**Hochschule**
**Bonn-Rhein-Sieg**
*University of Applied Sciences*

**Fachbereich Informatik**
*Computer Science Department*

# Master Thesis

## Master Program of Computer Science

# Requirement Analysis and Realization of Efficient Sparse Matrix-Vector Multiplications on Different Processor Architectures

**Jan Philipp Ecker**

**1. Examiner:** **Prof. Dr. Rudolf Berrendorf**
**2. Examiner:** **Prof. Dr. Peter Becker**

**Submitted:** **November 15th, 2016**

## Abstract

The Sparse Matrix-Vector Multiplication (SpMV) operation is a very important building block in high performance computing. Performance improvements are often reached by the development of new sparse matrix formats. In this work a theoretical approach, based on a comprehensive requirements analysis is used to develop new efficient sparse matrix formats. The work thereby considers the currently most important hardware platforms: Intel CPUs, Nvidia GPUs and the Intel MIC platform.
Three new formats are developed: CSR5 Bit Compressed (CSR5BC), Hybrid Compressed Slice Storage (HCSS) and Local Group Compressed Sparse Row (LGCSR). Additionally, an autotuning approach for the existing Dynamic Block (DynB) format is developed. Results show that the new formats achieve different performance on different platforms. The HCSS and LGCSR format perform very well on CPU systems, while CSR5BC is only suitable for the use on a GPU. Furthermore, the HCSS format outperforms all existing formats on the Intel Xeon Phi. The developed autotuning approach does achieve moderate performance increases. Using compiler optimizations, the sequential SpMV performance of the DynB format could be increased significantly.

**keywords: SpMV, HPC, CPU, GPU, MIC, CSR5BC, HCSS, LGCSR, DynB, Autotuning**

# Contents

# 1 Introduction

In many scientific problems systems of linear equations arise, e.g., by discretizing partial differential equations. The matrices which arise from these linear equation systems, are often very large and sparse. Many special matrix formats have been developed to store these matrices efficiently [77].

The Sparse Matrix-Vector Multiplication (SpMV) is an important operation that is required in most iterative solving algorithms for sparse linear systems, e.g., Jacobi, CG, GMRES. Often the major runtime is spent in the SpMV operation. Therefore, a lot of research is done to optimize the SpMV operation [77].

The optimization thereby often relies on the development of new sparse storage formats which allow a faster calculation of the SpMV operation. A wide variety of formats exist, that range from very simple to highly complicated. Over time many different optimization techniques have been developed to for example tackle hardware specific challenges. The most complex formats combine several different optimization techniques.

The most basic idea behind all sparse storage formats is to only store the non-zero elements with some required indexing structures. This drastically reduces the memory demand for storing the matrix and improves the SpMV runtime, as all calculates of zero-values can be omitted. More sophisticated optimizations utilize specific matrix properties like block structures.

Furthermore, the increasing complexity of recent hardware platforms require the development of specialized matrix formats. Many requirements have to be fulfilled to reach good performance when using sparse matrix operations on recent hardware platforms [8]. Even more requirements have to be considered in heterogeneous systems that utilize more than one hardware platform.

The objective of this work is the systematic development of new general purpose sparse matrix storage formats, which are suited for the use on the different hardware platforms. The relevant platforms are Intel Central Processing Units (CPUs) [28], Nvidia Graphics Processing Units (GPUs) [56] and the Intel Many Integrated Core architecture (MIC) [35]. The implementations may benefit by the use of hardware specific features. To achieve this goal a comprehensive requirements analysis should be done to determine the overall and hardware specific requirements. Furthermore an analysis of existing sparse matrix storage formats should be done to identify the currently utilized optimization techniques.

The focus is thereby on the development of general purpose storage formats and the optimization of the SpMV performance. This should be respected in the requirements analysis and in the analysis of existing optimization techniques. Considerations regarding the heterogeneous execution of the SpMV operation, i.e the simultaneous execution on different hardware platforms, are out of scope of this work.

This work should answer the question, whether it is possible to develop efficient sparse matrix formats for the different hardware platforms using the described theoretical approach, based on a requirements analysis. Thereby it is assumed, that not a single format can deliver optimal performance on all relevant hardware platforms.

This work is structured in the following way: Chapter 2 introduces three very basic sparse matrix formats, the SpMV operation and the, for this work, relevant hardware platforms. Furthermore, related work is presented. In Chapter 3 the requirements and constraints for the development of efficient sparse matrix formats are identified with a comprehensive requirements analysis. In Chapter 4 existing format optimization techniques are presented. The development of the new formats is described in Chapter 5. In Chapter 6 the implementation of the developed formats is described in more detail. The performance of the newly developed formats is evaluated in Chapter 7. Finally the findings and results of this work are summarized in Chapter 8.

# 2 Background

In this chapter the concepts and definitions, which are relevant for the understanding of this work, will be explained. The first section defines the term sparse matrix and explains the basic concepts of special sparse matrix storage formats. The following section explains the SpMV operation in detail. Afterwards the relevant parallel hardware platforms for this work are presented. In the last section related work will be discussed.

## 2.1 Sparse Matrices and Basic Sparse Storage Formats

The term sparse matrix is not clearly defined in the literature [22, 77]. In general, a sparse matrix is a matrix containing mostly zero elements. A common definition is, that a matrix is sparse, whenever there is an advantage of using special storage formats that only store the non-zero elements of the matrix. The advantage thereby can be a reduced memory demand of the matrix, but also a reduction of computational effort when using the matrix for mathematical calculations. The mathematical operations, that have to be executed on the matrix, are also important when selecting a sparse storage format [77].

In the following sections the three most basic sparse storage formats will be explained. Knowledge about the utilized concepts is very important for the understanding of the more complex formats discussed later in this work, as these are most often derived from them.

### 2.1.1 Coordinate Format

The Coordinate (COO) format [77] is one of the simplest sparse matrix storage formats. Three vectors are required for storing the matrix in a compressed manner. Two vectors are used for storing the row and column information of each non-zero element of the matrix. The third vector stores the non-zero values itself. Figure 2.1 depicts an example matrix and the resulting data using the COO storage scheme. The term $nnz$ is used to describe the number of non-zeros elements in the matrix. It should also be mentioned, that there is no specific order in which the elements have to be stored in this format. The order showed in the figure is used for presentation purposes only [77].

The memory demand of the format can easily be calculated. Different data types can be used for storing the index data and the non-zero entries itself. In the equation for the memory demand $m_{COO}$ these sizes are denoted as $S_{int}$ and $S_{float}$:

$$m_{COO} = nnz \times (S_{int} + S_{int} + S_{float})$$

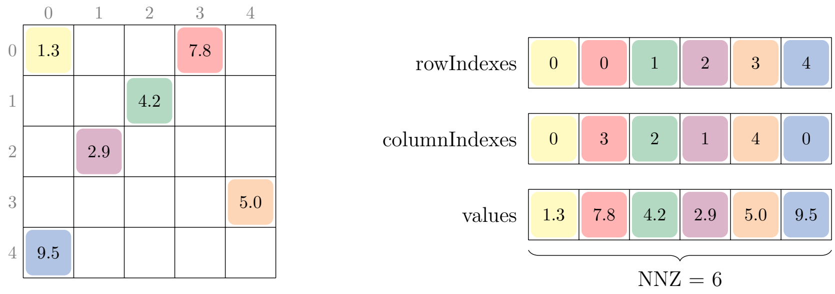


Figure 2.1: Structure of the COO format.

Most often values are stored using double precision, thus $S_{float}$ is defined to be 64 bit. The size of the used data type for the index structures most often depends on the size of the stored matrix. For most matrices the use of a 32 bit wide data type is sufficient. For very large matrices it can be necessary to use a 64 bit wide data type instead. The following equations show the difference in memory demand by using the two different data types:

$$\begin{aligned} m_{COO32} &= nnz \times (32\ bit + 32\ bit + 64\ bit) = nnz \times 128\ bit \\ m_{COO64} &= nnz \times (64\ bit + 64\ bit + 64\ bit) = nnz \times 192\ bit \end{aligned}$$

It can be seen, that the use of the bigger 64 bit data type increases the memory demand significantly.

The biggest advantage of the COO format is its simple structure, which makes it very easy to create. It is also possible to add new elements by expanding the existing vectors (in theory). But the format also has some significant disadvantages. It is not possible to iterate over the rows nor over the columns of the stored matrix, which can prevent efficient parallel implementations of matrix operations. It is also not possible to efficiently search for a specific element of the matrix, as every element can be stored at a any position in the vectors. The COO format has less importance for actual calculations and is more often used as a very simple matrix exchange format.

### 2.1.2 Compressed Sparse Row Format

The Compressed Sparse Row (CSR) format [77] is one of the most commonly used sparse storage formats. Figure 2.2 depicts a matrix represented in the CSR format. Very similar to the COO format it requires three vectors for storing the matrix data. Two are identical to the ones used by COO, one still holds the column indices and the other one is used to store the non-zero values itself. The elements are stored using row-major order, which is important for understanding the structure of the last vector. The n-th element of the vector points to the first element of the n-th row. This allows a very fast access to all elements of a given row. Additionally the vector can be used to calculate the number of non-zero elements per row. This can

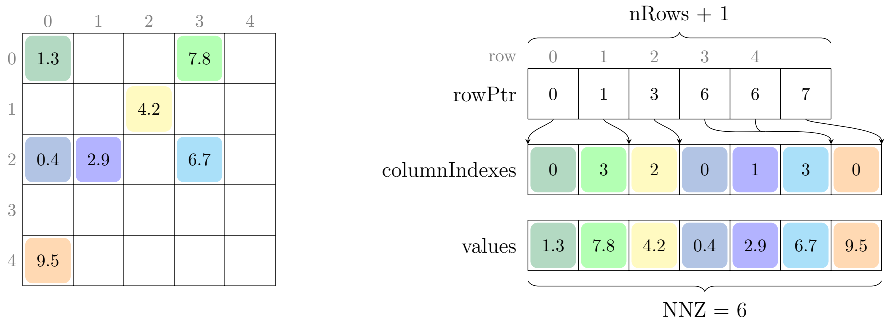


Figure 2.2: Structure of the CSR format.

be done by subtracting the offset of the row from the offset of the next row. The vector contains one additional element at the end, which points directly behind the last element of the last row. Because of this, the number of elements in the last row can also be calculated [77].

