Fixed and Variable Sized Block Techniques for Sparse Matrix Vector Multiplication with General Matrix Structures

Javed Razzaq, Rudolf Berrendorf, Soenke Hack, Max Weierstall
Computer Science Department
Bonn-Rhein-Sieg University of Applied Sciences
Sankt Augustin, Germany
e-mail:{javed.razzaq, rudolf.berrendorf, 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—**In this paper, several blocking techniques are applied to matrices that do not have a strong blocked structure. The aim is to efficiently use vectorization with current CPUs, even for matrices without an explicit block structure on nonzero elements. Different approaches are known to find fixed or variable sized blocks of nonzero elements in a matrix. We present a new matrix format for 2D rectangular blocks of variable size, allowing fill-ins per block of explicit zero values up to a user definable threshold. We give a heuristic to detect such 2D blocks in a sparse matrix. The performance of a Sparse Matrix Vector Multiplication for chosen block formats is measured and 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 can have advantages over fixed size formats.**

Keywords–Sparse Matrix Vector Multiplication; Blocking; Vector Intrinsics

I. Introduction

In many fields, such as natural or financial science, computational problems arise, in which the multiplication of a sparse matrix with a dense vector (SpMV) $Ax = y$ is an important operation that may be executed repeatedly [1]. Moreover, the SpMV may also be the most time consuming operation and consequently the bottleneck of such a computational problem. It is therefore desired to optimize the SpMV operation, to solve such a problem faster. 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. One category of sparse matrix formats that has a good optimization potential are block formats. The fundamental idea of block formats for sparse matrices is to exploit the block structure of nonzero elements in a matrix and to store dense blocks of nonzero values. Storing nonzero values together in a block can lead to an improved data locality and, by addressing more than one nonzero value by one index entry, the overall index structure and the memory indirections are reduced [2] [3]. By using a block, the index of a value is reused for the whole block and it is expected that the value will stay in the cache of the CPU or even in a register.

Another advantage of block formats is the possibility of unrolling the SpMV operation and the use of the processor's Single Instruction Multi Data (SIMD) extension [4], i.e., the processor's vector units. This 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, which use the same fixed block size for the whole matrix, and variable sized block formats, which 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 storing explicit zeros. The advantages of variable blocking formats are the exploitation of the matrix structure and the ability to store different sized blocks for a matrix.

Additionally, the two types can be combined with differed 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.

Sparse matrices with an inherent block structure (usually arising from a 2D/3D geometry) can certainly benefit from blocking techniques [10]. A question is, whether rather general matrices, without a clear block structure, can also benefit from blocking techniques.

The paper is structured as follows. In Section II, an overview on related work is given. In Section III, our own newly developed dynamic block format is described, including an algorithm for block determination, the SpMV operation and optimization. In Section IV an experimental setup is shown. In Section V experimental results, which compare and evaluate relevant blocking formats on matrices without an explicit block structure, are presented. At last, in Section VI 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.

The Coordinate Format (COO) [1] is the most simple format to store a sparse matrix. It consists of three arrays. The nonzero values, as well as the row and column index of each value are each stored explicitly in an array. The size of each array is equal to the number of nonzeros.

The Compressed Sparse Row (CSR) [11] [1] [12] format is one of the most commonly 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 row. This row index indicates the start of a new row within the other two arrays.

The Block Compressed Sparse Row (BCSR) [12] [2] format is similar to the CSR format, but instead of storing single nonzero values, the BCSR format stores blocks, i.e., dense submatrices. Only submatrices with at least one nonzero

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-506-7

element are stored. 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. The optimal block size differs for different matrices and different platforms. Advantages of the BCSR format are: possible reduction of the index structure, possible loop unrolling per block, using vector units through intrinsics [13] and many other low level optimization techniques [14]. 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 (MBR) [6] format is similar to the BCSR format. Like BCSR, MBR uses blocks of a fixed size $r \times c$. In addition to BCSR, bitmaps that encode the nonzero structure for each block are stored. 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 (BCCOO) [15] format 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 is archived. One disadvantage of the `bit_flag` array is, that an additional array is needed to execute the SpMV operation in parallel.

