The DynB Sparse Matrix Format Using Variable Sized 2D Blocks for Efficient Sparse Matrix Vector Multiplications with General Matrix Structures

Javed Razzaq, Rudolf Berrendorf, Jan P. Ecker, Soenke Hack, Max Weierstall
Computer Science Department
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Germany
e-mail:{javed.razzaq, rudolf.berrendorf, jan.ecker, soenke.hack, max.weierstall}@h-brs.de

Florian Mannuss
EXPEC Advanced Research Center
Saudi Arabian Oil Company
Dhahran, Saudi Arabia
e-mail: florian.mannuss@aramco.com

Abstract—**The Sparse Matrix Vector Multiplication is an important operation on sparse matrices. This operation is the most time consuming operation in iterative solvers and therefore an efficient execution of that operation is of great importance for many applications. Numerous different storage formats that store sparse matrices efficiently have already been established. Often, these storage formats utilize the sparsity pattern of a matrix in an appropiate manner. For one class of sparse matrices the nonzero values occur in small dense blocks and appropriate block storage formats are well suited for such patterns. But on the other side, these formats perform often poor on general matrices without an explicit / regular block structure. In this paper, the newly developed sparse matrix format DynB is introduced. The aim is to efficiently use several optimization approaches and vectorization with current processors, even for matrices without an explicit block structure of nonzero elements. The DynB matrix format uses 2D rectangular blocks of variable size, allowing fill-ins per block of explicit zero values up to a user controllable threshold. We give a simple and fast heuristic to detect such 2D blocks in a sparse matrix. The performance of the Sparse Matrix Vector Multiplication for a selection of different block formats and matrices with different sparsity structures is compared. Results show that the benefit of blocking formats depend – as to be expected – on the structure of the matrix and that variable sized block formats like DynB can have advantages over fixed size formats and deliver good performance results even for general sparse matrices.**

Keywords–Sparse Matrix Vector Multiplication; SpMV; Blocking; Vector Units; Autotuning

I. Introduction

Sparse matrices arise in many applications of natural science and engineering. The characteristic of sparse matrices is that almost all matrix values are zero and only very few entries (usually less than 1%) have a nonzero value. This sparseness property is used in special storage formats for such matrices to store only / mainly nonzero values along with index information. A performance critical operation on such matrices is the multiplication of a sparse matrix with a dense vector (SpMV) $\vec{y} \leftarrow A\vec{x}$ that may be executed many times, e.g., at each iteration step of an iterative solver. The efficiency of the SpMV operation highly depends on the used sparse matrix format, the matrix structure and how the SpMV operation is implemented and optimized according to the format. Many techniques are known to store a sparse matrix and perform the SpMV operation that take advantage of the nonzero structure of the matrix. One class of formats are block formats. Block storage formats exploit block structures of nonzero elements in a matrix and store dense blocks of values [1]. Block formats have several advantages for efficient SpMV executions. Storing nonzero values together in a block can lead to an improved spatial data locality and, by addressing more than one nonzero value by one index entry, the overall index structure, the memory indirections and the memory bandwidth demand are reduced [2] [3]. Another advantage of block formats is the use of the processor's Single Instruction Multi Data (SIMD) extension [4], i.e., the vector units of a processor. Such a block approach works for dense nonzero block structures in sparse matrices and increases the performance of the SpMV operation significantly, even if explicit zeros are used to fill the blocks [5].

There are two groups of blocking formats: fixed size blocking formats that use the same fixed block size for the whole matrix and variable sized block formats that use the structure of the matrix to build variable sized blocks. The advantages of fixed sized blocking formats are the possibility of optimizing the SpMV for certain, at compile time known, block sizes and the rather simple building of blocks by allowing and storing explicit zeros. The advantages of variable blocking formats are the exploitation of a non-regular matrix structure and the ability to store different sized blocks for a matrix. Furthermore, the two types can be combined with different other optimization techniques, like using bitmaps [6] [7] or relative indexing [8] [9]. There are also some block formats that do not fit in either of these categories or use both techniques [10].

Sparse matrices with an inherent block structure usually arise from a regular 2D / 3D geometry associated with the original problem. Such matrices can certainly benefit from blocking techniques [11]. A question is whether rather general matrices without a clear block structure can also benefit from blocking techniques.

Additionally, for different block sizes different implementations of the SpMV kernel may be optimal. Selecting an implementation among a set of different implementations for a large amount of different block sizes may be a task that can hardly be handled manually, due to the large parameter space. Thus, using an autotuning approach for this task may be beneficial.

The paper is structured as follows. In Section II, an overview on related work is given. In Section III, our own newly developed block format DynB is described, including the description of a low overhead algorithm for block detection, implementation issues of the specific SpMV operation and optimizations. In Section IV an autotuning approach for selecting optimal SpMV kernel implementations for different 2D block sizes is presented for the new format. The following Section V describes the experimental setup. Section VI shows performance results, which compare and evaluate relevant blocking formats on matrices without an explicit block structure. At last, in Section VII a conclusion is given.

II. Related Work

In this section, a comprehensive overview of block formats is given, including formats where blocks are used aside with other optimization techniques. Then, a short overview on further block detection methods is given. At last, autotuning methods are discussed.

As an introduction we start with a brief discussion of rather general sparse matrix formats. The Coordinate format (COO) [12] is the most simple and basic format to store a sparse matrix. It consists of three arrays. For a nonzero value, the value as well as the row and column index is stored explicitly at the i-th position in the three arrays, respectively. The size of each array is equal to the number of nonzeros. The order of values stored is of no concern. The Compressed Sparse Row format (CSR) [13] [12] [14] is one of the most used matrix format for sparse matrices. The index structure in CSR is in relation to COO reduced by replacing the row index for every nonzero value with a single index for all nonzero values in a row. This row index indicates the start of a new row within the other two arrays. As a consequence and in difference to COO, all values in a row must be stored consecutively.

For blocked formats, the Block Compressed Sparse Row format (BCSR) [14] [2] is similar to the CSR format. But instead of storing single nonzero values, the BCSR format stores blocks, i.e., dense submatrices of a fixed size. The matrix is partitioned into blocks of fixed size $r \times c$, where r and c represent the number of rows and columns of the blocks.Then, only submatrices with at least one nonzero element are stored. The optimal block size differs for different matrices and even different processor platforms. Advantages of the BCSR format are a possible reduction of the index structure, possible loop unrolling per block, using vector units through automatic compiler vectorization or using explicit intrinsics [15] and many other low level optimization techniques [16]. However, it may be necessary to store explicit zero values for blocks that are not fully filled with nonzero values. In the worst-case, this could lead to the same index structure as with CSR, but with additional zeros stored for each nonzero value.

The Mapped Blocked Row format (MBR) [6] is similar to the BCSR format. Like BCSR, MBR uses blocks of a fixed size $r \times c$. In addition to BCSR, bitmaps are stored that encode the nonzero structure for each block. An advantage of this bitmap array is, that only actual nonzero values need to be stored in the `values` array, even though filled-in zeros exist. In exchange for the reduced memory use, additional computation time is needed during the SpMV operation.

The Blocked Compressed Common Coordinate format (BCCOO) [17] uses fixed size blocks. It is based on the Blocked Common Coordinate (BCOO) format, which stores the matrix coordinates of a fixed sized block to address the value. BCCOO relies on a `bit_flag` to store information about the start of a new row. By using a bit array instead of an integer, a high compression rate of the index information is archived. One disadvantage of the `bit_flag` array is, that an additional array is needed to execute the SpMV operation in parallel with partition information.