The additional variable $N_{rows}$ is required for the calculation of the memory demand for the CSR format. $N_{rows}$ describes the number of rows of the stored matrix. The required memory $m_{CSR}$ for the general case and when using the different data types for the index structures can be calculated as following:

$$\begin{aligned} m_{CSR} &= nnz \times (S_{int} + S_{float}) + (N_{rows} + 1) \times S_{int} \\ m_{CSR32} &= nnz \times (32\ bit + 64\ bit) + (N_{rows} + 1) \times 32\ bit \\ m_{CSR64} &= nnz \times (64\ bit + 64\ bit) + (N_{rows} + 1) \times 64\ bit \end{aligned}$$

It is obvious, that the CSR format is more efficient compared to COO for most matrices, as it can be assumed, that there are much more non-zero elements compared to the number of rows.

The CSR format can be traversed by rows, which allows the efficient parallelization of the SpMV operation (explained in further detail in the next chapter). It also stores the matrix in a more compressed manner compared to the COO format, as the row pointer vector is expected to be much smaller than the row indices vector used by the COO format. The Compressed Sparse Column (CSC) should be mentioned here as well, a very similar storage scheme as CSR. The elements are stored using column-major order instead of row-major order. Instead of column indexes, the row indexes are stored. This allows the CSC format to be traversed by its columns.

### 2.1.3 ELLpack Format

The ELLpack (ELL) [77] format uses a special approach, which allows very efficient calculations on vector machines (e.g., GPUs or vector units in a CPU). Today most often the modified ELLpack-R (ELL-R) [87] format is used, and therefore is described in the following. Figure 2.3 shows a matrix stored in the ELL-R format.

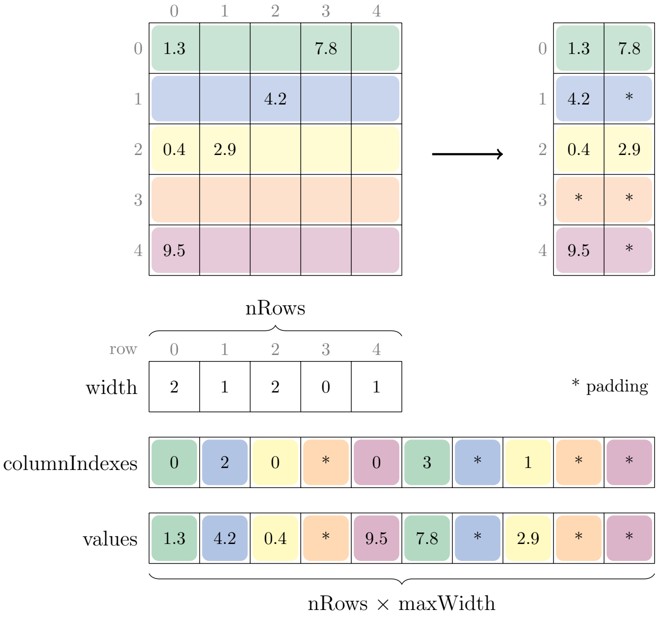


Figure 2.3: Structure of the ELL-R format.

Three vectors a required for storing the matrix data. The first vector stores the number of non-zero elements that exist in each row. The other two vectors are again used to store the column indices and the non-zero values itself. A specific order is used for storing the matrix elements. In a first step the matrix is reduced to its non-zero elements only (see Figure 2.3). The size of the resulting matrix depends on the number of rows and the longest row. All elements are shifted to the left, missing entries are filled with zero, which is called padding or fill-in. The elements of this reduced matrix are then stored in the vectors using column-major order. The only difference of the ELL-R format to the traditional ELL format is the use of the additional width array [77].

To calculate the required memory for the ELL-R format the additional variable *width* is required, which is the number of non-zero elements in the longest row. With identical data types as used before, the memory demand $m_{ELL}$ is calculated as following:

$$\begin{aligned} m_{ELL} &= N_{rows} \times width \times (S_{int} + S_{float}) + N_{rows} \times S_{int} \\ m_{ELL32} &= N_{rows} \times width \times (32\ bit + 64\ bit) + N_{rows} \times 32\ bit \\ m_{ELL64} &= N_{rows} \times width \times (64\ bit + 64\ bit) + N_{rows} \times 64\ bit \end{aligned}$$

It can be seen that the size highly depends on the *width* of the matrix, which can result in very huge data structures, if the matrix structure is disadvantageous (e.g., one very long row). The smallest memory consumption is achieved without any fill-in, which is the case with all rows containing the same number of non-zero elements. In this case $N_{rows} \times width$ basically equals $nnz$, which brings ELL to

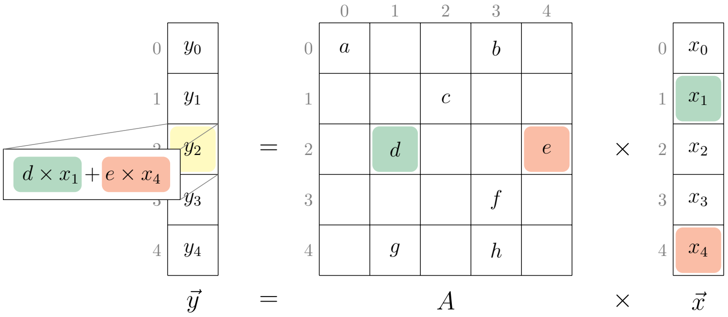


Figure 2.4: Simplified version of the SpMV operation.

nearly the same memory consumption as CSR. This shows, that in most cases the memory consumption of ELL is higher than the of the CSR format.

The rows of the ELL (and ELL-R) format are easily traversable, very similar to the CSR format. The special structure is advantageous especially for GPUs, which benefit from accesses to neighboring memory elements (see Section 2.3.2). One big disadvantage of the format is the required padding. This can lead to a very high memory demand if the matrix structure is disadvantageous. The worst case is a matrix with one fully populated row, which would require to store all elements of the dense matrix.

## 2.2 Sparse Matrix Vector Multiplication

The Sparse Matrix-Vector Multiplication operation is widely used in many different fields of applications. It is commonly required and used in many iterative solvers like CG, GMRES or Jacobi [77]. The SpMV is defined in the BLAS standard [10] as follows:

$$\vec{y} = \alpha \times A \times \vec{x} + \beta \times \vec{y}$$

With $\alpha$ and $\beta$ being scalars, $\vec{x}$ and $\vec{y}$ being vectors and $A$ being a matrix. The calculation of each element $y_i$ of the result vector $\vec{y}$ is defined as:

$$y_i = \sum_{j=1}^{m} \alpha \times a_{ij} \times x_j + \beta \times y_i$$

With $m$ being the number of columns of the matrix. In this work a simplified version of the SpMV operation is used, with $\alpha = 1$ and $\beta = 0$, which is depicted in Figure 2.4. It also depicts the calculation of one element of the result vector $y$. It can be seen in the figure, that zero-value elements of the matrix are ignored in the calculation. Many computational steps can therefore be omitted, which can lead to a significant performance improvement, compared to a dense calculation (which would not ignore zero elements).

## 2.3 Relevant Parallel Hardware Platforms

In this chapter the currently most relevant single node hardware platforms in High-performance computing (HPC) are described as target architectures of this work. Regarding the current TOP500 list [86], the Intel Core processor line [32], or more specific the Intel Xeon series [32] is the dominant CPU platform. In the TOP500 are also some systems using AMD Opteron [2], IBM Power [25] or Sparc [81] processors, but these architectures are currently much less relevant than the Intel Xeon processors. Therefore the focus in this work will be on Intel Xeon CPUs, especially regarding low level optimization.

Many of the systems in the TOP500 list utilize additional accelerators. Nearly all used accelerators are Nvidia Tesla GPUs [63] or Xeon Phi Coprocessors [35]. An insignificant number of systems also utilize AMD/ATI based GPUs [3]. The focus of this work will be on the Nvidia and Intel based accelerator platforms.

It will get obvious in the following sections, that all of these compute platforms are quite complex and can not be replaced by a simple model. A deeper understanding of the concrete architectures, or at least about some specific aspects, is necessary for the requirements analysis in the following chapter.

Following the Intel Core Processors, the Nvidia Tesla GPUs and the Intel Xeon Phi Coprocessor are described in more detail.

### 2.3.1 Intel Xeon Processors

The Intel Xeon processor series is part of the Intel Core product line. The Xeon series is designed for server and HPC applications, while most of the other products of the Intel Core line are supposed for the consumer market.

The latest microarchitecture from Intel is called Skylake. First Skylake EP products, which are relevant for HPC applications, will be available in 2017. The systems listed in the TOP500 list mostly rely on the currently available Haswell architectures and the previous architecture SandyBridge. In the following the Haswell architecture will be explained in more detail. Furthermore the relevant changes in the Skylake EP architecture are mentioned.

The Intel Haswell EP [28] architecture is the 4th generation of Xeon processors. The first products where produced in a 22 nm process, while the newest iteration, called Broadwell, is produced using a 14 nm process. The products implementing this architecture where released between late 2014 and mid 2015. Processors of the Broadwell-EP-line are available with up to 22 cores, which can process up to 44 threads in parallel using hyper threading. The Haswell architecture offers 4 memory channels and every core has 2.5 MB L3 cache, which allows a maximum L3 cache size per CPU of 55 MB for the largest chips. Later iterations of the Haswell architecture for the first time support DDR4 memory, which offers a higher memory bandwidth.

