIdx
8:     for mi ← 0 to mextent in parallel do
9:         for ri ← 0 to kextent in parallel do
10:            check if we reach the bounds of A.colIdx
11:            colIdxTile(mi,ri) ← A.colIdx(A.rowPtr(mi) + reduceDomain)
12:    for mi ← 0 to mextent in parallel do
13:        for ri ← 0 to kextent do
14:            check if we reach the bounds of A.colIdx
15:            O(i,j) ← sum(U(i,colIdxTile(start+reduceDomain)))
```

Recall that `A.rowPtr(i)` is a scalar integer value and has a single point interval for each row. Therefore, the bounds of `A.colIdx` are calculated as follows:

$$[A.rowPtr(i) + min(reduceDomain),\\ A.rowPtr(i) + max(reduceDomain)]$$

Line 11 in Algorithm 1 shows that we preload `A.colIdx` for `kMTile` rows and with `kNnzTile` indices of non-zero elements for each row, the inner reduction dimension iterating from 0 to `kNnzTile` - 1. For each row, the preload size is only related to the size of `reduceDomain` domain. We get the loop extent and allocation size is $[min(ri), max(ri)]$. Therefore, the loop extent for `A.colIdx` each row is equal to `ri`'s loop extent: $min(nnzPerRow - ro \times kNnzTile, kNnzTile)$, where `nnzPerRow` means the number of non-zero elements per row and ro means the outer reduction dimension split from `nnzPerRow` with a factor `kNnzTile`.

We find that this loop extent has a constant upper bound `kNnzTile`. Therefore, we can optimistically allocate `kNnzTile` shared memory size for `A.colIdx` each row in Algorithm 1 (line 5). Instead of pessimistically putting all of `A.colIdx` into shared memory as Halide will do, we only put chunks of `kNnzTile` size into shared memory while still maintaining the check of the dynamic memory bounds in the computation.

b) Row Load Balancing with Buffer Binding Index: Sputnik [21] observes that the NVIDIA Volta thread block scheduler employs a round-robin strategy. Therefore, Sputnik uses row swizzle load balancing for sparse matrix multiplication to balance the workload. It first sorts the row indices based on the number of non-zero items in each row and stores the sorted indices in an array called `balancedIdx`. As a result, the first GPU thread block will compute on `balancedIdx[0]`, which points to the longest row, and the second GPU block will compute the second-longest row.

To incorporate this row load balancing strategy into our DSL, we introduce the *bind_buffer* (line 5 in Figure 4) schedule primitive and a corresponding lowering pass to support indexing with a buffer. We use the *BindBuffer* function (lines 1–3 in Algorithm 2) to bind each row index access expression `exprM`

Algorithm 2 Lowering pass for buffer binding index.

```
1: function BINDBUFFER(balancedIdx)
2:     for all row index access expression exprM do
3:         exprM.bindedBuffer ← balancedIdx
4: function BUFFERBINDINGINDEXLOWERING(halideExprs)
5:     for all i ∈ [0, halideExprs.size()) do
6:         if halideExprs[i].bindedBuffer is not NULL then
7:             originalIndexExpr ← halideExprs[i]
8:             balancedIdx ← originalIndexExpr.bindedBuffer
9:             newIndexExpr ← balancedIdx(originalIndexExpr)
10:            halideExprs[i] ← newIndexExpr
11:    return halideExprs
```

with the `balancedIdx` buffer, which stores the row indices sorted by `nnzPerRow`. Lines 4–11 in Algorithm 2 shows the lowering pass. Internally, the computation iterates through the expressions to find every expression associated with a bound buffer, and replaces it with the row indices expression, and then invokes the loading expression of `balancedIdx` buffer with indices. As a result in the CUDA code shown in Figure 5 (lines 23 – 24), when the `rowPtr` is requested with an indexed GPU thread block `vm`, its index `vm` needs to be replaced with `balancedIdx[vm]`.

This way, when we assign GPU blocks to each row of the output matrix, the $ith \in \{0, m\}$ GPU block is responsible for computing the `balancedIdx[i]`th row of the sparse matrix, instead of the `i`th row. As Figure 3 illustrates, the `A.rowPtr` is indirectly accessed from the `balancedIdx` buffer.

c) 1-D Stride Register Tiling: Besides using shared memory preload, the other way is to reuse the value of the sparse matrix in the register when computing the result, i.e., register tiling, which is 1-dimension here.

As shown in Figure 3, we compute the output in registers using 1-D tiling. During the register tiling computations, we load the `A.colIdx` into the register from the shared memory. Since the row dimension remains constant, the register value of `A.colIdx` is reused in the tiling computations for different values of the input matrix `U`, thereby reducing the memory access of `A.colIdx`.

Additionally, rather than performing tiling in the continuous memory of the output matrix, we do tiling across a specific length of elements. This approach is necessary to ensure that the global memory access of the dense output matrix is contiguous and coalescing when accessed using GPU threads. If we were to tile adjacent elements, the global memory access with threads would be in stride but not coalescing, resulting in wasted global memory access transactions.