The Unaligned Block Compressed Sparse Row format (UBCSR) [5] [18] removes the row alignment of the BCSR format by adding an additional array. However, this optimization appears to be only applicable to a special set of matrices where blocks occur in a recurring pattern in a row and are all shifted.

The Variable Block Row format (VBR) [5] analyses rows and columns that are next to each other. Their nonzero values are stored in blocks, if they have the identical pattern of nonzero values in a row or in a column. Hereby, only completely dense blocks are stored by VBR. It is possible to relax the analyses of rows and columns by the use of a threshold, which allows VBR to store explicit zeros to build larger blocks [18].

The Variable Block Length format (VBL) [3] [19] [11], which is also referred as Blocked Compressed Row Storage format (BCRS), is likewise similar to the CSR format. But, rather than storing a single value, all consecutive nonzero values in a row are stored in 1D blocks. The blocks of the VBL format do not have a fixed size and only nonzero values are stored. VBL may reduce the index structure depending on the stored matrix, but compared to CSR an additional loop inside the SpMV is required to process all blocks in a row and to proceed all elements in a block.

The aim of the Compressed sparse eXtended format (CSX) [10] is to compress index information by exploiting (arbitrary but fixed) substructures within matrices. CSX identifies horizontal, vertical, diagonal, anti-diagonal and two-dimensional block structures in a pre-process. The data structure, which is used by CSX to store the location information, is based on the Compressed Sparse Row Delta Unit format (CSR-DU) [20]. The advantages of CSX are the index reduction by using the techniques of CSR-DU and, at the same time, the provision of a special SpMV implementation for each substructure. However, implementing CSX seems to be rather complex and to determine appropriate substructures in a matrix may cause perceptible overhead.

The Pattern-based Representation format (PBR) [7] aims to reduce the index overhead. Instead of adding fill-in or relying on dense substructures in a matrix, PBR identifies recurring block structures that are sharing the same nonzero pattern. For each pattern that covers more nonzero values than a certain threshold, PBR stores a submatrix in the BCOO format plus a bitmap, which represents the repeated nonzero pattern. For each of these patterns, an optimized SpMV kernel is provided or generated. Belgin et al. state in their work [7] that it is possible to use prefetching, vectorization and parallelization to optimize each kernel individually. Advantages of PBR are the possibility of providing special SpMV kernels for each occurring block pattern as well as low level optimization for these SpMV kernels.

The Recursive Sparse Blocks (RSB) format [21] [22] aims to reduce the index overhead while keeping locality. By building a quadtree, which represents the sparse matrix, the matrix is recursively divided into four quadrant submatrices, until a certain termination condition is reached. The termination condition for the recursive function is defined in detail by Martone et al. in [23] [24]. The submatrix is stored in the leaf node of the quadtree in COO or CSR format. All nodes before the leaf node do not contain matrix data and are pointers, which build the quadtree.

The Compressed Sparse Block format (CSB) [8] [9] aims to reduce the storage needed to store the location of a value within a matrix by splitting the matrix into huge square blocks. Further, row and column indices of each value are stored relatively to each block. Due to the relative addressing of the values, it is possible to use smaller data types for the row and column index arrays, which leads to an index reduction per nonzero. It is possible to order the values inside the `values` array to get better performance of the SpMV operation. The authors of the original work suggest a recursive Z-Motion ordering to provide spatial locality. The parallel SpMV implementation of CSB uses a private result vector per thread, but the implementation also provides an optimization in case the vector is not required for a block row [8].

In [25] it is shown that finding optimal nonoverlapping dense blocks in a sparse matrix is a NP complete problem. Here, the proof is given by using a reduction of the maximum independent set problem, which is known to be NP complete [26]. Moreover, [25] gives a greedy algorithm for finding 2×2 blocks within a sparse matrix. This algorithm is based on a decision tree for finding only dense blocks, allowing no fill-in within these blocks.

Other methods that can be used for blockfinding are for example the kd-tree [27] and r-tree [28] data structure / algorithm known from spatial databases. Both build (search) tree data structures using spatial location of points or objects within a search region.

Various other publications [29] [30] [31] [32] [33] discuss the use of autotuning approaches, which can be used for a wide range of optimizations. E.g., the selection of format parameters, specific optimization techniques or the selection of the best suited formats. Sophisticated auto-tuning approaches are based on complex models [31] [32] or mathematical and machine learning concepts [33].

In [29] Byun et al. present an auto-tuning framework for optimizing the CSR format, e.g., by fixed-sized blocking. This framework is used to find the optimal blocking for the resulting BCSR format for a given input matrix and used hardware platform.

III. Development of a 2D Variable Sized Block Format

In this section, a newly developed variable sized block format, called DynB, is described. The goal of DynB is, to find rectangular 2D blocks within a matrix to efficiently utilize a processor's vector units for the SpMV. At first, a simple and fast algorithm for the detection of variable sized 2D blocks is introduced. Then, the overall structure of the format is given. Afterwards, the SpMV kernel is presented and at last code optimization techniques are considered.

A. Finding Variable Sized Blocks

As described in Section II, the CSX format uses a sophisticated but time consuming algorithm to find even complex nonzero substructures within the entire matrix. Although the speedup of the SpMV operation may be high, many SpMV operations may be neccessary to compensate the cost of the detection algorithm. In contrast, the VBL format uses a simple and fast algorithm to find just 1D blocks within each row of a matrix. However, the speedup of the SpMV may not be as high as for CSX. For the introduced DynB format a fast algorithm to find rectangular 2D blocks over the entire matrix should be developed. With these 2D blocks, a reasonable runtime improvement for the SpMV operation should be achieved, by using advantages similar to BCSR, while possibly generating less fill-in.

```
Input: $A[\,][\,]$, $T$, $S_{max}$
Output: $B[\,][\,]$
for $i \leftarrow 1, nRows$
  for $j \leftarrow 1, nColumns$
    if $A[i][j] \neq 0 \wedge A[i][j] \notin B$
      $r \leftarrow 1$, $c \leftarrow 1$, $rr \leftarrow 0$, $cc \leftarrow 0$
      $added \leftarrow TRUE$
      while $added$
        $added \leftarrow FALSE$
        $rr \leftarrow r-1$, $cc \leftarrow c-1$
        $search$(next column $n$ with $A[i:i+rr][n] \neq 0$)
        $search$(next row $m$ with $A[m][j:j+cc] \neq 0$)
        if $r*(n+1-j) \leq S_{max} \wedge t(A[i:i+rr][j:n]) \geq T$
          $c \leftarrow n+1-j$
          $added \leftarrow TRUE$
        end if
        if $(m+1-i)*c \leq S_{max} \wedge t(A[i:m][j:j+cc]) \geq T$
          $r \leftarrow m+1-i$
          $added \leftarrow TRUE$
        end if
      end while
      $B \leftarrow B + A[i:i+rr][j:j+cc]$
    end if
  end for
end for
```

Figure 1: Fast Heuristic for the Detection of 2D Blocks.

