Topology-Aware Optimization of Big Sparse Matrices and Matrix Multiplications on Main-Memory Systems

David Kernert
Technische Universität Dresden
Dresden, Germany
david.kernert@sap.com

Wolfgang Lehner
Technische Universität Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

Frank Köhler
SAP SE
Walldorf, Germany
frank.koehler@sap.com

***Abstract*—Since data sizes of analytical applications are continuously growing, many data scientists are switching from customized micro-solutions to scalable alternatives, such as statistical and scientific databases. However, many algorithms in data mining and science are expressed in terms of linear algebra, which is barely supported by major database vendors and big data solutions. On the other side, conventional linear algebra algorithms and legacy matrix representations are often not suitable for very large matrices. We propose a strategy for large matrix processing on modern multicore systems that is based on a novel, adaptive tile matrix representation (AT MATRIX). Our solution utilizes multiple techniques inspired from database technology, such as multidimensional data partitioning, cardinality estimation, indexing, dynamic rewrites, and many more in order to optimize the execution time. Based thereon we present a matrix multiplication operator ATMULT, which outperforms alternative approaches. The aim of our solution is to overcome the burden for data scientists of selecting appropriate algorithms and matrix storage representations. We evaluated AT MATRIX together with ATMULT on several real-world and synthetic random matrices.**

I. INTRODUCTION

In the recent decade, big data matrices have appeared in many analytical applications of science and business domains. These include solving linear systems, principal component analysis [1], clustering and similarity-based applications [2], such as non-negative matrix factorization in gene clustering [3], but also algorithms on large graphs, for example multi-source breadth-first-search [4]. A very common – and often the most expensive – operation that is involved in all of the aforementioned applications is matrix multiplication, where at least one matrix is usually large and sparse. As an example, consider a similarity query from text mining: a term-document matrix $(\mathbf{A})_{ij}$ that contains the frequency of terms j for every document i, is multiplied with its transpose to get the cosine similarity matrix of documents $\mathbf{D} = \mathbf{A}\mathbf{A}^T$. In gene clustering [3] the core computation contains iterative multiplications $\mathbf{V}\mathbf{H}^T$ of the large, sparse gene expression matrix with a dense matrix.

Such applications used to be custom-coded by data analysts in solutions on a small scale, at best using math libraries or numerical frameworks. However, the vast growth of data volume, and the increasing complexity of modern hardware architectures has driven scientists to shift from handcrafted implementations to scalable alternatives, like massively parallel frameworks and database systems. While in-memory frameworks like R or MATLAB provide a good language environment to develop mathematical algorithms, their implementation is not out-of-the-box scalable. As a consequence, the demand of data scientists for a scalable system that provides a basic set of efficient linear algebra primitives attracted the attention of the database community [5], [6], [7]. Recently emerged systems like SCIDB [6] or SYSTEMML [7] reacted by providing a R or R-like interface, and deep integrations of basic linear algebra operations, such as sparse matrix-matrix and matrix-vector multiplications. However, with the decrease in random access memory (RAM) prizes, it has become feasible to store big analytical data sets in main memory database systems, and run linear algebra algorithms directly in the database engine [8].

In most math systems, such as R, MATLAB, as well as BLAS libraries, the user is required to predefine the final data structure of a matrix. However, the predefinition of matrix storage types is disadvantageous, since it mostly has a negative impact on the performance, e.g. as observed for sparse matrix chain multiplications [9]. However, not only the physical organization in-between multiple matrix operands influences performance. Single matrices are commonly stored as a whole in a static, homogeneous sparse or dense format (e.g., in SCALAPACK), resulting in poor memory utilization and processing performance, if the data representation was not chosen wisely. In fact, additional tuning potential can be leveraged when matrices are considered as *heterogeneous* objects. We observed speedup factors of more than 6x over state-of-the-art sparse matrix multiplication algorithms by splitting a single multiplication operation is into multiple optimized sub-multiplications. In fact, conventional multiplication algorithms are agnostic of density variations within matrices, whereas at the same time, there are many efficient routines for either plain sparse or plain dense matrices. This motivated us to rethink and redesign data structures, and processing practices for large, sparse data matrices.

In this paper, we significantly push the envelope towards a dynamic and adaptive physical organization of matrices and matrix multiplication. In the environment of a multicore main-memory platform, our solution stores and multiplies very large and sparse matrices, by using an optimized data layout based on the matrix non-zero topology. To accelerate the multiplication we use optimization techniques that are inspired from relational

query processing. In this context, we developed the adaptive tile matrix (AT MATRIX) data structure, and a shared-memory parallel matrix multiplication operator ATMULT. In particular, the contributions of this work are:

1) We propose a hybrid matrix representation AT MATRIX consisting of adaptive tiles to store large sparse and dense matrices of any individual non-zero pattern efficiently.
2) We present a time- and space-efficient matrix multiplication operator ATMULT that performs dynamic tile-granular optimizations based on density estimates and a sophisticated cost model.
3) Our work utilizes several methods based on database technologies, such as 2D indexing, cardinality estimation or just-in-time partial data conversions.
4) We evaluated our implementation using a wide range of synthetic and real world matrices from various domains.

Section II starts with the description of our adaptive tiled matrix data type, followed by a description of the partitioning algorithm that converts a a raw matrix into a AT MATRIX. In section III we present our matrix multiplication operator ATMULT, including our tile-granular optimization approach that uses just-in-time data conversions. Finally, the paper is completed by an extensive evaluation of ATMULT in section IV, and a conclusion.

II. ADAPTIVE TILE MATRIX

In this section, we address the problem of selecting an appropriate data representation for large sparse matrices. Typically, the matrix representation is chosen in accordance with the used algorithms, which often have distinct data access patterns. For this reason, many algorithms are particularly written for a certain data representation. As a consequence, choosing the right data structure has a significant impact on the processing performance. Naturally, the selection of data representation and algorithm should depend on matrix characteristics, such as the matrix density, which is the number of non-zero elements divided by the matrix dimensions $\rho = N_{nz}/(m \times n)$. However, we observed that the widely used, crude separation between plain dense and plain sparse matrices does not match the particularities of real world matrices. Their non-zero patterns are usually non-uniform and contain distinct regions of higher and lower densities. Therefore, our adaptive, heterogeneous data representation (AT MATRIX) is able to accommodate individual dense or sparse substructures for matrices of any nature.

A. Matrix Representations

To motivate our data structure, we shortly discuss state-of-the-art matrix representations, which are commonly used by efficient C++ libraries such as the basic linear algebra subprograms (BLAS) and its extensions, as well as software frameworks like R or MATLAB:

1) Dense Matrices: Dense matrices are usually stored in plain C++ arrays, or blocks of arrays, which does not leave a lot of room for optimization. Algorithms and in-place updates on dense matrices are significantly more efficient than on sparse matrix representations, due to the simple layout and direct index accessibility in both – column and row dimensions. Thus, performance-wise it can be promising to spend a bit more memory and use a dense array instead of a sparse representation.

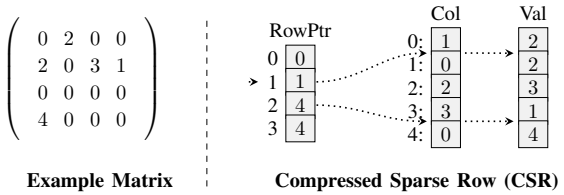


Fig. 1: The Compressed Sparse Row format.

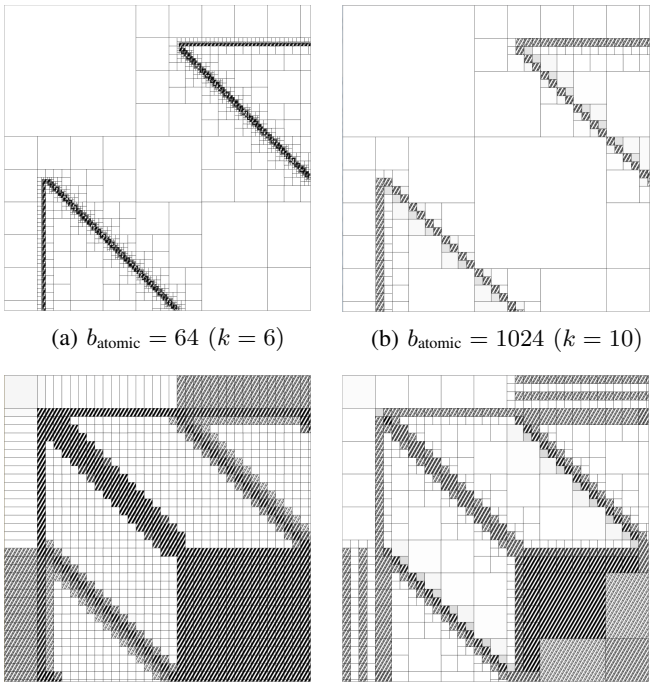
2) Sparse Matrices: In contrast to dense matrices, there are dozens of different data representations for sparse matrices. For a good overview of sparse matrix structures in numeric linear algebra we refer to the work of Saad et. al. [10]. Many representations are well designed for distinct non-zero patterns, such as band or diagonal-dominated matrices, but are poorly suited for general, rather uniformly populated matrices. Widely used for storing sparse matrices is the compressed sparse row format (CSR, aka. CRS – compressed row storage) shown in Fig. 1. It was first presented together with Gustavson's sparse matrix multiplication algorithm [11], and remained the input format for many algorithms in numeric libraries, such as the spgemm method of Intel's MKL [12]. Moreover, CSR-based algorithms tend to have the best performance for sparse matrix vector multiplications, as shown by Vuduc et. al. [13].

