Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques

Serif Yesil
syesil2@illinois.edu
University of Illinois at Urbana-Champaign

José E. Moreira
jmoreira@us.ibm.com
IBM Research

Josep Torrellas
torrella@illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

Recent processors have been augmented with matrix-multiply units that operate on small matrices, creating a functional unit-rich environment. These units have been successfully employed on dense matrix operations such as those found in the Basic Linear Algebra Subprograms (BLAS). In this work, we exploit these new matrix-multiply facilities to speed up Sparse Matrix Dense Matrix Multiplications (SpMM) for highly sparse matrices.

SpMM is hard to optimize. The sparsity patterns lead to a highly irregular memory access behavior. Additionally, each sparse matrix has unique characteristics, making it hard to find a single SpMM strategy that works well for all sparse matrices. The addition of matrix-multiply units makes this even more challenging.

In this paper, we address these challenges. First, we design Dense Dynamic Blocks (DDB), a method to utilize the new matrix units. DDB has two specialized versions: DDB-MM and DDB-HYB. DDB-MM is a strategy that only utilizes the matrix-multiply facilities. DDB-HYB is a hybrid approach that maximizes the floating-point throughput by utilizing both vector and matrix units. Furthermore, we design a prediction mechanism for identifying the best SpMM strategy for a given sparse matrix and dense matrix pair: SpMM-OPT. SpMM-OPT selects among vector unit oriented, matrix unit oriented, and hybrid strategies for the highest floating-point throughput while taking cache optimizations into account.

We experiment with 440 matrices from the well-known SuiteSparse matrix collection on a POWER10 system with vector and matrix units. We show that DDB-MM and DDB-HYB can achieve a floating-point throughput of up to 1.1 and 2.5 TFLOPs/s on a POWER10 single-chip module for double- and single-precision SpMM, respectively. Our analysis also shows that SpMM-OPT effectively chooses the best SpMM strategy and can achieve an average speedup of up to 2× compared to an optimized CSR baseline.

CCS CONCEPTS

• **Computing methodologies → Shared memory algorithms**; • **Mathematics of computing → Mathematical software performance**.

KEYWORDS

Sparse Matrix-Matrix Multiply, SpMM, Matrix-Multiply Assist, IBM POWER10

ACM Reference Format:
Serif Yesil, José E. Moreira, and Josep Torrellas. 2022. Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3524059.3532369

1 INTRODUCTION

Sparse Matrix Dense Matrix Multiplication (SpMM) is a fundamental building block for many complex applications such as linear solvers [9, 10], graph analytics [40], recommender systems, and machine learning [21, 45]. Most of these applications are also iterative in nature, executing SpMM many times with the same sparse input matrix. As a result, SpMM often consumes most of an application's execution time, making it an important target to optimize.

Various state-of-the-art CPU and GPU systems are now featuring matrix-multiply hardware facilities tailored for dense matrix operations. These specialized units can execute dense matrix-multiply operations on small matrices (e.g., blocks of size 4×4). Examples of these are NVIDIA's Tensor Cores [2, 5], IBM's POWER10 Matrix-Multiply Assist (MMA) facilities [12, 41], and Intel's AMX [6]. These units are successfully utilized to maximize performance for dense matrix operations [2, 5, 32].

However, many real-world matrices have sparsity. For example, in the domain of neural network training and inference, up to 90% of matrix entries can be zero due to activation function outputting zero or weights becoming zero [34, 37]. Current work represents these matrices as dense matrices and tries to skip unnecessary computations involving elements that are zero or become zero during the training phase [2, 18]. In contrast, in the domains of graph analytics and scientific computing, matrices are often extremely sparse. As a result, they are represented in sparse formats and result in very irregular computations. In this work, we focus on SpMM with these extremely sparse matrices.

Previous work on speeding up SpMM for extremely sparse matrices using matrix-multiply hardware facilities includes Blocked CSR (BCSR) or its variations, used with GPU Tensor Cores [48, 54]. BCSR can create $r \times c$ dense blocks from consecutive rows and columns of the sparse input matrix. However, such blocks are hard to find in sparse matrices, and dense blocks that contain many zeros cause under-utilization of the matrix-multiply units' throughput.

Previous approaches were designed to improve the irregular accesses in Sparse Matrix-Vector Multiplication (SpMV) [24, 26, 27, 43, 44] and, more recently, to utilize Tensor Units for SpMM on

ICS '22, June 28–30, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9281-5/22/06…$15.00
https://doi.org/10.1145/3524059.3532369

GPUs [48]. These techniques rely on finding consecutive rows that show a similar nonzero structure within a consecutive range of columns. Finding such rows typically requires expensive mechanisms. Fortunately, SpMM is a more computationally-intensive operation than SpMV. In SpMM, a single nonzero causes an access to a whole row of the input dense matrix, instead of just a single element from the dense vector in SpMV. Therefore, it is less crucial to form dense blocks from consecutive columns. Furthermore, although matrix-units have significantly higher floating-point operation throughput compared to vector units, the matrix-units can be underutilized due to zero padding introduced by blocking, leading to lower effective FLOPs/s. For this reason, we focus on efficient utilization of matrix-multiply units rather than improving locality.

We propose to use a more relaxed dynamic approach for blocking: Dense Dynamic Blocks (DDB). We extract $R \times 1$ blocks from consecutive R rows of the sparse matrix. These $R \times 1$ blocks are used to create a dynamically sized dense block to execute on matrix units. This approach, called DDB-MM, can still suffer from zero padding and underutilize the floating-point throughput of the processor. To address this issue, we propose a novel approach in which computations in SpMM are synergistically executed on matrix units or on traditional vector units to maximize the floating-point throughput. We call our scheme DDB Hybrid (DDB-HYB). DDB-HYB extracts variable-sized dense blocks from the sparse input matrix. In DDB-HYB, the matrix-multiply units process the blocks with a large fraction of nonzero elements; the remaining, sparser blocks use traditional vector units to maximize performance.

For many sparse matrices, the number of dense blocks extracted will be minimal or the dense $R \times 1$ blocks will have many zeros in them. Therefore, they will not benefit from DDB-MM or DDB-HYB significantly. For this reason, we also propose a simple metric, *Average Flop Throughput* (AFT), to assess the efficacy of DDB-HYB for a given matrix before its application. AFT can classify the matrices coarsely based on their potential to get benefits from matrix-multiply units. This coarse-grain classification can limit the search space, but it does not always choose the best SpMM strategy.

For this reason, we develop *SpMM-OPT*. SpMM-OPT is a machine learning approach with a carefully crafted feature set to select the best SpMM strategy in a functional unit-rich environment. It can choose between a vector unit oriented approach (CSR), a matrix unit oriented approach (DDB-MM), and a hybrid approach (DDB-HYB). It can also decide on cache optimizations by selecting an appropriate slicing factor for the dense matrix of SpMM.

We validate the performance of DDB-MM and DDB-HYB on a large selection of matrices from the SuiteSparse matrix collection. First, we establish an efficient baseline with a CSR implementation. Then, we assess the speedup of DDB-MM and DDB-HYB over this baseline. We observe that DDB-MM and DDB-HYB can achieve a floating-point throughput of up to 1.1 TFLOPs/s for double-precision SpMM (corresponding to 40% of LINPACK's throughput) and 2.5 TFLOPs/s for single-precision SpMM, on a single-chip POWER10 processor. Finally, we analyze the effectiveness of SpMM-OPT to select efficient SpMM strategies. Our analysis shows that SpMM-OPT effectively chooses the best SpMM strategy and can achieve an average speedup of up to 2× compared to an optimized CSR baseline.

2 BACKGROUND

In this paper, we use upper case letters for matrices (e.g., **A** and **B**). $\mathbf{A}[i, j]$ is the element in row i, column j of matrix **A**. To represent a slice of a matrix, we use the $\mathbf{A}[a : b, x : y]$ notation, which gives elements of **A** in rows a to b and columns x to y. A slice without any boundaries gives all the elements in that dimension. For example, $\mathbf{A}[k, :]$ represents all the elements in row k of matrix **A**.

An SpMM kernel multiplies a sparse $M \times T$ matrix **A** with a dense $T \times N$ matrix **B**, generating a dense $M \times N$ matrix **C**. A row i of matrix **C** is updated with the computation $\mathbf{C}[i, :] = \sum_j \mathbf{A}[i, j] \times \mathbf{B}[j, :]$. Since matrix **A** is sparse, a row of **A** ($\mathbf{A}[i, :]$) has few nonzeros. These sparse matrices require a compact representation.

This section describes the Compressed Sparse Row (CSR) representation of sparse matrices and its SpMM implementation, and discusses the Matrix-Multiply Assist (MMA) instructions of the IBM POWER10, used for validating our ideas.

2.1 CSR Format and SpMM Implementation

Large sparse matrices require compact in-memory representation. The compressed sparse row format (CSR) (or variants) is the most popular choice for basic linear algebra, graph processing, and machine learning frameworks [1, 33, 40].

As shown in Figure 1, CSR uses three arrays to represent a sparse matrix: `vals`, `col_id`, and `row_ptr`. The `vals` array stores all of the nonzero elements in the matrix. The `col_id` array stores the column index of each nonzero in the `vals` array. The `row_ptr` array stores the starting position in the `vals` and `col_id` arrays of the first nonzero element in each row of the sparse matrix. In this paper, we use CSR as our baseline.

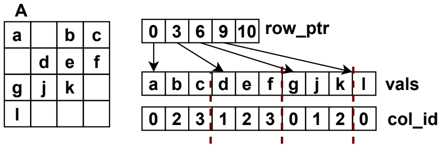


Figure 1: CSR format.

CSR is popular due to its ability to express sparse matrix computations effectively. Algorithm 1 shows the implementation of SpMM with CSR format.

Algorithm 1 Implementation of CSR SpMM (CSR-A).

```
1: for (i=0; i<M; i++) in parallel do
2:     for (e=row_ptr[i]; e<row_ptr[i+1]; e++) unroll ($K$) do
3:         for (j=0; j<N; j++) unroll ($P$) do
4:             C[i][j] = C[i][j] + (vals[e] × B[col_id[e]][j])
5:         end for
6:     end for
7: end for
```

Algorithm 1 iterates over the rows of matrix **A** in parallel (Line 1). For each row, it iterates over its nonzeros in Line 2. Finally, Line 3 iterates over the columns of the dense **B** matrix and updates the values of the **C** matrix.