Like most modern CPU architectures, Haswell offers vector units [24, p. 264ff.] in addition to normal arithmetic units. Vector units follow the Single Introduction

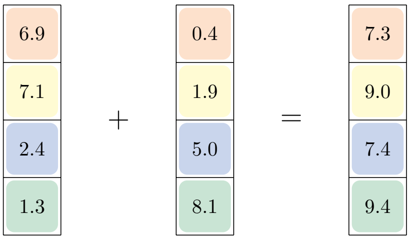


Figure 2.5: Example of a simple vector addition using vector units. All additions are calculated simultaneously.

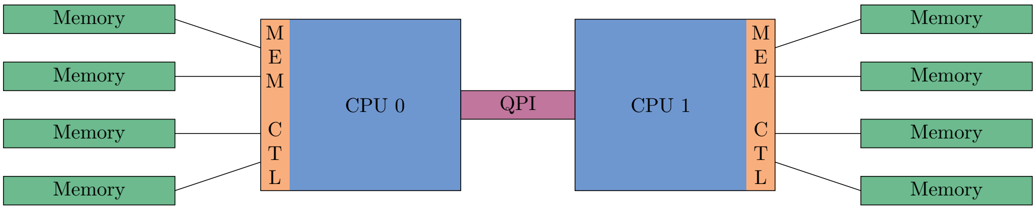


Figure 2.6: Simplified concept of a NUMA based CPU.

Multiple Data (SIMD) principle [24, p. 10], which allows efficient processing of large data sets. Figure 2.5 illustrates the basic concept of vector units. The maximal number of elements that can be processed at once by a vector unit depends on the width of the vector unit and the size of the data type. The Haswell architecture supports Advanced Vector Extensions 2 (AVX2) [31] instructions, which define 256 bit wide operations. This allows the simultaneous calculation of up to 8 single precision (32 bit) or 4 double precision (64 bit) elements. While the older AVX instructions mostly supported floating point operations, AVX2 also support many operations for integer calculations. Haswell also implements the Fused Multiply-Add 3 (FMA3) [26] instructions, which allows a very efficient calculation of problems of the form $a = b * c + d$.

Xeon processors of the EP-lines can be used in dual CPU setups, where they share the same memory space (shared memory systems). In such a configuration they act as Non-Uniform Memory Access (NUMA)-systems [24, p. 346ff.], which is further illustrated in Figure 2.6. In a NUMA based system, a single CPU does not have direct access to the whole system memory. If the required data resides in a memory location that belongs to another CPU, that data has to be transferred through the Intel QuickPath Interconnect (QPI). This results in different memory access times depending on the data location and the processor core accessing the data. The location of data is therefore of relevance for parallel programs, which will be explained in further detail in the next chapter.

The new Skylake architecture offers processors with up to 26 cores in the EP lineup [58]. The cache per core stays the same, which results in up to 65 MB of L3 cache. The number of memory lanes is increased from 4 to 6. New in the Skylake architecture is the optional integration of OmniPath [30] into the processors, a new

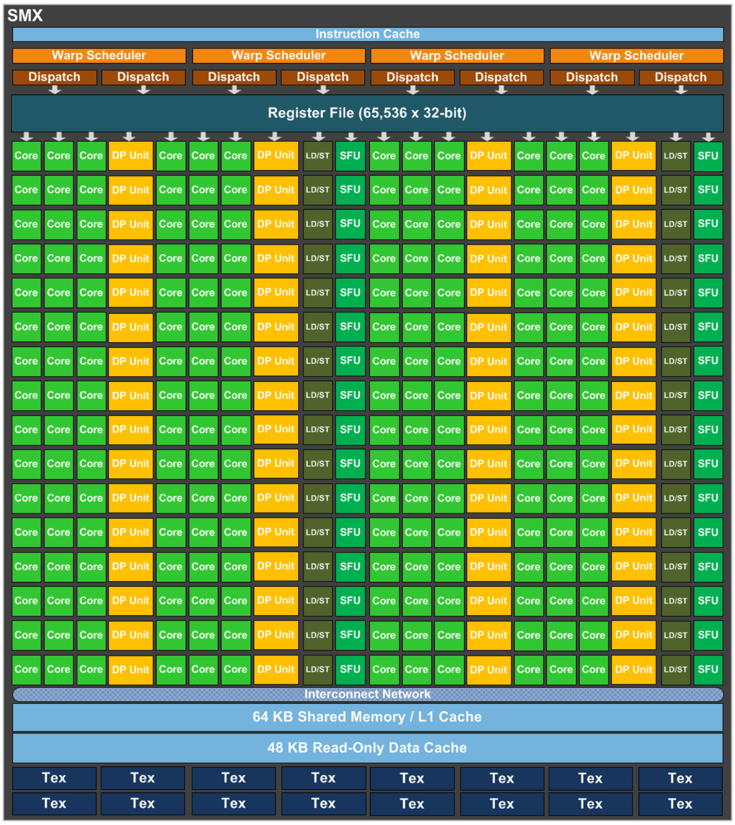


Figure 2.7: Block diagram of a single SMX unit [56].

network technology from Intel. Skylake also supports the new Advanced Vector Extensions 512 (AVX-512) [26] instructions, which support 512 bit wide operations. This doubles the number of elements that can be processed simultaneously, compared to Haswell.

### 2.3.2 Nvidia Tesla GPUs

The Tesla-line is the server and especially HPC segment of Nvidias GPU products. The currently available GPU architecture is Maxwell [60]. It is only used in very specialized segments of HPC, as it is specialized on half-precision calculations and offers only low double precision performance. This section therefore focuses on the previous Nvidia Kepler [59] architecture, which is widely used. The next architecture from Nvidia is Pascal, where first products of the Tesla-lineup should be available late 2016. The most important changes in the Pascal architecture are mentioned at the end of this section.

The architecture of modern GPUs is quite different compared to CPUs [39, p. 2ff.]. While a core of a CPU is optimized for the execution of sequential programs, GPUs are designed for highly parallel workloads on large datasets. The number of cores in a GPU is therefore much higher compared to a CPU, but they are also much smaller and contain less control logic. One important part of the GPU is the Next Generation Streaming Multiprocessor (SMX), which is shown in Figure 2.7. Each SMX has its own L1 cache and scheduling logic, while the biggest part of the SMX consists of cores and Special Function Units (SFUs). All SMX additionally share a common L2 cache. Very similar to the vector units in a CPU, the SMX follows the SIMD principle. On a higher abstraction level the SMX can be compared to a single core of a CPU which only consists of very large vector units. Every SMX can also handle multiple threads at the same time (very similar to hyper threading).

In difference to CPUs, modern GPUs are designed to handle a large amount of threads, which are dynamically scheduled, by a hardware scheduler, on the available hardware resources. The hardware can also switch very efficiently between multiple active threads, which is for example used to hide memory access latencies. The threads are organized in groups of size 32 called a *warp*. All threads of a warp share a common instruction pointer, which means each clock cycle all threads can only execute one common instruction on a specific, usually per thread different, memory location. Simplified this could be seen as a large vector unit. When multiple threads of the same warp follow different execution paths (branching) they diverge and the execution of the threads is serialized [62]. This can have significant impact on the performance of the executed program.

GPUs are designed to process a large amount of data, which resulted in the development of memory with a very high bandwidth. The high bandwidth comes with the drawback of higher memory access latency and some restrictions the programmer has so take care of when developing algorithms for GPUs. Nvidia GPUs coalesce memory transactions to neighboring elements in the memory, which reduces the number of address transactions to the memory. The peak memory bandwidth can only be reached with consecutive memory accesses of all threads [62]. Nvidia GPUs are also developed with weak coherence model, which means that data in the caches is not invalidated by other threads and the programmer again has to take care of synchronizing the cache contents.

The key differences in the upcoming Pascal architecture will be a feature called mixed precision, the use of 3D stacked memory and the introduction of NVLink [66]. The mixed precision mode introduces the possibility to do calculations in half-precision (16 bit) in addition to the typical single and double precision. The 3D stacked High Bandwidth Memory 2 (HBM2) is a new technology that allows a much more compact construction of memory. It allows to move the memory onto the same chip as the GPU which increases memory bandwidth, reduces the latency and improves the energy efficiency. The HBM2 memory thereby completely replaces the GDDR5 memory. NVLink [61] is a new high bandwidth interconnect technology between multiple GPUs or a GPU and the CPU.

### 2.3.3 Intel Xeon Phi Coprocessors

The Intel Xeon Phi is a relatively new hardware platform developed by Intel. It implements the MIC [29] and its first generation is called Knights Corner. The next, second generation, products are called Knights Landing and they start to be

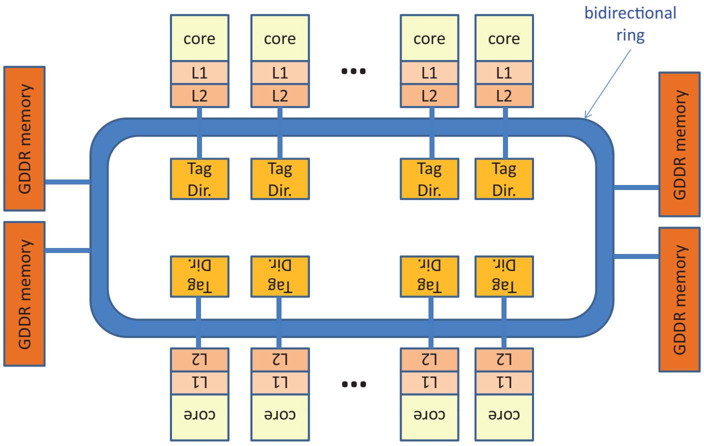


Figure 2.8: Simplified Xeon Phi Knights Corner [9].

available on the market at the time of writing.

As the hardware of the Knights Landing products is not widely available yet the focus will still be on the previous architecture Knights Corner. The first generation Xeon Phi is available as PCIe expansion card and its simplified architecture is depicted in Figure 2.8. The Xeon Phi consist of up to 61 CPU-like cores, which are interconnected by a bi-directional ringbus. Each core has its own private L1 and L2 caches and can process up to 4 hardware threads. The Xeon Phi also provides 512 bit wide vector units, with a hardware specific instruction set (distinct to AVX-512).