C. G-SDDMM Optimizations

a) Adaptive Warp Shuffles: A g-SDDMM operation is depicted in Figure 6. `G-Sparse`'s g-SDDMM kernels follow the edge parallel paradigm and tune the size of edge parallelism and the lane width inside the warp shuffle. We assign a specific number of edges to a block, use adaptive lanes for computing features to intermediate results, and apply warp shuffle instructions to reduce intermediate results in registers. Algorithm 3 provides simplified pseudo kernel code for computing g-SDDMM within a warp. Within $adaptiveLaneWidth$ number of lanes in a warp, line 5 computes a dot product between vertex feature matrix U and another vertex feature matrix V, storing the result into the register $regInThread$. Lines 6 – 7 accumulate the results in $regInThread$ within the lanes to yield the final result $regResult$. The g-SDDMM is implemented in our DSL, as demonstrated in Figure 7. Figure 8 also presents the specific CUDA code generated by our `G-Sparse` compiler.

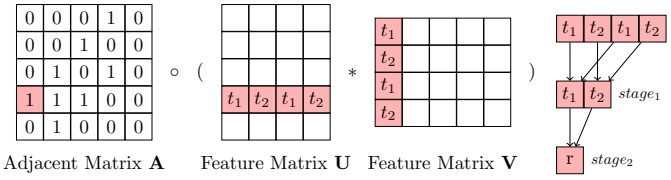


Fig. 6: A g-SDDMM operation. In $stage_1$, threads t_1 and t_2 separately compute the partial dot product result of matrix U and V. $Stage_2$ uses warp shuffle to sum up the partial results in t_1 and t_2 to get the final result r.

Algorithm 3 Adaptive warp shuffle.

```
1: function ADAPTIVEWARPSHUFFLE
2:     regResult ← 0.0f
3:     regInThread ← 0.0f          ▷ Calculate the partial results to
   regInThread in parallel in each thread.
4:     for all ro ← 0 to extent do
5:         regInThread ← sum(load(U) * load(V))
       ▷ Use warp shuffle to sum up the regInThread in each thread.
6:     for all ri ← 0 to adaptiveLaneWidth do
7:         regResult ← sum(warp_shuffle(regInThread, ri))
8:     store(regResult, O)
```

```
1  // SDDMM DSL Description.
2  Func sddmm("sddmm");
3  Var vnnz, vo, vi, vfactor, vlane;
4  Expr row = cooRowInd(vnnz);
5  Expr col = cooColInd(vnnz);
6  RDom rDomain(0, featureLength);
7  sddmm(vnnz) = sum(u(rDomain, row) * v(rDomain, col));
8  // Schedules.
9  sddmm.split(vnnz, vo, vi, kMTile).split(vi, vi, vlane,
   ↪  1).gpu_blocks(vo).gpu_threads(vi, vlane);
10 RVar ro, ri;
11 Func intm = sddmm.update().split(r, ri, ro,
   ↪  kLaneWidth).reorder(ri, ro).rfactor(ro, vfactor);
12 intm.compute_at(sddmm, vi).update().gpu_lanes(vfactor);
```

Fig. 7: The SDDMM DSL and schedules.

Although DGL and FeatGraph utilize techniques like warp shuffles, we find that DGL does not apply autotuning, and FeatGraph applies tuning only with the size of GPU blocks. Moreover, they do not apply warp shuffles for feature lengths less than the warp size of 32, thus not achieving optimal performance in some cases. Additionally, the lane width of warp shuffles for different feature lengths is fixed at 32 for both DGL and FeatGraph. However, as Table IV in Section IV-D demonstrates, 32 is not always the optimal lane width. We use an adaptive shuffling lane width for warp shuffle, ranging from 2 to 32 (powers of two), to achieve the best performance.

```
__global__ void warp_shuffle_sddmm(u, v, out) {
  float result = 0.0f;
  float resultIntm = 0.0f;
  // Calculate the partial results in parallel.
  for (int ro = 0; ro < outerLimit; ro++) {
   resultIntm +=
    u[threadIdx.x+ro*adaptiveLaneWidth+rowidx*featLen]
        *
    v[threadIdx.x+ro*adaptiveLaneWidth+colidx*featLen]
  }
  // Reduce to the final result.
  for (int r = 0; r < adaptiveLaneWidth; r++) {
   result += __shfl_down_sync(fullmask,resultIntm,r)
  }
  out[blockIdx.x*blockDim.x+threadIdx.y] = result;
}
```

Fig. 8: The generated SDMM pseudo-CUDA code. While warp shuffle is used by previous works in which the lane width is fixed at 32, `G-Sparse` introduces adaptive warp shuffle in which the lane width is adaptive for different feature lengths.

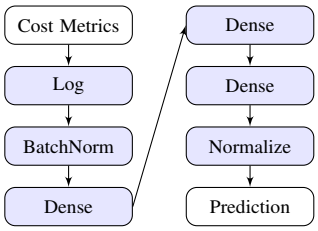


Fig. 9: G-Sparse cost model DNN architecture.

D. Autotuning with Cost Model

a) Cost Model: To make performance tuning easy in the search space, task schedules over the GPU threads can be abstracted using the DSL compiler. The following hyper-parameters constitute the search space:

- `kMTile` and `kNTile` represent the thread block tiling sizes for row and column dimensions of the output matrix, respectively.
