The Sparse Roofline Benchmark Specification

Editors: Willow Ahrens, Olivia Hsu, Kyle Deeds (Please add your name!)

June 2023

1 Introduction

Sparse matrix and tensor problems have received increased attention in recent years. We have developed this benchmark suite to encourage meaningful comparison among the varieties of solutions. Direct comparisons in sparse computing research are difficult due to the wide variety of sparse kernels, input-dependent performance, differences in intended use cases for each kernel, and a lack of realistic or theoretically interesting datasets.

Prior benchmark suites have encouraged competition to improve the performance of various subsets or combinations of sparse kernels. SparseBench was proposed by the Netlib to measure numerical improvements to sparse iterative solvers [3]. The GAP (Graph Algorithm Platform) benchmark suite describes entire graph applications composed of multiple sparse kernels [4]. PASTA (A Parallel Sparse Tensor Algorithm Benchmark Suite) was proposed to measure sparse tensor kernels, but only concerns kernels with a single sparse tensor[8]. [Olivia: I think we also need to address database collections of sparse data here, like SuiteSparse and FROSTT, and how people use them as pseudo benchmark data but it is incomplete. This will further place where our contribution is and motivate this project]

Our benchmark suite aims to measure a set of specific sparse kernels that completely covers the sparse performance engineering landscape. This requires several novel features.

[Hengrui: A bit unsure what kind of tensor generator we are looking into, maybe reference? Kernelized generator with pre-specified interactions for big tensor seems to be an open problem, like https://proceedings.mlr.press/v206/wesel23a.html.] First, we specify datasets or generators for all of the inputs to a given kernel, including situations where multiple inputs are sparse and their patterns interact. As an example, this situation arises in sparse matrix times sparse matrix multiply, where the sparsity of the output is not known at the outset and may belong to different distributions depending on how the inputs correlate.

Second, we specify explicit staging of sparse kernels to quantify the tradeoff between preprocessing time and runtime. Many optimizations to SpMV, for example, involve reformatting and reorganizing the matrix before multiplying it, under the assumption it will be used multiple times.

Third, we provide sparse tensor statistics for each problem (such as the sparsity of the output or the number of required operations), and generators for pathological cases. This helps quantify when implementations are reaching their theoretical limit, the roofline.

Our benchmark suite aims to provide researchers with the software and specifications to easily benchmark sparse problems on realistic datasets across multiple languages and architectures. We also provide an easily accessible database to tabulate the results: the best known approaches to sparse problems.

2 Reference Implementation

While the precise mathematical specification of the computation is specified in this document, we have also provided a reference implementation for each problem.

3 Tensor Formats

To enable cross-language compatibility, all tensors generators or dataset downloads will produce tensor files in Binsparse tensor formats [2]. Possible tensor format names such as CSR or COO will refer to Binsparse format codes. Binsparse tensors are composed of several data vectors. On disk, these vectors may be stored in any array data file format such as an HDF5 container or an `.npy` file. During benchmarking, inputs and output data are expected to correspond to in-memory versions of these same arrays.

For example, the CSR structure involves three vectors, `pointers_to_1`, `indices_1`, and `values`. The column indices of the stored values located in row i are located in the range [`pointers_to_1[i], pointers_to_1[i+1]`) in the `indices_1` array. The scalar values for each of those stored values is stored in the corresponding index in the values array. On disk, these arrays might be serialized to a text representation, but in memory in C, the data would be referenced by two pointers to integers and one pointer to floating-point numbers.

The precise filenames and paths will be specified along with the benchmark kernel that uses them. For example, one might use the RMAT matrix generator included with the suite to generate inputs, stored in `spmv/RMAT/A.bsp.h5` and `spmv/RMAT/x.bsp.h5` [Hengrui: In h5 format, especially in those sparse data filled with zeros, practitioner will use a low-rank approximation (e.g., 'On the Compression of Low Rank Matrices' by Cheng et.al) to store it at the cost of minimal loss of accuracy.] Subsequent sections of this document may specify that inputs live in "dense" formats or "sparse" formats. We say that the dense binsparse formats are those that store all possible entries, and sparse formats are those that allow one to store only a subset of the entries.

[Willow: There are more possibilities here, we orthogonalize concerns to leave discussion to another committee...]