Although even a straightforward implementation can use the vectorization facilities of current hardware, we can improve SpMM performance by utilizing vector registers and vector units more effectively through code transformations. Algorithm 1 shows two loop unrolling opportunities: (1) unrolling the loop in Line 2 by a factor of K, and (2) unrolling the loop in Line 3 by a factor of P.

By unrolling the loop in Line 2, we can process multiple nonzeros of matrix **A** at the same time. It allows us to keep values of **C** in vector registers, decreasing the number of store operations for **C**. On the other hand, by unrolling the loop in Line 3, we improve the throughput of vector instructions by creating more parallelism across columns of the **B** and **C** matrices. Algorithm 1, which we name as CSR-A version, focuses on reusing the values of **A**. Another approach would be to maximize the reuse for **C**. Algorithm 2, which we name CSR-C, shows how register blocking can be achieved for values of **C**. In CSR-C, P values of row i of **C** are kept in registers while processing all nonzeros of row i of **A**. In the CSR-C version, **C** is kept in vector registers by sacrificing the reuse for **A**.

Algorithm 2 Implementation of CSR SpMM with reuse for **C** (CSR-C).

```
for (i=0; i<M; i++) in parallel do
    for (p=0; p<N; p+=P) do
        for (e=row_ptr[i]; e<row_ptr[i+1]; e++) unroll (K) do
            for (j=p; j<p+P; j++) unroll (P) do
                C[i][j] = C[i][j] + (vals[e] × B[col_id[e]][j])
            end for
        end for
    end for
end for
```

2.2 POWER10 Matrix-Multiply Unit: MMA

Without loss of generality, we target hardware that is capable of executing $\mathbf{A}[0:R-1, 0:K-1] \times \mathbf{B}[0:K-1, 0:P-1]$ dense matrix multiplication (or $R \times K \times P$, for short), either directly or through primitives that can be combined to do so. Such hardware provides two sets of primitives: (1) instructions to move data to/from accumulators used during the matrix multiply operation, and (2) instructions that perform a matrix-multiply or a matrix-multiply and accumulate operation.

Our testbed is a POWER10 system, where an $R \times K \times P$ matrix-multiplication can be implemented with MMA capabilities [32]. MMA provides 8 accumulators, each holding either a 4×4 or a 4×2 matrix of single- or double-precision elements, respectively. Among other operations, MMA can perform outer-products of single- and double-precision vectors, with the results added to an accumulator. Table 1 shows a summary of the MMA instructions.

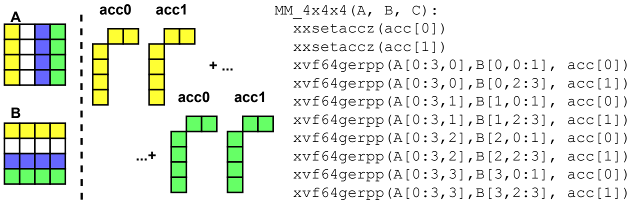
Table 1: Summary of relevant MMA instructions.

Instruction	Description
xxmtacc(acc[k])	Moves values from vector registers to accumulator k
xxmfacc(acc[k])	Moves values from accumulator k to vector registers
xxsetaccz(acc[k])	Set values of accumulator k to zero
xvf32gerpp(x, y, acc[k])	Performs a 4x4 single-precision outer-product with x[0:3] (vector register) and y[0:3] (vector register), and accumulates the result on accumulator k
xvf64gerpp(x, y, acc[k])	Performs a 4x2 double-precision outer-product with x[0:3] (vector register pair) and y[0:1] (vector register), accumulating the result on accumulator k

The **xxmtacc** instruction moves data from four vector registers to an accumulator, while **xxmfacc** does the opposite, and **xxsetaccz** clears an accumulator. The single-precision **xvf32gerpp** instruction performs a 4×4 outer-product of two 4-element vectors in registers x and y and adds the result to the current value in an accumulator. The corresponding double-precision **xvf64gerpp** instruction performs the outer-product of a 4-element vector in register-pair x and a 2-element vector in register y, adding the result to an accumulator.

A (double-precision) matrix multiply is performed as follows. First, we initialize an accumulator by either transferring values from vector registers to the accumulator (**xxmtacc**) or setting all accumulator values to zero (**xxsetaccz**). Then, we perform outer products and accumulate the values of a 4×2 dense matrix by using **xvf64gerpp** instructions. Finally, we use **xxmfacc** to transfer the values from the accumulator to vector registers, which will contain the result of the matrix multiplication [12].

Thanks to their fine-grained design, MMA instructions can implement small dense matrix multiplications with various sizes. As an example, Figure 2 shows an implementation for a double-precision $4 \times 4 \times 4$ matrix-multiplication. Note that we are using two accumulators: acc[0] and acc[1].



```
MM_4x4x4(A, B, C):
   xxsetaccz(acc[0])
   xxsetaccz(acc[1])
   xvf64gerpp(A[0:3,0],B[0,0:1], acc[0])
   xvf64gerpp(A[0:3,0],B[0,2:3], acc[1])
   xvf64gerpp(A[0:3,1],B[1,0:1], acc[0])
   xvf64gerpp(A[0:3,1],B[1,2:3], acc[1])
   xvf64gerpp(A[0:3,2],B[2,0:1], acc[0])
   xvf64gerpp(A[0:3,2],B[2,2:3], acc[1])
   xvf64gerpp(A[0:3,3],B[3,0:1], acc[0])
   xvf64gerpp(A[0:3,3],B[3,2:3], acc[1])
```

Figure 2: A $4 \times 4 \times 4$ matrix-multiplication of $\mathbf{A} \times \mathbf{B}$, and its double-precision code using two 4x2 accumulators. Matrix A is stored in column-major order, matrix B in row-major order, and the output in row-major order.

3 OPTIMIZING SPMM

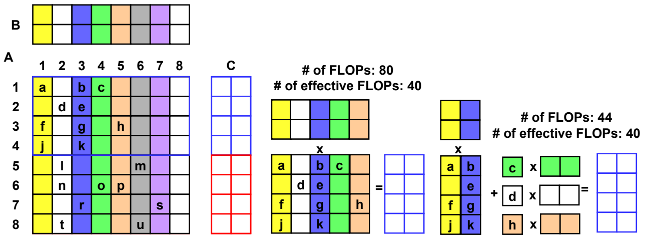
In this section, we describe how we can reformulate SpMM to utilize both matrix and vector units and how our Dense Dynamic Blocks approach can leverage the discrepancy in effective floating-point throughput of matrix and vector units to maximize performance.

Sparse matrices vary significantly in terms of nonzero structure. While some matrices can greatly benefit from utilizing matrix units with dense $R \times 1$ blocks, for others performance can degrade. Therefore, choosing a good SpMM strategy is crucial. In section 3.3, we describe the optimization search space for SpMM in a functional unit-rich processor. Furthermore, we develop *SpMM-OPT*, a machine learning approach with a carefully crafted feature set to select the best SpMM strategy in a functional unit-rich environment. We describe SpMM-OPT in Section 3.4.

3.1 Reformulating SpMM for Matrix-Multiply Units

A matrix-multiply unit is capable of executing an $R \times K \times P$ matrix-multiply operation, where typical dimensions of R, K, and P are 2–16. Furthermore, we expect that a unit can handle variable-sized matrix-multiply operations. In the rest of this section, we use a unit that provides $4 \times 4 \times 2$ matrix-multiply operations and can also handle $4 \times 2 \times 2$, and $4 \times 1 \times 2$.

We present two dynamic blocking approaches that we call DDB-MM and DDB-HYB. As an example, consider Figure 3a. It shows an 8×8 sparse matrix (**A**) and two 8×2 dense matrices (**B** and **C**). Note that **B** is shown transposed to emphasize the access patterns for **B**. The rows of **B** and columns of **A** that will be accessed together are shown with the same color.



(a) Matrices for $C = A \times B$. (b) DDB-MM (c) DDB-HYB

Figure 3: Performing an $8 \times 8 \times 2$ SpMM with 4×1 blocks.

First, we consider the need for finding blocks with consecutive columns. DDB-MM relaxes this requirement by identifying $R \times 1$ blocks from R consecutive rows of the sparse matrix. These $R \times 1$ blocks are then used to create a larger dense block (an $R \times n$ dense matrix from n $R \times 1$ blocks). Figure 3b shows this approach for the first $R = 4$ rows of the sparse matrix **A**. Such an approach can utilize the high floating-point throughput of matrix units. However, DDB-MM can still have drawbacks. If not all $R \times 1$ blocks are dense we will introduce zero padding, which results in underutilization of the floating-point throughput of the processor.

As an example, consider IBM POWER10. A POWER10 core has two pipelined MMA units, each capable of executing 4×2 double-precision outer-product instructions. Let us consider only the case of double-precision. If all the column elements are nonzero, the MMA units of a POWER10 core deliver an aggregate throughput of 32 effective FLOPs/cycle. For the POWER10 MMA, if the columns have 1, 2, or 3 zero elements, the effective throughput falls to 24, 16, and 8 FLOPs, respectively. However, a POWER10 core also has four pipelined vector units, each capable of computing a 1×2 (double-precision) outer-product for an aggregate throughput of 16 effective FLOPs per cycle. Note that, in the vector units, there is no need to process zeros introduced due to padding. Therefore, it only makes sense to process columns with 2 or more nonzero elements in the MMA, where we would obtain 16 or more effective FLOPs per cycle. This suggests that we should process some columns of the sparse matrix in the MMA units and some columns in the vector units. Our hybrid approach (DDB-HYB) takes advantage of this throughput discrepancy.

The aforementioned observations also hold for single precision operations, where MMA units can execute 4×4 outer product instructions and vector units can execute 1×4 outer product instructions.