- `kNnzTile` represents the shared memory tiling size in each nnzPerRow.
- `kNRegTile` denotes the size of 1-D stride register tiling.
- `kLaneWidth` represents the lane width of warp shuffles.
- Whether to use row load balancing, 2-D shared-memory optimizations for `A.colIdx` or `A.values`, and warp shuffles.

Nevertheless, the search space is so large that recent GNN models are hard to tune. Taking executing g-SpMM operations over `REDDIT` dataset as an example. When the length of features equals 256, the size of the search space could be estimated as 1.5 billion sizes. Conducting a brute-force search will take weeks or even months with every candidate's real run time on specific architectures.

To address this challenge, we propose an autotuning method based on a cost model-driven technique. Inspired by [26], [34], [35], our cost model improves their performance on sparse matrix operations with a neural network structure and training target. Most of the previous works are designed for dense matrix computations. In contrast, our cost model is designed specifically for sparse matrix computations. We ultilize metrics calculated in Algorithm 4 for g-Spmm and g-SDDMM respectively. Additionally, we also add the shape information of the input and output matrix to the metrics for training. While previous works aim to predict the schedules directly, we use the cost model to predict the cost of each schedule in the search space and then fine-tune it with genetic search or random sampling. Finally, we choose the best configuration from the fine-tuned options.

Algorithm 4 G-SpMM and G-SDDMM Cost Metrics

```
function GSpMMCostMetrics(nnz, numNodes, featLen, kMTile, kNTile, kNRegTile, kNnzTile)
    outMemAccessEff ← min(kMTile / 8.0, 1.0)
    nnzRatio ← nnz / ((2*numNodes*featLen)+numNodes+nnz+1)
    accessEff ← outMemAccessEff * nnzRatio
    parDegree ← 1.0 / kNRegTile
    dataReuse ← kNRegTile
    threadBlockSizePenalty ← 128.0 / (kMTile*kNTile)
    if kMTile*kNTile <= 128 then
        threadBlockSizePenalty ← 1.0
    sharedMemSize ← kNnzTile * kMTile
    outMemAccessEffThreadXFactor ← 2.0
    if kMTile >= 8 then
        outMemAccessEffThreadXFactor ← 1.0
    costMetrics ← (outMemAccessEff, parDegree, dataReuse, threadBlockSizePenalty, sharedMemSize, outMemAccessEffThreadXFactor)
    return costMetrics
function GSDDMMCostMetrics(featLen, kMTile, kLaneWidth)
    memAccessEffFactor ← min(kMTile / 8.0, 1.0)
    memAccessEffFactor2 ← min(kLaneWidth / 8.0, 1.0)
    totalNumThreads ← kMTile*kLaneWidth
    warpShuffleFactor ← 1.0 * kLaneWidth / featLen
    featLenFactor ← 2.0
    if featLenFactor < 32 then
        featLenFactor ← 1.0
    costMetrics ← (memAccessEffFactor, memAccessEffFactor2, totalNumThreads, warpShuffleFactor, featLenFactor)
    return costMetrics
```

Our cost model employs a simple and trainable DNN. As illustrated in Figure 9, the first layer of our model structure is a log function, followed by a second layer consisting of batch normalization [36] and three dense layers. The final layer is the normalization layer. The loss is calculated using mean squared error, and we normalize the real data to better reflect the true trend: $loss = MSE(predict_y, normalize(y))$. $Predict_y$ represents the predicted cost, while $normalize(y)$ refers to the normalization of the actual cost, specifically representing the run time observed on the hardware. Ultimately, we utilize the Pearson correlation coefficient to measure the accuracy of our training and prediction.

b) Genetic Search and Random Sampling: After the cost model predicts the cost of the schedules in the search space, we choose 5 to 20 schedules that exhibit the lowest cost values as predicted by the cost model to fine-tune the performance with genetic search and random sampling algorithms.

We customize a genetic search algorithm to search for the optimal schedule, similar to the approach in [25]. The

population size is set at 16, the elite size is 6, and the search concludes after three generations of breeding. We employ a roulette wheel algorithm for candidate selection during the crossover. The mutation probability is set at 20% to escape local optima.

Furthermore, crossover and mutation procedures can generate invalid configurations (e.g., the number of threads exceeding 1024). In such cases, we choose the most similar yet valid candidate from the remaining candidates, where similarity is measured by Euclidean distance. A candidate is returned once it has only one gene differing from the invalid candidate. Consequently, the time complexity of the similarity calculation is linear to the generation size, and the best time complexity is constant. Lastly, a multi-threaded compilation of the schedules is used to accelerate the process.

Fitness: The fitness is designed as the reciprocal of the execution time on the GPU hardware:

$$fitness = 1/execution\_time(candidate) \tag{3}$$

The candidate is executed 10 times, and we take the average time results of these runs. Before that, we execute once to warm up.

Random sampling: We employ random sampling to find the optimal performance when requiring a fast search time. After trimming the search space, we randomly sample the remaining candidates. Three candidates are selected and executed. Ultimately, the best-performing candidate is chosen as the final schedule.