4 Timing

The time to transfer input/output data to and from the input file should not be measured. Timing should begin with inputs resident in memory with the specified binary format (see Section 3). The timing should end after the output is resident in memory with the specified binary format. This benchmark may apply to a diversity of architectures with different memory hierarchies, from distributed memories to special purpose hardware accelerators. The only requirement is that the outputs are resident in the same memory as the inputs, and that only one copy of the input is used. [Grace: Some clarification about how to handle this in the case of accelerators with distinct scratchpads and accumulators should be done. Do scratchpads and accumulators count as 'the same' memory?]

Some accelerators have high bandwidth costs to and from the chip (making timings for problems resident in accelerator memory less realistic for those who wish to use the data off-chip). However, this may be accounted for naturally as the problem size grows to exceed the on-chip memory.

4.1 Staging

Each benchmark is separated into stages based on which inputs the application is allowed to read at each stage. For example, to enable a clear distinction between matrix preprocessing and runtime in SpMV kernels, the first stage allows access to just the matrix. Any vector-agnostic preprocessing can be timed within the first stage. The second stage allows access to the vector, and thus computation of the output. We recognize that many divisions between stages are possible; we have selected separations into stages that we believe are meaningful for applications.

4.2 Results Aggregation

4.2.1 Deterministic data

In situations where the input sparsity patterns are fixed, applications may run as many executions as they wish, reporting the minimum time. Applications should report whether the cache was cleared at the start of each execution.

4.2.2 Randomized data

In situations where the input sparsity patterns are drawn from a distribution, applications must choose a random seed for the generator, and report the mean and variance of the execution times, as well as confidence intervals for each statistic. Implementations are free to choose how many executions are used, but a confidence interval of at least 95% is required for the mean time.

The software implementation provides a convenient command-line utility to compute these statistics, as well as an in-software utility, so implementers need only record the runtime on each random input.

For each random trial input, implementers should follow the guidelines for deterministic data, taking the minimum of as many execution times as they wish and reporting whether the cache was cleared before each.

5 Correctness and Validation

All kernels in the benchmark suite are composed of one or more statements of the form

$$A_{i\ldots} = \sum B_{j\ldots} \cdot C_{k\ldots} \cdot \ldots, \tag{1}$$

where i, j, k may contain overlapping indicies and the operators $+$ or $\cdot$ may be substituted with other operators.

5.1 Allowed Optimizations

Reference implementations are provided for all kernels in the benchmark suite. Historically, we recongize that many optimizations have been proposed which are typically excluded from benchmarks, such as Strassen's algorithm. However, a goal of this benchmark is to encourage exploration of the unique properties of sparse kernels that arise from the constraint that one must do "all the same operations" as the reference implementation. We therefore add a category for limited optimizations, and considerations for those who wish to perform further algorithmic optimizations.

5.1.1 The Semiring Category

Both theoretical lower bounds and popular implementations of sparse kernels, such as GraphBLAS, have noticed that a wide variety of sparse optimizations can be described by the mathematical construct of semirings. Thus our "limited" optimizations may utilize any property derived from operators forming semirings. This is the default category we expect most optimizations to fall under, and includes the associative and commutative property. This does not include the distributive property. In this categorization, we treat floating point numbers as real numbers for the purpose of applying mathematical properties.

5.1.2 Beyond Semirings

For implementations that wish to break the above rule, any mathematically equivalent optimization is allowed. This category allows optimizations such as Strassen's algorithm.

5.2 Floating-Point Accuracy

Although the previous section treats floating point numbers as real numbers, in practice they behave differently. We do not require any particular exception handling behavior.

5.2.1 Error Bounds

Implementations involving floating point numbers must satisfy a relative error bound on (1) of the following form, where ϵ is a declared parameter and n is the number of elements in the sum:

$$\forall_{i\ldots} |A\_ref_{i\ldots} - A\_res_{i\ldots}| \leq n\epsilon \sum |B_{j\ldots} \cdot C_{k\ldots} \cdot \ldots|$$

If only `Semigroup` level optimizations are used, then this equation will hold for the default value of machine ϵ.

This equation is inspired by the standard error bound for recursive summation, taken from [7], and our generalized form can be adapted to several strategies for summation.

6 Datasets

What makes a choice of dataset a good choice?

- There is a citation we can make
- There is a clear story/good argument connecting this data and its use (on a given kernel) to an application that people care about.
- The dataset is taken from another already existing benchmark
- The dataset exercises a particular extremal complexity case (e.g. expander graphs, or things coming from the lower bounds group)
- That the dataset should be representative of a real application of a kernel (i.e. make sense together)

6.1 Tensor Datasets

Tensors may be generated by a generator, or downloaded from a dataset online. The tensor datasets are listed here, together with their keys. For example, the key `SuiteSparse/Boeing/ct20stif.bs` refers to the `Boeing/ct20stif` matrix in the `SuiteSparse` collection. The `download.jl` script included with the suite will download and reformat datasets to binsparse or TTX format.