The next generation Xeon Phi, code named Knights Landing was introduced in June 2016 [36]. It is available as PCIe accelerator and as socketed version, where it replaces the host CPU. Figure 2.9 shows its completely new architecture. Instead of a ringbus, a 2 dimensional mesh network, consisting of 36 tiles, is used. In addition to the 6 available DDR4 memory channels, the new platform offers 16 GB of Multi-Channel DRAM (MCDRAM). MCDRAM is a high bandwidth and on package memory and comparable to HBM2 from Nvidia. The MCDRAM can be used as additional cache, extension of the DDR4 memory or it can be programmed specifically.

Each tile of the architecture consists of two processing cores, which share 1MB of L2 cache. Every core can handle up to 4 hardware threads and has 2 dedicated Vector Processing Units (VPUs), which implement the AVX-512 instructions. Overall the new Xeon Phi consist of up to 72 cores which can handle up to 288 concurrent hardware threads.

## 2.4 Overall Architecture Comparison

In the previous sections the three relevant hardware platforms for this work have been described. Table 2.1 gives an additional brief overview over the discussed

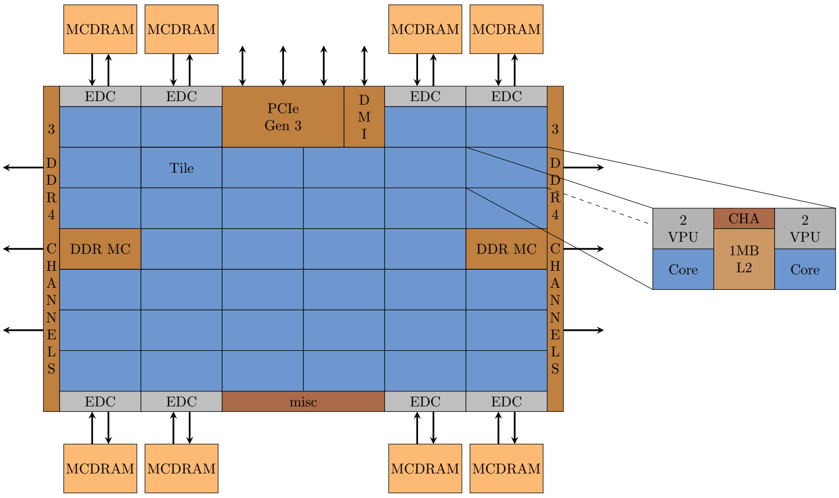


Figure 2.9: Architecture of the Xeon Phi Knights Landing [36].

hardware platforms and generations. For each hardware platform the currently available and the next upcoming hardware generation is listed (except Skylake, as no specific information are available yet).

The table shows clearly the much higher raw performance of GPUs and the new Xeon Phi platform compared to the classical CPUs. Also significant differences in the available memory bandwidth can be seen. One of the biggest changes in the upcoming hardware generations is the use of HBM2 and MCDRAM which highly improve the memory performance. In comparison to the GPUs, CPU-based systems offer much higher memory capacities. The new Xeon Phi Knights Landing architecture allows a moderate amount of memory by using MCDRAM and DDR4 memory at the same time.

Not presented in the table is the single thread performance of the different platforms, which can be expected to be much higher for CPU-based systems. One reason for this is the much lower memory latency which is also not shown in the table. While this may change with the new HBM2 and MCDRAM memory, the CPU architecture is most specialized for sequential workloads. Overall a trend to higher parallelism and increasing vector unit sizes is obvious.

## 2.5 Related Work

The SpMV operation, using various formats on different platforms, is a well researched scientific area. However, there exist only a few publications, that try to analyze the underlying optimization techniques used in the different formats in a comprehensive manner. The work of Shahnez et al. [79] gives a brief overview of existing storage formats and a basic insight into the problems, which are tackled by some specific formats. The publication of Langr [48] also contains an analysis of existing storage formats, but the focus is on space efficiency of very large matrices

| **Architecture (Product)** | **Performance [in GFLOPS]** | **Memory Bandwidth [in GB/s]** | **Max. Memory [in GB]** |
|---|---|---|---|
| Intel Haswell (E5–2699v4) | 634 | 77 | 1536 DDR4 |
| Nvidia Kepler (Tesla K40) | 1430 | 288 | 12 GDDR5 |
| Nvidia Pascal (P100) | 4700 | 720 | 16 HBM2 |
| Knights Corner (7120P) | 1210 | 352 | 16 GDDR5 |
| Knights Landing (7290) | 3470 | 490 MCDRAM<br>115 DDR4 | 16 MCDRAM<br>384 DDR4 |

Table 2.1: Summary of the most important hardware features of the relevant hardware platforms and generations.

that can not be stored in memory, instead of the performance.

The newer publication of Langr et al. [49] describes evaluation criteria for sparse matrix formats. The work also contains a survey of existing matrix format. Some of the presented criteria will be considered in the evaluation of this work.

The work of Böckem [13] was published at the time of writing of this work and provides a comprehensive survey of existing matrix formats. The focus of the work is on the presentation and coarse categorization of the formats. In contrast to this work, no focus was on the optimization techniques of the identified formats.

The work of Koza et al. [43] describes fundamental GPU specific issues when calculating the SpMV operation. Most of these issues are still applicable to other hardware platforms like CPUs. This has some relevance for this work and will be considered in the requirements analysis. In contrast to this work no new matrix format was developed using the findings. Additionally, this work will not focus on a single platform.

The publications [34, 89, 90] describe some possible optimization techniques for the SpMV operation. Most of these are low-level optimizations, which are considered in the following chapters. These publications do not include extensive requirement analysis. The focus is thereby on the programmatically optimization of existing matrix formats and not the development of new sparse storage structures.

Various other publications [12, 14, 17, 47, 50] discuss the use of auto-tuning 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. Autotuning approaches based on complex models [14, 17] or mathematical and machine learning concepts [50] are out of scope of this work. In this work, autotuning is used for finding a improved implementation of the SpMV operation itself. A similar technique is used by Byun et al. [12], who present a autoutuning framework for optimizing the CSR format. This framework is used to find the optimal blocking for the Blocked Compressed Sparse Row (BCSR) format for a given input matrix and used hardware platform. The focus of the autotuning approach developed in this work is on the optimization of the SpMV operation for the Dynamic Block (DynB) format [72], which uses dynamic block sizes. Furthermore, it should be evaluated if the findings of these older publications still hold true compared to optimizations done by recent compilers.

There exist various publications, which present new optimization techniques and sparse storage formats e.g. [38, 42, 44, 51, 53, 85]. They all have in common that the focus is on the presentation of the new format and the used optimization techniques. In none of these publications a comprehensive requirements analysis for the used platforms and the relevant operations is done. The development process of most of these papers could be described as an empirical approach, while the focus of this work is on using a systematic theoretical approach.

# 3 Requirements Analysis

In this chapter the requirements for executing the SpMV as efficient as possible are analyzed, which is done on an abstract level intentionally. The requirements are deduced from the underling problems which have to be identified first. It is also important to mention, that the line between the requirements and solutions is very thin. In this chapter the most abstract requirements should be identified, while solutions to these problems will be developed in the following chapters. There will be a distinction between identified requirements and constraints. While requirements can be fulfilled by specific optimization techniques, the constraints are more general, but still important considerations which have to be respected in the development process of new formats. The rest of this chapter is structured as follows. First the problem of memory boundedness will be discussed, followed by the description of computational related problems and requirements. Afterwards additional general problems will be described. The chapter closes with a short summary of all findings.

## 3.1 Problem of Memory Boundedness

The most important performance constraints of the SpMV operation are related to the memory subsystem. One reason for this is, that the SpMV operation is a memory bandwidth bounded problem. This is stated in many scientific publications [42, 69, 74] and also has been analyzed in more detail [89]. Following a simple model will be developed to further investigate the cause of this constraint.

By calculating the relation between the time for transferring all required data $t_{trans}$ and executing all calculations $t_{calc}$ the limiting factor can be identified. If the time for transferring all required data is bigger than the execution time, the problem is memory bandwidth bounded. This can be expressed as the following equation:

$$\frac{t_{trans}}{t_{calc}} > 1$$

The time $t_{trans}$ is calculated by dividing the amount of moved data $d$ in Byte by the available bandwidth $b$ in Byte per second. Very similar the time $t_{calc}$ can be determined by dividing the number of calculations $c$ in Floating Point Operation (FLOP) by the available processing performance $p$ in Floating Point Operations Per Second (FLOPS):

$$t_{trans} = \frac{d}{b} \qquad\qquad t_{calc} = \frac{c}{p}$$

```
1 for $i = 0$ to $nRows$ do
2     $tmp = 0$
3     for $j = rowPtr[i]$ to $rowPtr[i+1]$ do
4         $tmp \mathrel{+}= values[j] \times x[columnIndices[j]]$
5     end
6     $y[i] = tmp$
7 end
```

**Algorithm 1:** SpMV algorithm for the CSR format.

Overall this results in the following equation, which is separated in two factors on the right hand side. The first factor is the relation between the actual available computational performance $p$ and memory bandwidth $b$. These two variables can be seen as given by the used hardware platform. For this comparison the second factor, which is defined by the amount of moved data $d$ and the number of calculations $c$ is more interesting.

$$\frac{t_{trans}}{t_{calc}} = \frac{p}{b} \times \frac{d}{c}$$

To investigate this in further detail, it is required to determine the amount of moved data as well as the number of executed calculations. The storage scheme of the used format is important when calculating the amount of moved data. The CSR format (see Section 2.1.2) is used here as an example with 32 bit index data and 64 bit double precision values. The result will differ when other formats are used, but it can be assumed that the overall findings for the CSR format also apply for other formats.