Figure 3c shows how we can change the matrix-multiply sequence to improve throughput and reduce padding for the first four rows of input matrix **A**. We separate the processing of high and low-density columns. The columns with high density (yellow and blue) use matrix-multiply units, while the remaining columns use vector units. This approach reduces the number of superfluous FLOPs, due to zero padding, significantly. For example, the 4×1 blocking shown in Figure 3b would perform 80 FLOPs, while the DDB-HYB approach would perform 44 FLOPs for the given 4-row block. Note that the block only needs 40 FLOPs. Although DDB-HYB separates a given row-block into a dense and sparse portion, our method doesn't execute distinct SpMM operations. Instead, for a given row-block, the sparse portion is executed right after the dense portion.

Note that DDB-HYB is an intermediate design point between DDB-MM and CSR. If the column-blocks of a given row-block in a sparse matrix are represented in compressed form, then DDB-HYB turns into CSR format.

3.2 Dense Dynamic Blocks: DDB-MM and DDB-HYB

Our Dense Dynamic Blocks (DDB) technique is built on the previous observations. DDB examines the structure of a sparse matrix and finds $R \times 1$ dense blocks, which we call *dense column blocks*. In DDB-MM, each group of R consecutive rows is represented solely as $R \times 1$ dense blocks, independent of the amount of padding necessary. In DDB-HYB, we categorize these dense column blocks in terms of their floating-point throughput potential into high throughput (*htp*) and low throughput (*ltp*). For the POWER10 MMA unit, column blocks with 2 or more nonzeros are categorized as *htp*, while column blocks with a single nonzero are *ltp*. DDB-HYB then splits the sparse matrix into a *blocked* submatrix and a *compressed* submatrix. The former is composed of *htp* dense column blocks, which can leverage the MMA units; the latter is composed of the *ltp* dense column blocks and is represented in a CSR-based format to utilize vector units and maximize floating-point throughput. Finally, DDB-HYB forms large, multi-column dense blocks in the blocked submatrix during execution, based on the capabilities of the matrix-multiply units. Note that CSR is the base case for DDB in the absence of a dense portion. In the rest of this section, we explain the matrix format and SpMM implementation for only DDB-HYB. DDB-MM can be obtained by ignoring the data structures and operations required for the sparse portion of DDB-HYB.

3.2.1 DDB-HYB's Matrix Format. Given an $M \times T$ sparse matrix, DDB-HYB creates $R \times T$ submatrices that contain R consecutive rows each. In each submatrix, it finds the *htp* and *ltp* column blocks. An *htp* column block is represented with a dense $R \times 1$ array. In an *htp* column block, an element not present in the original sparse matrix is represented as a zero (or the annihilator of SpMM). The *htp* column blocks are then placed one next to the other. Since the column numbers of these *htp* column blocks are non-contiguous, we need to record the *ids* of the column blocks. Therefore, like in CSR, we use an `offset` array of size $\lceil M/R \rceil$ to point to the beginning of the *htp* column blocks and column ids from each $R \times T$ submatrix. This is the representation of the *blocked* submatrix obtained from the original sparse matrix. As an example, the left-most part of Figure 4 shows the corresponding representation obtained from matrix **A** given in Figure 3a.

The *ltp* column blocks constitute the *compressed* submatrix obtained from the original sparse matrix and are represented using the CSR format. The rightmost part of Figure 4 shows the corresponding representation obtained from matrix **A**.

3.2.2 SpMM with DDB-HYB. By using our technique, we can cast an SpMM into a set of $R \times K \times P$ matrix multiplications. We propose two algorithms with different objectives: MMA-A (Algorithm 3) and MMA-C (Algorithm 4). The former improves the reuse for

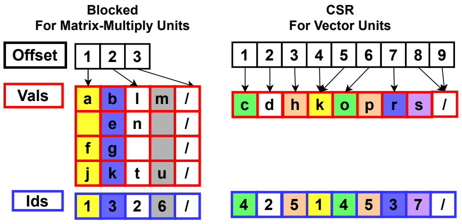


Figure 4: Representing the matrix in Figure 3a with DDB-HYB.

blocks of **A**; the latter minimizes stores to **C** and transfers between accumulators and vector registers.

Algorithm 3 MMA-A: DDB SpMM Improving Block Reuse.

```
 1: procedure SpMM($A, B, C$)
 2:     // $A$: $M \times T$ sparse matrix
 3:     // $B$: $T \times N$ dense matrix, $C$: $M \times N$ dense matrix
 4:     for (r=0; r<M; r+=R) in parallel do
 5:         // Processing blocked portion for rows r to r+R
 6:         for (k=offset[r]; k<offset[r+1]; k+=K) do
 7:             cols = ids[k:k+K] // Array of column ids
 8:             aRxK = vals[k*R:(k+K)*R]
 9:             for (p=0; p<N; p+=P) do
10:                 cRxP = C[r:r+R, p:p+P]
11:                 // B matrix is sliced with column ids
12:                 bKxP = B[cols, p:p+P]
13:                 moveToAcc(cRxP) // Transfer to accumulator
14:                 MM_RxKxP(aRxK, bKxP, cRxP)
15:                 moveFromAcc(cRxP) // Transfer from accumulator
16:                 C[r:r+R, p:p+P] = cRxP
17:             end for // remainder loop omitted
18:         end for // remainder loop omitted
19:         // Processing compressed portion for rows r to r+R with CSR
20:     end for
21: end procedure
```

In both algorithms, we parallelize over row blocks ($R \times T$ submatrices) of sparse matrix **A**. We first process the dense blocks of the corresponding rows (Lines 6-18 for Algorithm 3 and Lines 6-18 for Algorithm 4). Then, in both algorithms, we process the compressed portion of the same set of rows using the CSR representation. Note that both CSR-A (Algorithm 1) and CSR-C (Algorithm 2) can be used for the compressed portion. This part is omitted for brevity.

Algorithm 4 MMA-C: DDB SpMM Minimizing Transfers.

```
 1: procedure SpMM($A, B, C$)
 2:     // $A$: $M \times T$ sparse matrix
 3:     // $B$: $T \times N$ dense matrix, $C$: $M \times N$ dense matrix
 4:     for (r=0; r<M; i+=R) in parallel do
 5:         // Processing blocked portion for rows r to r+R
 6:         for (p=0; p<N; p+=P) do
 7:             cRxP = C[r:r+R, p:p+P]
 8:             moveToAcc(cRxP) // Transfer to accumulator
 9:             for (k=offset[r]; k<offset[r+1]; k+=K) do
10:                 cols = ids[k:k+K] // Array of column ids
11:                 aRxK = vals[k*R:(k+K)*R]
12:                 // B matrix is sliced with column ids
13:                 bKxP = B[cols, p:p+P]
14:                 MM_RxKxP(aRxK, bKxP, cRxP)
15:             end for // remainder loop omitted
16:             moveFromAcc(cRxP) // Transfer from accumulator
17:             C[r:r+R, p:p+P] = cRxP
18:         end for // remainder loop omitted
19:         // Processing compressed portion for rows r to r+R with CSR
20:     end for
21: end procedure
```

In Algorithm 3, for each row block, we retrieve K dense column blocks of R rows of matrix **A**, forming an $R \times K$ dense block (Line 8). We iterate over the columns of the dense matrices, loading the corresponding dense block of **C** (Line 10) and dense dynamic block of **B** (line 12). Since the column ids in matrix **A** are not consecutive, loading the dense block from **B** is a specialized slicing operation. The block of **B** is loaded using the column ids (`cols`). Next, we move the values of **C**'s block to the accumulator and perform an $R \times K \times P$ matrix-multiply. Finally, the values are transferred back from the accumulator, and **C** is updated in memory.

Algorithm 4 is similar to Algorithm 3, except that it first iterates over the columns of **B** and **C**. Therefore, it first loads the block of **C** and transfers it to the accumulator (Lines 7- 8). Note that this approach increases the number of loads for blocks of **A** (Line 11) while minimizing the number of transfers to and from accumulators.

3.2.3 Choosing the Dimensions R, K, and P. The dimensions of the matrix-multiply operation are determined by both the hardware and the structure of the sparse matrix. R is determined by the first dimension of matrix-multiply operation. For instance, a POWER10 core has 4×2 (4×4) double-precision (single-precision) accumulators. Therefore, we choose $R = 4$ for our matrix-multiply kernel and for the number of consecutive rows that we can use to extract dense column blocks. We observe that $R = 4$ matches both the hardware parameters and is a good size to find dense column blocks in sparse matrices, as also noted by previous work [44].

P is mainly driven by the hardware. A POWER10 core has 8 accumulators capable of holding 4×2 (4×4) matrices for double (single) precision floating-point numbers. Therefore, we can process up to 16 (32) columns of **C**. That is, $P = 16$ for double- and $P = 32$ for single-precision.

Finally, K is similar to an unrolling factor. It is driven by two considerations: (1) the capacity of the vector registers and (2) the number of dense column blocks in the blocked portion of the R-row sparse submatrix. Thanks to its dynamic nature, DDB-HYB can handle matrices that have any number of dense column blocks in individual R-row sparse submatrices.

3.3 SpMM Optimization Problem

CSR, DDB-HYB, and DDB-MM lie on a spectrum of optimizations. For example, DDB-MM addresses the needs of matrices with highly regular structures where consecutive rows have similar nonzero structures. In contrast, CSR is needed when we have highly irregular nonzero distributions for consecutive rows of the matrix. And DDB-HYB strikes a balance between CSR and DDB-MM, maximizing floating-point throughput while minimizing zero padding.

In addition to the effective utilization of functional units available in the processor, we need to consider the reuse approach: CSR-A vs. CSR-C for the compressed portion and MMA-A vs. MMA-C for the blocked portion of the sparse matrices. With these reuse approaches, the SpMM execution search space becomes even more complex.