The algorithm we developed to find 2D block structures of nonzero elements is a fast greedy heuristic. It tries to find possible block candidates that should be as large as possible, even if nonzeros are not direct neighbors, i.e., fill-ins of explicit zeros are allowed up to a certain amount per block. Consequently, a threshold T is used that indicates how dense a block candidate, which has been found by the heuristic, needs to be in order to be stored as a block. That means T is a measure for how many fill-in is allowed in a block. The nonzero density $t(block)$ of a block has to satisfy the relation

$$\begin{aligned} t(block) &= nnz_{block}/blocksize \\ &= nnz_{block}/(nnz_{block}+zeros) \\ &= nnz_{block}/(r*c) \\ &\geq T \end{aligned}$$

where nnz_{block} represents the number of nonzero values in the block and r, c the number of rows, columns of that block.

The algorithm shown in Figure 1 describes a simplified version of the heuristic, which is used to find the blocks in a matrix. The heuristic takes a sparse matrix $A[][]$, the desired threshold T (maximum portion of nonzero values in a block) and a maximum blocksize S_{max} (usually according to the size of the vector units) as an input. It gives the converted blocked Matrix $B[][]$ as output. The algorithm iterates rowwise over the nonzero elements of the original matrix. If a nonzero value of the original matrix is not already assigned to a block, a new 1×1 block will be created. Then this block will be

$$A = \begin{pmatrix} 0 & \boxed{a_1 \quad a_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_3 \quad a_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 \quad a_5 & 0 & 0 & 0 & a_6 & a_7 \\ 0 & 0 \quad a_8 & 0 & 0 & 0 & a_9 & a_{10} \\ a_{11} & 0 \quad 0 & a_{12} & a_{13} & a_{14} & 0 & 0 \\ a_{15} & 0 \quad 0 & a_{16} & 0 & a_{17} & 0 & 0 \\ a_{18} & 0 \quad 0 & a_{19} & a_{20} & a_{21} & 0 & 0 \\ 0 & \boxed{a_{22}} \quad 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
values        =  {a1,a2,a3,a4,0,a5,0,a8,
                   a6,a7,a9,a10,
                   a11,a15,a18,
                   a12,a13,a14,a16,0,a17,a19,a20,a21,
                   a22}
block_start   =  {0,8,12,15,24}
row_index     =  {0,2,4,4,7}
column_index  =  {1,6,0,3,1}
block_row     =  {4,2,3,3,1}
block_column  =  {2,2,1,3,1}
```

Figure 2: The DynB Format storing Matrix A with a Threshold of 0.75.

expanded successively with new columns and rows in each iteration of the while loop. Adding a new column or row means adding the column/row with the next nonzero element and all fill-in columns/rows with zeros that are located between the outermost block column/row and the column/row with the next nonzero. Column/rows are only added to the block if the nonzero density of the block after adding these columns/rows would be large enough. If not enough nonzero elements would be added, i.e., the **if** statements for both column and row fail, the heuristic will finish the block. After all blocks are found, the memory for the DynB data structure is allocated and filled with the actual values and index structure. This data structure is described in the following section.

B. Structure of the Format

The DynB format relies on six arrays. In the `values` array the nonzero values (plus fill-in zeros) are consecutively stored in block order and rowwise within a block. The `block_start` pointer stores the starting position of each block in the `values` array. The `row_index` and the `column_index` store the location of the upper left corner of each block. This is similar to the COO format for single values, but here, fewer indices are stored explicitly, because the indices are used to address a whole block of values. Finally, the `block_row` and `block_column` arrays store the column and row size of each two dimensional block, i.e., the block size is variable. Below, the purpose of the six arrays are described as well as why certain data types were chosen and how many entries they contain:

- `values[nnz+zeros]` : **double** contains the values of the matrix.
- `rowIndex[blocks]` : **int** stores the row index in which a block starts.
- `columnIndex[blocks]` : **int** stores the column index in which a block starts.
- `blockStart[blocks]` : **int** stores the start point of each block inside the `values` array.
- `blockRow[blocks]` : **unsigned char** stores the number of rows a block contains. The unsigned char data type is used because the maximum allowed block size is 64, according to the size of vector units, which means that $blockRow \times blockColumn \leq 64$ must hold.
- `blockColumn[blocks]` : **unsigned char** stores the number of columns a block contains.
- `nonZeroBlocks` : **int** stores the quantity of blocks.
- `threshold` : **float** needs to be set prior to the conversion of a matrix into the DynB format. The threshold needs to be positive and smaller or equal to 1.0 (e.g., $1.0 = 100\%$ nonzero values, $0.5 = 50\%$ nonzero values in a block are allowed).

All data types are choosen as small as possible to reduce memory bandwidth demands, which are critical in SpMV operations. Figure 2 shows in an example how a matrix A is stored using the DynB format.