The algorithm of the SpMV operation for the CSR format is shown in Algorithm 1. It can be seen, that actual calculations only appear in the inner most loop (line 4). The line contains a multiplication and an addition, which can be handled as a single Fused Multiply-Add (FMA) instruction. As we try to show the memory boundedness we assume the best case regarding number of required operations. So it is assumed that 2 FLOP are executed in each iteration. The inner most loop is repeated once for every non zero element, which means it is overall repeated $nnz$ times. Additional integer operations occur in the heads of the loops, but as they are not processed by the floating point units they are ignored here.

$$c = nnz \times 2FLOP$$

For each iteration of the inner most loop 64 bit or 8 Byte have to be read from $values$. Additionally, 32 bit or 4 Byte are read from $columnIndices$. As the loop is repeated $nnz$ times this leads to the following amount of data being transferred:

$$d_1 = nnz \times (8Byte + 4B) = nnz \times 12B$$

Additional accesses to the $\vec{x}$ vector occur in the inner most loop, but as we try to proof the memory boundedness we assume the smallest amount of data moved and therefore a perfect caching for the vector. This means, each of the $n$ elements of the vector is accessed only once in this model. Additionally for each iteration of the outer loop accesses to the $rowPtr$ array are required. Algorithm 1 was kept simple to allow better readability, which results in two accesses to the $rowPtr$ array in each iteration. The algorithm can be changed so that only one access per iteration is required (see Algorithm 2 in the appendix). Also the $\vec{y}$ vector is accessed once in each iteration. This results in the following amount of data $d_2$ being transferred.

$$d_2 = n \times (4B + 8B + 8B) = n \times 20B$$

This finally results into the following equation. The interesting part is the second factor in the equation, which consists of a sum of a constant 6 and a second summand which depends on the sparsity of the matrix. It can be assumed that the number of rows $n$ is always smaller than the number of non-zeros $nnz$, the value of the second summand is therefore between 10 and 0.

$$\frac{t_{trans}}{t_{calc}} = \frac{p}{b} \times \frac{nnz \times 12B + n \times 20B}{nnz \times 2FLOP} = \frac{p}{b} \times \frac{B}{FLOP} \times (6 + \frac{n}{nnz} \times 10)$$

In the hypothetical case that the available memory bandwidth $b$ of a system is equal to the compute performance $p$ this results into the following equation:

$$\frac{t_{trans}}{t_{calc}} = 1 \times (6 + \frac{n}{nnz} \times 10)$$

The second factor has been shown to be positive and bigger than 6. The relation between $t_{trans}$ and $t_{calc}$ is therefore bigger than 1, which shows the memory bandwidth boundedness of this case. For real systems it can be assumed, that the available memory bandwidth $b$ is much smaller than the available performance $p$. Using the theoretical peak memory bandwidth and compute performance of the Intel Haswell CPU presented in Table 2.1 (see Section 2.4) results into the following equation:

$$\frac{t_{trans}}{t_{calc}} = \frac{634}{77} \times (6 + \frac{n}{nnz} \times 10) = 8,2 \times (6 + \frac{n}{nnz} \times 10)$$

As the first factor increases by the factor of about 8, the equation shows an even stronger memory bandwidth limitation. It can be assumed, that the disproportion between the available memory bandwidth and compute performance will not vanish in near future. In fact it is plausible, that the gap is getting wider. It is therefore shown that the SpMV is, and probably will be in near future, a memory bounded problem.

**Requirement R1** (reduce amount of moved data)**.** The most obvious implication of the memory bandwidth limitation is to reduce the amount of data that has to be moved. This is indeed one of the major reasons for using sparse storage formats in the first place. It was already shown in Section 2.1 that even the very simple formats have a different memory footprint. As previously shown the factor between the amount of moved data and the processing power is quite big. It is therefore very unlikely, that the storage demand of the matrix can be reduced to the point where the problem is no longer memory bound. But than it would be feasible to invest some processing power to further reduce the memory demand of the format. This could for example be done with advanced compression techniques, which has to be investigated in more detail in the following chapters.

**Requirement R2** (allow / improve data reuse of the $\vec{x}$ vector)**.** Another way of reducing the amount of moved data is by data reuse. As all elements of the matrix are only read once, there is no data reuse at all. Each element of the $\vec{y}$ vector has also only to be touched once, if the format allows row based calculations. The $\vec{x}$ vector is therefore the only data structure that can benefit from data reuse. The previously developed model which was used to show the memory boundedness, did assume a perfect data reuse and therefore represents the best case regarding the required memory bandwidth. The model can be extended to consider more realistic caching scenarios. Therefore a new variable *cmr* which describes the cache miss ratio is introduced. The accesses to the $\vec{x}$ vector occur in the inner most loop of the Algorithm 1 and therefore are executed *nnz* times. Taking the *cmr* variable into consideration this results in the following amount of overall moved data:

$$nnz \times 12B + nnz \times 8B \times cmr + n \times 12B$$

The range of the cache miss ratio can be defined as $1 \geq cmr \geq \frac{n}{nnz}$. The upper bound of $cmr = 1$ represents the worst case in which no data could be cached and *nnz* accesses on the $\vec{x}$ vector are required. The lower bound is given by the assumption that every element of the $\vec{x}$ vector is at least accessed once and therefore $n$ accesses to the vector are required.

Valuating the impact of the caching on the overall amount of data moved requires the definition of average number of non-zeros per row. It can be assumed that every row has at least one non-zero element and therefore $n \leq nnz$ is always true. Defining a upper bound for the average number of non-zero elements per row is not that easy as it strongly depends on the type of problem. For simplicity it is assumed here, that the average row contains less than 100 non-zero elements. The bigger *nnz* is compared to $n$, the bigger is the influence of the caching. The upper bound of $nnz = 100 * n$ is therefore used for the valuation of the caching. The minimum and maximum amount of data movement are now determined by the upper and lower bound of the cache miss ratio *cmr*.

The caching has therefore the maximum effect when $nnz = 100 * n$ and $cmr = \frac{n}{nnz}$. When considering best and worst case caching behavior the following amount of data $d_{\min}$ and $d_{\max}$ has to be transferred:

$$\begin{aligned}
d_{\min} &= nnz \times 12B + nnz \times \frac{n}{nnz} \times 8B + n \times 12B \\
&= 100 \times n \times 12B + n \times 20B = n \times 1220B \\
d_{\max} &= nnz \times 12B + nnz \times 1 \times 8B + n \times 12B \\
&= 100 \times n \times 20B + n \times 12B = n \times 2012B
\end{aligned}$$

It can be seen that the overall amount can be reduced from $n \times 2012B$ to $n \times 1220B$ which is about 40% less, when assuming perfect caching. It is therefore worth to consider caching effects in the development process of a format.

**Requirement R3** (improve access latency on the $\vec{x}$ vector)**.** Beside the reduction of the amount of moved data it is also important to consider the access latency to the main memory. Required data has to be requested early enough so that it is available when it is needed. The access latency can be hidden or reduced by using prefetching techniques [32]. Prefetching is used to load relevant data from the main memory into the processors caches before they are actually required. This can reduce the access times significantly, as the access latency to the cache is much lower compared to the main memory. The prefetching can be done in software, but most modern processors also offer hardware prefetchers, which are able to detect simple access patterns and automatically try to prefetch relevant data elements [27]. Software prefetching can be done by the developer, but is also done by the compiler, depending on the used optimization level.

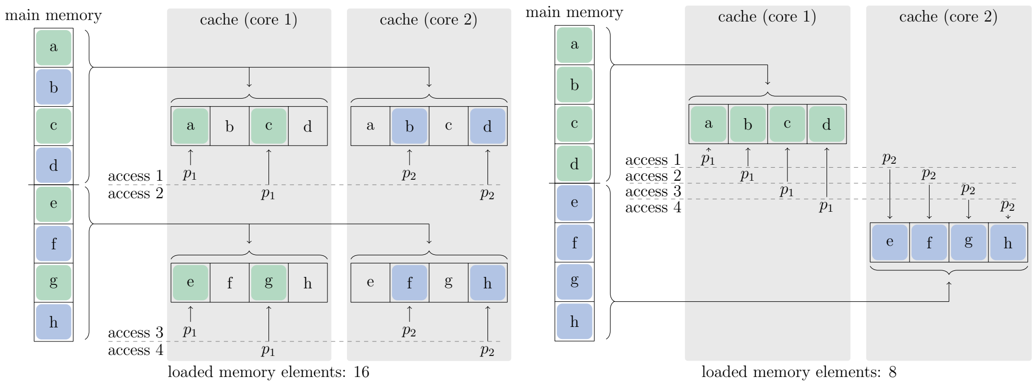
The matrix data is accessed in a continuous manner in most cases (depending on the used storage scheme and SpMV approach). As this is a very simple access pattern it is assumed, that the data can be prefetched efficiently by hardware prefetchers, and is therefore not relevant regarding the access latency. The $\vec{y}$ vector is only written and no reads are required and it is therefore also irrelevant for the access latency as write buffers exist.

The accesses to the $\vec{x}$ vector depend on the structure of the matrix and can be highly irregular. Additionally indirect memory accesses are used. It is unlikely that a hardware prefetcher can successfully prefetch the required data elements.

Research has shown, that the access latency to the L1 and L2 caches on a current CPU based system are in the area of 10 cycles, while main memory accesses require over 150 cycles [8]. Assuming no prefetching or cache reuse at all, this means 15 times higher access times for all accesses on the $\vec{x}$ vector. Some of the access times may be hidden by other hardware features like out of order execution [32]. The out of order execution allows the execution of other, independent, parts of a program, while the system is waiting for data. Overall the memory latency on the $\vec{x}$ vector remains to be a problem.

**Requirement R4** (allow / improve consecutive memory accesses)**.** The access pattern of the different threads in a parallel executed program can have a significant impact on the performance. The most important factor here are cache lines or minimal memory transaction sizes. On all relevant hardware platforms the accesses to the main memory are organized in transactions of a certain minimal size. For the CPU based systems this is the size of a single cache line (64 Byte). When an memory location is accessed, the whole cache line corresponding to this location has to be transferred.