Although the above mentioned data structures and corresponding algorithms are widely used even for very large matrices, we identified several reasons for why one should refrain from using them naively. First of all, without a detailed knowledge about the data and the corresponding non-zero pattern, it is unlikely that a user will pick the *best* data structure for a sparse matrix with respect to algorithm performance. Moreover, we observed in our experiments that plain CSR-based matrix multiplication does not scale well on multicore machines. This can be explained with the decreased cache locality due to large matrix dimensions. Similar observations were also made by related work [14]. Finally, large sparse matrices often have a topology with distinct areas of a significantly higher density. Using dense subarrays reveals a significant optimization opportunity by exploiting efficient dense multiplication kernels.

B. Adaptive Tiles

Due to the bad scaling behavior of naive sparse data representations, and the performance improvements that adhere when using dense algorithms for dense subparts of matrices, we made two major design choices for our AT MATRIX structure: first, for a better cache locality of dense and sparse algorithms, each matrix is *tiled* with a variable tile size. Second, tiles with a high population density are stored as dense array, tiles of a lower density have a CSR representation, resulting in a *heterogeneous* matrix structure. The individual tile representation is chosen according to a density threshold value that minimizes the runtime costs of the algorithm under consideration, which is matrix multiplication in this work.

Note that we distinguish in our notation between matrix *tiles* and logical, atomic *blocks*: tiles define the bounding box of the physical representation that accommodates the corresponding part of the matrix. They have variable sizes. In contrast, a logical block is *atomic* and only refers to a square area of



(a) $b_{\text{atomic}} = 64$ $(k = 6)$ (b) $b_{\text{atomic}} = 1024$ $(k = 10)$

(c) Estimated $\hat{\rho}_C$ (d) Result $C = \text{R3} \cdot \text{R3}$

Fig. 2: The TSOPF_RS_b283 matrix (R3) as AT MATRIX using different granularities (2a, 2b), and its estimated (2c) and actual self-multiplication result (2d). The grayscale indicates the population density of sparse tiles, dense tiles are marked with a diagonal pattern.

the fixed size $b_{\text{atomic}} \times b_{\text{atomic}}$. An atomic block is our unit of granularity. Hence, its internal density is approximated as uniform, and no heterogenity is resolved below the block size. Except for corner situations (e.g. at matrix boundaries that are not aligned to the block grid,) a physical matrix tile is greater or equal than a logical block, and could potentially span a region of multiple logical blocks. The tiles are adaptive, thus, the size is chosen such that an AT MATRIX tile covers the greatest possible matrix part of either a low or a high population density, without exceeding the maximum tile size.

1) Maximum Tile Size: For dense tiles, the maximum tile size $\tau^d_{\text{max}} \times \tau^d_{\text{max}}$ is fixed and chosen such that α tiles fit in the last level cache (LLC) of the underlying system. A parameter $\alpha \geq 3$ should reasonably preserve the cache locality for binary algorithms like a tiled matrix-matrix multiplication. Regarding sparse tiles, there are two upper bounds for the size: the first is variable and thus that a sparse tile with density ρ does not occupy more memory than $\frac{1}{\alpha}$ of the LLC size. The other bound is a bit more subtle: it is dimension-based, i.e. chosen such that at least β 1D-arrays with the length of one tile-width should fit in the LLC, which is motivated by the width of accumulator arrays used in sparse matrix multiplication kernels. Hence, the system-dependent maximum tile sizes are calculated as:

$$\tau^d_{\text{max}} = \sqrt{\frac{\text{LLC}}{\alpha \mathcal{S}_d}} \tag{1}$$

$$\tau^{\text{sp}}_{\text{max}} = \min\left\{\sqrt{\frac{\text{LLC}}{\alpha\rho\mathcal{S}_{sp}}}, \frac{\text{LLC}}{\beta\mathcal{S}_d}\right\}, \tag{2}$$

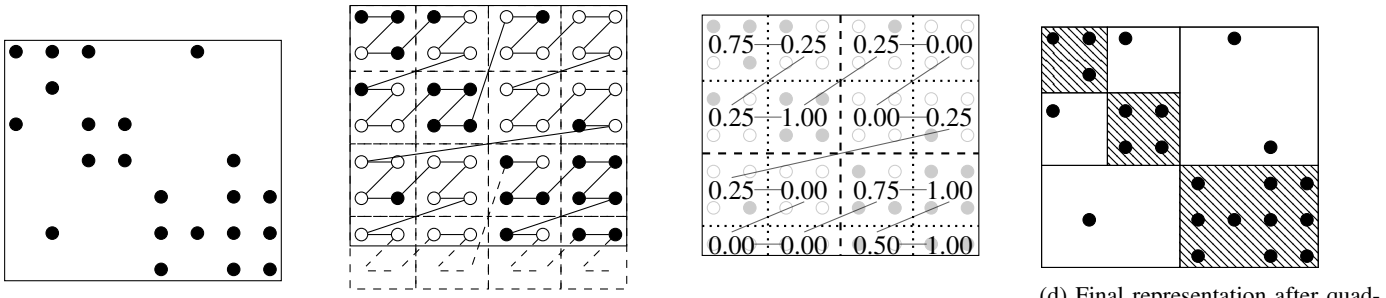
with $\mathcal{S}_{d/sp}$ referring to the size in bytes of a matrix element: $\mathcal{S}_d = 8$ bytes in the dense array representation and $\mathcal{S}_d = 16$ bytes in the sparse CSR representation due to the additional storage that is required for the element coordinates. For our experiments, we chose $\alpha = \beta = 3$, which assures that at least three matrix tiles fit in the LLC at the same time. Naturally, cache locality also depends on concurrency, as well as on the data and respective thread distribution on NUMA sockets. Hence, the ideal tile size and values of α, β might deviate from our heuristic selection, leaving room for further tuning.

2) Minimum Tile Size: The minimum size of matrix tiles defines the granularity of the AT MATRIX, and is equal to the logical block size. The selection of the granularity is generally trade-off between overhead and optimization opportunity: the finer grained the tiles are, the higher the administrative cost of the AT MATRIX, whereas very coarse grained tiles might not be able to resolve the heterogeneous substructure of a matrix. However, atomic blocks that are too small do not only result in a higher overhead of any algorithm dealing with the partitioned AT MATRIX structure, but also increase the recursion depth of our initial partitioning algorithm, which we describe in section II-C. The latter also requires that the dimensions of atomic blocks are a power of two, i.e. $b_{\text{atomic}} = 2^k$. Naturally, k has to be adapted to the system configuration, since the optimal minimum tile size depends on hardware parameters such as the cache size. For a system with a last level cache of 24MB[1], our multiplication experiments have shown the best results for $k = 10$, which yields $b_{\text{atomic}} = 1024$, and is equal to the maximum dense tile size τ^d_{max} of Eq. (1).

Fig. 2a-2b shows the AT MATRIX layout for the matrix R3 (see Tab. I for details) and the two different granularities $k = 6$ and $k = 10$. The heterogeneous tiling yield a significant performance gain by using different multiplication kernels, which is exploited by our ATMULT operator that we present in section III. In fact, the multiplication result matrix in Fig. 2d shows a substructure with distinct sparse and dense regions, underlining the motivation for a topology-aware matrix data type.

Unlike our AT MATRIX approach, a naive matrix tiling with *fixed* block size, as it is done in some implementations [15], [7], often results in adding overhead without any positive impact on the runtime performance. This holds in particular for hypersparse matrices, if blocks are badly dimensioned and barely contain any element. Therefore, our sparse tiles are adaptive: the tile size depends on the number of non-zero elements, and grows until the maximum tile size is hit as of equation (2). As a consequence, sparse, and in particular large, hypersparse matrices without notable dense subregions are not splitted, as long as they do not exceed a certain dimension threshold. Assuming an LLC of size 24 MB and following (2), a sparse matrix with dimensions of $300,000 \times 300,000$ and a

[1]Comparable with our test environment



(a) Raw input matrix $\mathbf{A}$.

(b) Z-Curve ordering of $\mathbf{A}$ and logical atomic blocks.

(c) Density map of $\mathbf{A}$ in the reduced Z-space.

(d) Final representation after quadtree recursion.

Fig. 3: Schematic illustration of the quadtree partitioning using a sparse 7×8 matrix and a 2×2 block granularity.

homogeneous density of $\rho = 10^{-5}$ would be stored in a single, sparse tile. Thus, no additional overhead is added by our AT MATRIX structure, which adds a tile substructure only if it is beneficial for later processing.

C. Partitioning Process

We developed a recursive partitioning method that identifies regions of different density and sizes in the matrix topology, and creates the corresponding matrix tiles in the AT MATRIX representation. This process can be divided into the components loading, reordering, identifying, and materialization, and is illustrated in Fig. 3 for an exemplary, small sparse matrix.