The Unaligned Block Compressed Sparse Row (UBCSR) [5] [16] format 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 (VBR) [5] format 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 [16].

The Variable Block Length (VBL) [3] [17] [10] format, which is also referred as Blocked Compressed Row Storage (BCRS) format, 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 an additional loop inside the SpMV is required to proceed through a block.

The aim of the Compressed sparse eXtended (CSX) [18] format 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 (CSR-DU) [19] format. 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 determaning the substructures may cause perceptible overhead.

The Pattern-based Representation (PBR) [7] format 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 optimisation for these SpMV kernels.

The Recursive Sparse Blocks (RSB) [20] [21] format 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 [22] [23]. 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 (CSB) [8] [9] format 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-Morton ordering to provide spatial locality. The parallel SpMV implementation of CSB, uses a private result vector per thread, but also provides a optimization in case the vector is not required for a block row [8].

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 algorithm for the determination 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 (and probably time consuming) algorithm to find 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 a row of the matrix. However, the speedup of the SpMV may not be as high as for CSX. For DynB a simple algorithm to find rectangular 2D blocks over the entire matrix should

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-506-7

Input: $A[\,][\,]$, T, S_{max}
Output: $B[\,][\,]$
1: **for** $i \leftarrow 1, nRows$
2: **for** $j \leftarrow 1, nColumns$
3: **if** $A[i][j] \neq 0 \wedge A[i][j] \notin B$
4: $r \leftarrow 1,\ c \leftarrow 1,\ rr \leftarrow 0,\ cc \leftarrow 0$
5: $added \leftarrow TRUE$
6: **while** $added$
7: $added \leftarrow FALSE$
8: $rr \leftarrow r-1,\ cc \leftarrow c-1$
9: $search$(next column n with $A[i:i+rr][n] \neq 0$)
10: $search$(next row m with $A[m][j:j+cc] \neq 0$)
11: **if** $r*(n+1-j) \leq S_{max} \wedge t(A[i:i+rr][j:n]) \geq T$
12: $c \leftarrow n+1-j$
13: $added \leftarrow TRUE$
14: **end if**
15: **if** $(m+1-i)*c \leq S_{max} \wedge t(A[i:m][j:j+cc]) \geq T$
16: $r \leftarrow m+1-i$
17: $added \leftarrow TRUE$
18: **end if**
19: **end while**
20: $B \leftarrow B + A[i:i+rr][j:j+cc]$
21: **end if**
22: **end for**
23: **end for**

Figure 1: Heuristic for Dynamic 2D Blocks.

$$A = \begin{pmatrix} 0 & a_{01} & a_{02} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{03} & a_{04} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{05} & 0 & 0 & 0 & a_{06} & a_{07} \\ 0 & 0 & a_{08} & 0 & 0 & 0 & a_{09} & a_{10} \\ a_{11} & 0 & 0 & a_{12} & a_{13} & a_{14} & 0 & 0 \\ a_{15} & 0 & 0 & a_{16} & 0 & a_{17} & 0 & 0 \\ a_{18} & 0 & 0 & a_{19} & a_{20} & a_{21} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

`values` $= \{a_{01}, a_{02}, a_{03}, a_{04}, 0, a_{05}, 0, a_{08}, a_{06}, a_{07}, a_{09}, a_{10}, a_{11}, a_{15}, a_{18}, a_{12}, a_{13}, a_{14}, a_{16}, 0, a_{17}, a_{19}, a_{20}, a_{21}, a_{22}\}$
`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.

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.

The algorithm, we developed to find 2D block structures of nonzero elements, is a 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 $t(block) = nnz_{block}/blocksize = nnz_{block}/(nnz_{block} + zeros) = nnz_{block}/(r*c) \geq T$, 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 Fig. 1 describes a simplified version of the heuristic, which is used to find the blocks in a matrix, in pseudo code. 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} (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 original matrix. If a nonzero of the original matrix is not already assigned to a block, a new 1×1 block will be created. Then this block will be 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 the 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 (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]` : `unsingned` **`char`** stores the number of rows a block contains. The unsigned char data type is used because the maximum block size is 64, according to the size of the vector units, which means that $blockRow \times blockColumn \leq 64$.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-506-7

- `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 conversion of a matrix into the DynB format. The thresholds 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).