The SpMM performance is also affected by the cache behavior. The nonzero structure of the sparse matrix and the scheduling of rows and row-blocks to threads affect the cache behavior. For example, if there are many common column ids among the rows or row-blocks, we can observe temporal locality for the dense input matrix. This locality is limited by the number of columns in the dense input and output matrices due to L1 and L2 cache sizes. For

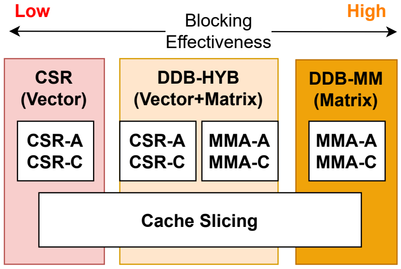


Figure 5: SpMM optimization space.

instance, a sparse matrix may deliver high floating-point throughput when multiplied by a dense matrix with 32 columns. However, throughput can significantly decrease if the number of columns in the dense matrix is increased to 64 due to limited cache capacity. This is where *cache slicing becomes effective*. By slicing the input (**B**) and output (**C**) dense matrices into blocks of columns, we can reduce the memory footprint of the two dense matrices during a single iteration and utilize caches more effectively. As a result, we can do multiple passes over the sparse matrix with a higher floating-point throughput, maximizing the overall performance.

Figure 5 shows a summary of the optimization space for SpMM. Next, we discuss the details of SpMM-OPT, which can navigate this complex space to identify the best SpMM strategy for a given ⟨sparse matrix, dense matrix⟩ pair.

3.4 SpMM-OPT: An ML Approach for Method Selection

SpMM-OPT finds an SpMM execution strategy for a given pair ⟨sparse matrix, dense matrix⟩. The strategy is defined by a tuple ⟨FU_St, Blocked_St, Compressed_St, SF⟩. FU_St is the strategy for utilizing the functional units (either CSR, DDB-MM, or DDB-HYB). Blocked_St is the reuse approach to process the blocked portion of the sparse input matrix. It is only applicable for the DDB-MM and DDB-HYB approaches, and it can have values in {MMA-A, MMA-C}. Compressed_St is the reuse strategy for the compressed portion and is used for the CSR and DDB-HYB approaches. The possible values of Compressed_St are in {CSR-A, CSR-C}. SF is the slicing factor, which is the number of columns in the blocks of the dense matrices.

SpMM-OPT works as follows. We detect the potential of matrices by using the *Average Floating Point Throughput* (AFT) metric. AFT categorizes each matrix into high AFT (HAFT) and low AFT (LAFT) categories. HAFT matrices have a higher potential to benefit from the hybrid or matrix unit-oriented techniques DDB-HYB and DDB-MM. In contrast, LAFT matrices prefer the CSR method.

SpMM-OPT trains separate machine learning models for HAFT and LAFT matrices with different numbers of columns in the dense matrix. In this paper, we consider the number of columns in the dense matrix to be equal to $D \in \{16, 32, 64, 128, 256\}$, each corresponding to a power of two cache lines for double and single-precision values. In total, SpMM-OPT trains 20 different ML models for double-precision (DP) and single-precision (SP) SpMM operations: $\{\mathtt{SP}, \mathtt{DP}\} \times \{\mathtt{HAFT}, \mathtt{LAFT}\} \times \{16, 32, 64, 128, 256\}$. Since $\mathtt{SF} \leq D$, each model has a different number of classes. For example, if $D = 64$ we can only have slice sizes $\mathtt{SF} \in \{16, 32, 64\}$. The given ⟨sparse matrix, dense matrix⟩ pair is run through the corresponding ML model, obtaining a ⟨FU_St, Blocked_St, Compressed_St, SF⟩ tuple for an efficient SpMM strategy. Each of these ML models is a Support Vector Machine (SVM) for which we optimized the parameters with a grid search.

3.4.1 *Categorizing Sparse Matrices with AFT.* Sparse matrices show widely different characteristics in terms of the distribution of nonzeros. The AFT metric estimates the potential FLOPs throughput of the sparse matrix by considering the distribution of the number of nonzeros in the dense column blocks.

To compute AFT, we first create a density histogram H with R elements. H has R elements because a single column block can have from 1 to R nonzeros. A single entry (H_i) in the density histogram gives the percentage of nonzeros that reside in column blocks with density level i (i.e., with a number of nonzeros is equal to i), where $1 \leq i \leq R$. Using the density histogram, we calculate the AFT as $AFT = \sum_{i=1}^{R} H_i * t_i$, where t_i is the effective FLOP throughput that we can achieve when a column block has i nonzeros. For example, with POWER10 MMA units, we can have up to 4 elements in a single column block ($R = 4$).

For double-precision floating-point values, we use the MMA units' outer-product instructions for column blocks with 2-4 nonzeros in them, which can achieve an effective FLOP throughput of $t_2 = 16$, $t_3 = 24$, and $t_4 = 32$ FLOPs/cycle, respectively. On the other hand, we use vector instructions for the column blocks with a single element, with a throughput $t_1 = 16$ FLOPs/cycle. For single-precision, the throughput is doubled.

If the AFT is above a certain threshold, we classify matrices as HAFT. Otherwise, we classify them as LAFT. We chose this threshold experimentally to be 20 by performing a sensitivity analysis.

In addition to the H distribution, we also calculate the distribution T of dense column blocks with varying densities. In this case, T_i gives us the fraction of dense column blocks with i nonzeros among all dense column blocks. T_i is calculated as follows: $T_i = \frac{\textit{\# dense column blocks with i nnzs}}{\textit{\# of all dense column blocks}}$.

3.4.2 *System Features.* We categorize the features of the ML system into three categories: (1) size, (2) blocking, and (3) locality features.

(1) Size Features. These features model the size characteristics of a given sparse matrix. We use the number of rows and the number of nonzeros in the matrix.

(2) Blocking Features. These features model the blocking behavior. As described in the AFT calculations (Section 3.4.1), we first find the density of each 4×1 block of the matrix by considering 4-row blocks. Each element of the H and T distributions becomes a parameter in the SpMM-OPT.

(3) Locality Features. We consider two sets of locality features: $\mathtt{uniq4}_X$ and $\mathtt{uniq256}_X$. To calculate $\mathtt{uniq4}_X$, we logically divide the sparse matrix **A** into submatrices of 4 rows and X columns. Then, we count the number of such submatrices of **A** that have at least one nonzero, and divide the count by the total number of nonzeros in **A**. This metric gives us a measure the locality of the nonzeros: if $\mathtt{uniq4}_X$ is close to 1, the memory accesses to the input dense matrix **B** potentially have low locality. On the other hand, if

uniq4$_X$ is close to 0, the memory accesses potentially have high locality. uniq256$_X$ is computed similarly for submatrices of 256 rows and X columns. The uniq256$_X$ metric is used to estimate the potential locality that comes from scheduling a chunk of rows to a single thread. We experimentally choose 256 as our scheduling parameter—i.e., 256 rows of the sparse matrix are scheduled to the same thread together. In both uniq4$_X$ and uniq256$_X$, we use $X = \{1, 4, 8, 16, 32, 64\}$ values to capture different characteristics.

In addition to the uniq4$_X$ and uniq256$_X$ features, we also use, as indicators of cache capacity utilization: the average number of nonzeros per row in CSR, the average number of nonzeros per row for the compressed portion of DDB-HYB, the average number of column-blocks per row-block in DDB-MM, and the average number of column-blocks per row-block in the dense portion of DDB-HYB. Table 2 summarizes the features we use and their names.

Table 2: Summary of Features.

Feature	Description
nrows, nvals	Encode the size characteristics of the matrix
H_i	The distribution of the nonzeros to dense column blocks
T_i	Density distribution of dense column blocks
aveBlked$_{\text{DDB-HYB}}$	Average number of column-blocks per row-block in the dense portion of DDB-HYB
aveComped$_{\text{DDB-HYB}}$	Average number of nonzeros per row in the compressed portion of DDB-HYB
ave$_{\text{CSR}}$	Average number of nonzeros per row in CSR
ave$_{\text{DDB-MM}}$	Average number of column-blocks per row-block in DDB-MM
uniq4$_X$	Unique access ratio due to blocking
uniq256$_X$	Unique access ratio due to scheduling

3.4.3 Normalization of features.

Since we use SVMs as our ML mechanism, we need to normalize the features to the $[0, 1]$ range. Four of our features are already ratios normalized to this range: uniq4$_X$, uniq256$_X$, H_i, and T_i.

The aveBlked$_{\text{DDB-HYB}}$, aveComped$_{\text{DDB-HYB}}$, ave$_{\text{CSR}}$, and ave$_{\text{DDB-MM}}$ features can have significantly different values depending on the structure of the matrix. For these variables, we apply one-hot encoding with small changes. First, we create a 9-bit one hot encoded representation of these variables by considering the ranges: $[0, 2)$, $[2, 4)$, $[4, 8)$, $[8, 16)$, $[16, 32)$, $[32, 64)$, $[64, 128)$, $[128, 256)$, $[256,)$. If the value of the feature falls under one of these ranges, the corresponding bit is set to 1. Additionally, for each of these nine ranges, we create a *magnitude* variable whose value is the normalized value of the metric in that particular range. For example, if the average number of nonzeros in the CSR format is 6, then the 3^{rd} bit of the 9-bit variable will be set. Moreover, the magnitude variable of the $[4 - 8)$ range will have a value of 0.5. The number of rows and number of nonzeros are normalized by considering the minimum and maximum values observed in the training and test sets.

4 ESTABLISHING CSR BASELINE

We compare the performance of our DDB technique against a CSR baseline on IBM's POWER10 processor. We establish the quality of that CSR baseline by comparing it against SpMM routines in the well-known production library MKL [4] on an Intel Xeon Platinum 8268 platform [3]. We use the Intel Compiler v2020 for compilation and MKL v2020. The corresponding math library for POWER10 (ESSL) does not have SpMM routines. That is why we use this indirect approach to establish the quality of our baseline.

We emphasize that MKL and the Xeon 8268 are used solely to establish the quality of our CSR implementation. That implementation is then tuned for POWER10 and used as the baseline in experimental results for POWER10. Our results always compare our method on POWER10 to that tuned CSR baseline on POWER10.

We test our CSR implementation with different unrolling parameters —as described in Algorithms 1 and 2—and various numbers of columns in the dense matrices. Details of datasets and unrolling parameters are found in Section 5. Both MKL and CSR are tested with 24 threads (a single socket). Each thread is bound to a physical core using *numactl*, and memory is allocated in the local node. For these experiments, we use 50 matrices including matrices used in previous work [30].