Algorithm 1 Recursive Quadtree Partitioning

```
1: function RECQTPART(ZMatrix Src, ZBlockCnts[], AT MATRICES Tgt, zStart, zEnd)
2:     range ← zEnd − zStart
3:     if range == 0 then
4:         if ZBlockCnts[zStart] == −1 then
5:             return (OUT_OF_BOUNDS, 0)
6:         else
7:             ρ ← CALCDENSITY(ZBlockCnts[zStart])
8:             return (FORWARD, ρ)
9:     else
10:        stride ← (zEnd − zStart + 1)/4
11:        UL ← RECQTPART(Src, ZBlockCnts[], Tgt, zStart, zStart + stride)
12:        UR ← RECQTPART(Src, ZBlockCnts[], Tgt, zStart + stride, zStart + 2 · stride)
13:        LL ← RECQTPART(Src, ZBlockCnts[], Tgt, zStart + 2 · stride, zStart + 3 · stride)
14:        LR ← RECQTPART(Src, ZBlockCnts[], Tgt, zStart + 3 · stride, zStart + 4 · stride)
15:    if UL,UR,LL,LR homogeneous
               and range <MAXSIZE then
16:        return (FORWARD, AVG(UL, UR, LL, LR))
17:    else
18:        return (MATERIALIZED, MATERIALIZETILES(UL, UR, LL, LR, ρ_0^R))
```

1) Locality-Aware Element Reordering: At first, the matrix data is loaded into a temporary, unordered staging representation, which is simply a table of the matrix tuples – containing the coordinates and values of each matrix element. In order to identify areas of different densities in the two-dimensional matrix space, it is crucial to bring the staged raw matrix into a format that preserves the 2D-locality of matrix elements in memory. Obviously, locality is not well preserved for the widely used row-major and column-major layouts: two elements that are placed in the same column and in neighboring rows of an $n \times n$ row-major matrix have a distance of $n \cdot \mathcal{S}_d$ bytes in memory (and equivalently two adjacent elements in one row in the memory layout of a column-major matrix). Hence, column (or row) reads result in a strided memory access throughout the complete memory segment of the matrix, which is obviously unfavorable for locality-aware algorithms that process data in two-dimensional space.

Instead, our algorithm recurses on two-dimensional quadtrees by using a storage layout that preserves locality in both dimensions. Therefore, all elements of the matrix are reordered according to a space-filling curve. This technique is inspired from indices for spatial or multidimensional data, such as in Hilbert R-trees [16] or UB-trees [17]. Besides the Hilbert-curve, there are several space-filling curves that provide a quadtree ordering, and many are equally suited for our recursive method. We decided in favor of the Z-curve, or Morton order [18], since the Z-value can be efficiently computed with bit interleaving [17]. The locality of two matrix-elements is effectively preserved within one quadrant, which is an internal node of the quadtree. This property is recursive, such that the four child quadrants of any node are always stored consecutively in memory, as sketched in Fig. 3b-3c. As a matter of fact, the quadrant dimensions are aligned to a power of two, since with each recursion the quadrant covers exactly a fourth of its parent area. In order to compute the minimal square Z-space that is required to cover the complete matrix with a Z-curve, both matrix dimensions are logically padded to the next largest common power of two. This results in a Z-space size of $K = 4^{\max\{\lceil \log_2 m \rceil, \lceil \log_2 n \rceil\}}$.

2) Identifying Sparse and Dense Submatrices: Alg. 1 sketches our recursive partitioning routine, which takes the Z-ordered source matrix, the AT MATRIX target, two Z-coordinates, and the buffer array *ZBlockCnts[]* as arguments. The latter stores the number of non-zero elements of each atomic block $b_{\text{atomic}} \times b_{\text{atomic}}$, and is precalculated using a single pass over the source matrix. As mentioned before, our atomic block dimensions are aligned to a power of two in

order to match the recursive partitioning routine. The resulting *ZBlockCnts* array is also Z-ordered, but due to the blocking, the initial Z-space of size K is reduced by the factor of b^2_{atomic} (Fig. 3c), and the recursion depth by a factor of $\log_2 b_{\text{atomic}}$.

The algorithm recurses on the *ZBlockCnts* array by using the Z-coordinates *zStart* and *zEnd*. The sub-ranges in-between *zStart* and *zEnd* determine the upper-left (*UL*), upper-right (*UR*), lower-left (*LL*) and lower-right (*LR*) subsquare of the corresponding recursion step in the Z-space (line 11-14). The anchor is reached when *range* is zero (line 3), and consequently, the subsquares have the minimal block size. On the bottom-up recursion path, it is checked if all four neighboring blocks are homogenous – i.e., if all of them are of the same type, and neither of the two maximum block criteria (Eqs. 1&2) is met. If so, the homogenous blocks are logically melted into a four times larger block, which is then returned (*forwarded*) to the recursion level above. If there are neighboring blocks of different density types at any recursion level, then each of the four corresponding blocks is *materialized* into either a sparse or a dense matrix tile (line 18). Depending on the matrix shape and dimension, the padding of the Z-space can result in a few logical blocks in the *ZBlockCnts* array that are outside of the actual matrix bounds, like the lower parts in Fig. 3c. However, these are ignored in the recursion and do not contribute to the submatrix materialization. Fig. 3d sketches the resulting AT MATRIX from Alg. 1 on the sample matrix of Fig. 3a.

In total, the computational complexity of the restructuring algorithm for a matrix $\mathbf{A}$ with N^A_{nz} elements is in $\mathcal{O}\left(N^A_{nz} + K/b^2_{\text{atomic}} + N^A_{nz} \log N^A_{nz}\right)$. The sort term stems from both the preceding Z-ordering as well as from the sparse tile materialization. In contrast, the complexity of a sparse matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{A}$ is given as $\mathcal{O}\left(N^A_{nz}\sqrt{N^C_{nz}}\right)$ in related work [19], [20]. We show in our experiments that the partitioning costs are usually compensated by the performance gain of a single multiplication, except for situations where N^C_{nz} is small or K/b^2_{atomic} is extremely large (hypersparse matrices).

3) Quality Considerations: Our prior aim of the heterogenous tiling is to optimize the processing performance of large sparse matrices, with the focus on matrix multiplication. Unfortunately it is barely possible to judge the quality of the obtained partitioning for a single matrix under this aspect, since the performance of a multiplication heavily depends on the second matrix, which is not known before runtime.

A key aspect of the partitioning algorithm is to decide if a given matrix tile with density $\rho_{(ij)}$ is treated as sparse or dense, which is part of the homogenity check of Alg. 1. In particular, it is checked whether $\rho_{(ij)}$ exceeds the density read threshold value ρ^R_0. The latter is chosen according to the *density turnaround point*: it is effectively the intersection of the multiplication cost functions, at which the dense algorithm starts to be more time-efficient than the sparse algorithm. Since the cost functions are multidimensional, this point is not clearly defined, and is only approximated by ρ^R_0. The eight-fold multiplication cost model will be explained in more detail in section III-C.

However, depending on the actual characteristics of second matrix tile at runtime, the threshold ρ^R_0 might be far off the real turnaround point of the corresponding cost function. In this case, the matrix tile might not be present in the optimal representation. Nevertheless, such a situation is detected by our runtime multiplication optimizer of ATMULT, which triggers a tile conversion from a sparse to a dense representation (or vice versa) at runtime. In the worst case, every matrix tile is later converted into another representation. We simulated this situation in our evaluation (section IV, Fig. 9c- 9a) by multiplying a heterogeneous sparse matrix with a full matrix, which, however, resulted in a conversion overhead of not more than 10% of the total runtime. Since the cost model is not only dependent on the second matrix density, but also on the system configuration, ρ^R_0 is besides b_{atomic} the second adaptable tuning parameter.

The memory consumption of the AT MATRIX can either be lower or higher than that of a pure sparse matrix representation, but is always lower than a plain dense array. The worst case is present when all tiles have densities slightly above ρ^R_0. Then, the matrix would be stored as dense, consuming $\mathcal{S}_d/(\rho^R_0 \mathcal{S}_{sp})$ as much memory as the sparse representation (maximum 2x in our configuration).

III. MATRIX MULTIPLICATION

In this section we describe ATMULT, our matrix multiplication operator for general, large matrices. The ATMULT operator supports three independent operand types for the following: left input $\mathbf{A}$, right input $\mathbf{B}$ and output matrix $\mathbf{C} \rightarrow \mathbf{C}'$ of the equation $\mathbf{C}' = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$. Each matrix type can be one of the following: a plain matrix structure such as dense arrays or sparse CSR matrices, as they are commonly used in numerical algebra libraries, or a heterogeneous AT MATRIX. The latter provides a performance improvement by a cost-based optimization of tile-multiplications, and just-in-time transformations.