```
for (int i = 0; i < nonZeroBlocks; ++i){
  //general SpMV for any blocksize
  for (int ii = 0; ii < blockRow[i]; ++ii){
    double s = 0.0;
    int jj =  blockStart[i] + (blockColumn[i]*ii) ;
      for (int j = 0 ; j < blockColumn[i]; ++j, ++jj){
        s += values[jj] * x[columnIndex[i]+j];
      }
      y[rowIndex[i]+ii]+=s;
  }
}
```

Figure 3: SpMV implementation of DynB for general blocks.

C. SpMV Kernel

The SpMV implementation of DynB iterates over the blocks, which have been build before. A general and simplified code version is shown in Figure 3. Initially, beside a generic code version able to handle arbitrary block sizes, we implemented additionally optimized code versions for special and often found block structures (single nonzero values, horizontal and vertical 1D blocks and the general case of all other 2D block sizes). Further block kernels were implemented for the autotuning (see Section IV-A).

D. Code Optimization

It was already shown in [34] that using vector intrinsics to address the vector units of a processor can lead to a performance gain for the SpMV operation. However, with this technique the programmer needs to write code on an assembler level, which can be tedious and error prone. Another approach, which showed good results in [34], is to leave the utilization of vector units solely to the compiler. For the Intel Compiler icc/icpc, automatic compiler vectorization is enabled for the optimization level `-O2` and higher levels [15]. The compiler can use various optimization techniques and auto-vectorize code, where possible. To achieve this, the compiler has to be provided with appropriate information. E.g., by using the `-x` compiler option the information on the target processor architecture / instruction set can be provided [15]. Without this option, the compiler uses a default (older) instruction set that can not utilize abilities of current vector units. A programmer can give the compiler additional hints, e.g., where data can be assumed as aligned, if the compiler is not able to

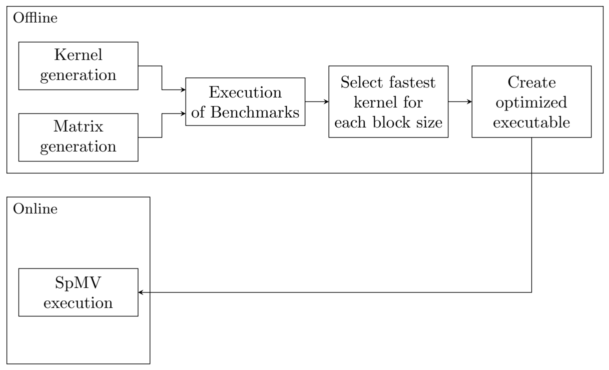


Figure 4: Simplified Process of the DynB Autotuning.

detect this automatically [15]. Furthermore, different `pragmas` exist to force the compiler to execute different actions. With `#pragma simd` a compiler is requested to vectorize a loop even if the compiler ascertains that this is not a good idea. There are many other possible optimizations as well for the blocked SpMV. Loop unrolling can be executed manually by the programmer, if the blocksizes are known beforehand. With `pragma unroll`, the compiler can be forced to unroll a loop. Furthermore, special cases of the blocked SpMV can be considererd, e.g., for 1D blocks, resulting in single loops instead of nested loops.

IV. Autotuning for 2D Blocks

By enabling variable block sizes, many blocks of different sizes may be found. Due to the design decision (and optimization) of using `unsigned char` and therefore 8 Bits to code a block size, the DynB format supports 280 different block sizes. It would be possible to compute the SpMV operation using one general kernel code for arbitrary block sizes that iterates over both dimensions of the blocks (see Figure 3 for such a general code version). This very simple implementation is not expected to deliver the optimal performance, as it contains two additional loops with unknown iteration counts, which can not easily be optimized by a compiler. It is expected that the optimal implementation for each block size is at least slightly different. E.g., the use of vector units may be beneficial for larger, but not for very small blocks, e.g., single values. Additionally, the optimal strategy to handle specific block sizes is processor specific. Finding the optimal implementation for each possible block size manually may be time consuming. Thus, an autotuning approach is used for the DynB format to find offline optimal implementations. The focus of the developed autotuning approach is on the optimization of the SpMV operation for the DynB format, which uses dynamic block sizes.

A. Description of the Autotuning Approach

The developed autotuning approach for the DynB format has similarities to the pOSKI framework [29]. This framework is used for finding the optimal blocking for a given input matrix and used hardware platform. The autotuning approach developed in this work focuses on the identification of optimal implementations for all possible block sizes of the DynB format.

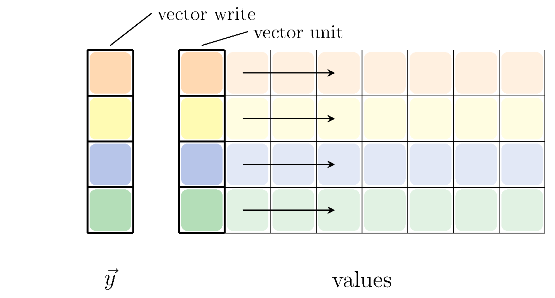
The basic idea of the autotuning approach is to identify the optimal implementation for each block size individually, using a large set of possible implementations and synthetic benchmark matrices with different sparsity pattern. The simplified process of the autotuning is presented in Figure 4. In a first step, the possible SpMV kernels and the benchmark matrices have to be generated. A large set of matrices is thereby created, with each matrix containing only one specific block size. Afterwards, the SpMV is executed using all kernels and the benchmark matrices, while the execution time is measured. The gathered information can then be used to identify the fastest implementation for each specific block size. These implementations are then used to generate an optimized executable, which is used to execute the actual SpMV operation. The complete autotuning is required only once for the specific hardware platform and can be executed offline. This means there is no overhead for the the SpMV operation applied on an actual matrix, caused by the autotuning. In the following, the different steps are explained in more detail.

In the first step, the kernel generation, the required SpMV kernel source code for the for all possible block sizes is generated. Many kernels can be generated automatically, because they follow a fixed pattern. There are additional kernels of relevance, e.g., implementations using intrinsics, that can not easily be generated automatically and therefore hand-tuning is necessary is this cases. The following list shows the set of kernels used for most block sizes:

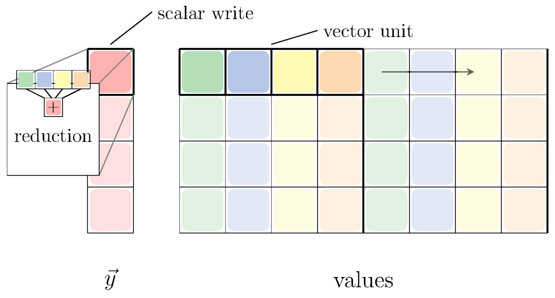
- **normal**: The default kernel, normally used in the general case. Consists of two loops with variable loop count.
- **loop**: Very similar implementation as the **normal** kernel. Instead of variable loop counts, the known block sizes are used as static loop counts.
- **singleLoop**: Special kernel for one dimensional blocks only. Implementation is identical with the **loop** kernel, but only using one of the loops. The other loop is not required, as its iteration count would be 1.
- **unroll**: Identical loop implementation as the **loop** kernel. Additionally the `#pragma unroll` directive is used to generated unrolled code with different unroll factors.