`SuiteSparse`	The University of Florida SuiteSparse Matrix Collection [6]
`FROSTT`	The Formiddable Repository of Open Sparse Tensors and Tools [9]
`ImageNet`	`https://www.kaggle.com/competitions/imagenet-object-localization-challenge/`

[Hengrui: I suggest fMRI datasets and HMP datasets, https://adni.loni.usc.edu/, which are kind of popular and known to be lacking this kind of benchmark]

6.2 Tensor Generators

Tensor generators are listed here, together with a description of their parameters. All tensor generators that use random numbers additionally take a seed argument s.

6.2.1 RMAT

The RMAT matrix generator [5] creates graphs with power-law degree distributions, which presents an interesting challenge to load balancing for parallelization. The generator works by taking the Kronecker product of a seed matrix with itself several times to produce a matrix of probabilities of including nonzeros. An artifact of RMAT matrix generation is that heavy rows are often grouped together (a ”staircase” pattern), so we choose to apply a random row/column permutation to the rows and columns. RMAT is used in the Graph500 and GAP benchmarks [1, 4]. Additionally, the tensor generalization (which takes Kronecker products of a tensor seed with itself) is used in the PASTA Benchmark Suite [8].

The RMAT matrix generator classically uses a generating matrix

$$G = \begin{bmatrix} A & B \\ C & D \end{bmatrix},$$

though we will specify the entire tensor G, rather than the parameters A, B, C, D. Additionally, we will specify r, the number of seed matrices which should be joined with a Kronecker product.

7 Kernels

7.1 SpMV

Sparse Matrix Vector multiplication (SpMV) [Hengrui: abbr.]

7.1.1 Input Arguments

m: a positive integer.

n: a positive integer.

A: an $m \times n$ sparse matrix.

x: an n-length dense vector.

7.1.2 Output Arguments

y: an m-length dense vector.

7.1.3 Computation

$$y_i = \sum_j A_{ij} x_j$$

7.1.4 Sparse Format Variants

[Gilbert: This text should be reduced. It's just here to suggest] There should be a list here of options that have already been defined in the format section for A and x (and any other inputs). In the case of SpMV, this just means specifying format for A, because x and y are dense.

[Willow: This is the section that would specify whichever formats we think are interesting for SpMV.]

7.1.5 Benchmark Stages (Provide timings for each stage)

SpMV is frequently used repeatedly on the same matrix. Therefore, it is the subject of many optimizations which are independent of the vector x. Thus, we separate the benchmark into two stages:

- **(Stage 1)** The kernel may read m, n, and A.
- **(Stage 2)**: The kernel may read x.

7.1.6 Datasets

`spmv/kronecker10`: 100 random kronecker networks at 10 percent sparsity, using the same parameters as GAP and Graph500 [1].

`spmv/kronecker10/A`: $\texttt{RMAT}\left(r = 27, G = \begin{bmatrix} 0.57 & 0.19 \\ 0.19 & 0.05 \end{bmatrix}\right)$

`spmv/kronecker10/x`: $\texttt{rand}\left(m = 2^{27}\right)$

`spmv/OSKI`: the sparse matrices used in evaluation of "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply" [10].

7.1.7 Variants to Measure

[Hengrui: We perhaps want to select some problems with ground truth so that we can have the actual error.] Here are the combinations of settings that should actually be measured.

Everything after this line is a draft:

7.1.8 Variations

Also include SpMV transpose.

Mention $A = A^T$, possible additional optimizations, including"Fast Bilinear Algorithms for Symmetric Tensor Contractions", E. Solomonik, J. Demmel, Computational Methods in Applied Mathematics, Feb 2020. https://doi.org/10.1515/cmam-2019-0075

7.2 Iterative (Jacobi? PCG?) Solve

Input (Stage 1): Matrix given (e.g. "Boeing/ct20stif" or power-law graph laplacian) Input (Stage 2): Vector (e.g. uniform random dense vector) Compute: $y_j = A_ij * x_j$ Timings: Report time start stage 1 to start stage 2 Report time start stage 2 to end Report aggregate time Some combination of: 1. Allow reorganization of algorithm up to equivalence under arbitrary semirings (treat unsupported functions as uninterpreted) (e.g. thou shalt not do Strassen) 2. Allow reorganization of algorithm up to equivalence under infinite precision 2.5 (2 and 3) 3. Anything you want up to declared empirical accuracy If algorithm is iterative, run steps until tolerance If algorithm is non-iterative, report achieved empirical accuracy compared to double.