Algorithm 2 Sequential version of ATMULT

```
1:  function ATMULT(AT MATRICES A, B, C)
2:      ρ̂_C ← ESTIMATEDENSITY(ρ̂_A, ρ̂_B)
3:      ρ_D^W ← min{ρ_0^W, WATERLVLMETHOD(ρ̂_C, MEMLIMIT)}
4:      for all tile-rows ti in A do
5:          for all tile-cols tj in B do
6:              C_{ti,tj} ← ((ρ̂_C)_{ti,tj} ≥ ρ_D ? DenseTile :
                                                    SparseTile)
7:              for all matching tiles k in A_{ti} and B_{tj} do
8:                  w ← CALCULATEREFWINDOW(A_{ti,k}, B_{k,tj})
9:                  A'_{ti,k}, B'_{k,tj} ← OPTIMIZE(A_{ti,k}, B_{k,tj}, C_{ti,tj})
10:                 TILEMULTIPLY(A'_{ti,k}, B'_{k,tj}, C_{ti,tj}, w)
```

The pseudocode for ATMULT is shown in Algorithm 2. For illustrational purposes, we first sketch the *sequential* version of ATMULT, and address the parallelization later separately. In general, multiplications that involve one or more AT MATRICES are internally processed as multiple mixed multiplications of *dense* or *sparse* matrix tiles in a block-wise manner, as depicted in Fig. 4. The resulting matrix tile multiplications are then processed by the basic multiplication kernels. However, unlike naive block-wise matrix multiplication, one particularity of our ATMULT operator is that matrix tiles may have different block sizes. This entails that some multiplications involve only the matching subpart of a matrix tile, since a matrix multiplication is only defined for matrices with a matching contraction (inner)

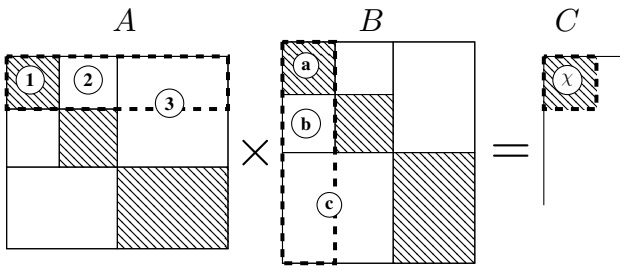


Fig. 4: Referenced submatrix multiplication.

dimension k. Fig. 4 sketches a situation, where the matrix tile $\mathbf{C}_{(\chi)}$ results from the accumulation of three tile-multiplications $\mathbf{A}_1\mathbf{B}_a$, $\mathbf{A}_2\mathbf{B}_b$ and $\mathbf{A}_3\mathbf{B}_c$. However, for the latter only the upper half of tile $\mathbf{A}_{(3)}$ is multiplied with the left half of tile $\mathbf{B}_{(c)}$. We denote this operation as *referenced submatrix multiplication*.

A. Multiplication Kernels

The contribution of this paper is the optimal processing of tiled matrix multiplication rather than the particularities of low-level algorithms of the multiplication kernels, since sparse and dense matrix multiplication kernels are continuously improved and tuned for modern hardware [21], [14], [22], [23], [24]. Our ATMULT approach acts in the layer above the kernels, which are merely invoked in form of multiplication tasks (line 10). The optimization is decoupled from the implementations, given that the cost functions of all kernels are known to the optimizer. Hence, the idea is to keep the ability to use existing high performance libraries as multiplication kernels wherever possible. Therefore, our architecture uses only common matrix representations, such as arrays or CSR, so that our approach benefits from a broad variety of libraries, which provide efficient implementations for the corresponding kernels. Thus, new algorithms from the high performance community that are based on the same basic data structures could just be *plugged in* to our system.

Nevertheless, for some of the multiplication kernels many high performance libraries such as IntelMKL offer no reference implementation, for example for a *dense* $\times$ *sparse* $\rightarrow$ *dense* multiplication. In total, there are $2^3 = 8$ different kernels for the basic matrix types that are either sparse or dense. We implemented the sparse and mixed sparse-dense multiplication kernels ourselves, using shared-memory parallel implementations based on the sparse accumulator approach. The latter is inspired from the classical row-based CSR algorithm of Gustavson [11], but used analogously in many column-based CSC algorithm implementations, e.g. in MATLAB [25] or CombBLAS [26], [22]. For space reasons, we will not discuss each multiplication kernel, and refer to related work for more details, in particular to [11].

B. Referenced Submatrix Multiplications

The ability of multiplying only certain subparts of matrix tiles is a basic building block of ATMULT. Arbitrary rectangular subparts can be referenced via four coordinates, namely the coordinates of the *upper left* edge (x_{ul},y_{ul}), and the coordinates of the *lower right* edge (x_{lr},y_{lr}), which are both relative to the upper left end of the matrix tile $(0, 0)$. In algorithm 2 the referenced coordinates are encoded in the reference window w.

Fortunately, the implementation of referencing in the multiplication kernels is less intrusive than one might expect, in particular for dense matrix multiplications: the BLAS function interface *gemm* (**ge**neral **m**atrix **m**ultiply) already distinguishes between the embracing array size that could differ from the actual matrix dimensions, by providing the additional parameters *lda*, *ldb*, and *ldc* (leading array dimension in $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$). So the only adaption we made is transforming the submatrix coordinates into an offset that points to the beginning of the matrix in the array, and passing the respective *lda* and *ldb* to the gemm kernel.

Regarding sparse kernels, referencing subranges in the *row* dimension is equally trivial, since matrix rows in the CSR representation are indexed. In contrast, matrix elements are not index-addressable in the column direction, which complicates the adaption of our sparse multiplication kernels to process column ranges. We altered the row-based algorithm to process only the column indices that are contained in the referenced row range. To avoid undesired column range scans, we sorted the elements in each row by column id at creation time to enable binary column id search. Moreover, the number of non-zero elements in a sparse tile of size $\tau_{\max}^{\text{sp}}$ is restricted (Eq. 2), which limits the number of elements in a column range. In fact, we observed in our experiments that the overall performance improvement due to dynamic multiplication optimization overcompensates the remaining overhead due to referenced submatrix multiplications.

C. Dynamic Multiplication Optimizer

As described in our prior work [9], each of the above mentioned kernel functions has a different runtime behavior that is reflected by a comprehensive cost model. The runtime of each kernel depends on the dimensions $m \times k$ of matrix $\mathbf{A}$, $k \times n$ of matrix $\mathbf{B}$, the densities ρ_A, ρ_B, ρ_C of both input matrices, as well as the *estimated* result density $\hat{\rho}_C$. A common situation is that some matrix tile, or even a subpart of a matrix tile, exceeds a characteristic density threshold of the cost model. Then it might be beneficial to convert the respective part into a dense representation prior to the multiplication operation. Hence, the optimizer dynamically converts the input matrix representations whenever it leads to a reduction of the multiplication runtime for the respective part multiplication (line 9 in Alg. 2). In contrast to a single multiplication operation, the target tile $\mathbf{C}_{(\chi)}$ (Fig. 4) is written accumulatively in multiple tile-multiplications (i.e., in each multiplication of the corresponding block-row $\mathbf{A}_1\mathbf{B}_a$, $\mathbf{A}_2\mathbf{B}_b$ and $\mathbf{A}_3\mathbf{B}_c$). Hence, the physical representation of a $\mathbf{C}_{(\chi)}$ tile is selected according to its *final* density, which is taken from the density estimate $\hat{\rho}_{C(\chi)}$.

There is a significant difference in the cost model regarding the density turnaround point between the matrix $\mathbf{A}, \mathbf{B}$-tiles that are *read* from (as part of the multiplication), and the $\mathbf{C}$-tiles that are *written*. In general, writing a sparse tile is much more expensive than reading it, compared to a dense tile, which has a smaller read/write asymmetry. This is why we introduced two density thresholds: one for tiles that are read ρ_0^R, and one for tiles that are written ρ_0^W, whis has usually a much lower value. The latter also explains the good

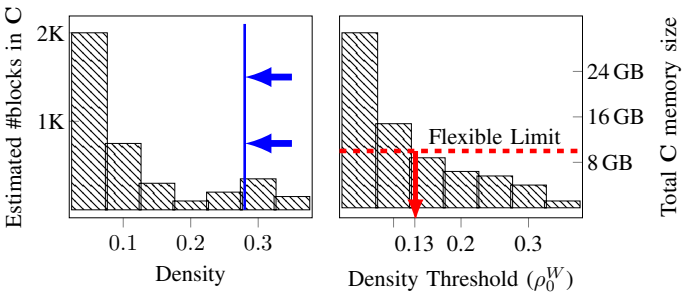


Fig. 5: *Left)* One-dimensional histogram of logical block densities. *Right)* The memory consumption depending on the density threshold. The water-level method approaches the intersection of the total memory consumption with the *flexible limit* (dashed line) from the right.