Fig. 2 shows how a matrix A is stored using the format.

C. SpMV Kernel

The SpMV implementation of DynB iterates over the blocks, which have been build before. It is shown in Fig. 3 in a general and simplified version. Additionally, we implemented optimized code version for some block structures (1×1, $1 \times column$, $1 \times row$ and other block sizes).

D. Optimization

It was shown that, using vector intrinsics, to adress the vector untis of a processor, can lead to a performance gain for the SpMV operation [14]. However, with this technique the programmer needs to write code on an assembler level, which can be teadious and error prone. Another approach, which showed good results in [14], is the use of compiler optimization. With the `fast` and `Ofast` option of the Intel Compiler [13] and GNU Compiler [24], processor specific code that may also adress the vector units efficiently, using the highest available instruction set, can be generated by the compiler. For the Intel Compiler, vectorization is enabled for `O2` and higher levels [13].

IV. Experimental Setup

The experiments to evaluate block formats were run on a system with an Intel Xeon E5-2697 v3 CPU (Haswell architecture) [25] and the Intel C++ Compiler [13]. A set of 111 large test matrices from the Florida Sparse Matrix Collection [26] and SPE reference problems [27] was taken as test matrices. The chosen matrices do *not* have an overall explicit nonzero block structure. The specifications of the matrices were: real, square, more than $5,000,000$ nnz, no graph or model reduction problem, no pattern format. Additionally the matrices *sherman1-5*, *nlpkkt-problems*, *bone010*, *boneS10*, *Cube_Coup_dt0*, *ML_Geer* were used. Compiler optimization and vecotor intrinsics were used, if possible. The following matrix formats were chosen to be compared in the experiments:

- DynB (variable): own implementation according to Section III, with and without intrinsics, 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], with and without intrinsics.
- CSX (variable): library taken from the authors of the original work on CSX [18] [28], no influence on implementation.
- BCSR (fixed): own implementation according to [12], with and without intrinsics, supported block dimensions: 2×2, 3×3, 4×4