Table 3 reports, for each precision and method, the average execution time normalized to the fastest version for each matrix. The best possible value is 1. For both double-precision (DP) and single-precision (SP) SpMM, it can be shown that there is at least one tuned implementation of CSR-A and CSR-C that is significantly faster than MKL.

Table 3: Average of normalized execution times for MKL and CSR implementations for double- (DP), and single-precision (SP) SpMM on an Intel processor. The values are normalized to the fastest execution time observed for each matrix. Lower is better.

Columns		16	32	64	128	256
DP	***CSR-A***	1.11	1.10	1.08	1.06	1.04
	CSR-C	1.05	1.05	1.06	1.08	1.09
	MKL	1.21	1.19	1.23	1.25	1.30
SP	***CSR-A***	1.17	1.13	1.11	1.08	1.08
	CSR-C	1.07	1.06	1.07	1.07	1.11
	MKL	1.24	1.21	1.18	1.24	1.27

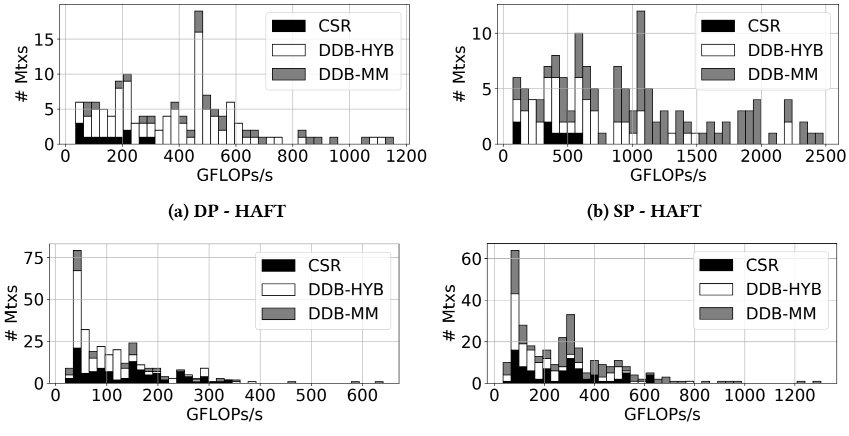
5 EXPERIMENTAL SETUP

System Setup. Our POWER10 system has one single-chip module (SCM) with 15 SMT8 cores, equivalent to 30 SMT4 cores [41]. Each SMT4 core has 32 KB private L1 and 1MB private L2 caches. Also associated with each SMT4 core there is a 4 MB local component of the L3 cache (LLC). POWER10 supports PowerISA 3.1, which includes Vector-Scalar Extensions (VSX) and Matrix-Multiply Assist (MMA) facilities. The system memory is 485 GB. We used IBM Advance Toolchain for Linux on Power Systems version 15.0.0-alpha (GCC 11.0.0), which provides compiler built-ins for MMA and VSX instructions. Details of our system are summarized in Table 4.

Table 4: POWER10 system setup.

System	Component	Properties
POWER10	***CPU***	30 SMT4 cores; 32 KB private L1, 1 MB private L2, 4 MB local L3 per core
	Memory	485 GB
	ISA	PowerISA 3.1, MMA, VSX
	Software	Advance Toolchain 15.0.0-alpha (GCC 11.0.0), OS Kernel: 4.18.0-277.el8.ppc64le

Input Matrices. We test 440 matrices from the Sparse Suite [17] matrix collection. We select the matrices used in previous work [30]. Additionally, we test matrices with 100 thousand to 10 million rows, to make sure that matrices are large enough to give consistent performance results and small enough to fit into memory when we have a large number of columns in the dense matrices.



(a) DP - HAFT

(b) SP - HAFT

(c) DP - LAFT

(d) SP - LAFT

Figure 6: Floating-point throughput distribution of HAFT and LAFT matrices.

Experiments. We always use the best performing version of CSR as our baseline implementation (CSR-A and CSR-C without slicing). For DDB-MM, we use both MMA-A and MMA-C versions. For DDB-HYB, we use MMA-A and MMA-C for the blocked (i.e., dense) portion, and CSR-A and CSR-C for the compressed portion, hence obtaining four different implementations of DDB-HYB.

All our versions are implemented in C++ using OpenMP. We have manually vectorized all implementations by using compiler built-ins for POWER10. Similarly, for MMA-A and MMA-C, we have used the built-ins provided for MMA instructions and enabled *-O3* optimizations.

We test both double-precision (64-bit) and single-precision (32-bit) floating-point arithmetic. We repeat each experiment for 110 iterations. We ignore the first 10 iterations as warm-up and report the average execution time of the last 100 iterations.

Choosing SVM parameters and Testing SpMM-OPT. We use Scikit-Learn to train and test our ML models in our experiments [35]. We use a Support Vector Machine (SVM) classifier with linear kernel for our individual ML models (LinearSVC in Scikit-Learn). A linear SVM kernel has a regularization parameter (c) that needs to be tuned. In order to tune c, we use a grid search approach. We test $\{0, 1, 10, 100, 1000, 10000\}$ values for the c parameter. To select the best c parameter, we have considered the average speedup observed with respect to the baseline CSR implementation (best of CSR-A and CSR-C without slicing). In the end, we choose a single $c = 1$ parameter to use in all ML models. The speedups from using SVMs generated with $c = 1$ are always in the 10% neighborhood of the maximum average speedup observed for any values in the grid search. While testing SpMM-OPT, we use the 10-fold approach and report the aggregate results. The 10-fold approach is commonly used to create training and test sets from a single data set. It creates 10 disjoint training and test set pairs that include 90% and 10% of all samples in training and test sets, respectively.

6 EXPERIMENTAL RESULTS

In this section, we analyze the effectiveness of various DDB techniques and the effectiveness of SpMM-OPT.

6.1 Performance of Blocking with MMA Facilities

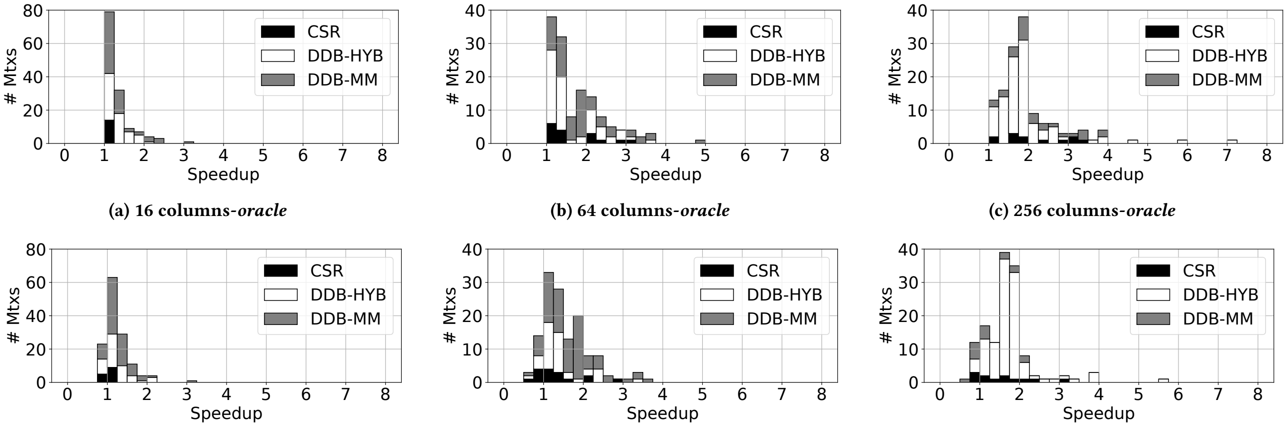
First, we analyze the performance potential of DDB-MM and DDB-HYB. We test all $\langle \texttt{method}, \texttt{slicing} \rangle$ pairs against each sparse matrix and number of columns in the dense matrices. We measure the FLOPs/s (single- and double-precision) for each case.

Figure 6 shows the distribution of the highest FLOPs/s that can be obtained for each matrix with our SpMM implementations for double- and single-precision operations. In each of the figures, each column is a performance bin and the y-axis is the number of matrices for which the best observed FLOPs/s falls on that bin. The performance bins also show the breakdown of the SpMM methods achieving the highest floating-point throughput for each matrix.

We observe that the maximum FLOPs/s for our matrices are 1.15 TFLOPs/s for double-precision and 2.5 TFLOPs/s for single-precision computations. These maximums are achieved by utilizing the MMA units with the DDB-MM method. The maximum performance achieved by the DDB-MM method (with double-precision) is approximately 40% of the 2.9 TFLOPs/s observed in the LINPACK benchmark on a POWER10 single-chip module.

Furthermore, it can be shown that MMA units are needed to achieve more than 608 GFLOPs/s with double-precision values and more than 1023 GFLOPs/s with single precision values. The highest performance achieved by utilizing MMA units is approximately 2× higher than the highest performance that can be obtained by only using the vector units.

As expected, for HAFT matrices, we observe the highest floating-point throughput with the DDB-MM and DDB-HYB methods. For double-precision, we see that DDB-HYB is the fastest method for the majority of the matrices. For single-precision, we see that DDB-MM is more commonly the fastest, because single-precision SpMM has a higher FLOPs/byte ratio. MMA techniques are also useful for LAFT matrices. In DDB-HYB, a portion of the matrix is represented in CSR. However, even a slight improvement in the dense portion has a significant overall impact on performance. DDB-HYB is the fastest method for 247 of the matrices with double-precision SpMM. DDB-MM is the fastest method for 211 matrices for single-precision SpMM.