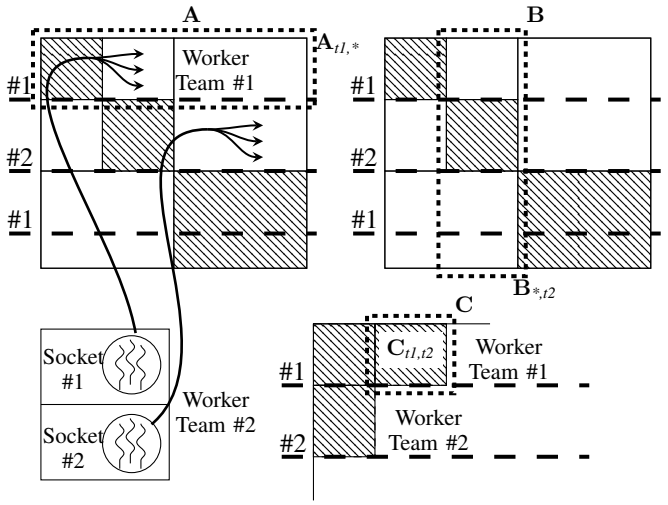


Fig. 6: Parallel resource distribution and NUMA partitioning for a ATMULT operation on a two-socket system.

results of the spspd_gemm kernel in our evaluation compared to spspsp_gemm. However, storing a rather sparse result matrix in a dense array is an excessive waste of memory that should be avoided, even when the cost-based optimization might decide so. In a resource-managed system, such as a DBMS, there are usually some operational service level agreements (SLAs) to meet, for example a restriction of the total memory consumption. Therefore, our ATMULT approach employs a flexible *write* density threshold ρ_D^W depending on the desired memory space limit of the target matrix (line 3). This adaption to runtime-available resources might sacrifice performance in favor of a lower memory consumption.

D. Density Estimation

There are several reasons for having a prior estimate of the block-density structure of the result matrix, and they are barely different from the motivation of cardinality estimation for relational join processing: first, the optimizer's decision naturally depends on the output density, which is a impactful parameter in the cost model of sparse multiplication kernels. Second, knowing the result density is crucial for memory resource management, which includes memory restrictions that also effect the choice of data representations in the optimizer, e.g. in order to avoid out-of-memory situations. However, the exact non-zero structure can only be found through the actual execution of the multiplication. Therefore, we make use of the density estimation procedure based on probability propagation, which was presented in [9]. In particular, we use the "density map" estimator to obtain an estimate at negligible runtime cost compared to the actual multiplication. The estimator takes the density maps of the input matrices $\mathbf{A}$ and $\mathbf{B}$ and returns a density map estimate of the resulting matrix $\mathbf{C}$ (e.g. in Fig. 2c). For more details about the theory background of the estimation we refer to [9], section 4.3.

E. Memory Resource Flexibility

In order to determine the common local write density threshold ρ_D^W for *target* tiles with respect to the flexible memory consumption limit for the total matrix, we employ a "water level method" that works on the density map estimate $\hat{\rho_\mathbf{C}}$ described above (line 3 in Alg. 2): consider $\hat{\rho_\mathbf{C}}$ as a two-dimensional histogram with a density bar ρ_{ij} for each logical matrix ($b_{\text{atomic}} \times b_{\text{atomic}}$) block (ij) – the higher the density ρ_{ij} of block (ij), the higher its bar in the histogram. The idea is to start with a high water level that covers all bars in the two-dimensional histogram. If the level is lowered, the blocks with the highest density bars will be visible first – hence, storing these blocks as dense is most promising with regard to the potential performance improvement. The level will be continuously lowered until the accumulated memory consumption of all dense and sparse blocks hits the memory limit. The method can easily be transformed from two-dimensional space into a one-dimensional space, as shown in Fig. 5: instead of a 2D-histogram, we create a 1D histogram that displays the absolute number of logical blocks per density bin ρ_j. Lowering the water level can be imagined as a sliding vertical line (Fig. 5 *left*) from the right to the left histogram side. All blocks right of the line contribute with dense size BS_d to the memory consumption, whereas all blocks on the left-hand side only with the size of the sparse representation $\rho_i \cdot BS_{sp}$. The resulting density threshold equals the intersection of the accumulated histogram (Fig. 5 *right*) with the flexible memory limit.

F. Parallelization

Our implementation of ATMULT (Algorithm 2) is parallelized in two stages: first, pairs (*ti*,*tj*) of tile-rows *ti* of matrix $\mathbf{A}$ and tile-columns *tj* of matrix $\mathbf{B}$ are formed. For instance, in Fig. 6 the pair ($\mathbf{A}_{t1,*}$,$\mathbf{B}_{*,t2}$) is marked. Each pair represents a set of tile-multiplication tasks that write a target tile $\mathbf{C}_{ti,tj}$ in the result matrix. All tile-multiplications referring to a particular tile-row-column pair are executed one after another, and by the same worker team. Nevertheless, multiple worker teams are running in parallel on different pairs, e.g. team #1 and #2 in Fig. 6. Moreover, all multiplication kernels in our ATMULT operator are internally shared memory-parallel. As a result, there are in total *two* levels of parallelization: the *intra-tile* parallelization (number of worker teams) and *inter-tile* parallelization (number of threads in a team). Thus, the parallel resources should be distributed among these levels, which is generally nontrivial: assigning all threads to the intra-

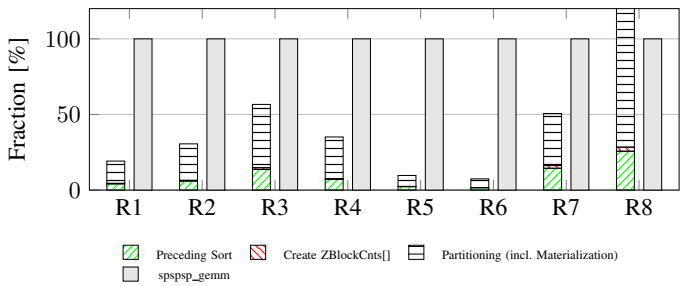


Fig. 7: Duration of the components in the partitioning process, relative to a sparse matrix multiplication (executed once).

tile level might lead to over-parallelization, when blocks are small and very sparse. In contrast, leaving all threads to the coarse-grained inter-tile level might have a negative influence on the last level cache re-usage. If many different tiles are touched concurrently, this could pollute the cache. To avoid cache pollution, we spawn as much worker teams as there are sockets on the system, so that each worker team writes to its local memory node. The number of threads per team is related to the number of cores on the socket.

On modern multicore machines, the RAM is distributed across multiple sockets that contain one or more processors each, leading to effects of non-uniform memory access (NUMA). To reduce remote accesses, our matrices are distributed across the memory nodes as part of the partitioning process. Since it is generally unknown whether a matrix will take part as the left or the right operand in a matrix multiplication, all matrices are in the same way horizontally partitioned. Consequently, $\mathbf{A}_{ti,*}$ and $\mathbf{B}_{ti,*}$ tile-rows are distributed round-robin-wise. The worker thread teams of ATMULT are pinned to the socket of the respective $\mathbf{A}_{ti}$ tile-row. Since the worker threads dynamically allocate the target tiles in $\mathbf{C}_{ti,tj}$ and due to the Linux first touch policy [27], $\mathbf{C}$ effectively inherits the tile-row memory distribution scheme from matrix $\mathbf{A}$. For our implementation we used a task scheduling framework [28] that is used in the SAP HANA database [29], since it offers options to pin tasks to CPU cores and memory nodes.

IV. Evaluation

In this section we present measurements of the partitioning routine and the ATMULT matrix multiplication runtimes using a variety of real-world matrices. Moreover, we added synthetic matrices using an RMAT graph generator [30] to systematically evaluate the influence of data skew.

A. Experiment Setting

We used a four-socket Intel E7-4870 system with 4×10 cores (80 physical threads via hyperthreading) @2.40 GHz and a total of 1 TB RAM. Regarding the tunable parameters, our system configuration yields $b_{\text{atomic}} = 1024$ ($k = 10$) for the minimum tile size and $\rho_0^R = 0.25$ as tile read density threshold.

Tab. I lists the sparse matrices that we used in the experiments. All real-world matrices (Ri) except the Hamiltonian matrices are taken from the Florida Sparse Matrix Collection[2], which contains matrices of various domains, such as power networks, genomics or structural problems. The Hamiltonian matrices were provided by a nuclear physics group we cooperated with. In general, we tried to select matrices of different shapes and sizes in order to compare our approach on a wide scale of data. However, to understand the implications of matrix size, density, and data skew on our algorithm better, we added synthetic matrices (Gi). These were generated using an RMAT [30] recursive graph generator, and are listed in the lower half of Tab. I. RMAT-generated matrices are configurable via the parameters: dimension, number of non-zero elements, and the four values a, b, c, d. The latter control the relative fractions of non-zero elements that are contained in the upper left, upper right, lower left and lower right part of the submatrix at each recursion step.

B. Heterogeneous Partitioning

Fig. 7 shows the relative duration of the following components of the partitioning process: the preceding sort to create the Z-order, the creation of the *ZBlockCnts* array, and the recursive partitioning routine itself, including the tile materializations. The partitioning time is dominated by the materialization, which includes copying and reordering into CSR, or row-major array representation. Except for matrix R8, the duration of the partitioning process is smaller than a single execution of the traditional multiplication algorithm. Matrix R8 matches a case we mentioned in section II-C2, where the non-zero size of the output matrix is relatively small, but the dimensions are large.