- **novec**: Identical loop implementation as the **loop** kernel. Additionally the `#pragma novector` directive is used to prevent a vectorization of the code.
- **simd**: Identical loop implementation as the **loop** kernel. Additionally the `#pragma simd` directive is used to force vectorization of the code.
- **plain**: The kernel is implemented without any loops. All operations are manually unrolled.
- **intrinsic**: Similar to the **plain** kernel, the kernel is implemented without loops. The calculation is implemented using low level intrinsic functions. The kernels have to be written manually.

In the basic DynB format the elements of every block are stored in row-major order. For the autotuning every kernel is additionally generated for a column-major order organization of the blocks. The creation of the format has been changed as well to allow both block types. This may allow a more efficient vector unit utilization.

Figure 5 shows the usage of vector units with a column-major order and row-major order. In column-major order, vector units are used to process multiple rows at once, which allows a very efficient calculation. Memory accesses can be easily aligned and consecutive. Furthermore, after calculating

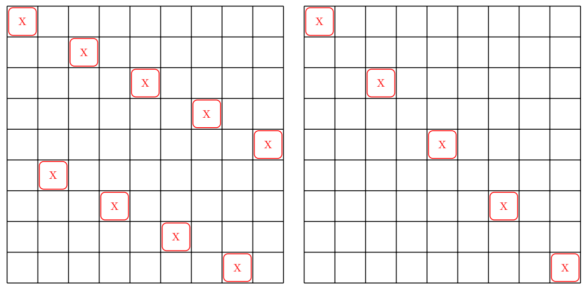


(a) Column-major Order

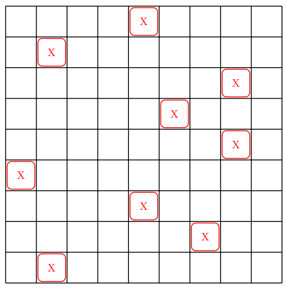


(b) Row-major Order

Figure 5: Calculation of 2D Blocks using Vector-Units.



(a) Round Robin (b) Diagonal



(c) Random

Figure 6: Illustration of the three different structures used for the synthetic matrices of the DynB autotuning. Every red square represents a block in the DynB format.

the last elements of the rows, the result of the vector unit is simply written to the $\vec{y}$ vector.

On the other hand, in row-major order, vector units process elements of the same row. This requires an additional reduction operation at the end of every row, as the partial results of the vector unit lanes have to be combined.

The matrix generation for the benchmark matrices is also done in the first autotuning step. For each block size a set of synthetic benchmark matrices is generated. Thereby, the matrices contain one specific block size only. Furthermore, for each block size three different distributions of the elements and two different nonzero densities are created. This is done to analyze, if the matrix structure has an impact on the individual kernel performance.

Figure 6 presents the three used distributions. In every distribution the blocks are placed with a safety margin around them, to prevent the greedy block finding algorithm of the DynB format to combine multiple blocks into one bigger block. The first structure is called round robin and it distributes the elements evenly over the columns of the matrix. Starting in the first row and column, the blocks are placed in increasing columns. When the end of the matrix is reached, the column index is reset. The resulting patterns resamble squashed diagonals. The second structure is a simple diagonal pattern, because of the safety margin there is not an element in every row. The last structure selects the column index of the blocks randomly. The number of entries per row is still fixed, also the safety margin is still be respected.

The second step of our autotuning uses the generated kernels and matrices and measures the SpMV execution time. Furthermore, for each kernel multiple versions should be used, using different compilers, optimization levels and inlining of code.

In the third step the fastest kernel for each individual block size has to be identified. This can simply be achieved by comparing the measured runtimes of all kernels for one specific block size. Further complexity is introduced by the different matrix structures used, which may require a comparison over a larger data set.

The last step of the offline autotuning is the creation of an optimized executable for the SpMV execution. The kernels identified in the previous step are combined into one large SpMV kernel, to provide a proper implementation for every possible kernel. Further analysis may be required to identify if it is suitable to provide an implementation for every of the 280 possible kernels. For each block in a matrix, the proper kernel has to be selected at runtime. The selection of the kernel has therefore to be very efficient, to prevent excessive branching. This will be described in more detail in the following Section IV-B, where the implementation is described.

The developed autotuning potentially can increase the performance of the DynB format. The default implementation does handle most of the block sizes identically. For small blocks the loop overhead of the general implementation might be too high, while for larger blocks the use of vector units may be beneficial. One possible problem with the use of individual kernels is the introduced branching that is necessary to handle the different sizes. For every block the correct kernel has to be identified and executed, which can potentially slow down the SpMV. Furthermore, the amount of program code could result in problems with the instruction caches. If a large number of different kernel implementations is used, the required code could not fit in the available caches.

Another problem may occur because of the developed autotuning process itself. The initial assumption of the autotuning is, that the performance result of the individual kernels and synthetic matrices can be used to determine the proper kernel for a real matrix. It is also assumed, that the performance numbers of the sequential execution can be used to find the optimal kernels for a parallel executions. It is possible that these assumptions do not hold true, which would result in wrong findings.

B. Implementation

The autotuning approach can be automated using scripts (e.g., we used Python) in combination with the SpMV implementation. As described in the previous section, most of the kernels can be generated automatically. These kernels have one basic pattern, which can be adapted to different block sizes. This is different for some more specialized kernels, which need to be written by hand. One example for this are implementations using intrinsics. The kernel creation scripts take manually written kernels and the general templates to create the source code for the kernels for every block size.

The benchmarking script uses the kernel source code to compile a special version of the DynB SpMV operation. The SpMV operation is executed using the synthetic matrices and the execution time is measured. The results can ce stored, e.g., in a database. This step is repeated several times for every kernel using different compilers and optimization options. Finally, the selection of the fastest kernels and the creation of the optimized executable can also be automated using scripts.