(a) 16 columns-*oracle*

(b) 64 columns-*oracle*

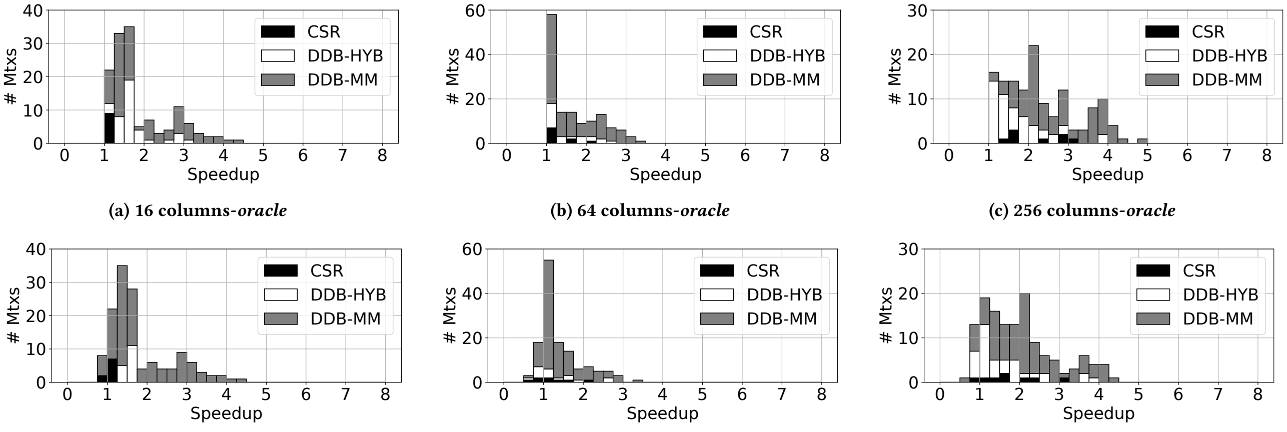
(c) 256 columns-*oracle*

(d) 16 columns-SpMM-OPT

(e) 64 columns-SpMM-OPT

(f) 256 columns-SpMM-OPT

Figure 7: Speedup for HAFT matrices (double-precision).



(a) 16 columns-*oracle*

(b) 64 columns-*oracle*

(c) 256 columns-*oracle*

(d) 16 columns-SpMM-OPT

(e) 64 columns-SpMM-OPT

(f) 256 columns-SpMM-OPT

Figure 8: Speedup for HAFT matrices (single-precision).

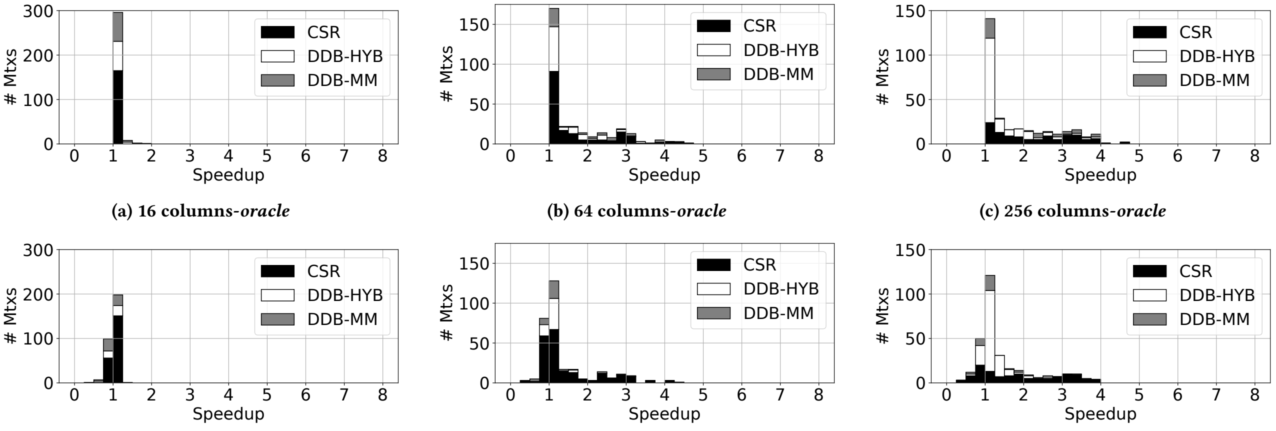
Finally, we observe that slicing has a positive performance potential. For example, 147, 68, 19, and 19 of the 440 matrices achieve the highest FLOPs/s with 16-, 32-, 64-, and 128-column dense matrices, respectively, for double-precision SpMM. For single-precision SpMM, we observed that 116, 106, 82, and 18 of the 440 matrices obtain the highest throughput with 16-, 32-, 64-, and 128-column dense matrices, respectively. 256-column dense matrices achieve the top throughput in the remaining cases (187 cases for double-precision and 118 cases for single-precision).

6.2 Performance of SpMM-OPT

To analyze the effectiveness of SpMM-OPT, we compare its performance to an *oracle* method that selects the best SpMM strategy by exhaustively searching all possibilities. Figures 7-10 show the distribution of the speedups with *oracle* and SpMM-OPT. The speedups are calculated by normalizing the floating-point throughput of the selected SpMM strategy with *oracle* or SpMM-OPT to the floating-point throughput of the CSR implementation without slicing. In these figures, each bar represents a 0.25 range. The figures show the speedups for dense matrices **B** and **C** of different numbers of columns (16, 64, and 256 columns).

6.2.1 Prediction for HAFT matrices. Figure 7 shows the speedup distribution for HAFT matrices with double-precision SpMM with *oracle* (top row) and SpMM-OPT (bottom row). We observe that *oracle* achieves an average speedup of 1.31, 1.76, and 2.02× for dense matrices of 16, 64, and 256 columns, respectively. SpMM-OPT attains 1.25, 1.55, and 1.70× for dense matrices of 16, 64, and 256 columns, respectively. For single-precision SpMM (Figure 8), the average speedups that we observe are higher. The average speedups of *oracle* for the same set of columns in the dense matrices are 1.85, 1.66, and 2.34× while SpMM-OPT attains 1.79, 1.41, and 1.99×, respectively. As observed in the previous section, the reason behind this is the higher FLOP density per byte in single-precision cases.

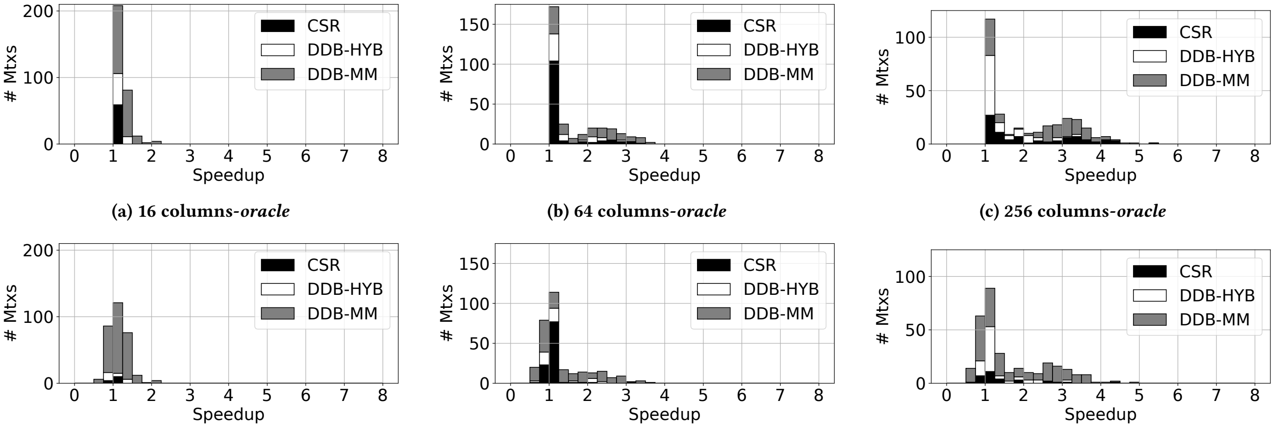
We observe that SpMM-OPT is successful at predicting the SpMM method and slicing factor—therefore attaining speedups close to those of *oracle*. Although there are matrices in the speedup range of $0.75 - 1.0$ (i.e., showing no speedup over CSR), generally, these are matrices with a speedup $\geq 0.9\times$. One reason behind the differences between *oracle* and SpMM-OPT is that SpMM-OPT cannot identify all extremes, such as cases with very high speedup values found by *oracle*.



(a) 16 columns-*oracle* **(b) 64 columns-*oracle*** **(c) 256 columns-*oracle***

(d) 16 columns-SpMM-OPT **(e) 64 columns-SpMM-OPT** **(f) 256 columns-SpMM-OPT**

Figure 9: Speedup for LAFT matrices (double-precision).



(a) 16 columns-*oracle* **(b) 64 columns-*oracle*** **(c) 256 columns-*oracle***

(d) 16 columns-SpMM-OPT **(e) 64 columns-SpMM-OPT** **(f) 256 columns-SpMM-OPT**

Figure 10: Speedup for LAFT matrices (single-precision).

With HAFT matrices, we observe that DDB-HYB and DDB-MM methods are frequently selected. For example, one of DDB-HYB and DDB-MM is selected for more than 80% of the matrices. The remaining cases with speedups over 1 are attained by CSR with slicing.

6.2.2 *Prediction for LAFT matrices.*

Figures 9 and 10 show the speedup distributions for the LAFT matrices for double and single-precision SpMM, respectively. For LAFT matrices, we expected to see that optimized CSR would be the best method for most matrices. However, we see that DDB-HYB and DDB-MM are selected for a significant number of LAFT matrices as well.

With SpMM-OPT, many matrices fall under the $0.75 - 1.00\times$ speedup range. Similar to HAFT matrices, these generally show $\geq 0.9\times$ speedups. Thus, the average performance is not significantly affected. In the worst case, with dense matrices of 16 columns where slicing is not applicable, we see an SpMM-OPT average speedup of 0.98× for double-precision SpMM (with a harmonic mean of 0.97×). DDB-HYB generally behaves as the optimized CSR since matrices do not have many dense blocks.

For the single-precision case, the average speedup of SpMM-OPT for dense matrices of 16 columns is 1.14× (with a harmonic mean of 1.10×). For single-precision matrices, we also observe that DDB-MM gives a performance boost in many cases.

6.2.3 *Effect of Slicing.*

For both HAFT and LAFT matrices, cache slicing improves the performance significantly. Table 5 shows the percentage of matrices that select different slicing parameters (i.e., slice size equal to 16, 32, 64, 128, or 256 columns) for a given number of columns in the dense matrix (16, 32, 64, 128, or 256 columns). The slice size has to be equal or smaller than the number of columns. Each row of the table is for a given number of columns in the dense matrix, while each column is for a slice size.