For SpMV, we compare to the naive implementation and take the L2-norm of the residual.

7.3 SpMSpV

"Sparse-Sparse Matrix-Vector Multiply" i.e. the matrices and vectors are all sparse.

$$y_i += A_{i,j} * x_j$$

7.4 SpMM (SpGEMM?)

Input (Stage 1): Matrix Input (Stage 2): Vector Compute: $y_ij = A_ik * x_kj$ Timings: Report time start stage 1 to start stage 2 Report time start stage 2 to end Report aggregate time

SpGEMM from multi-source BFS: Computation: Compute BFS parents starting from a particular set of vertices. Distribution: If the graph is fixed, you can precompute the BFS parents (not interesting). Thus, this problem is only well-posed on graph generators. I'm not sure if the set of vertices matters, could pick randomly.

SpGEMM from setup phase of algebraic multigrid: Computation: Compute $R_l A_l P_L$ as described in `https://doi.org/10.1145/3571157`. Distribution: matrices described in `https://doi.org/10.1145/3571157` SpGEMM with various levels of size prediction difficulty and output reuse: Computation: Compute C = A*B Distribution: Generate A and B (using RMAT, perhaps?) so that expected number of nonzeros in A and B remain fixed but expected number of nonzeros in C can be very high, very low, or highly variable

7.5 ATA

7.6 MTTKRP

Computation: A single MTTKRP Distribution: The dense matrices are random, but the tensor can be fixed or random.

7.7 Candecomp

Computation: Some fixed number of MTTKRPs of the same tensor in alternating transposition orders on the same factor matrices Distribution: The tensor can be fixed or random here as well. Stencil (Same as SpMV, but matrix is a stencil) Computation: SpMV or Congugate gradient Distribution: Matrix is a toeplitz matrix for a particular stencil, vector is random

7.8 FFT

Computation: SpMV Distribution: Matrix is an FFT matrix, vector is random

7.9 Dynamic PageRank

Computation: Compute Pagerank of a matrix Stage 1. The first (n - d) rows and columns of a matrix are given Stage 2. The final d rows and columns are given Computation: Do 20 iterations of pagerank Notes: This incentivizes the implementation to use the first portion of the matrix to accelerate the computation in stage 2. (Gilbert: This is just a power iteration? Willow: It is. I was trying to emphasize the dynamic graph kernel problem, where updates to a graph are received in real time and we are trying to maintain some statistics on the graph as the updates come in. Gilbert: I'm not sure page-rank or similar are incrementalized in practice? Maybe. Maybe worth considering social network clustering algorithms here instead?)

7.10 Quantum Circuits

Computation: Compute the distribution of the quantum circuits outputs Distribution: 1 Tensor for each circuit element (10s-100s total) Transformer Model: Computation: Compute the output of a single encoder block in a transformer model with k attention heads Distribution: Randomly initialized weights with some sparsification algorithm applied

$$\begin{aligned}
Q[S,E,K] &= \sum_{C}(I[S,C] * WQ[C,E,K]) \\
K[S,E,K] &= \sum_{C}(I[S,C] * WK[C,E,K]) \\
V[S,E,K] &= \sum_{C}(I[S,C] * WV[C,E,K]) \\
AP[S,S',K] &= \sum_{E}(Q[S,E,K] * K[S',E,K])/8 \\
Z[S,E,K] &= \sum_{S}^{'}(Sigmoid(AP[S,S',K]) * V[S',E,K]) \\
CZ[S,E] &= \sum_{k}(Z[S,E,K] * WZ[K,E]) \\
FF[S,FF] &= Relu(CZ[S,E] * WFF1[E,FF]) \\
O[S,E] &= Relu(FF[S,FF] * WFF2[FF,E])
\end{aligned}$$

7.11 CountSat

Computation: Compute the number of satisfying solutions for a sat problem. This is a tensor contraction along each variable of the product of all clause tensors according to matching variables. Distribution: Randomly generated sat clauses with variables of length 3-10 (Existing Sat datasets? Are random sat problems unsatisfiable w.h.p.?)