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.

```
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.

V. Results

In this section we present selected results of the executed experiments. When boxplots are shown, the quartiles over the results for all 111 matrices are given, whiskers extend to the last datapoint within $1.5 \times interquartile\ range$ and outliers are drawn as points.

Fig. 4 shows the execution times of the SpMV for the implemented formats with different configurations, if possible. For all formats the SpMV was executed with compiler optimization level `O0`, `O3` and `fast` and an implementation using intrinsics (with `fast`). For DynB only the results for selected configurations are presented. For the CSX only one result is presented, because the library settings could not be controlled. Overall it can be seen that, using compiler optimization `O0` (no optimization and vectorization) and `O3` results in slower execution times than using `fast` and intrinsics, which confirms the results found in [14]. This can be explained, because with the compiler option `fast` processor specific code is generated. Hence, with `fast` and intrinsics the CPU specific vector instructions can be used. Moreover, the difference between `fast` and intrinsics for DynB and BCSR seems to be marginal. For DynB, there seem to be hardly any differences dependent on the threshold T, except when looking at the outliers. For BCSR, the compiler with the `fast` option does even a better optimization than handwritten intrinsics. For the VBL format, the intrinsic implementation performed better than the `fast` optimization. Comparing the VBL intrinsics with the CSX shows that these two versions are on a similar level. However, the (one-time) creation times for the VBL format were much shorter than for the CSX format, due to the simpler heuristics used in VBL. For the DynB format the implementation of the heuristic is currently a prototype and needs improvement in runtime. A possible reason for the better performance of the 1D VBL format compared to the 2D formats DynB and BCSR could be, that for 1D blocks there are no jumps within the result vector y of the multiplication $Ax = y$. Thus, 1D blocks may benefit from better spatial locality, while still being large enough to use vector units efficiently.

Fig. 5 shows the coefficient of variation of the SpMV execution time for the DynB format for the 111 test matrices, over all thresholds (optimization `fast`). It can be seen that, for some matrices varying the threshold T has a significant impact on the execution time. This is due to the different blocks that were found by the heuristic. Fig. 6 shows the found blocks and their execution times according to the threshold for the *nlpkkt80* matrix. This matrix has the highest coeffiecient of variation. It can be seen that, for several thresholds the same blocksizes were found. Consequentliy, the execution times for

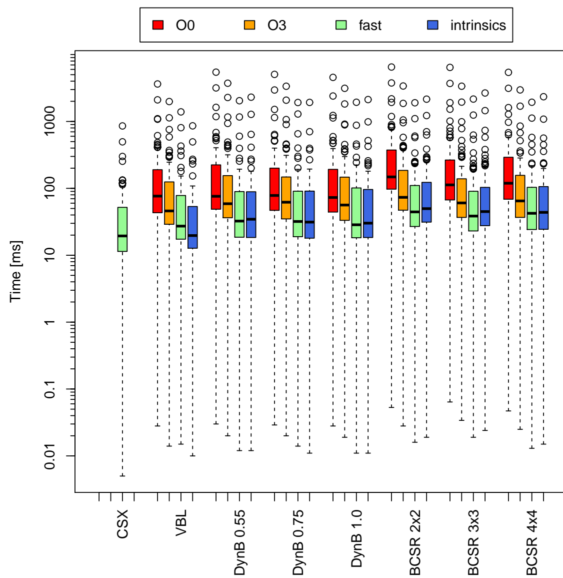

Figure 4: SpMV with all Blocking Formats, Different Configurations.

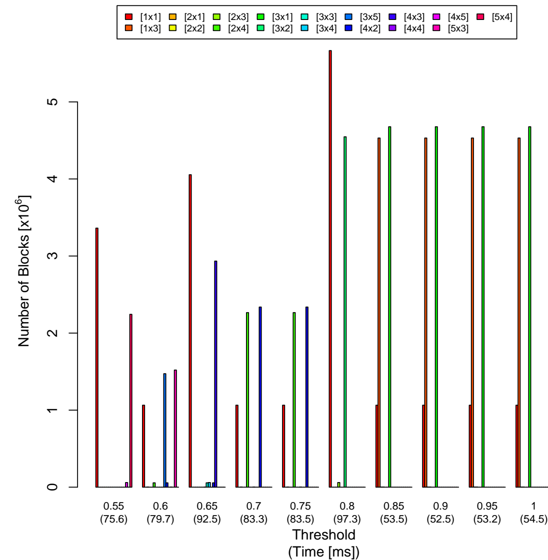

Figure 6: Blocks Found for DynB with Different Thresholds, *nlpkkt80* Matrix.