IV. Evaluations

In Section IV-A, we evaluate the performance of g-SpMM and g-SDDMM. In Section IV-B, the ablation study shows the importance of our optimizations. In Section IV-C, the productivity of our DSL compiler is compared to the human expert. In Section IV-D, we evaluate the autotuning with the cost model. Finally, in Section IV-E we evaluate the GNN model end-to-end training and inference performance.

Evaluation Environment: For all experiments, we maintain a consistent setup. We utilize an NVIDIA V100 GPU with 16GB of device memory and employ CUDA 11.1. Sparse tensors feature 32-bit indices, while dense tensors use 32-bit floating-point values in row-major format.

Benchmarks: All g-SpMM kernels and g-SDDMM kernels have been tested using the REDDIT dataset from Hamilton et al. [6], as well as the OGBN-PROTEINS and OGBN-PRODUCTS datasets from the Open Graph Benchmarks [37]. In the REDDIT dataset, nodes represent forum posts, while edges indicate that users have commented on both connected posts. The OGBN-PROTEINS dataset features nodes as proteins and edges as biologically meaningful associations between them. In the OGBN-PRODUCTS dataset, nodes symbolize products purchased on Amazon, and edges signify that the connected products were bought together. These three datasets are widely used as benchmarks for evaluating GNN models. Basic statistics for each dataset are provided in Table III.

TABLE III: Graph dataset statistics.

Dataset	# Nodes	# Edges	Sparsity(%)
REDDIT	232,965	114,615,892	99.789
OGBN-PROTEINS	132,534	79,122,504	99.550
OGBN-PRODUCTS	2,449,029	123,718,280	99.998

Evaluation Methodology: We evaluate `G-Sparse` by comparing it with state-of-the-art GPU implementations, including DGL v0.9.1 [16], FeatGraph [19], Sputnik [21], and cuSPARSE [20], to demonstrate its performance advantages in kernel performance benchmarks, end-to-end model training, and inference benchmarks. We use DGL to conduct performance evaluations and integrate `G-Sparse` kernel implementations into DGL for comparison. In end-to-end model benchmarks, DGL employs the cuSPARSE vendor library whenever possible.

To ensure fairness, all kernel execution measurements are initiated from DGL's Python interface, as performance overhead exists when calling native functions from Python.

A. Overall Performance

We investigate the kernel performance of g-SpMM and g-SDDMM under various scenarios by comparing them across different feature lengths ranging from 1 to 1024, with each length being a multiple of 2, for each dataset.

a) G-SpMM Performance: For simplicity, we focus on the g-SpMM with the `copy_lhs` binary operation and the `sum` reduction type, which can be supported by cuSPARSE.

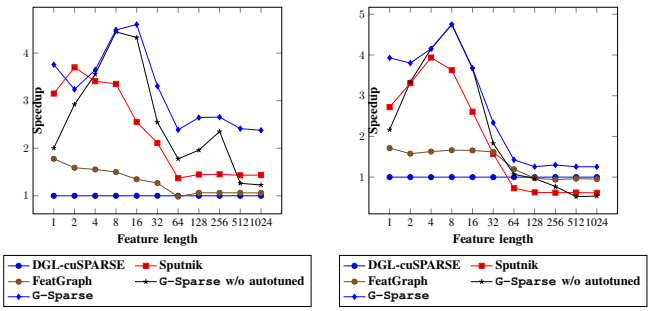
As illustrated in Figures 10, our performance surpasses cuSPARSE, DGL, FeatGraph, and Sputnik for most tested datasets and feature lengths.

Compared to the DGL with cuSPARSE implementation, our kernels achieve an average speedup of $3.2\times$, $2.6\times$, and $1.5\times$ on REDDIT, OGBN-PROTEINS, and OGBN-PRODUCTS, respectively. We observe a greater acceleration on the REDDIT dataset, likely because it is more unbalanced than OGBN-PROTEINS and has more non-zeros per row on average than OGBN-PRODUCTS, as Table V shows. Consequently, our row balance and data reuse optimizations have a more significant impact on its performance.

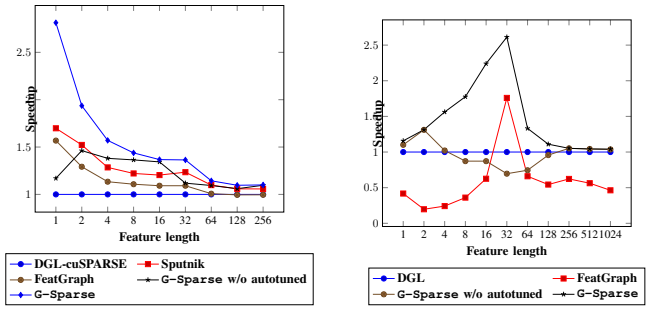
Compared to FeatGraph, we achieve up to a $3.4\times$ speedup and an average of $1.9\times$ speedup. FeatGraph's performance is comparable to cuSPARSE but falls short when the feature length is smaller than 32. Although both FeatGraph and the latest version of DGL apply the same optimization techniques, FeatGraph can adjust and tune the number of CUDA blocks for different datasets and feature lengths, resulting in higher performance than DGL.