7.12 Graph Neural Networks

Computation: Compute one iteration of message passing Distribution: Random graph models, random sparsified weights

$NeighborMessages[u,v,k'] = A[u,v]*(Sum_k(X[v,k]*WX[k,k'])+BX[k'])$

$AggNeighborMessages[u,k'] = Sum_v(NeighborMessage[u,v,k'])$

$NewX[u,k] = softmax_k(WX'[u,k](X[u,k]+AggNeighborMessages[u,k])+BX'[k])$

7.13 SubGraph Counting

Computation: Compute the number of subgraphs of a particular kind (triangle, flower, petal, etc.) Distribution: Random graph models, directed vs undirected subvariants

8 Future Benchmark Verticals

8.1 Quantum Chemistry Stuff

Oh cool, quantum chemistry stuff!

8.2 Morgue from Earlier Notes

Produce a List of all Computation Kernels (Acronyms Definitions) SpMV? SpMM? SpDM3? SpMSpV? SPGEMM? SpLU? SpTRSV/M?

SpMV: “Sparse Matrix-Vector Multiply” Dense x,y, Sparse A y[i] += A[i, j] * x[j] SpMSpV: “Sparse-Sparse Matrix-Vector Multiply” Sparse A, x, y y[i] += A[i, j] * x[j] SpGEMM: “Sparse-Sparse Matrix Multiply” Sparse C, A, B, often all matrices are square C[i, j] += A[i, k] * B[k, j] Masked SpGEMM: Sparse (sometimes Boolean) S, C, Sparse A, B C[i, j] += S[i, j] * A[i, k] * B[k, j] SDDMM: “Sampled Dense Dense Matrix Multiply” Sparse (sometimes Boolean) S, C, Dense A, B C[i, j] += S[i, j] * A[i, k] * B[k, j]

SpMM: “Sparse Matrix Multiply” Dense C, B, Sparse A, often J small constant C[i, j] += A[i, k] * B[k, j]

MTTKRP: “Matricized Tensor Times Khatri-Rao Product” Sparse B, Dense A, C, D A[i, j] += B[i, k, l] * C[k, j] * D[l, j] SpTTM: “Sparse Tensor Times Matrix” Sparse A, B, Dense C, usually K ¿¿ L or L ¿¿ K A[i, j, k] += B[i, j, l] * C[k, l] SpMTTKRP: Sparse A, B, C, D A[i, j] += B[i, k, l] * C[k, j] * D[l, j] SpLU: “Sparse LU Factorization” Sparse A, produce permutations P, Q, sparse lower triangular L, and sparse upper triangular U such that PAQ=LU SpTRSV: “Sparse Triangular Solve” Sparse triangular matrix A, dense vector y, produce dense vector x such that Ax = y SpTRSM: Sparse triangular matrix A, dense matrix Y, produce dense matrix X such that AX = Y

Willow: I’ve taken a stab at listing some kernel definitions. One might group them broadly by: 1 argument sparse: SpMV, SpMM, SDDMM,MTTKRP, SpTTM 2+ argument sparse: SpGEMM, SpMSpV, SpMTTKRP Loop carry dependency: SpLU, SpTRSV, SpTRSM

References

[1] Graph 500 | large-scale benchmarks. URL: `https://graph500.org/`.

[2] GraphBLAS/binsparse-specification: A cross-platform binary storage format for sparse data, particularly sparse matrices. URL: `https://github.com/GraphBLAS/binsparse-specification`.

[3] SparseBench home page. URL: `https://netlib.org/benchmark/sparsebench/`.

[4] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite, May 2017. arXiv:1508.03619 [cs]. URL: `http://arxiv.org/abs/1508.03619`.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM)*, Proceedings, pages 442–446. Society for Industrial and Applied Mathematics, April 2004. URL: `https://epubs.siam.org/doi/10.1137/1.9781611972740.43`, `doi:10.1137/1.9781611972740.43`.

[6] Timothy A. Davis and Yifan Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, November 2011. URL: `http://dl.acm.org/citation.cfm?doid=2049662.2049663`, `doi:10.1145/2049662.2049663`.

[7] N. Higham. The Accuracy of Floating Point Summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, July 1993. URL: `https://epubs.siam.org/doi/abs/10.1137/0914050`, `doi:10.1137/0914050`.

[8] Jiajia Li, Yuchen Ma, Xiaolong Wu, Ang Li, and Kevin Barker. PASTA: a parallel sparse tensor algorithm benchmark suite. *CCF Transactions on High Performance Computing*, 1(2):111–130, August 2019. `doi:10.1007/s42514-019-00012-w`.

[9] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools, 2017. URL: `http://frostt.io/`.

[10] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 26–26, November 2002. ISSN: 1063-9535. `doi:10.1109/SC.2002.10025`.