When multiple threads access memory locations in an interleaved pattern this can result in a significant increase of the overall transferred data. This is illustrated by Figure 3.1. Additionally a simple model is used to demonstrate the effect of



(a) interleaved accesses

(b) consecutive accesses

Figure 3.1: Comparison of different access patterns and the related cache behavior.

interleaved memory accesses. Assuming a cache line length in number of memory locations $c$, number of threads $p$ and the number of accessed memory locations $n$ (e.g., length of a vector). Further assuming that all memory locations are accessed in a perfect interleaving pattern and that $n$ is much bigger than $c$ and $p$. Depending on the number of threads and the size of the cache lines there are three possible cases: $c < p$, $c = p$ and $c > p$.

In the first case there are more threads than elements in a single cache line, which means that the first $c$ threads access elements of the same cache line and thus have to load the identical cache line. As this happens to every cache line, the overall amount of transferred data is increased by the factor $c$. The same is true for the second case where the number of threads and elements in a cache line is identical.

In the last case there are more elements in a cache line than there are threads accessing elements. This results in each thread accessing multiple elements of the same cache line. Every thread has to access elements of every cache line, which means that the amount of transferred data is increased by the factor $p$.

The minimum of the variables $c$ and $p$ can be used to express the general case with $d$ being the amount of data transferred: $d = n \times \min(c, p)$. In an optimal case every data element has to be transferred only once which means $d = n$. This shows that a bad data access pattern can significantly increase the amount of data being transferred by the factor $\min(c, p)$.

A proper access pattern is also required to make optimal use of the hardware prefetcher. As already described in the previous Requirement R3, a regular access pattern is required for the hardware prefetcher to work.

Additionally a consecutive access pattern can be advantageous when using vector units, as a more scattered data layout requires the use of gather operations to access all the required data elements. Even though current hardware platforms provide very efficient gather and scatter operations for the available vector units, the performance can increase when they are not required [32].

**Constraint C1** (prevent interleaved write operations on the $\vec{y}$ vector)**.** In the previous requirement the importance of the access patterns for reading operations was mentioned. When taking writing operations into consideration additional effects can occur, which can decrease the performance even further. This considerations are only relevant for the accesses to the $\vec{y}$ vector, as it is the only data structure actually written to.

A very important aspect is the problem of cache line invalidation. In the previous Requirement R4 a simple model was introduced to show the effect of parallel read operations. The same model is used to explain the effect of cache line invalidation. It has been shown, that multiple threads will access the same cache line, when an interleaved access pattern is used. There is no problem when multiple caches hold the same cache line for each of the threads as long as all accesses are read only.

The situation changes as soon as data of the cache line is modified. In every architecture that ensures cache coherence with invalidation based protocols (e.g., CPU based systems), all copies of a cache line have to be invalidated when one thread writes to it. When a cache line is invalidated, it has to be requested again from the main memory upon the next memory access that is related to that cache line. Additionally the next read of that cache line is stalled until the write of the first thread finished, i.e., the cache line is written back to the main memory. This behavior can induce significant latencies which results in stalled threads and performance loss.

The worst case scenario is, multiple threads constantly writing to the same cache line, which results in continuous cache line invalidation and slow synchronization using the systems main memory. Even though this problem is not directly present on current GPU architectures, which implement a weak coherence model, the developer would still have to set manual synchronization points to ensure the coherence. This results in the same effect of slow memory synchronization using the systems main memory.

The access of multiple threads to elements of the $\vec{y}$ vector that reside in the same cache line should therefore be prevented. The data partitioning in the SpMV calculation is important and should consider in which cache lines the elements of the $\vec{y}$ vector reside. Especially interleaving accesses of multiple threads on neighboring rows, by different threads should be prevented.

**Constraint C2** (NUMA-awareness)**.** For shared-memory systems (e.g., dual socket Intel Xeon processors) another memory related constraint exists. In NUMA based systems (see Section 2.3.1) the physical location of data can have a large impact on the performance [34]. Problematic are memory accesses on data that does not reside in the CPU local memory, which increases the access latency significantly. Additionally it is also important to distribute the data between the memory of the different CPUs so that the full memory bandwidth of the system can be utilized [8]. If all data reside in the memory of one CPU only, in a dual CPU setup, the overall available memory bandwidth is halved.

This results in multiple important factors for the SpMV operation. Firstly each thread should work on a fixed set of data, that resides in the local memory of the corresponding CPU core. The threads should therefore also avoid accessing data that correspond to other threads. Another important aspect is the memory allocation, which influences the placement of the data in the memory. Most current systems use a first touch policy [22, p. 186f.] for the actual memory placement of data. In a system with first touch policy the memory placement is decided at the time of the first memory access instead of at the time of the memory allocation. It is therefore important, that the initial write operation after the allocation is done by the correct thread. In the format creation process each thread should write its part of the format data structures to ensure a proper placement in the memory.

**Constraint C3** (prevent synchronized writes on the $\vec{y}$ vector)**.** Another major performance bottleneck can be synchronization. Depending on the used hardware platform and the amount of parallelism, synchronized memory accesses can be very inefficient [8]. Regarding the SpMV operation, this is only relevant for the $\vec{y}$ vector, as this is the only data structure that is actually written to. The COO format is an example where synchronized writes are required, when the SpMV is processed in parallel. This is because the matrix elements are stored without a particular order. When processed in parallel, in a straight forward way, any thread could process elements of any row, which results into a read and write to the $\vec{y}$ vector. As it could not be ensured that two processes try to write to the same element of $\vec{y}$, synchronized writes are required.

The CSR format on the other hand can be calculated without the use of synchronous write operations (see Algorithm 1). This is possible as it can be ensured that one process calculates all partial results of one row and only one write operation per element of the $\vec{y}$ vector is required. Memory synchronization can reduce the memory throughput and should therefore be avoided.

## 3.2 Problem of Computational Unit Utilization

Even though the SpMV operation is memory bound, the computational part can not be ignored. Nowadays available hardware platforms are highly parallel and many aspects have to be considered to reach decent performance. If the hardware is not utilized properly the performance can degrade significantly, possibly enough to reduce the overall SpMV performance. Following the identified relevant requirements regarding the utilization of computational units, that should be considered in the development process, are presented.

**Requirement R5** (allow / improve utilization of vector units)**.** All hardware platforms relevant for this work contain vector units (Xeon and Xeon Phi) or could be described as vector processors (GPUs). It is very important to utilize these vector units to reach decent computational performance. If the vector units are not used, a significant part of the overall functional unit can be idle, degrading the performance significantly. Assuming a vector unit that can process $q$ elements at once, the performance drops by the factor $\frac{1}{q}$. On current CPU based systems $q$ is typically 4 or 8, while it is 32 on most GPU based systems. It is therefore obvious, that a proper utilization of the available vector units is very important.

**Constraint C4** (ensure proper memory alignment)**.** One problem that is also introduced by vectorization is memory alignment [22]. Most vector operations are optimized to work on specific memory boundaries, typically depending on the width of the data loaded or stored by the operation. While it is possible to use unaligned load and store operations, these come with a performance penalty.

For reaching good performance it is therefore necessary to ensure a properly aligned memory allocation, respecting the required memory boundaries. Additionally, even more important, the data has be structured in such way, that the memory accesses in the SpMV operation also respect the boundaries. This for example may require the introduction of additional padding.

**Constraint C5** (prevent branching)**.** Another computation related problem is branching, which happens in programs for example when conditional statements are used. This means the program can, at some point, follow different execution paths. All for this work relevant platforms are sensitive to branching for some degree and for different reasons.

For GPUs, performance decreases if different threads follow different execution paths (see Section 2.3.2). The reason is a serialization of the corresponding part of the program [62]. CPU-based systems suffer performance losses for a different reason [22]. Todays CPUs typically use quite deep pipelines, to allow a high computational throughput with the cost of a higher latency. In case of branching the next instruction is only known, when the condition at the branching point is evaluated. Modern CPUs try to predict the most probable branch and queue the corresponding instructions. In the case that the prediction was correct, no performance is lost. In the other case, the correct instructions have to be queued, which introduces additional latency.

**Requirement R6** (allow / improve load balancing of the SpMV)**.** Since some years there is no increase or even a decrease in processor clock speeds [22, p. 23] This can be explained with reaching physical limitations and because of energy efficiency reasons. The trend of modern processors and accelerators is therefore to increase the parallelism of the processors. One very important issue with highly parallel systems is the load balance of all the running processes. Nearly all parallel programs suffer from load imbalances which lead to waiting times for the processes that finished early. Every time a process is waiting, the related functional units are idle and the reached processing performance can degrade significantly.

It is therefore important to improve the load balance of the SpMV operation as good as possible. Often the format has only a indirect role for the load balancing process, but the format can improve the load balance by an intelligent design. Additionally the data structure can allow and prevent fine-granular load balancing techniques. If for example a format allows row based accesses, a load balancing can be implemented row based. If only accesses to a slice (a group of rows) are possible, the possible load balancing techniques work on much coarser items. The developed format should allow a proper load balancing or even improve it by an intelligent internal data layout.

## 3.3 Additional General Problems

Beside the already mentioned requirements derived from the memory boundedness and the utilization of the computational units, there exist further more practical and general requirements.

**Constraint C6** (allow efficient format creation)**.** The creation of a matrix in a specific format can be very time consuming. Creation thereby often can mean conversion from the CSR or COO format into the specific format. For many formats the creation or most part of the creation can only be done sequentially. Whether the creation time is relevant mostly depends on the specific application. When matrices are created only once and do not change structurally for many SpMV operations, the creation time has only a minor influence on the overall runtime. When the number of iterations is lower or the matrix structure does change, which would require a recreation of the matrix, the creation time has much more importance. As this work will focus on matrices that do not change structurally, this constraint has only minor importance, but still should not be completely ignored.