An important part of the implementation is the integration of the optimized block kernels into the SpMV operation of the DynB format. As already discussed in the previous section, 280 different block sizes and kernels are possible, which potentially introduces a lot of additional branching. To handle this efficiently a jumper table should be used either by function pointer arrays generated by a programmer or by a switch case that is handled by the compiler. Many compilers are able to create a jumper table from a switch case if certain limitations are respected, e.g., the number of states are within a certain limit. This behavior has been verified for the Intel compiler, by analyzing the generated assembler code. The analysis also showed, that the optimization can be applied in the case that not only consecutive numerical values are used. In this case, the missing values are filled with jump directives to the default case of the switch case statement.

V. Experimental Setup

The experiments to evaluate block formats were run on a system with an Intel Xeon E5-2697 v3 CPU (Haswell architecture) [35] and the Intel C++ Compiler version 2017 [15]. A set of 78 large test matrices from the Florida Sparse Matrix Collection [36] and SPE reference problems [37] was taken as test matrices. Most of the chosen matrices do *not* have an overall explicit block structure of nonzeros. Compiler optimization and AVX2 instruction set were used, if possible. The following block matrix formats were chosen to be compared in the experiments. They represent a selection of 1D and 2D block formats with fixed and variable sizes as well as more arbitrary pattern (CSX) which have shown to be well performing, at least on matrices with an explicit block structure. In parentheses is shown whether fixed or variable sized blocks can be used.

- DynB (variable): own implementation according to Section III, threshold T varied from 0.55 (slightly more nonzeros than fill-in) to 1.0 (only nonzeros, no fill-in).
- VBL (variable): own implementation according to [3].

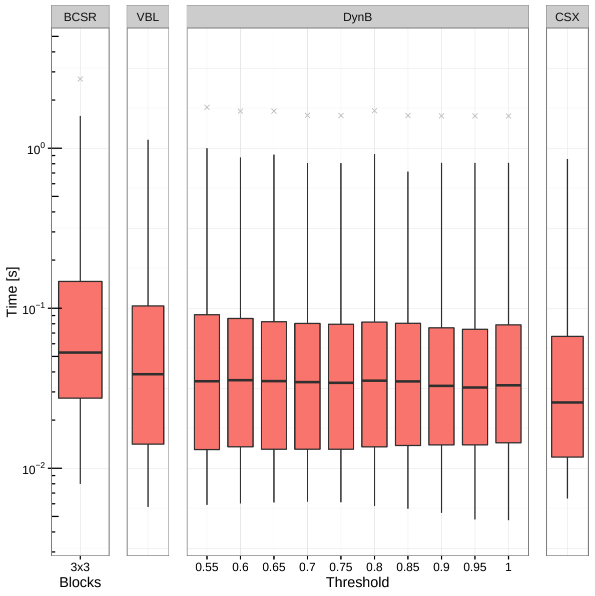


Figure 7: SpMV with all Blocking Formats, Best implementation.

TABLE I: Count of minimal SpMV execution times

BCSR	VBL	DynB	CSX
0	4	32	42

- CSX (variable): library taken from the authors of the original work on CSX [10] [38], no influence on implementation.
- BCSR (fixed): own implementation according to [14], block dimensions: 2×2, 3×3, 4×4

Moreover, the autotuning approach for the DynB format was measured for selected block kernels. The other block kernels were chosen to be optimized by the compiler, giving it in all cases of the switch statement the exact block dimensions as constants. For all experiments, the SpMV operation was executed 100 times and the median of these execution times was taken as the resulting execution time, to exclude uncertainty of the measurements. Subsequently, this is referred to as execution time.

VI. Results

In this section we present selected results of the executed experiments. When boxplots are shown, the quartiles over the results for all matrices are given, whiskers extend to the last datapoint within $1.5 \times \mathit{interquartile\ range}$ and outliers are drawn as points.

A. Comparison of the Formats

Figure 7 shows the execution times of the SpMV for the formats of interest. Different optimizations were applied and the on average best implementation was chosen (for DynB optimizations see Section III-D). For BCSR different blocksizes are possible. In the figure, the results for the 3×3 blocks are shown which have shown the best results over all supported blocksizes.

The base BCSR version showed the weakest performance of all formats. An explanation is the introduced fill-in of

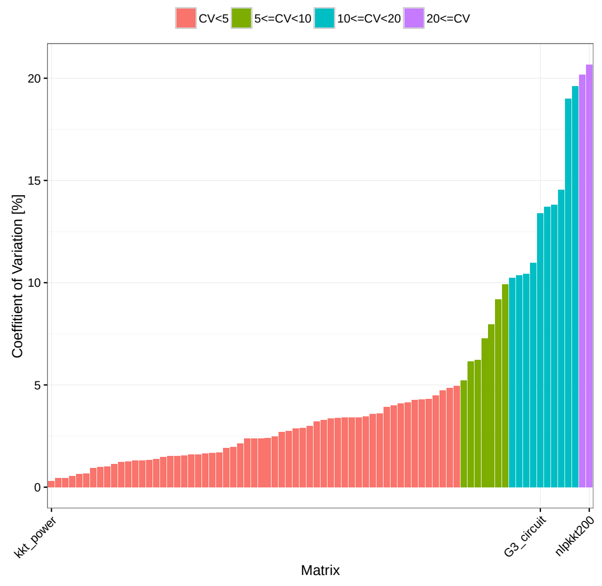


Figure 8: Coefficient of Variation of SpMV with DynB over all Thresholds per Matrix.

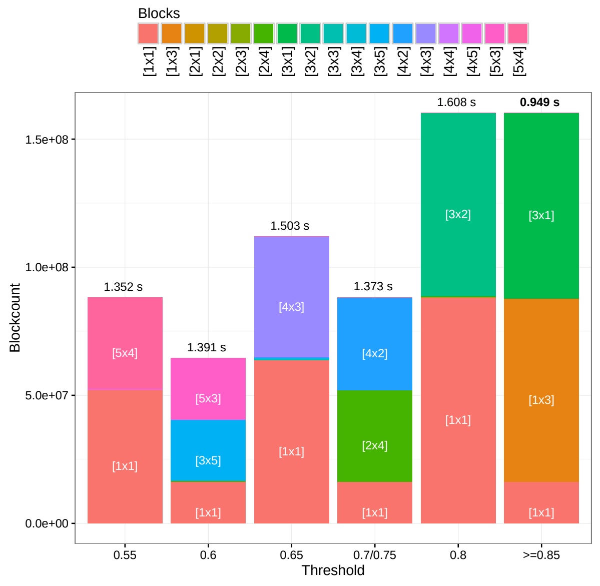


Figure 9: Blocks Found for DynB with Different Thresholds, *nlpkkt200* Matrix.

nonzero values of the format for the fixed 2D block sizes for the rather general matrices used (i.e., without regular block structures). Therefore, a first conclusion is already that the BCSR format should be used only for matrices where an appropriate nonzero pattern exists in the matrix.

Comparing VBL, DynB and CSX shows that these formats are on a similar SpMV performance level. However, the (one-time) creation times for the VBL format were much shorter than the complex detection algorithm for the CSX format, due to the simpler heuristics used in VBL. For the DynB format, there seem to be onyl minor differences dependent on the threshold T.

Table I summarizes the best ranking of the examined formats, i.e., when a format with any setting resulted in the minimal execution times of the SpMV operation. It can be seen that the DynB is the second best format for this setting behind the CSX. However, the algorithm for block detection in the CSX is much more complicated than the detection of the rectangular blocks for the DynB format. Additionally, parallelizing the CSX format is difficult, for example as it is possible that blocks overlap over a row which is circumvented in the creation process of the DynB format. Overlapping blocks in rows computed by different threads requires some form of (costly) synchronization, e.g., atomic operations or reductions on private buffers which can be quite costly.

B. Analysis of the DynB Format

In this subsection the DynB format is analyzed in more depth regarding the influence of the threshold T and the autotuning.

1) Influence of the Threshold T: Here, we begin initially with the standard / base version of the DynB and used compiler optimization level `O3` and the AVX2 instruction set. Figure 8 shows the coefficient of variation of the SpMV execution time for the DynB format for the test matrices, over all thresholds T for the allowable fill-in of a block. It can be seen that, for some matrices varying the threshold T has a significant

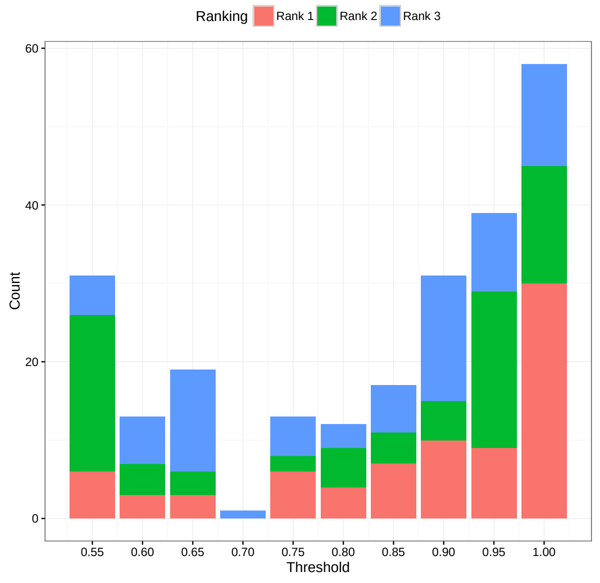


Figure 10: Ranking of DynB Thresholds.

impact on the execution time. This is due to the different blocks that were found by the heuristic. Figure 9 shows the found blocks and their execution times according to the threshold for the example matrix *nlpkkt80*. The class of *nlpkkt* matrices have shown the highest coefficient of variation. It can be seen that, for several different thresholds the same block sizes were found. Consequently, the execution times for the same block sizes do not differ significantly. Moreover, when blocksize 1×1 is predominant (i.e., such blocks consist of a single nonzero value), the execution times are highest. Here, a lot of overhead arises due to the indices that have to be stored for only single values. The best execution times are achieved, when the threshold is higher, i.e., less fill-in occurs, and (for