Compared to Sputnik, we obtain up to a $2.1\times$ speedup and an average of $1.4\times$ speedup. While Sputnik employs hierarchical 1-D dimensional tiling, we utilize 2-D shared memory optimization, which offers better data reuse efficiency. Furthermore, Sputnik does not apply the 1-D stride register tiling as we do.

b) G-SDDMM Performance: `G-Sparse` consistently outperforms or matches DGL and FeatGraph for g-SDDMM kernels in all tested cases. Figure 6 illustrates the performance

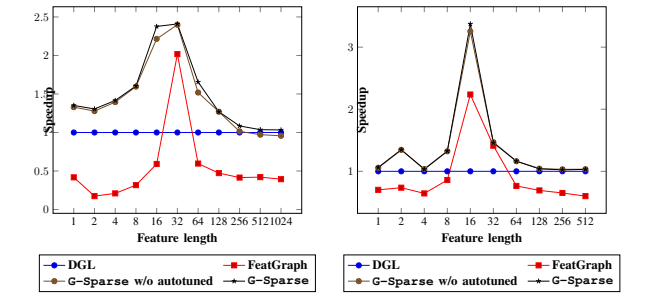


(a) REDDIT (**g-SpMM**) (b) OGBN-PROTEINS (**g-SpMM**)



(c) OGBN-PROTEINS (**g-SpMM**)

(d) REDDIT (**g-SDDMM**)



(e) OGBN-PROTEINS (**g-SDDMM**) (f) OGBN-PRODUCTS (**g-SDDMM**)

Fig. 10: G-SpMM and g-SDDMM benchmarks for REDDIT, OGBN-PROTEINS and OGBN-PRODUCTS datasets. When the feature length is 512 (g-SpMM) and 1024 (g-SpMM and g-SDDMM), we encountered out-of-memory on both of the baselines and `G-Sparse` on OGBN-PRODUCTS.

comparison for g-SDDMM kernels. `G-Sparse` achieves approximately $1.02\times$ to $3.37\times$ speedup and an average of $1.46\times$ speedup over DGL.

Although both DGL and FeatGraph employ warp shuffle optimizations, we achieve higher performance for feature lengths smaller than 32. This is because DGL and FeatGraph only apply warp shuffle optimizations for feature lengths equal to or greater than 32. Since the lane width of warp shuffles can be an integer multiple of 2, we can perform warp shuffles for feature lengths ranging from 2 to 32.

B. Ablation Study

We conduct an ablation test for g-SpMM and g-SDDMM kernels with different feature lengths on the REDDIT dataset.

As shown in Table II, 2-D shared memory tiling and row balancing greatly impact performance and do not exhibit significant performance changes with variations in feature lengths. We observe that warp shuffles provide considerable benefits for g-SDDMM kernels but are not robust to the size of the feature lengths.

Conversely, 1-D stride register tiling influences approximately 10% of the performance and shows no benefit when the feature length is 8. We find that this is likely because when the feature length is smaller than the warp size of 32, the GPU warp is not fully utilized, leaving no room for register tiling.

TABLE IV: Autotuned schedule option for g-SDDMM on REDDIT. The optimal width of `G-Sparse` lanes varies with the feature length, while DGL uses fixed lane width 32.

Feature length	`G-Sparse` threads	Lane Width	
		`G-Sparse`	DGL
8	128	2	N/A
32	32	4	32
128	8	16	32
512	4	32	32

TABLE V: Row length coefficient of variation(COV) and average non zeros per row(AVG).

Dataset	COV	AVG
REDDIT	1.63	492
OGBN-PROTEINS	1.04	597
OGBN-PRODUCTS	1.88	51

TABLE VI: Autotuning time taken of g-SpMM on REDDIT.

Feature Length	Cost Model Inference (s)	Genetic Search (s)	Total (s)
8	1.74	7.03	8.77
16	1.77	7.77	9.54
32	1.76	8.45	10.21
64	1.76	9.79	11.55

C. Productivity

Our `G-Sparse` implementations are not only more efficient but also require fewer lines of code compared to DGL. `G-Sparse` codes are $3.6\times$ and $4.4\times$ shorter than DGL without counting DGL's utility code lines while achieving $2.5\times$ and $1.45\times$ average speedup over DGL for g-SpMM and g-SDDMM computations, respectively.

In terms of details, DGL requires 313 core code lines along with an additional 176 code lines for g-SpMM computations, while our `G-Sparse` implementation only needs 86 core code lines. For g-SDDMM computations, DGL requires 319 core code lines with the same extra utility code lines, while our `G-Sparse` implementation only needs 72 core code lines.

D. Autotuning Evaluation

We apply autotuning across all experiments to obtain the best-performing kernel. Our autotuning system combines a cost model with a genetic search algorithm, taking approximately 8 to 20 seconds for each experiment. Table VI shows the autotuning time of g-SpMM for the REDDIT dataset. Autotuning results can be saved and reused, allowing us to save time when autotuning the same dataset and the operations with the same shapes.