**Constraint C7** (allow efficient matrix element updates)**.** The ability to update matrix elements is much simpler compared to the previous requirement of matrix creation. The matrix structure is unchanged, and only the values of the matrix elements are changed. One major aspect when updating matrix elements is the process of finding the location where a specific element is stored. This can differ highly for different matrix formats. The COO format for example does not offer any additional internal ordering, which means that any element could be stored at any position inside the format. In the worst case every element has to be searched to find one specific matrix element.

Row wise or column wise traversable formats like the CSR format allow a much more efficient search. As it is possible to directly identify all elements of a specific row or column, only these elements have to be searched, what should be a much smaller search space for most typical sparse matrices.

In practice, many applications are developed using one specific matrix format like CSR. It is often unrealistic to introduce a new, specialized, matrix format to be used in the whole software. A common use-case is the conversion of the matrix into a specialized matrix format only for specific parts of the software, like a linear solver. If this part of the software is executed multiple times for the same matrix, but with different values, it can be advantageous to reuse the previously created matrix structures and only update the matrix elements accordingly. If the position of the non-zero value can be calculated based on the position in the CSR matrix, this can be done very efficiently.

**Constraint C8** (allow asynchronous computation of the SpMV operation)**.** Another requirement is the ability to compute the SpMV operation asynchronously on another platform. Taking modern accelerators like Xeon Phi or modern GPUs into consideration, the ability of executing parts of the SpMV operation asynchronously on an accelerator is very important. Accelerators often come with their own memory and have no direct or only slow access to the hosts main memory. It is therefore mandatory to transfer all required data into the device memory. The computational units are unused while that data is transferred when no asynchronous computation is possible. The overall processing time can therefore be reduced by allowing asynchronous computations which allows the parallel transfer of data while the processing can already be started on other parts of the matrix.

Another additional benefit of asynchronous computations is, that possibly larger problems can be solved. The device memory of the accelerators is often much smaller than the main memory of modern CPU based systems, which can be a limiting factor for the maximum problem size. With asynchronous computations not all data has to be in the device memory at every time, and therefore is the device no longer a bottleneck. Many of the described advantages also apply to distributed systems, where data has to be transferred between multiple computation nodes. This has no further relevance for this work, but could be interesting for further research.

**Constraint C9** (prevent parameters or offer simple heuristics)**.** More complicated formats occasionally offer tuning parameters which allow further hardware and matrix specific optimization (e.g. SELL-C-$\sigma$ [45], ELL-BRO [85]). This can further increase the performance of the SpMV operation, but can also make the use of the format much more complicated. The developed format should be simple to use and should therefore at least offer simple heuristics for potential tuning parameters.

## 3.4 Summary

In this chapter various different requirements and constraints for the development of an efficient sparse matrix storage format have been identified. It is unlikely that all the requirements can be considered in the development as it is assumed that some optimizations may fulfill one requirement but have a negative effect on others. Similarly not all constraints may be considerably. The best suited optimization techniques have to be identified and the negative side effects have to be considered in the development process. All mentioned requirements and constraints are summarized in Tables 3.1 and 3.2 to provide a better overview of all mentioned aspects.

| # | **Short Description** |
|---|---|
| R1 | reduce amount of moved data by reducing the overall memory demand |
| R2 | allow / improve data reuse of the $\vec{x}$ vector |
| R3 | improve access latency on the $\vec{x}$ vector |
| R4 | allow / improve consecutive memory accesses |
| R5 | allow / improve utilization of vector units |
| R6 | allow / improve load balancing of the SpMV |

Table 3.1: Summary of all determined requirements of the SpMV operation.

| # | **Short Description** |
|---|---|
| C1 | prevent interleaved write operations on the $\vec{y}$ vector |
| C2 | NUMA-awareness |
| C3 | prevent synchronized writes on the $\vec{y}$ vector |
| C4 | ensure proper memory alignment |
| C5 | prevent branching |
| C6 | allow efficient format creation |
| C7 | allow efficient matrix element updates |
| C8 | allow asynchronous computation of the SpMV operation |
| C9 | prevent parameters or offer simple heuristics |

Table 3.2: Summary of all determined constraints for the development process.

# 4 Analysis of Optimizations in Existing Formats

In the last chapter the requirements for developing sparse matrix storage formats, that can be used efficiently for calculating the SpMV operation, have been identified. Based on an extensive survey of existing spare matrix storage formats, the focus of this chapter is on the identification of optimization techniques. The investigation thereby primarily includes optimization techniques which are important for the design process of a new sparse matrix format. Low-level optimizations (e.g., loop unrolling, prefetching or the use of processor intrinsics) are not relevant in the design process, as they can be applied in the implementation stage of most formats.

At the time of writing a bachelor-thesis has been published, presenting a comprehensive survey of existing matrix formats [13]. The focus of the work is thereby on the description and categorization of existing matrix formats. The work has been used to identify additional matrix formats, which were not already found in the own survey.

In the following sections different groups of optimization techniques are described with their advantages and possible problems. Also the relation to the identified requirement and constraints is discussed.

## 4.1 Blocking and Pattern Detection

One very early optimization was the approach of blocking and one of the first formats implementing it was the BCSR format [5]. Various different blocking approaches exist which will be explained in more detail in this section. A very simple blocking approach is illustrated in Figure 4.1. It shows the use of two dimensional blocks with a fixed size as they are used in the BCSR format. One major aim of most blocking approaches is the reduction of memory, that is required for storing the index information (Requirement R1). When dense blocks are used, only the coordinates of the block have to be stored and the positions of all non-zero elements can be determined by their position within the block. Another aspect of blocking is cache reuse (Requirement R2) and vector unit utilization (Requirement R5). As a block contains elements with the same or at least very similar column indices the cache reuse of the $\vec{x}$ vector can be improved [34]. Fixed block sizes can possibly also ensure proper memory alignment (Constraint C4). If the size and the structure of a block is known, highly optimized SpMV kernels can be created which allow a very efficient calculation [34].

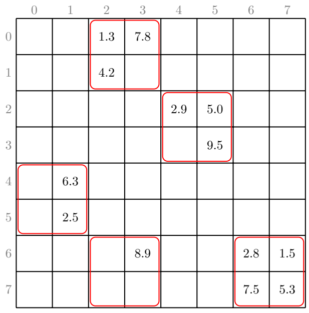


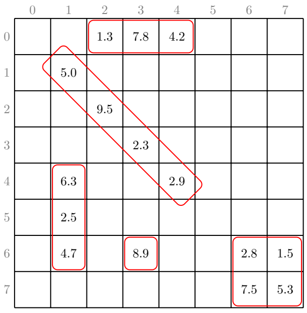
Figure 4.1: Illustration of a simple blocking approach with blocks of fixed size.

The BCSR format, as already mentioned, was one of the first blocked storage formats. It uses blocks of a fixed size and works on fixed alignments only, which means that possible positions of blocks are predefined by the block size. Only those blocks are stored, which contain at least one non-zero element. The alignment restriction allows a very efficient creation of the format (Constraint C6) and also simplifies the SpMV calculation. The main drawback of it is a possibly much higher amount of padding, depending on the matrix structure and block size.

Instead of using a fixed grid for finding and storing the blocks, the Variable Block Row (VBR) format [76] defines a grid depending on the non-zero structure. The matrix is therefore divided in ever smaller parts until all dense blocks are identified. The format creation requires much more effort compared to the BCSR format. Dynamic blocks require a more complex structure of the format, as additional meta information have to be stored. The Variable Block Length (VBL) format [71] is an example for a format that uses a one dimensional blocking. The format stores non-zero elements of continues columns into single blocks of variable length. The publication also discusses the possibility of using reordering to increase the size of the dense sub-structures and therefore increase the efficiency of the blocking. The Unaligned BCSR (UBCSR) format [88] got rid of the alignment restriction of the BCSR format and allows the storage of differently sized blocks.

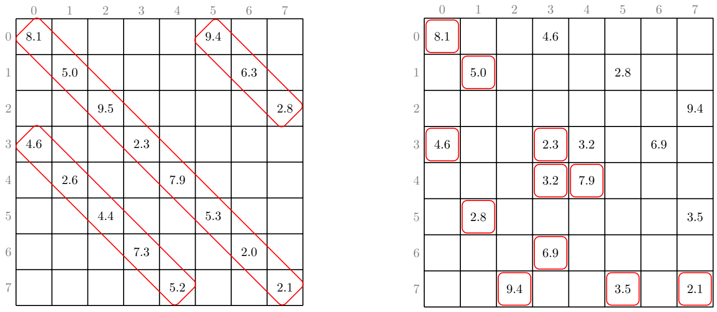
Most of the presented blocked formats are based on the CSR format, but there also existed blocking approaches based on other formats. The ELL based format Blocked ELLpack (BELLpack) [14] was proposed with further optimizations which mostly handle the padding issues introduced by ELL. Yan et al. [92] presented the Blocked Compressed COO (BCCOO) format which is based on the COO format.

Beside the approaches based on simple rectangular block structures, formats have been proposed that take advantage of more complex non-zero patterns. Figure 4.2 illustrates the general concept of pattern based approaches. Pattern based approaches also aim to reduce the overall memory demand of the format (Requirement R1). In difference to simple blocked formats, pattern based approaches can handle matrices with more complex structures. One issue with pattern based formats is the process of identifying patterns, which can have a high computational complexity [38, p. 111ff.]. This potentially leads to longer creation times



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | | 1.3 | 7.8 | 4.2 | | | |
| 1 | | 5.0 | | | | | | |
| 2 | | | 9.5 | | | | | |
| 3 | | | | 2.3 | | | | |
| 4 | | 6.3 | | | 2.9 | | | |
| 5 | | 2.5 | | | | | | |
| 6 | | 4.7 | | 8.9 | | | 2.8 | 1.5 |
| 7 | | | | | | | 7.5 | 5.3 |

Figure 4.2: Illustration of a pattern detection based approach.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 8.1 | | | | | 9.4 | | |
| 1 | | 5.0 | | | | | 6.3 | |
| 2 | | | 9.5 | | | | | 2.8 |
| 3 | 4.6 | | | 2.3 | | | | |
| 4 | | 2.6 | | | 7.9 | | | |
| 5 | | | 4.4 | | | 5.3 | | |
| 6 | | | | 7.3 | | | 2.0 | |
| 7 | | | | | 5.2 | | | 2.1 |

(a) diagonal matrix

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 8.1 | | | 4.6 | | | | |
| 1 | | 5.0 | | | | 2.8 | | |
| 2 | | | | | | | | 9.4 |
| 3 | 4.6 | | | 2.3 | 3.2 | | 6.9 | |
| 4 | | | | 3.2 | 7.9 | | | |
| 5 | | 2.8 | | | | | | 3.5 |
| 6 | | | | 6.9 | | | | |
| 7 | | | 9.4 | | | 3.5 | | 2.1 |

(b) symmetric matrix

Figure 4.3: Illustration of the utilization of diagonal and symmetric properties.