From the table, we see that *oracle* very often chooses to use multiple slices. Therefore, slicing is needed. For example, in HAFT matrices, for double-precision SpMM under *oracle*, 85.2%, 64.4%, and 53.3% of the matrices benefit from slicing for 64, 128, and 256 columns, respectively. Furthermore, SpMM-OPT often captures these best slicing parameters. The same behavior is seen for single-precision SpMM and for LAFT matrices.

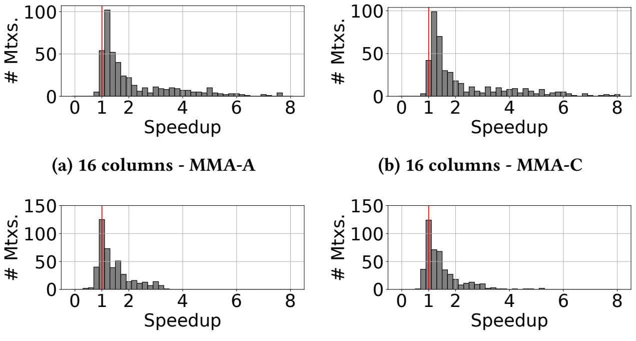
Table 5: Slicing parameters selected by *oracle* and SpMM-OPT for HAFT and LAFT matrices. Entries show the percentage of cases with a given slice size (shown in table columns) for a given number of columns in the dense matrix (shown in table rows). Entries in a row add to 100%.

	HAFT										LAFT									
	Slice Size for *oracle*					Slice Size for SpMM-OPT					Slice Size for *oracle*					Slice Size for SpMM-OPT				
DP	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
16 Cols	100	-	-	-	-	100	-	-	-	-	100	-	-	-	-	100	-	-	-	-
32 Cols	29.6	70.4	-	-	-	23.0	77.0	-	-	-	45.6	54.4	-	-	-	37.6	62.4	-	-	-
64 Cols	29.6	55.6	14.8	-	-	31.1	64.4	4.4	-	-	44.0	17.9	38.1	-	-	51.3	8.8	39.9	-	-
128 Cols	27.4	34.1	3.0	35.6	-	19.3	29.6	0.7	50.4	-	38.4	13.7	7.2	40.7	-	51.0	7.2	1.6	40.2	-
256 Cols	24.4	22.2	2.2	4.4	46.7	15.6	21.5	0.0	3.0	60.0	36.8	12.4	5.2	4.2	41.4	43.5	5.9	1.6	2.9	46.1
SP	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
16 Cols	100	-	-	-	-	100	-	-	-	-	100	-	-	-	-	100	-	-	-	-
32 Cols	24.4	75.6	-	-	-	8.1	91.9	-	-	-	32.2	67.8	-	-	-	17.0	83.0	-	-	-
64 Cols	21.5	34.8	43.7	-	-	14.1	51.9	34.1	-	-	30.3	25.4	44.3	-	-	32.4	24.8	42.8	-	-
128 Cols	21.5	32.6	36.3	9.6	-	17.0	48.1	29.6	5.2	-	29.0	23.1	16.6	31.3	0.0	34.3	20.9	10.1	34.6	-
256 Cols	20.7	28.9	27.4	2.2	20.7	17.8	43.0	25.9	0.0	13.3	28.7	21.5	14.7	4.9	30.3	40.8	23.5	8.2	1.0	26.5

6.2.4 *DDB-MM vs. Rectangular Blocks.*

We compare the DDB-MM approach to using rectangular 4 × 4 blocks. Figure 11 shows the distribution of the speedups obtained by using DDB-MM for dense matrices of 16 and 64 columns with double-precision values with respect to using rectangular blocks. We observe that DDB-MM achieves an average speedup of 2.21× and 2.35× with MMA-A and MMA-C methods with dense matrices of 16 columns. The average speedups with dense matrices of 64 columns are 1.45× and 1.48× for MMA-A and MMA-C methods, respectively. In the vast majority of cases, it is best not to use 4 × 4 blocks.

The histograms in Figure 11 cover all the 440 matrices in our suite. It can be shown that rectangular 4 × 4 blocks perform relatively better for HAFT matrices than for LAFT matrices. For a few matrices that naturally have 4×4 blocks, DDB-MM and rectangular blocks obtain similar performance. However, on average, DDB-MM outperforms 4 × 4 blocks by 1.49× and 1.08× for 16 and 64 columns, respectively, for HAFT matrices.



(a) 16 columns - MMA-A

(b) 16 columns - MMA-C

(c) 64 columns - MMA-A

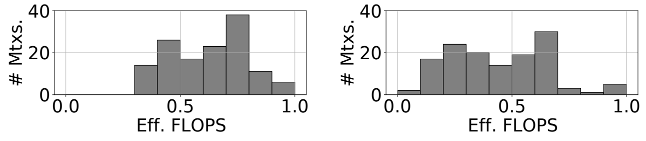
(d) 64 columns - MMA-C

Figure 11: Speedup of DDB-MM with respect to using rectangular 4 × 4 blocks with MMA-A and MMA-C techniques.

The main reason why using rectangular blocks is not effective is because it introduces superfluous FLOPs. Figure 12 shows the ratio of effective FLOPs for HAFT matrices when DDB-MM and 4 × 4 rectangular blocks are used. The figure shows how many matrices have a given effective FLOPs ratio.

Using DDB-MM significantly increases the effective FLOPs ratio compared to 4 × 4 blocks. With 4 × 4 blocks, almost half of the matrices have less than 50% effective FLOPs ratio—i.e., 50% or more of FLOPs executed do not contribute to the final result. In these cases, MMA matrix units operate inefficiently.

On the other hand, with DDB-MM, only 30% of matrices have less than 50% effective FLOPs ratio. Moreover, only 10% of all the matrices have an effective FLOPs ratio of 40% or less. Such matrices with low effective FLOPs ratio can be executed efficiently by DDB-HYB or CSR, and SpMM-OPT can select a suitable strategy to use.



(a) Effective FLOPs with DDB-MM

(b) Effective FLOPs with 4 × 4 Blocks

Figure 12: Distribution of the effective FLOPs ratio for (a) DDB-MM and (b) 4×4 rectangular blocks for HAFT matrices.

7 RELATED WORK

SpMM is optimized for both CPU and GPU systems. In the CPU domain, the main concern is to improve the locality of memory references. For example, Compressed Sparse Blocks [9, 14] improves performance by tiling for caches, while ApST [20] augments the CSR representation to improve locality. Additionally, [30] proposes a formulation to find the best tiling parameters for registers and caches.

SpMM optimizations for GPUs generally target load balancing issues and efficient use of cache capacity in GPUs [22, 49]. These techniques include methods to manage warp level parallelism and to improve memory coalescing. Moreover, since SpMM operations are important in training and inference of Graph Neural Networks (GNNs), several other optimizations have been proposed for GPUs in the GNN domain [21, 23], including slicing and locality-aware scheduling.

Blocking for matrix-multiply units for SpMM and for multiplication of two sparse matrices has been explored in the GPU domain with the BCSR approach for Tensor Cores [48, 54]. Our work is different in that we have a dynamic structure and we target MMA units with finer-granularity outer-product operations.

Blocking techniques have been heavily explored for SpMV [26, 27, 36, 38, 43, 44]. While BCSR is limited by having static block sizes, Unaligned BCSR [44] and VBR [38] improve performance by introducing dynamically-sized blocks. Blocking for SpMV improves performance by eliminating irregular accesses, while potentially suffering from zero-padding. Our difference from these methods is that our goal is to utilize matrix and vector units synergistically to improve FLOP throughput, rather than to optimize padding or improve irregular access patterns. Instead of finding rectangular blocks, we find column blocks that are dynamically grouped to create a block for MMA units.

Previous vectorization techniques for SpMV, such as ELLPACK and Sell-c-σ also create a compact representation by grouping consecutive rows of a matrix [16, 19, 29, 50]. However, these techniques do not identify the common columns across the consecutive rows and only focus on minimizing padding. For example, the dense column blocks identified by Sell-c-σ would have multiple column ids appearing in the same dense-column block.

Previous work has evaluated auto-tuning approaches [15, 24, 43]. Recently, some work [21, 30] has proposed to use slicing to improve SpMM performance. It uses a model-based mechanism to tune the slicing parameter. Our work, on the other hand, also targets the effective utilization of functional units. Previous works have also utilized ML for performance prediction of sparse matrix computations [7, 31, 39, 51]. Their aim is to select an optimal matrix format to execute SpMV efficiently. SpMM-OPT differs from previous work in multiple aspects. For example, instead of only choosing a format, SpMM-OPT makes many decisions simultaneously in a comprehensive optimization framework.

Other examples also exist to leverage ML to predict performance for other primitives, such as sparse matrix sparse matrix multiplication [47]. Furthermore, [42, 53] leverage ML techniques in an algorithm selection setting. In [42], the authors design ML techniques to choose efficient execution strategies for several primitive operations such as dense matrix-multiply operations and sorting. On the other hand, [53] proposes an ML framework to choose a parallel reduction technique among the mechanisms described in [52]. [53] can be effective for the SpMV primitive, which has parallel irregular reduction operations.

Although we do not explore reordering the sparse matrix in this paper, we can use sparse matrix reordering [8, 11, 13, 19, 25, 28, 36, 46] to improve the performance of our mechanisms. DDB uses consecutive rows while relaxing the order of columns. If we can inexpensively find similar rows and group them together, we can further improve the performance of our techniques.

8 CONCLUSIONS

In this paper, we introduced Dense Dynamic Blocks (DDB), a new technique that executes SpMM on matrix-multiply units and vector units to improve performance. DDB enables a matrix unit-oriented strategy via DDB-MM or a hybrid strategy via DDB-HYB. However, we observed that not all sparse matrices can utilize matrix units efficiently, and thus, benefit equally from DDB. Moreover, the cache behavior of SpMM for a given sparse matrix also affects the performance significantly. As a result, we have designed a machine learning method called SpMM-OPT that can navigate this complex search space. SpMM-OPT identifies the best SpMM strategy for a given sparse matrix and dense matrix pair. SpMM-OPT selects among vector unit-oriented, matrix unit-oriented, and hybrid strategies to attain the highest floating-point throughput while also taking into account cache optimizations.