We compare autotuning results with those obtained by domain experts who manually tuned the scheduling over several days. As seen in Figure 10, kernels with autotuning outperform those without autotuning in all cases, achieving a maximum speedup of 3.7×. This can be attributed to the fact that autotuning combines domain expertise with comprehensive search spaces that are too large for experts to consider every aspect thoroughly.

We find that autotuning is more adept at identifying the optimal solution for allocating GPU resources, such as the number of threads per block and number of registers, which are often mutually constrained, compared to domain experts. As demonstrated in Table IV, the optimal number of threads for warp shuffle in g-SDDMM does not always equal the feature length or 32 (the warp size). Because more warp shuffle instructions require more threads and registers, and the total number of threads and registers is limited, autotuning is employed to discover the best configuration. Moreover, the performance is close between `G-Sparse` and `G-Sparse` without autotuning on Figures 10(e) and 10(f). We observe that, in these cases, the schedules of autotuning are highly consistent with those written by our experts. Overall, autotuning leads to higher or the same performance as the experiments without autotuning in all cases.

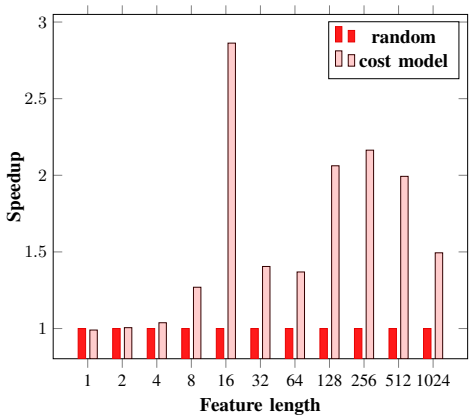


Fig. 11: Kernel speedups of cost model than random sampling with the same number of trials.

Cost Model Evaluation: We train the cost model on the performance data of the REDDIT dataset for 1000 epochs. Pearson correlation coefficients are used to assess the prediction performance of our cost model. For inference correlation testing, the REDDIT, OGBN-PROTEINS, and OGBN-PRODUCTS datasets are measured, with the feature lengths of the REDDIT dataset differing from the training ones.

We train the cost model using three different kernels on the REDDIT dataset. Each kernel has approximately 20 to 1500 performance samples, depending on its search space size. For training results, we predict the performance of training samples and then measure their correlation with real performance measurements. For testing results, correlation coefficients are calculated using different kernels that were not part of the training process on all three datasets. This process is repeated for g-SpMM and g-SDDMM, respectively. As observed, even though the cost model is trained on the REDDIT dataset, the training correlation reaches 0.88 and 0.99, while the inference correlation achieves 0.74 and 0.89 for g-SpMM and g-SDDMM, respectively. This indicates the relatively good generalization of our cost model.

Figure 11 demonstrates that the cost model is more efficient than random sampling. With the same number of trials (three), the best performance of the kernel from the cost model achieves a 1.0× to 2.9× speedup on REDDIT compared to the random sampling method. When the feature length is less than 8, there are only tens or hundreds of candidates to try, so the cost model and random sampling exhibit little difference.

TABLE VII: End-to-end training (one epoch) and inference speedups on REDDIT, where GraphSage uses full batch.

	Model	DGL (s)	`G-Sparse` (s)	Speedup
Training	GCN	0.4165	0.2742	1.52×
	GAT	0.5832	0.4243	1.37×
	GraphSage	0.5509	0.2520	2.19×
Inference	GCN	0.2094	0.1274	1.64×
	GAT	0.2858	0.1971	1.45×
	GraphSage	0.2688	0.1197	2.25×

E. GNN Model Evaluation

Herein, we compare the end-to-end performance of GCN [5], GAT [38], and GraphSage [6] models with different kernels on the REDDIT dataset. The `G-Sparse` implementations are integrated into DGL for performance testing, and DGL utilizes the cuSPARSE library if cuSPARSE supports the computation. In all three models, the number of layers is two, and the hidden size is 256 for each layer. GCN and GraphSage use g-SpMM with the aggregation type of `sum`, while GraphSage can also use other types such as mean and max. GAT employs both g-SpMM and g-SDDMM with more aggregation types and incorporates both vertex and edge feature embeddings.

The full-batched training is used for all models. The reasons come from two aspects. On one hand, our core interest lies in the performance improvement brought by the accelerated sparse computations and the full-batched training has a stable speed per epoch. On the other hand, the current implementation of `G-Sparse` still has one limitation that uses the number of non-zeros in the sparse matrix as the input feature to our DNN-based autotuner (refer to Algorithm 4). We plan to overcome this limitation in our future work to integrate DSL compilation with autotuning techniques to effectively accelerate sampling-based GNN learning. Nonetheless, the mini-batched training is supposed to demonstrate consistent acceleration results with the full-batched, since the overhead of graph sampling in mini-batched training can be removed or hidden in practice. For instance, two popular approaches can be used. One is to generate sub-graph examples beforehand and store them on disk for future use in training/inference [39]–[41]. The other is to exploit parallel data prefetching in data loaders [16], [42], [43].