C. Performance Comparison

Fig. 8 shows the matrix multiplication performance of our ATMULT approach relative to the plain *sparse* $\times$ *sparse* $\rightarrow$ *sparse* multiplication kernel (spspsp_gemm). We chose it as baseline ($\equiv 1$) because it is similar to the algorithm used in R or MATLAB, which however, only have a sequential sparse matrix multiplication implementation. In addition, we included the spspd-, spdd-, and ddd_gemm (IntelMKL) kernels in the measurement.

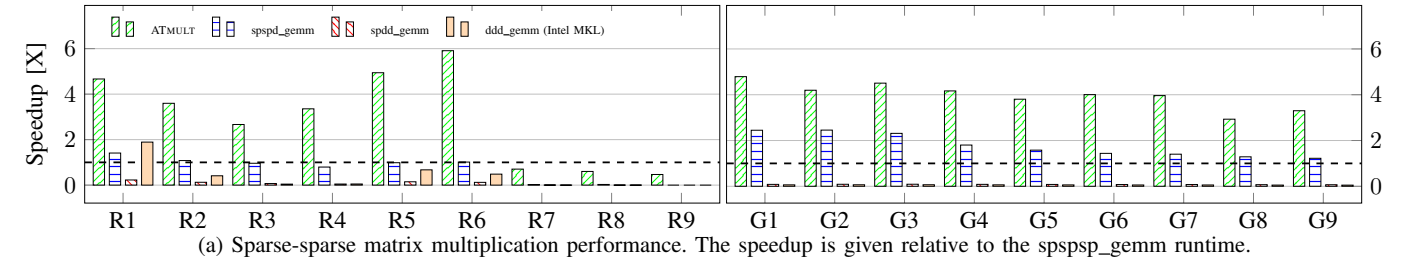
We observe that ATMULT outperforms the alternative approaches in most cases. Only for matrices R7-R9 the ATMULT performance is slightly behind the spspsp_gemm performance. This can be explained by the matrix characteristics: matrices R7-R9 do not contain any region of a higher density, hence, offering little optimization potential. As a consequence, the partitioning overhead just adds to the ATMULT execution runtime. Obviously, the other approaches that involve plain dense matrix representations (spspd-, spdd- & ddd_gemm) have an even worse performance for R7-R9. Nevertheless, for most of the other instances, the spspd_gemm kernel (dense target array) seems to be the better alternative to spspsp_gemm, since the result matrix is often significantly denser than the input matrices. The advantage of ATMULT over the naive algorithms is most clearly when there are distinct regions of a significantly higher local density in the matrix non-zero pattern, as for example for matrix R3, which is illustrated in Fig. 2. But even for matrices that have a dense result and a relatively uniform non-zero pattern (R4, G1, G2), we observe that ATMULT can outperform the others, particularly the spspd_gemm kernel.

[2]http://www.cise.ufl.edu/research/sparse/matrices/index.html

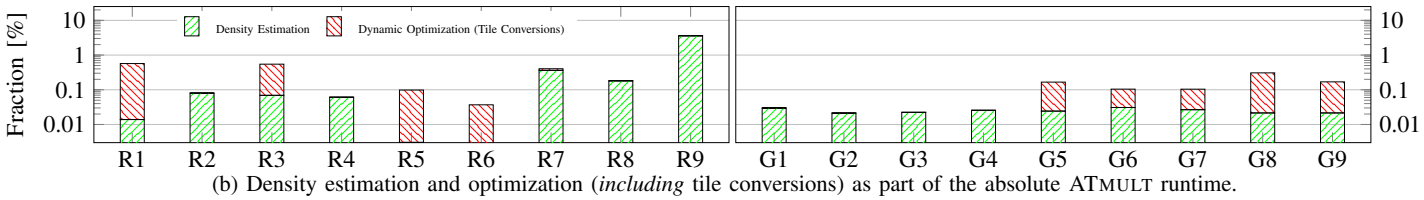
TABLE I: Sparse matrices of different dimensions and population densities. The $\rho = N_{nz}/(n \times n)$ value denotes the population density of each matrix. All matrices are square ($n \times n$.) The binary size is given for the sparse triple/coordinate format (COO) consisting of $\langle$*int, int, double*$\rangle$.

Number	Name	Matrix Domain	Dimensions	N_{nz}	ρ [%]	Bin. Size	Result Size
	Real-World Matrices						
R1	Hamiltonian1	Nuclear Physics	17040 × 17040	42.95 M	14.8	687 MB	4.64 GB
R2	human_gene	Gene Expr. (BioInf.)	22283 × 22283	24.67 M	5.0	395 MB	3.58 GB
R3	TSOPF_RS_b2383	Power Network (Eng.)	38120 × 38120	32.31 M	2.2	517 MB	6.24 GB
R4	mouse_gene	Gene Expr. (BioInf.)	45101 × 45101	28.97 M	1.4	463 MB	7.72 GB
R5	Hamiltonian2	Nuclear Physics	52928 × 52928	188.93 M	6.7	3.02 GB	42.1 GB
R6	Hamiltonian3	Nuclear Physics	77205 × 77205	319.30 M	5.4	5.11 GB	88.1 GB
R7	barrier2-4	Semicond. Device (Eng.)	113 K × 113 K	2.13 M	0.016	34 MB	993 MB
R8	pkustk14	Structural Problem (Eng.)	152 K × 152 K	11.20 M	0.048	179 MB	907 MB
R9	msdoor	Structural Problem (Eng.)	416 K × 416 K	19.17 M	0.011	230 MB	955 MB
	Generated Matrices	Parameters $\{a, b, c, d\}$					
G1	RMAT1	$\{0.25, 0.25, 0.25, 0.25\}$	100 K × 100 K	20 M	0.2	320 MB	55.5 GB
G2	RMAT2	$\{0.35, 0.22, 0.22, 0.21\}$	100 K × 100 K	20 M	0.2	320 MB	60.0 GB
G3	RMAT3	$\{0.45, 0.18, 0.18, 0.19\}$	100 K × 100 K	20 M	0.2	320 MB	62.3 GB
G4	RMAT4	$\{0.55, 0.15, 0.15, 0.15\}$	100 K × 100 K	20 M	0.2	320 MB	59.2 GB
G5	RMAT5	$\{0.61, 0.13, 0.13, 0.13\}$	100 K × 100 K	20 M	0.2	320 MB	55.0 GB
G6	RMAT6	$\{0.64, 0.12, 0.12, 0.12\}$	100 K × 100 K	20 M	0.2	320 MB	52.6 GB
G7	RMAT7	$\{0.67, 0.11, 0.11, 0.11\}$	100 K × 100 K	20 M	0.2	320 MB	50.3 GB
G8	RMAT8	$\{0.70, 0.10, 0.10, 0.10\}$	100 K × 100 K	20 M	0.2	320 MB	47.2 GB
G9	RMAT9	$\{0.73, 0.09, 0.09, 0.09\}$	100 K × 100 K	20 M	0.2	320 MB	43.7 GB

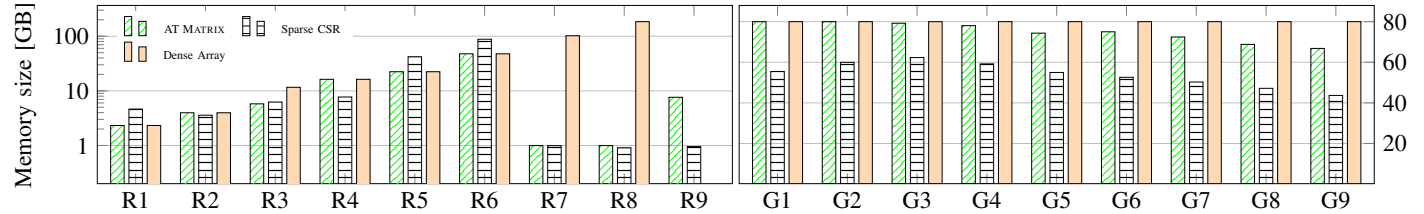
increasing skew



(a) Sparse-sparse matrix multiplication performance. The speedup is given relative to the spspsp_gemm runtime.



(b) Density estimation and optimization (*including* tile conversions) as part of the absolute ATMULT runtime.



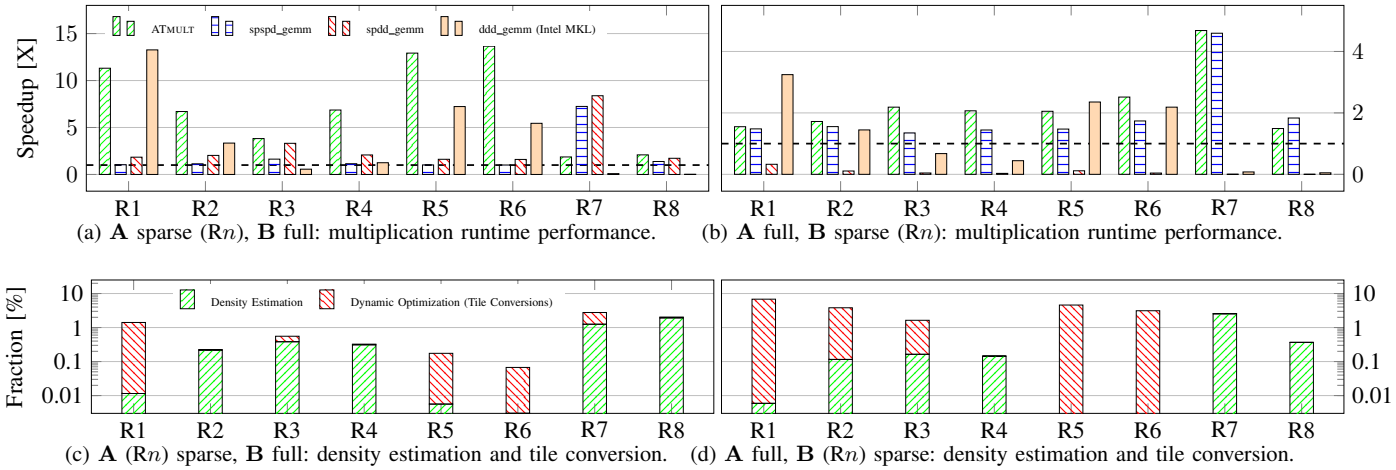
(c) Memory consumption of $\mathbf{C}$ in the AT MATRIX, *sparse* (CSR) and *dense* representation.