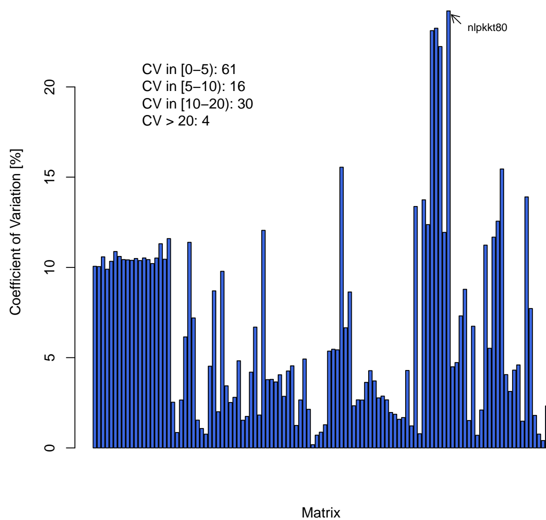

Figure 5: Coefficient of Variation of SpMV with DynB over all Thresholds per Matrix.

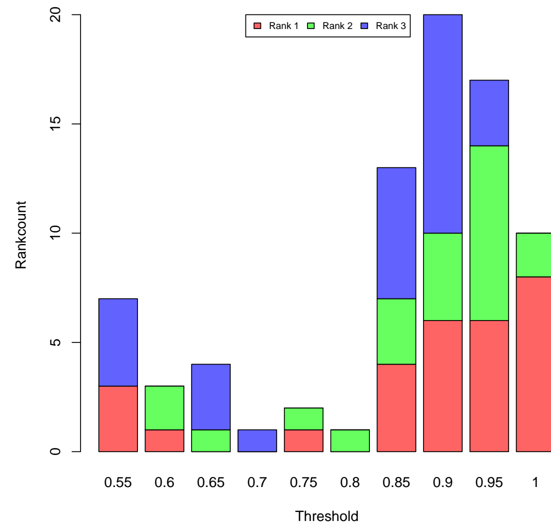

Figure 7: Ranking of DynB Thresholds.

same blocksizes do not differ. Moreover, when blocksize 1×1, i.e., the block consists of a single nonzero, is predominant 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 this matrix) a lot of 1D blocks are found. For the *G3_circuit* matrix the results are similar, 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 varience is the *kkt_power*. For this matrix, changing the threshold did not result in different blocks, due to its structure. Hence, the execution time was the same for all thresholds.

Fig. 7 shows the count of the ranking (rank 1 to rank 3, related to time) of the thresholds across all matrices (optimization `fast`), i.e, how often a threshold resulted in the fastest, 2nd fastet and 3rd fastest time. Overall it can be seen that, a threashold of 0.9 could lead mostly to a ranking. Although a threshold of 0.9 more often lead to rank three, this might be a good indicator that this is a well enough threshold for general use. A threshold between 0.7 and 0.8 did not result in a good ranking for the testmatrices. A lower threshold of 0.55 (adding more fill-in) could, in some cases, result in better rankcounts again.

This is further shown in Fig. 8. Here, the normalized times ($Time \in [0.0, 1.0]$), for selected matrices with different

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-506-7

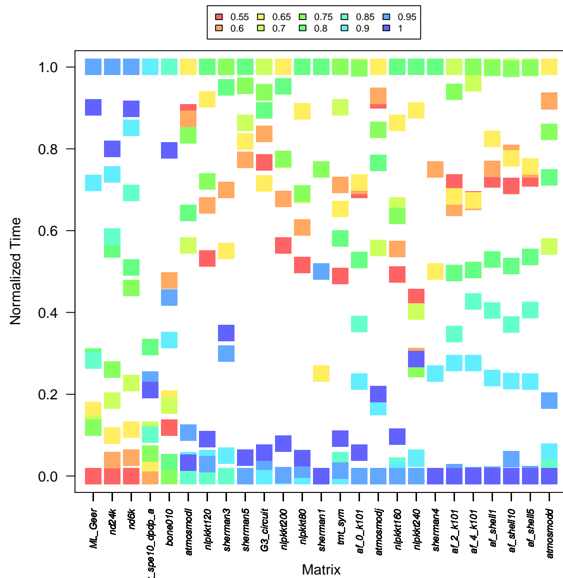

Figure 8: Normalized Times for selected Matrices with DynB, Different Thresholds.

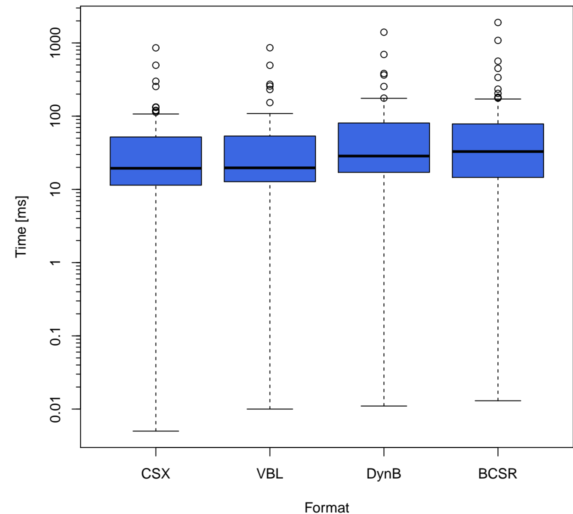

Figure 9: SpMV with all Blocking Formats, Best Results.

TABLE I: Selected DynB Times for ML_Geer

Threshold	Predominant Block	Compileroption	Time [ms]
0.55	8×8	`fast` `O0`	175.2 554.5
0.95	2×2	`fast` `O0`	251.0 513.9

TABLE II: Outliers of SpMV in Fig. 9

Matrix	CSX	VBL	DynB	BCSR
circuit5M				x
Cube_Coup_dt0	x			
dielFilterV2real		x		x
dielFilterV3real		x	x	x
Flan_1565	x		x	
HV15R	x	x	x	x
matrix_spe5Ref_dpdp_a	x			
matrix_spe5Ref_dpdp_b	x			
matrix_spe5Ref_dpdp_c	x			x
matrix_spe5Ref_dpdp_d	x			x
matrix_spe5Ref_dpdp_e	x			
ML_Geer	x			
nlpkkt120				x
nlpkkt160	x	x	x	x
nlpkkt200	x	x	x	x
nlpkkt240	x	x	x	x

structures, is given for different thresholds (optimization `fast`). It can be seen that, not for all matrices a higher threshold leads to short execution times of the SpMV operation. For example, the matrix *ML_Geer* shows the best results with lowest threshold (thus more fill-in). Table I shows the absolute results for this matrix. With a higher amount of fill-in it is possible to find more 8×8 blocks. Looking at the times dependent on compileroptions, it can be seen that, when `fast` (and thus vectorization) is used SpMV is faster when using vectorization. Moreover, the `fast` works better when the 8×8 blocks are predominant. Thus this matrix has shortest SpMV execution times with a threshold of 0.55 and compileroption `fast`. Another interesting fact that can be derived fom Table I is, when `O0` is used the higher threshold with predominant 2×2 blocks is faster than the lower threshold with 8×8 blocks.