TABLE VIII: Feature comparison of G-Sparse with other works

	GNNs	Flexibility	Autotuning	Efficiency
cuSPARSE [20]	no	no	no	middle
Sputnik [21]	no	no	no	high
DGL [16]	yes	no	no	middle
FeatGraph [19]	yes	yes	limited	middle
G-Sparse	**yes**	**yes**	**yes**	**high**

Table VII displays the average time required for running inference and an epoch of training. The table demonstrates that DGL with G-Sparse outperforms the original DGL for all models, with the average speedup of $1.69\times$ for training and $1.78\times$ for inference.

We have verified that the training accuracy and the test accuracy obtained by using our kernels in DGL, within the same number of epochs, match that of DGL for each pair of the model and dataset. We trained the three models with 200 epochs to check for accuracy. The test accuracy of G-Sparse matches that of the original DGL implementations - 94.0% for GCN, 93.7% for GraphSage (full batch), and 93.1% for GAT.

V. Related Work

Sparse computations in GNNs dominate the overall runtime in training and inference. Recently, a few implementations have been proposed to speed up the sparse computations in GNNs. DGL [16] and FeatGraph [19] utilize node and edge parallelism and perform thread-level parallelism for feature dimension [44]. DGL employs the cuSPARSE [20] library when the latter supports the required computation. Table VIII compares the features of G-Sparse with both DGL and FeatGraph.

Sputnik [21] primarily targets DNNs and introduces shared memory optimization, vector instructions, GPU row swizzle load balancing, and other optimization techniques. Table VIII compares the features of G-Sparse with Sputnik. spECK [45] optimizes the load balancing problem for sparse computation. Ge-SpMM [18] optimizes the performance of g-SpMM for GNNs, supporting generic SpMM sparse computation and using the coalesced row caching method to access global memory efficiently. ES-SpMM [46] co-designs edge sampling and SpMM kernel to fit the graph into shared memory, reusing data to accelerate sparse computations. FusedMM [17] mainly supports generic GNN sparse computation on CPUs, optimizing load balancing and efficient use of memory bandwidth.

Halide [25], TVM [24], Cortex [28], AKG [29], Triton [30], Ansor [26], and Tiramisu [31] are all DSL compilers targeting image processing and DNNs, primarily consisting of dense matrix computations. TACO [47] is a tensor algebra compiler capable of expressing sparse computations. Scheduled TACO [48] supports split, collapse, and reorder schedules. However, scheduled TACO does not support non-rectangular buffer bound inference and bound bind index expression introduced by us for sparse computations.

We implement G-Sparse as a DSL by extending Halide to describe the computations in GNNs. Additionally, we integrate more optimizations into the schedule strategy, including data reuse and row balancing, and introduce autotuning to obtain optimal results by exploring more schedule space.

VI. Conclusions

We propose G-Sparse to accelerate generalized sparse computation in GNNs, which utilizes DSL compiler to separate the algorithm and schedule. G-Sparse extends Halide by introducing non-rectangular buffer bound inference and buffer binding index to enable DSL description and code generation for sparse kernels. Furthermore, G-Sparse introduces 2-D shared memory tiling, row balancing, 1-D stride register tiling, adaptive warp shuffles, and other optimizations in the schedule to increase data reuse efficiency and improve row balancing significantly. G-Sparse introduces a novel DNN-based cost model combined with genetic search autotuning, which can automatically search for optimal results without human intervention. As a result, our kernels get up to $4.75\times$ faster than previous technologies. By integrating G-Sparse into the DGL, we end up with a model training and inference performance speedup of $1.37\times \sim 2.25\times$ compared to DGL.

Our autotuning still takes seconds to generate the optimal program, and is only used on GPU hardware. Using autotuning to generate optimally performant programs for a wide variety of hardware, datasets, and GNN models on time is one of the future research directions. Finally, we believe that compiler-driven acceleration technologies like our G-Sparse will play an important role in propelling the development of large graph models that serve as the foundation models in graph intelligence.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. The authors from Chinese Academy of Sciences are both in the Institute of Software. This work is supported by Ant Group through Ant Research Intern Program, the National Science Foundation of China (NSFC) Grant No. 62002350 and the Youth Innovation Promotion Association of Chinese Academy of Sciences, China.

Appendix

This appendix outlines a set of guidelines for conducting artifact evaluation. We begin by detailing the artifact check-list and the evaluation environment, and then present a step-by-step workflow for evaluating and reporting the results.

A. Artifact Check-list (Meta-Information)

- **Algorithm:** Compiler-driven sparse computations for Graph Neural Networks on GPUs.
- **Program:** C/C++ code and Halide DSLs.
- **Binary:** Python package with C/C++ plugin libraries.
- **Model:** GCN, GAT and GraphSAGE.
- **Data set:** REDDIT dataset from Hamilton et al., as well as the OGBN-PROTEINS and OGBN-PRODUCTS datasets from the Open Graph Benchmarks, where the DGL framework will download.

- **Run-time environment:** Linux
- **Hardware:** NVIDIA V100 or P100 GPUs.
- **Output:** The output is provided in the output stream.
- **How much disk space is required (approximately)?:** 10 GB for datasets.