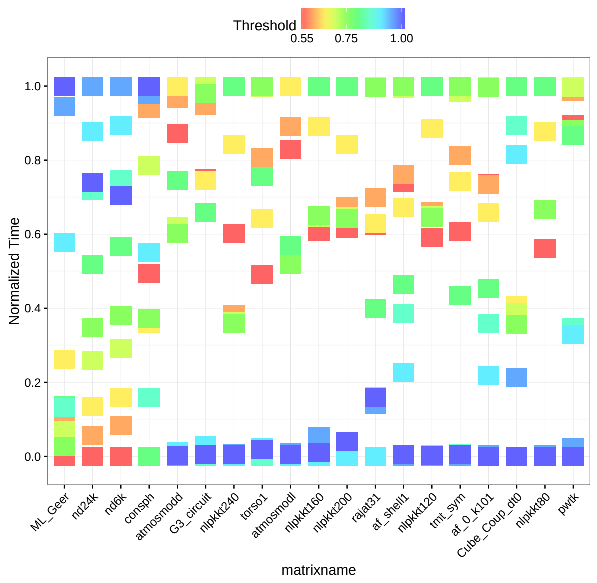


Figure 11: Normalized Times for selected Matrices with DynB, Different Thresholds.

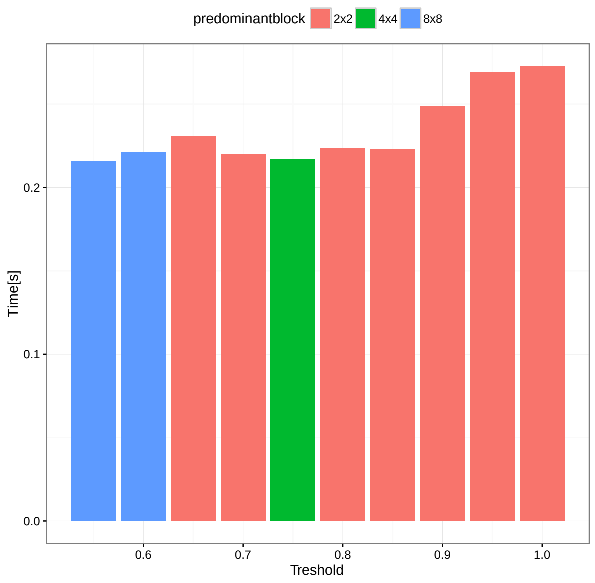


Figure 12: SpMV of ML_Geer for different Threshold.

this matrix) a lot of 1D blocks are found.

For the *G3_circuit* matrix the results are similar (not explicitly shown here), but its coefficient of variation is lower, what can be explained by the lower number of nonzeros, so execution time is primarily lower. The matrix with the lowest coefficient of variance is the *kkt_power*. For this matrix, changing the threshold did not result in different blocks, due to the structure of the matrix. Hence, the execution time was the same for all thresholds.

Figure 10 shows the count of the ranking (rank 1 to rank 3, related to time) of the thresholds across all matrices, i.e., how often a threshold resulted in the fastest, 2nd fastet and 3rd fastest time. Overall, it can be seen that higher thresholds (less or no fill-in) could lead mostly to a good ranking. Medium threshold did not result in a good ranking for the test matrices. However, in some cases, a low threshold (sufficient amount of fill-in) result in better rank counts again.

This is further shown in Figure 11. Here, the normalized times ($Time \in [0.0, 1.0]$) for selected matrices with different structures is given for different thresholds. It can be seen that not for all matrices a higher threshold leads to short execution times of the SpMV operation. For example, the matrices *ML_Geer*, *nd24k* and *nd6k* show the best results with a low threshold (thus more fill-in). Figure 12 shows the absolute results for the matrix *ML_Geer*. With a higher amount of fill-in it is possible to find more larger blocks (8×8 or 4×4). Moreover, when many small blocks are found the use of the AVX2 instruction set is even disadvantegous. With this instruction set the vector size of 4 with the Haswell architecture is assumed. Thus the shortest SpMV execution times can be achieved with different settings, dependent on the threshold.

2) Influence of the Autotuning: The results of the autotuning approach can be seen in Figure 13. Here, the autotuning described in Section IV-A was used. Four different settings are shown:

1) autotuned transposed
2) autotuned untransposed
3) comp. transposed
4) comp. untransposed

Items 1 and 2 are partially autotuned kernels, with 30 autotuned blocks and the rest of the kernels transposed or untransposed and optimized by the compiler (see Section IV-A). Compiler guided (comp.) denotes that the compiler is provided with the constant block sizes per kernel (constant propagation is therefore possible at compile time) and the compiler is thus able to optimize these kernels. The compiler guided version was also executed with transposed and untransposed blocks. It can be seen that all approaches mostly lead to a reasonable speedup. However, the compiler guided optimization, i.e., providing the Intel compiler with the information about (constant) blocksizes at compile time, results in the best speedup, even compared to the code version proposed by autotuning. This is consistent with the results presented in [34], where different optimization setting where examined.

3) Possible Applications: As described in Section I the SpMV is a central operation in iterative solvers, such as Conjugate Gradient (CG) or Generalized Minimal Residual (GMRES) [12]. Finding dense subblocks with the algorithm described in Figure 1 can be used in a one time preprocessing step before executing the actual solver. Then, the SpMV operation can be executed repeatedly with the same matrix within the iterative solver. Matrices where 1×1 blocks are *not* the predominant block size found by the algorithm are most suitable for the use with the DynB format. There are 44 matrices from our testset for which this is true. These matrices mostly arise from 2D / 3D problems with different origin, e.g., *RM07M* (CFD problem) [36], *Serena* (gas reservoir FEM problem) [36], *Geo_1438* (geomechanical problem) [36], *SPE* matrices (reservoir simulation problems) [37]. But there are also matrices that arise from other problems, like *TSOPF_RS_b2383* (power network problem) [36], *nlpkkt* matrices (optimization problems) [36] that are suitable for the DynB format. Spyplots of some of these matrices are given

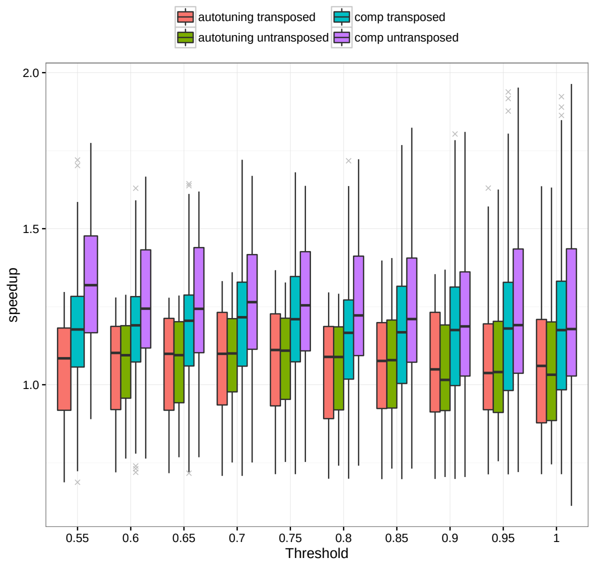
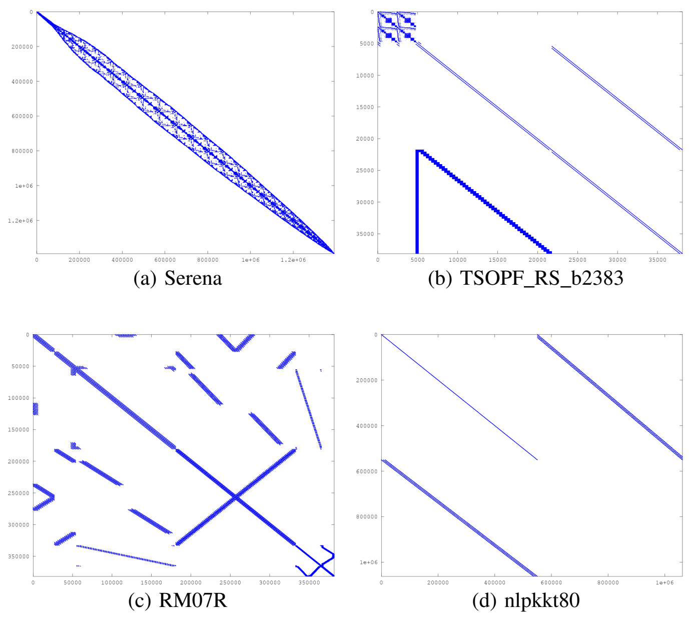


Figure 13: SpMV with DynB Format, Different Optimizations.


(a) Serena (b) TSOPF_RS_b2383

(c) RM07R (d) nlpkkt80

Figure 14: Example Matrices from the Testset.

in Figure 14. It can be seen that the matrices may have very different structures and they also have different properties, such as positive definiteness or symmetry. However, they all have a lot of densely packed regions, i.e., arbitrary grouped nonzero entries that can be exploited by the block finding alogorithm.

VII. Conclusions

In this paper, a new matrix format DynB for storing variable sized 2D blocks was introduced. The aim of the new format is to utilize *any* nonzero block structures in sparse matrices, because dense blocks can be handled efficiently with current and even more future processor architectures. For the DynB format, a simple and fast algorithm for finding blocks of different size and a related implementation of the SpMV operation was presented. Furthermore, several code optimization techniques, such as using vector intrinsics and using autotuning, were examined.

The execution of the SpMV operation on a large set of sparse matrices with different nonzero structures was examined and compared to other known block formats. Here, the formats with variable sized blocks had an advantage over the BCSR with fixed size blocks. For the DynB format, the structure of the matrix can have a significant impact on the dimension of the found blocks and thus on the execution time of the SpMV operation. Moreover, the choice of an appropiate threshold for DynB is dependent on the matrix structure. Several optimization approaches were introduced and combined in an autotuning technique for the DynB format. However, results showed that, when constant propagation is used for the block dimensions for every block kernel, the compiler optimizations showed the best results. Future work on the DynB format will include improvements in finding variable sized rectangular blocks and examining different parallelization techniques.

Acknowledgements