Fig. 8: Sparse $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ multiplication experiments where $\mathbf{A} = \mathbf{B}$, using real-world and synthetic matrices (see Tab. I for details about matrices).

This is an indication that part of the performance gain results from an increased cache locality due to the tiling that is caused by the block size limit of Eq. 2.

Regarding the memory consumption depicted in Fig. 8c, the AT MATRIX result matrix ($\mathbf{C}$) created by ATMULT tends to have an equal or lower size than the output size of alternative approaches, and is even sometimes lower than the sparse CSR format (matrices R1, R3, R5, and R6). The latter case arrives when there are dense regions $\rho > (\mathcal{S}_d/\mathcal{S}_{sp})$ that are stored more efficiently in a dense array.

To confirm our implications in a more systematic manner, we generated large RMAT matrices G1-G9 with increasing



(a) **A** sparse (Rn), **B** full: multiplication runtime performance.
(b) **A** full, **B** sparse (Rn): multiplication runtime performance.
(c) **A** (Rn) sparse, **B** full: density estimation and tile conversion.
(d) **A** full, **B** (Rn) sparse: density estimation and tile conversion.

Fig. 9: Multiplication of a sparse with a dense matrix: Performance speedup of ATMULT and optimization time breakdown.

skew in the matrix non-zero pattern. For equal parameters $a \equiv b \equiv c \equiv d$ the non-zero element distribution is nearly uniform. In contrast, the higher the parameter a, the more non-zero elements are placed in the upper left quarter of each recursion step, and hence, the higher is the skew of the matrix. All generated matrices with their parameters are also listed in Tab. I.

The right-hand side of Fig. 8a illustrates how the skew affects the runtimes of the sparse-sparse matrix multiplication. We observe that ATMULT outperforms spspsp_gemm by a factor of 3-5x, and spspd_gemm by a factor of about 2-2.5x. With increasing skew, the internal execution of ATMULT changes, which can be inferred from the increased optimization efforts in Fig. 8b. Interestingly, the speedup over spspsp_gemm is slightly shrinking, which is caused by a decreasing runtime of the latter, which is correlated to the output size (see Fig. 8c). The skew reduces the number of non-zero elements in the multiplication result matrix, as more element multiplications $(A)_{ik}(B)_{kj}$ are falling into the same target coordinates (ij). In contrast to the naive spspd_gemm approach, the memory size of the ATMULT result reflects this trend, as internally more result tiles will be stored sparse instead of dense.

Besides sparse-sparse matrix multiplication, which is for example used in graph algorithms, many applications involve the multiplication of a sparse with a full, dense ($\rho = 1.0$) matrix. Therefore, we included mixed sparse-dense multiplication in the experiments, and distinguish between: Fig. 9a) {**A**: sparse, **B**: dense}, and 9b) {**A**: dense, **B**: sparse}. The dense (full) matrices are formed such that the number of elements $m \cdot k$ is in the same order of magnitude as the number of sparse matrix elements N_{nz}. Hence, the dense matrix is rectangular, and the independent dimension is calculated as $n = \gamma N^A_{\text{nz}}/k$ for b) and $m = \gamma N^B_{\text{nz}}/k$ for c), respectively (we chose $\gamma = 3$).

Similar to the results of the sparse-sparse multiplication, we observe that ATMULT outperforms the alternatives for all mixed sparse-dense test instances except for the following: The relatively dense and small matrix R1 is multiplied most efficiently with ddd_gemm (MKL). Although ATMULT also uses this kernel internally, our dynamic optimizer has to convert some initial sparse tiles into dense tiles, adding overhead to the total execution time. Second, ATMULT is outperformed by sp(d|sp)d_gemm for the light, hypersparse matrix R7. Here, the overhead results from the implicit slicing of **A** in the multiplication, due to referenced submatrix multiplications caused by the actual partitioning of **B**. Such situations could be avoided by a dynamic re-tiling of the left-hand matrix as a part of a pre-multiplication optimization, which, however, is left for future work.

D. Runtime Optimizations

Finally, we evaluate how the total runtime of a ATMULT multiplication operation breaks down into its processing steps: the density estimation, the dynamic optimization including tile conversions, and the multiplication runtime itself. We observed that the actual tile multiplication runtime is by far the most expensive operation in ATMULT for all test instances, which is why we only show the relative fraction of ATMULT time that is spent with density estimation and optimization (tile conversion) in Figures 8b, 9c, and 9d. First of all, we observe from the right of Fig. 8b that the dynamic optimization time is almost zero for nearly uniform matrices, and grows with increasing skew. It reaches a peak of about 7.5% of the total runtime for the dense-sparse multiplication with matrix R1 in Fig. 9d. This peak can be explained by the topology of the AT MATRIX R1: many tiles have a density slightly below the read density threshold, thus, they are stored in the sparse representation. In fact, the optimizer then may decide to convert these tiles into a dense representation prior to the tile multiplication, adding conversion time to the optimizer runtime. Nevertheless, in the vast majority of the cases the overall multiplication time including the conversion is still less than the sole multiplication time when using a plain sparse or dense data representation.

The part of the density estimation is for most instances with less than 0.1% of ATMULT runtime negligible. As a matter of fact, the runtime of the density estimation procedure is independent from the number of non-zero elements, but depends on both matrix dimensions and the logical block size, which together define the size of density grid. As a consequence,

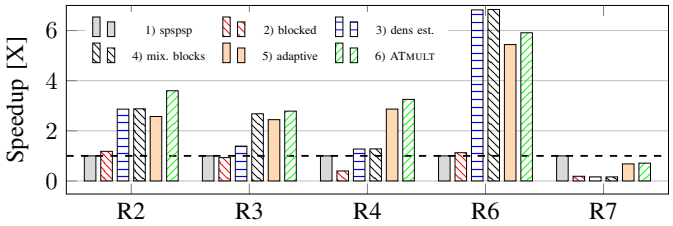


Fig. 10: Relative multiplication performance of $\mathbf{A}$ and $\mathbf{B}$ are sparse. Impact of optimization steps on performance.

it becomes more significant (5% for matrix R9) for hypersparse matrices with very high dimensions.

E. Impact of Single Optimization Steps

In our final experiment, we evaluated step-by-step the influence of adding different optimization components, and how they improve the multiplication runtime performance. Outgoing from the regular spspsp_gemm multiplication, we incrementally add the optimization steps that are contained in our ATMULT approach. They can be separated as follows:

1) Baseline: spspsp_gemm on unpartitioned sparse matrices.
2) Fixed-size, sparse-only tiles: The matrix is tiled into a fixed grid of sparse tiles, and product tiles are also sparse.
3) Fixed-size, sparse-only tiles, and density estimation: Same as 2), but with target density estimation. Target tiles are dense if the density exceeds ρ_0^W.
4) Fixed-size, mixed tiles, and density estimation: Same as 3), but the matrix blocks exceeding ρ_0^W are stored as dense.
5) Adaptive, mixed tiles, and density estimation: In contrast to 4), tiles are adaptive, but without dynamic tile conversion
6) Adaptive, mixed tiles, density estimation, and dynamic tile conversion optimization (ATMULT)

The way how the independent steps affect the performance significantly depends on the matrix topology. Therefore, we chose five different instances of the real-world matrices, and assembled the relative multiplication performance results in Fig. 10. The first observation is that, if the target tiles are sparse-only (2), then a naive fixed-size $b_{\text{atomic}} \times b_{\text{atomic}}$ blocking of a sparse matrix yields barely in an improvement. However, a significant performance boost for some matrices (R2, R6) is achievable by enabling density estimation and dense target tiles (3). In contrast to (2), dense target tiles are written more efficiently in an accumulative manner, which is in a way enabling the fixed-size tiling optimization to reveal its actual improvement potential. If we further add the storage type optimization (4), all fixed-size tiles of matrix $\mathbf{A}$ that exceed the density threshold are stored in a dense representation. Consequently, we see a performance jump for matrices that have distinct dense substructures, such as R3. For some of the matrices, e.g. R6, the optimization potential is already fully exploited by the steps up to (4). Adaptive tiling incurs some overhead that lowers the performance in these cases, but only by less than 20%. However, the fixed-block approaches (2-4) bear an even higher overhead for larger and slightly sparser matrices, e.g. R4, where the adaptive tile multiplication (5), which is yet improved by the dynamic tile conversions (6), outperforms them by 3x. More significantly, for the relatively sparse matrix R7, the ATMULT runtime is close to the naive spspsp_gemm, since, if at all, a very coarse grained tiling is chosen. In contrast, all fixed size tiling approaches fail completely, and are orders of magnitude slower.