Finally, Fig. 9 shows a summary for all formats. Here, only the minimal execution time of a format (over all configurations) is given. It can be seen, that the varible formats perform better than the static BCSR for these matrices without explicit blockstructure. For the variable formats, CSX and VBL perform best. Table II shows the different outliers of the formats. It can be seen that *nlpkkt160*, *nlpkkt200*, *nlpkkt240* and *HV15R* are outliers across all formats. These matrices have the biggest ammount of nonzeros ($> 200,000,000$) in the testset. The other outliers are mostly different between the formats.

VI. Conclusions

In this paper, an overview of different variable and fixed blocking techniques for SpMV was given. Moreover, a new matrix format for storing variable sized 2D blocks, called DynB, was introduced. For this format, a prototype algorithm for finding variable blocks and an implementation of the SpMV operation was presented. Furthermore, several optimization techniques, such as using vector intrinsics, were examined. For this, the execution time of SpMV using DynB and three other blocking formats was measured. Results showed, for a test set of 111 matrices, that using the `fast` option of the Intel Compiler could lead to good results, by effectively using CPU specific vector instructions. Using vector intrinsics with hand tuned code for the use of vector units did not result in better performance compared to just using the compiler option `fast`. Furthermore, variable blocking techniques showed better performance than static blocking techniques for these matrices. 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. Future work on the DynB format will include improvements in finding variable sized rectangular blocks as well as further optimization and parallelization of block handling inside the SpMV operation.

ACKNOWLEDGEMENTS

Jan Ecker and 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] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd ed. SIAM, 2003.

[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, 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.

[11] Y. Saad, “Sparskit: a basic tool kit for sparse matrix computations,” http://www-users.cs.umn.edu/~saad/software/SPARSKIT/, 1994, [retrieved: August, 2016].

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

[13] User and Reference Guide for the Intel C++ Compiler 15.0, https://software.intel.com/en-us/compiler_15.0_ug_c ed., Intel Corporation, 2014, [retrieved: August, 2016].

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] ——, “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.

[23] 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.

[24] GCC, the GNU Compiler Collection, Free Software Foundation, https://gcc.gnu.org/, [retrieved: August, 2016].

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

[26] 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.

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

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

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-506-7