We tested DDB and SpMM-OPT with matrices from the well-known SuiteSparse matrix collection on a POWER10 system with vector and matrix-multiply units. We observed that DDB-MM and DDB-HYB can achieve a floating-point throughput of up to 1.1 and 2.5 TFLOPs/s for double- and single-precision SpMM, respectively. SpMM-OPT was able to effectively choose high-performance SpMM methods, achieving an average speedup of up to 2× compared to an optimized CSR baseline.

ACKNOWLEDGMENTS

This work was supported in part by the IBM-Illinois Discovery Accelerator Institute, and by NSF under grants CNS-1763658, CNS-1956007, CCF-2028861, and CCF-2107470.

REFERENCES

[1] 2015. Intel Math Kernel Library Inspector-executor Sparse BLAS Routines. https://software.intel.com/en-us/articles/intel-math-kernel-library-inspector-executor-sparse-blas-routines

[2] 2017. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. [Online; accessed 02-April-2021].

[3] 2019. Intel Xeon Platinum 8268 Processor. https://ark.intel.com/content/www/us/en/ark/products/192481/intel-xeon-platinum-8268-processor-35-75m-cache-2-90-ghz.html. [Online; accessed 02-April-2021].

[4] 2020. Intel oneAPI Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/comp-onents/onemkl

[5] 2020. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf. [Online; accessed 02-April-2021].

[6] 2021. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html. [Online; accessed 02-April-2021].

[7] Walid A. Abu-Sufah and Asma Abdel Karim. 2013. Auto-tuning of Sparse Matrix-Vector Multiplication on Graphics Processors. In *Supercomputing - 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7905)*, Julian M. Kunkel, Thomas Ludwig, and Hans Werner Meuer (Eds.). Springer, 151–164. https://doi.org/10.1007/978-3-642-38750-0_12

[8] Ramesh C. Agarwal, Fred G. Gustavson, and Mohammad Zubair. 1992. A High Performance Algorithm Using Pre-Processing for the Sparse Matrix-Vector Multiplication. In *Proceedings Supercomputing '92, Minneapolis, MN, USA, November 16-20, 1992*, Robert Werner (Ed.). IEEE Computer Society, 32–41. https://doi.org/10.1109/SUPERC.1992.236712

[9] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 1213–1222. https://doi.org/10.1109/IPDPS.2014.125

[10] Hartwig Anzt, Stanimire Tomov, and Jack J. Dongarra. 2015. Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product. In *Proceedings of the Symposium on High Performance Computing, HPC 2015, part of the 2015 Spring Simulation Multiconference, SpringSim '15, Alexandria, VA, USA, April 12-15, 2015*, Layne T. Watson, Josef Weinbub, Masha Sosonkina, and William I. Thacker (Eds.). SCS/ACM, 75–82. http://dl.acm.org/citation.cfm?id=2872609

[11] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 22–31. https://doi.org/10.1109/IPDPS.2016.110

[12] Puneeth Bhat, Jose Moreira, and Satish Kumar Sadasivam. 2021. Matrix-Multiply Assist (MMA) Best Practices Guide. https://www.redbooks.ibm.com/redpieces/pdfs/redp5612.pdf. [Online; accessed 02-April-2021].

[13] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, Sadagopan

Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM, 587–596. https://doi.org/10.1145/1963405.1963488

[14] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA ’09)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[15] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. 2007. Performance Optimization and Modeling of Blocked Sparse Kernels. *Int. J. High Perform. Comput. Appl.* 21, 4 (Nov. 2007), 467–484. https://doi.org/10.1177/1094342007083801

[16] Jee Whan Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM, 115–126. https://doi.org/10.1145/1693453.1693471

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

[18] Zhangxiaowen Gong, Houxiang Ji, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2020. SparseTrain: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT ’20)*. Association for Computing Machinery, New York, NY, USA, 279–292. https://doi.org/10.1145/3410463.3414655

[19] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V. Çatalyürek, Srinivasan Parthasarathy, and P. Sadayappan. 2018. Efficient sparse-matrix multi-vector product on GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, Tempe, AZ, USA, June 11-15, 2018*, Ming Zhao, Abhishek Chandra, and Lavanya Ramakrishnan (Eds.). ACM, 66–79. https://doi.org/10.1145/3208040.3208062

[20] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP’19)*. Association for Computing Machinery, New York, NY, USA, 300–314. https://doi.org/10.1145/3293883.3295712

[21] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: a flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 71. https://doi.org/10.1109/SC41405.2020.00075

[22] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC ’20)*. IEEE Press, Article 72, 12 pages.

[23] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP ’21)*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/3437801.3441585

[24] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization Framework for Sparse Matrix Kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158. https://doi.org/10.1177/1094342004041296 arXiv:https://doi.org/10.1177/1094342004041296

[25] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *PPoPP ’20: 25th ACM SIGPLAN Symposium on Principles and Practiceof Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 376–388. https://doi.org/10.1145/3332466.3374546

[26] V. Karakasis, G. Goumas, and N. Koziris. 2009. Perfomance Models for Blocked Sparse Matrix-Vector Multiplication Kernels. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*. IEEE Computer Society, 356–364. https://doi.org/10.1109/ICPP.2009.21

[27] Vasileios Karakasis, Georgios I. Goumas, and Nectarios Koziris. 2009. A Comparative Study of Blocking Storage Methods for Sparse Matrices on Multicore Architectures. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009, Vancouver, BC, Canada, August 29-31, 2009*. IEEE Computer Society, 247–256. https://doi.org/10.1109/CSE.2009.223

[28] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[29] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2013. A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR* abs/1307.6209 (2013). arXiv:1307.6209 http://arxiv.org/abs/1307.6209

[30] Süreyya Emre Kurt, Aravind Sukumaran-Rajam, Fabrice Rastello, and P. Sadayappan. 2020. Efficient tiled sparse matrix multiplication through matrix signatures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 87. https://doi.org/10.1109/SC41405.2020.00091

[31] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI ’13)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/2491956.2462181

[32] José E. Moreira, Kit Barton, Steven Battle, Peter Bergner, Ramon Bertran, Puneeth Bhat, Pedro Caldeira, David Edelsohn, Gordon Fossum, Brad Frey, Nemanja Ivanovic, Chip Kerchner, Vincent Lim, Shakti Kapoor, Tulio Machado Filho, Silvia Melitta Mueller, Brett Olsson, Satish Sadasivam, Baptiste Saleil, Bill Schmidt, Rajalakshmi Srinivasaraghavan, Shricharan Srivatsan, Brian W. Thompto, Andreas Wagner, and Nelson Wu. 2021. A matrix math facility for Power ISA(TM) processors. *CoRR* abs/2104.03142 (2021). arXiv:2104.03142 https://arxiv.org/abs/2104.03142

[33] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP’13)*. Association for Computing Machinery, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[34] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 27–40. https://doi.org/10.1145/3079856.3080254

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[36] Ali Pinar and Michael T. Heath. 1999. Improving Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1999, November 13-19, 1999, Portland, Oregon, USA*. ACM, 30. https://doi.org/10.1145/331532.331562

[37] Minsoo Rhu, Mike O’Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 78–91. https://doi.org/10.1109/HPCA.2018.00017

[38] Youcef Saad. 2021. SPARSEKIT: A Basic Toolkit for Sparse Matrix Computations. https://www-users.cs.umn.edu/~saad/PDF/RIACS-90-20.pdf. [Online; accessed 02-April-2021].

[39] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS ’15)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/2751205.2751244

[40] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP ’13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[41] William J. Starke, Brian W. Thompto, Jeffrey Stuecheli, and José E. Moreira. 2021. IBM’s POWER10 Processor. *IEEE Micro* 41, 2 (2021), 7–14. https://doi.org/10.1109/MM.2021.3058632

[42] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. 2005. A Framework for Adaptive Algorithm Selection in STAPL. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP ’05)*. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/1065944.1065981

[43] Rich Vuduc, James Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin C. Lee. 2002. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA, November 16-22, 2002, CD-ROM*, Roscoe C. Giles, Daniel A. Reed, and Kathryn Kelley (Eds.). IEEE Computer Society, 35:1–35:35. https://doi.org/10.1109/SC.2002.10025

[44] Richard W. Vuduc and Hyun-Jin Moon. 2005. Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure. In *Proceedings of the First International Conference on High Performance Computing and Communications* (Sorrento, Italy) *(HPCC'05)*. Springer-Verlag, Berlin, Heidelberg, 807–816. https://doi.org/10.1007/11557654_91

[45] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[46] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1813–1828. https://doi.org/10.1145/2882903.2915220

[47] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. 2019. IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3330345.3330354

[48] Takuma Yamaguchi and Federico Busato. 2021. Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores. https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/. [Online; accessed 02-April-2021].

[49] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 672–687.

[50] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas. 2020. Speeding Up SpMV for Power-Law Graph Analytics by Enhancing Locality & Vectorization. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–15. https://doi.org/10.1109/SC41405.2020.00090

[51] Buse Yilmaz, Bariş Aktemur, María J. Garzarán, Sam Kamin, and Furkan Kiraç. 2016. Autotuning Runtime Specialization for Sparse Matrix-Vector Multiplication. *ACM Trans. Archit. Code Optim.* 13, 1, Article 5 (March 2016), 26 pages. https://doi.org/10.1145/2851500

[52] Hao Yu and Lawrence Rauchwerger. 2000. Adaptive Reduction Parallelization Techniques. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (Munich, Germany). Association for Computing Machinery, New York, NY, USA, 311–322. https://doi.org/10.1145/2591635.2667180

[53] H. Yu, D. Zhang, and L. Rauchwerger. 2004. An adaptive algorithm selection framework. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 278–289. https://doi.org/10.1109/PACT.2004.1342561

[54] Orestis Zachariadis, Nitin Satpute, Juan Gómez-Luna, and Joaquín Olivares. 2020. Accelerating sparse matrix–matrix multiplication with GPU Tensor Cores. *Computers & Electrical Engineering* 88 (Dec 2020), 106848. https://doi.org/10.1016/j.compeleceng.2020.106848