V. Related Work

As this work has overlaps with multiple research areas, we subdivide the discussion into the major subtopics:

A. Sparse Matrix Representations

A lot of effort has been invested in implementing efficient algorithms and data structures for sparse matrices in recent decades, starting from early FORTRAN77 implementations in the 70s. Gustavson [11] proposed the CSR representation together with a sparse matrix multiplication algorithm, which remained relevant until today.

Many recent works of the high performance community present implementations of sparse matrix vector multiplication kernels either on multicore CPUs [31], [13], GPUs [32], [33] or CPU+GPU [24]. The data representations used are either CSR, ELLPACK storage (ELL), the coordinate storage format (COO), or blocked representations, such as block-ELL [32] or block-CSR (BCSR). The latter stores small, fixed-size dense blocks instead of single matrix elements.

Although there are many different sparse matrix representations for different topologies, such as triangular, band matrices, diagonal or almost diagonal matrices (a good overview is given here [10]), Vuduc [13] observed that CSR tends to have best performance for sparse matrix-vector multiplication (spgemv) on a wide class of matrices, supporting our decision to chose CSR as the core representation for sparse matrix tiles.

B. Sparse Matrix-Matrix Multiplication

Common sparse multiplication algorithms, including Gustavson's and in the one used by MATLAB [25], are based on the sparse accumulator approach, which processes single rows (Gustavson) or columns (MATLAB) of the target matrix one after the other. Both algorithms are single core implementations, but there have been many works on a scalable equivalent for parallel processors in the meantime: Buluc et. al [22] present a distributed multiplication algorithm, where they store the matrix in tiles of hypersparse blocks. The algorithm, which is contained in the Combinatorial BLAS library [26], however, turned out to be comparatively slow on multicore systems, as observed in [14]. The authors of the latter paper present their own approach based on a partitioned Gustavson algorithm, where matrix $\mathbf{A}$ is divided in horizontal blocks and $\mathbf{B}$ in vertical blocks. Vertical partitioning of $\mathbf{B}$ is also used in the algorithm of Matam et. al. [21] for a hybrid CPU+GPU system.

From a theoretical perspective, Yuster & Zwick [34] show that an optimal algorithm can be achieved by separating the matrix into a dense part and a sparse part, and apply the respective optimal algorithm for each part. Most vendor-provided numerical libaries like Intel MKL [12] and cusparse [35] only provide an implementation for sparse-dense matrix multiplication, and are thus often not directly applicable for large, sparse-sparse matrix multiplications.

C. Matrix Tiling

Many of the related papers [23], [7] proposed a fine-grained matrix-partitioning into small, fixed-size blocks. Vuduc et al. [23] used a variable block structure for spgemv, which is an *unaligned* BCSR, to achieve register-blocking. However, their maximum block size is 3×3 – hence, their focus is rather on microscopic tuning than on high-level tile optimizations. Other work [15] motivate a macroscopic, fixed block size (2048×2048) by arguing that two matrix tiles should fit entirely into main memory, in the context of a disk-based system.

VI. Conclusion

With the increase in data volume and computation effort in many analytical applications, efficient processing of large sparse matrices becomes performance-critical and requires a redesign that goes beyond simple BLAS libraries and low-level tuning of sparse algorithms. In this paper we presented AT MATRIX as an adaptive storage layout for large matrices of any topology. Moreover, we showed how our matrix multiplication operator ATMULT accelerates matrix multiplication by utilizing density estimates, and a cost-based selection of multiplication kernels. Our approach outperformed common multiplication algorithms, similar to those that are still used for example in MATLAB or R, by a factor of up to 6x, while maintaining configurable memory restrictions. Nevertheless, our optimization approach is general and orthogonal to the multiplication kernels. At present, we have not yet made use of several performance tweaks in our custom kernels, and expect further improvement potential by implementing them.

To put it in a nutshell, we presented how methods inspired from database technology can improve linear algebra computations, and took a step into the direction of relieving data scientists from the complexity of the connections between matrix characteristics, algorithmic complexities, optimization and the hardware parameters of their system.

References

[1] C. Constantin, "Principal Component Analysis - A Powerful Tool in Computing Marketing Information," *Bull. Trans. University Brasov*, vol. 7, no. 2, 2014.

[2] M. Hahmann, D. Habich, and W. Lehner, "Modular Data Clustering - Algorithm Design beyond MapReduce," in *EDBT Workshops*, 2014.

[3] W. Liu, T. Wang, and S. Chen, "Regularized Nonnegative Matrix Factorization for Clustering Gene Expression Data," in *BIBM, IEEE*, Dec 2013.

[4] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, ser. Software, Environments, Tools. SIAM, 2011.

[5] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein *et al.*, "MAD Skills: New Analysis Practices for Big Data," *VLDB*, vol. 2, no. 2, Aug. 2009.

[6] P. G. Brown, "Overview of SciDB: Large Scale Array Storage, Processing and Analysis," in *SIGMOD*. ACM, 2010.

[7] M. Boehm, S. Tatikonda, B. Reinwald *et al.*, "Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML," *VLDB*, vol. 7, no. 7, 2014.

[8] D. Kernert, F. Köhler, and W. Lehner, "SLACID - Sparse Linear Algebra in a Column-oriented In-memory Database System," in *SSDBM*. ACM, 2014.

[9] D. Kernert, F. Köhler, and W. Lehner, "SpMacho - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation," in *EDBT*, 2015.

[10] Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*, version 2 ed., University of Minnesota Department of Computer Science and Engineering, Jun. 1994.

[11] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, Sep. 1978.

[12] MKL, intel Math Kernel Library, http://software.intel.com/en-us/intel-mkl.

[13] R. W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," Ph.D. dissertation, University of California, Berkeley, CA, USA, January 2004.

[14] M. A. Patwary, N. R. Satish, N. Sundaram *et al.*, "Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms," in *High Performance Computing*, ser. LNCS. Springer, 2015, vol. 9137.

[15] B. Huang, S. Babu, and J. Yang, "Cumulon: Optimizing Statistical Data Analysis in the Cloud," in *SIGMOD*. ACM, 2013.

[16] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree Using Fractals," ser. VLDB. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[17] V. Markl, *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*, ser. DISDBIS. Infix Verlag, St. Augustin, Germany, 1999, vol. 59.

[18] G. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. IBM, 1966.

[19] R. R. Amossen and R. Pagh, "Faster Join-Projects and Sparse Matrix Multiplications," in *ICDT*. ACM, 2009.

[20] R. Pagh and M. Stöckel, "The Input/Output Complexity of Sparse Matrix Multiplication," in *Algorithms - ESA 2014*, ser. LNCS. Springer, 2014, vol. 8737.

[21] K. Matam, S. Indarapu, and K. Kothapalli, "Sparse Matrix-matrix Multiplication on Modern Architectures," in *Int. Conf. High Perf. Comp.*, ser. HiPC, Dec 2012.

[22] A. Buluç and J. R. Gilbert, "Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments," *SIAM J. Scientific Computing*, vol. 34, no. 4, 2012.

[23] R. W. Vuduc and H.-J. Moon, "Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure," ser. HPCC. Springer, 2005.

[24] G. Schubert, G. Hager, H. Fehske, and G. Wellein, "Parallel Sparse Matrix-Vector Multiplication As a Test Case for Hybrid MPI+OpenMP Programming," ser. IPDPSW. IEEE, 2011.

[25] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse Matrices in Matlab: Design and Implementation," *SIAM J. Matrix Anal. Appl.*, vol. 13, no. 1, Jan. 1992.

[26] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, Nov. 2011.

[27] C. Lameter, "NUMA (non-uniform memory access): An overview," vol. 11, no. 7, p. 40.

[28] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement," in *VLDB*, 2015.

[29] F. Färber, N. May, W. Lehner, P. Groß e, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database – An Architecture Overview." *IEEE Data Eng. Bull.*, vol. 35, no. 1, 2012.

[30] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, ch. 43, pp. 442–446.

[31] A. N. Yzelman and R. H. Bisseling, "Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods," *SIAM J. Scientific Computing*, vol. 31, no. 4, Jul. 2009.

[32] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs," in *Proc. 15th ACM SIGPLAN PPoPP*, ser. PPoPP. ACM, 2010.

[33] N. Bell and M. Garland, "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors," in *Proc. Conf. H. Perf. Comp. Netw., Stor. An.*, ser. SC. ACM, 2009, pp. 18:1–18:11.

[34] R. Yuster and U. Zwick, "Fast Sparse Matrix Multiplication," *ACM Trans. Algorithms*, vol. 1, no. 1, Jul. 2005.

[35] CuSparse, nvidia cusparse Library, http://docs.nvidia.com/cuda/cusparse.