Simon Scholl at Bonn-Rhein-Sieg University helped us in many discussions. We would like to thank the CMT team at Saudi Aramco EXPEC ARC for their support and input. Especially we want to thank Ali H. Dogru for making this research project possible.

References

[1] J. Razzaq, R. Berrendorf, S. Hack, M. Weierstall, and F. Mannuss, “Fixed and variable sized block techniques for sparse matrix vector multiplication with general matrix structures,” in Proc. Tenth Intl. Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2016), pp. 84–90, 2016.

[2] E.-J. Im, K. Yelick, and R. Vuduc, “Sparsity: Optimization framework for sparse matrix kernels,” The International Journal of High Performance Computing Applications, vol. 18, no. 1, pp. 135–158, 2004.

[3] A. Pinar and M. T. Heath, “Improving performance of sparse matrix-vector multiplication,” in Proc. ACM/IEEE Conference on Supercomputing (SC’99), pp. 30 – 39. IEEE, Nov. 1999.

[4] S. Williams et al., “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in Proc. ACM/IEEE Supercomputing 2007 (SC’07), pp. 1–12. IEEE, 2007.

[5] R. W. Vuduc, “Automatic performance tuning of sparse matrix kernels,” Ph.D. dissertation, University of California, Berkeley, 2003.

[6] R. Kannan, “Efficient sparse matrix multiple-vector multiplication using a bitmapped format,” in Proc. 20th International Conference on High Performance Computing (HiPC), pp. 286–294. IEEE, 2013.

[7] M. Belgin, G. Back, and C. J. Ribbens, “Pattern-based sparse matrix representation for memory-efficient smvm kernels,” in Proc. 23rd International Conference on Supercomputing (SC’09), ser. ICS ’09, pp. 100–109. ACM, 2009.

[8] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks,” in Proc. 21th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA’09), pp. 233–244. ACM, 2009.

[9] A. Buluc, S. Williams, L. Oliker, and J. Demmel, “Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication,” in Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS’2011), pp. 721–733. IEEE, 2011.

[10] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, “An extended compression format for the optimization of sparse matrix-vector multiplication,” IEEE Transactions on Parallel and Distributed Systems, vol. 24, no. 10, pp. 1930–1940, Oct. 2013.

[11] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in Proc. 12th IEEE Intl. Conference on Computational Science and Engineerging (CSE-09), pp. 247–256. IEEE, 2009.

[12] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd ed. SIAM, 2003.

[13] ——, "Sparskit: a basic tool kit for sparse matrix computations," http://www-users.cs.umn.edu/~saad/software/SPARSKIT/, 1994, [retrieved: August, 2016].

[14] R. Barrett et al., Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd ed. SIAM, 1994.

[15] User and Reference Guide for the Intel C++ Compiler 17.0, https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide ed., Intel Corporation, 2017, [retrieved: February, 2017].

[16] R. Berrendorf, M. Weierstall, and F. Mannuss, "SpMV runtime improvements with program optimization techniques on different abstraction levels," Intl. Journal On Advances in Intelligent Systems, vol. 9, no. 3 & 4, pp. 417–429, 2016.

[17] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: yet another SpMV framework on GPUs," in Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), pp. 107–118. ACM, 2014.

[18] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in Proc. First Intl. Conference on High Performance Computing and Communications (HPCC'05), pp. 807–816. Springer-Verlag, 2005.

[19] V. Karakasis, G. Goumas, and N. Koziris, "Performance models for blocked sparse matrix-vector multiplication kernels," in Proc. 38th Intl. Conference on Parallel Processing (ICPP'09), pp. 356 – 364. IEEE, 2009.

[20] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in Proc. 5th Conference on Computing Frontiers (CF'08), pp. 87–96. ACM, 2008.

[21] M. Martone, S. Filippone, S. Tucci, P. Gepner, and M. Paprzycki, "Use of hybrid recursive csr/coo data structures in sparse matrix-vector multiplication," in Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, pp. 327–335. IEEE, 2010.

[22] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, "Assembling recursively stored sparse matrices." in IMCSIT, pp. 317–325, 2010.

[23] ——, "On the usage of 16 bit indices in recursively stored sparse matrices," in Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2010 12th International Symposium on, pp. 57–64. IEEE, 2010.

[24] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, "Utilizing recursive storage in sparse matrix-vector multiplication-preliminary considerations." in CATA, pp. 300–305, 2010.

[25] A. Pinar and V. Vassilevska, "Finding nonoverlapping dense blocks of a sparse matrix," Electronic Transactions on Numerical Analysis, vol. 21, pp. 107 – 124, 2004.

[26] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., 1979.

[27] J. L. Bentley, "Finding nonoverlapping dense blocks of a sparse matrix," Commun. ACM, vol. 18, no. 9, pp. 509 – 517, 1979.

[28] A. Guttman, "R-trees: A dynamic index structure for spatial searching," SIGMOD Rec., vol. 14, no. 2, pp. 47 – 57, 1984.

[29] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," EECS Department, University of California at Berkeley, Tech. Rep. UCB/EECS-2012-215, Nov. 2012.

[30] Y. Kubota and D. Takahashi, "Optimization of sparse matrix-vector multiplication by auto selecting storage schemes on GPU," in Proc. Computational Science and Its Applications - ICCSA 2011, vol. 6783, pp. 547–561. Springer-Verlag, 2011.

[31] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in Proc. Principles and Practices of Parallel Programming (PPoPP'10), pp. 115–125. ACM, Jan. 2010.

[32] A. Elafrou, G. I. Goumas, and N. Koziris, "A lightweight optimization selection method for sparse matrix-vector multiplication," arXiv.org, vol. abs/1511.0249, Dec. 2015.

[33] C. Lehnert, R. Berrendorf, J. P. Ecker, and F. Mannuss, "Performance prediction and ranking of SpMV kernels on GPU architectures," in Proc. 22th Intl. European Conference on Parallel and Distributed Computing (Euro-Par 2016), ser. LNCS, P. Dutot and D. Trystram, Eds., no. 9833, pp. 90–102. Springer, 2016.

[34] R. Berrendorf, M. Weierstall, and F. Mannuss, "Program optimization strategies to improve the performance of SpMV-operations," in Proc. 8th Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2016), pp. 34–40. IARIA, 2016.

[35] Intel® Haswell, Intel, http://ark.intel.com/products/codename/42174/Haswell, [retrieved: August, 2016].

[36] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," ACM Trans. Math. Softw., vol. 38, no. 1, pp. 1:1–1:25, Nov. 2010.

[37] SPE Comparative Solution Project, Society of Petroleum Engineers, http://www.spe.org/web/csp/, [retrieved: August, 2016].

[38] V. Karakasis, T. Gkountouvas, and K. Kourtis, CSX library v0.2, https://github.com/cslab-ntua/csx, [retrieved: August, 2016].