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes.
- **How much time is needed to complete experiments (approximately)?:** 20 minutes.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.8254020

B. Description

- **How to access:** (1) The source code, benchmarks, and Python package are provided at `https://doi.org/10.5281/zenodo.8254020`, and (2) Download the source files `g-sparse.zip` and `gpc-1.0-cp310-cp310-linux_x86_64.whl` from the above URL.
- **Hardware dependencies:** NVIDIA V100 or P100 GPUs.
- **Software dependencies:** `CUDA 11.1/11.7`, `DGL 0.9.1/1.0.0`, `PyTorch 1.9.0/2.0.0`, and `Python 3.10`.

C. Installation

A Python package is provided for installation, and users can run the command `pip install gpc-1.0-cp310-cp310-linux\_x86\_64.whl` to make it.

D. Experiment Workflow

First, download the source codes and install the Python package as described in Section B. Second, extract the contents of the g-sparse.zip file to a designated directory. Finally, (1) to evaluate the kernels, run the script `run_kernels.sh` in the `benchmarks` directory, and (2) to evaluate the GAT, GCN and GraphSAGE models, run the script `run.sh` in the `benchmarks/gat`, `benchmarks/gcn`, and `benchmarks/graphsage` directories, respectively.

E. Evaluation and Expected Results

To get the results, execute the `run_kernels.sh` and `run.sh` scripts located in the `benchmarks` directories. The execution will generate performance results, which will be displayed in the output stream of the terminal.

References

[1] M. Zitnik, M. Agrawal, and J. Leskovec, "Modeling polypharmacy side effects with graph convolutional networks," *Bioinformatics*, vol. 34, no. 13, pp. i457–i466, 2018.

[2] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.

[3] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1263–1272.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[6] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[7] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018.

[8] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Advances in neural information processing systems*, 2018, pp. 4558–4567.

[9] Y. Zhou, Y. Cao, Y. Liu, Y. Shang, P. Zhang, Z. Lin, Y. Yue, B. Wang, X. Fu, and W. Wang, "Multi-aspect heterogeneous graph augmentation," in *Proceedings of the ACM Web Conference 2023*, 2023, p. 39–48.

[10] C. Huan, S. L. Song, Y. Liu, H. Zhang, H. Liu, C. He, K. Chen, J. Jiang, and Y. Wu, "T-gcn: A sampling based streaming graph neural network system with hybrid architecture," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2023, p. 69–82.

[11] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019.

[12] C. Wang, Z. Lin, X. Yang, J. Sun, M. Yue, and C. Shahabi, "Hagen: Homophily-aware graph convolutional recurrent network for crime forecasting," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 4.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, 2019, pp. 8024–8035.

[15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[16] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs." 2019.

[17] M. K. Rahman, M. H. Sujon, and A. Azad, "Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 256–266.

[18] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–12.

[19] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, "Featgraph: A flexible and efficient backend for graph neural network systems," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[20] NVIDIA, "The api reference guide for cusparse, the cuda sparse matrix library," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[21] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[25] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing

parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[26] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 863–879.

[27] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to interval analysis*. SIAM, 2009.

[28] P. Fegade, T. Chen, P. B. Gibbons, and T. C. Mowry, "Cortex: A compiler for recursive deep learning models," *arXiv preprint arXiv:2011.01383*, 2020.

[29] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li *et al.*, "Akg: Automatic kernel generation for neural processing units using polyhedral transformations," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1233–1248.

[30] P. Tillet, H.-T. Kung, and D. Cox, "Triton: An intermediate language and compiler for tiled neural network computations," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019, pp. 10–19.

[31] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 193–205.

[32] NVIDIA, "Cuda c++ programming guide," 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[33] Y. Liu and B. Schmidt, "Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 82–89.

[34] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[35] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *arXiv preprint arXiv:1805.08166*, 2018.

[36] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*. PMLR, 2015, pp. 448–456.

[37] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

[38] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[39] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, p. 5171–5181.

[40] M. Zhang, P. Li, Y. Xia, K. Wang, and L. Jin, "Labeling trick: A theory of using graph neural networks for multi-node representation learning," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021.

[41] H. Yin, M. Zhang, Y. Wang, J. Wang, and P. Li, "Algorithm and system co-design for efficient subgraph-based graph representation learning," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2788–2796, 2022.

[42] Y. Bai, C. Li, Z. Lin, Y. Wu, Y. Miao, Y. Liu, and Y. Xu, "Efficient data loader for fast sampling-based gnn training on large graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2541–2556, 2021.

[43] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2094–2105, 2019.

[44] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu," in *European Conference on Parallel Processing*. Springer, 2018, pp. 672–687.

[45] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "speck: Accelerating gpu sparse matrix-matrix multiplication through lightweight analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 362–375.

[46] C.-Y. Lin, L. Luo, and L. Ceze, "Accelerating spmm kernel with cache-first edge sampling for gnn inference," *arXiv preprint arXiv:2104.10716*, 2021.

[47] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[48] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A sparse iteration space transformation framework for sparse tensor algebra," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.