(Constraint C6). Another problem can be the number of used patterns. A high number of different patterns can lead to a large number of possible execution paths (Constraint C5).

The Pattern-based Representation (PBR) format [6] uses a relatively easy approach for the identification of patterns. Very similar to the BCSR format a grid with a fixed size is assumed. Repeating patterns are searched in each of the cells of the grid.

Karakasis et al. [38] proposed a more sophisticated pattern detection strategy, used in the Compressed Sparse eXtended (CSX) format. The patterns have to be defined by hand beforehand, but they are searched in the complete matrix and not only in a fixed size grid as for the PBR format.

## 4.2 Exploitation of Other Structural Properties

The techniques described in the previous section exploit specific structural properties of a matrix. This is a very basic concept, that was already used in some of the first published sparse matrix formats.

For example the utilization of diagonal structures, as shown in Figure 4.3a, can be very advantageous. The Diagonal (DIA) format [77] stores all diagonals of a matrix that contain a non-zero element using two arrays. One array is used to store the actual non-zero elements and depending on the structure padding. The second array stores the position, or offset, of the diagonals. Exploiting the diagonal structure of a matrix is a very efficient way for reducing the memory demand (Requirement R1). It is very efficient, as no explicit row or column information is required. This means only the non-zero values itself and a small offset array is required for storing the matrix.

In diagonal formats the properties of diagonal matrices can be utilized to improve the performance of the SpMV operation. The accesses on the $\vec{x}$ vector for example are consecutive and can therefore be very good prefetched (Requirement R3 and R4). It also allows good memory alignment (Constraint C4) and good vectorization (Requirement R5). Depending on the position and number of diagonals, the cache reuse of the $\vec{x}$ vector can also be very high, which reduces the required memory bandwidth (Requirement R2).

The biggest disadvantage of these type of formats is their sensitivity on changes in the matrix structure. Very similar to the ELL format (see Section 2.1.3) a small number of additional non-zero elements can result in a lot of padding that increases the memory demand significantly.

There exist other formats that exploit the diagonal properties of a matrix. One example is the Banded Diagonal (BDIA) format [83], that stores bands of diagonal elements instead of single diagonals, which further reduces the memory demand.

The properties of symmetrical matrices can also be exploited, at least in theory. Symmetrical matrices have the same non-zero entries below and above the main diagonal. This could be utilized by storing the non-zero elements only once, which could cut the memory demand in half (Requirement R1). This is illustrated by Figure 4.3b.

The main problem of this approach is the implementation of a parallel SpMV-operation. The required data structure can not allow a row based access on the matrix elements, which unavoidably introduces the need for additional synchronization on the $\vec{y}$ vector (Constraint C3). Not many approaches could be found utilizing the symmetric property. The publication of Krotkiewski et al. [46] describes one approach, using MPI parallelism.

## 4.3 The ELL Storage Scheme

The ELL storage scheme was developed to allow efficient operations on vector based machines and today they are mostly used on GPU based systems. The ELL format uses an interleaving pattern for storing the non-zero elements as described in Section 2.1.3.

The format works very well for vector processors (Requirement R5) like GPUs, with appropriate matrices, because all elements are accessible in the right order (Requirement R4). Additionally all rows are padded to the same length which results in a perfect data alignment (Constraint C4). The padding is also the biggest drawback of all ELL based storage formats, as it can increase the storage demand dramatically (Requirement R1).

The padding can be reduced by slicing the matrix horizontally into multiple parts as introduced by the Sliced ELLpack (SELL) format [57] and the Sliced ELLR-T (SELLR-T) format [16]. The padding is applied in each slice individually, which can reduce the overall required padding significantly.

The ELL-Warp [91] format uses an additional global reordering of the matrix rows, by their length, to further reduce the padding. As similarly long rows are grouped together by the sorting, a slicing can more efficiently reduce the required padding. A global reordering can have negative effects on the locality, what is why Kreutzer et al. [4, 45] proposed a local reordering for their SELL-C-$\sigma$ and SELL-P formats. The area of the reordering is thereby multiple slices big, which allows some improvements regarding the padding without completely loosing locality. The ELLpack Sparse Block (ESB) format [53] is very similar to SELL-C-$\sigma$, but was mainly developed for the Xeon Phi platform instead of GPUs. It uses a bit-masking that is used with the vector units to reduce the required memory bandwidth by not loading padding elements.

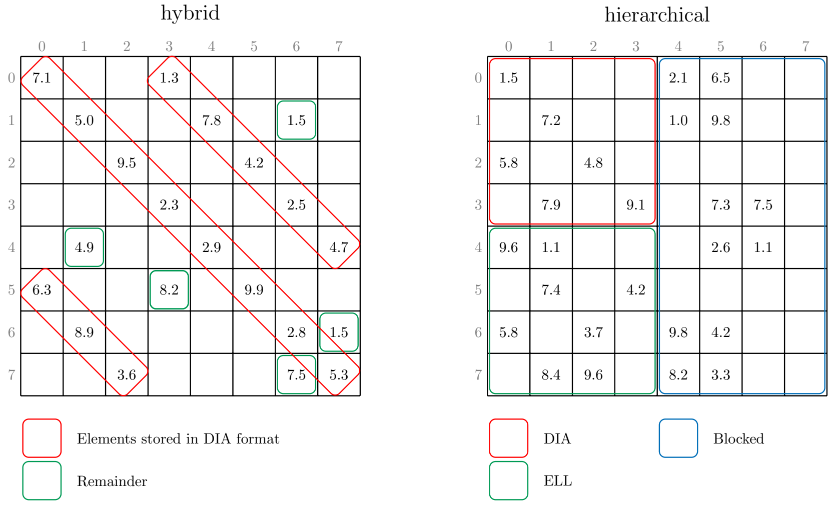
The ELL scheme is also used in blocked formats like BELLpack [14]. Dense blocks are thereby stored in an interleaving pattern. The padding is applied on a block level, with the same drawbacks as for the normal ELL format. The same techniques like slicing and reordering can be used to reduce the amount of padding.

## 4.4 Hybrid and Hierarchical Storage Formats

Hybrid and hierarchical storage formats store parts of the matrix using different matrix formats. The difference between these two type of formats is not clearly defined. Most often hybrid formats consist of two formats. The first format usually utilizes specific matrix properties, like ELL or DIA. The second format is used to store elements which can be disadvantageous for the first format. This is illustrated by Figure 4.4 with the first format being DIA. The elements in the second matrix are typically called remainder elements.

Hierarchical formats can utilize a larger number of different formats. Sparse matrices often consist of different regions with different structural properties. One hierarchical concept is the use of multiple formats for efficiently storing these different parts. The matrix is therefore partitioned into multiple sub-matrices. Figure 4.4 illustrates this one version of a hierarchical format. As the concepts of hybrid and hierarchical matrices are very similar the advantages and disadvantages are very similar and are therefore discussed together.

One of the first formats using a hybrid approach was the Hybrid (HYB) format [7] which combines the ELL format with the COO format. This is used to reduce the padding of the ELL format, very similar to the example presented in Figure 4.4. Matam et al. [54] proposed the combination of the ELL and CSR format. Very similar the BCSR Decomposed (BCSR-DEC) format [37] uses a combination of



hybrid

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 7.1 | | | 1.3 | | | | |
| 1 | | 5.0 | | | 7.8 | | 1.5 | |
| 2 | | | 9.5 | | | 4.2 | | |
| 3 | | | | 2.3 | | | 2.5 | |
| 4 | | 4.9 | | | 2.9 | | | 4.7 |
| 5 | 6.3 | | | 8.2 | | 9.9 | | |
| 6 | | 8.9 | | | | | 2.8 | 1.5 |
| 7 | | | 3.6 | | | | 7.5 | 5.3 |

hierarchical

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1.5 | | | | 2.1 | 6.5 | | |
| 1 | | 7.2 | | | 1.0 | 9.8 | | |
| 2 | 5.8 | | 4.8 | | | | | |
| 3 | | 7.9 | | 9.1 | | 7.3 | 7.5 | |
| 4 | 9.6 | 1.1 | | | | 2.6 | 1.1 | |
| 5 | | 7.4 | | 4.2 | | | | |
| 6 | 5.8 | | 3.7 | | 9.8 | 4.2 | | |
| 7 | | 8.4 | 9.6 | | 8.2 | 3.3 | | |

Figure 4.4: Example for the basic concept of hybrid (left) and hierarchical (right) storage formats.

BCSR and CSR. This is useful, as the BCSR format is also very sensitive regarding the matrix structure and can require a lot of padding.

The Cocktail format [83] is one example for an hierarchical approach. The matrix is partitioned depending on the matrix structure and various different matrix formats are used for representing them. A completely different hierarchical approach is used in the Hierarchical Sparse Matrix Storage (HiSM) format [82]. It is a combination of the BCSR and the COO format. The format stores large BCSR blocks and prevents padding by using a bit compressed COO format for storing the non-zero elements in these blocks.

The use of hybrid or hierarchical approaches can have many advantages, depending on the used formats. It is therefore not possible to mention general advantages and the corresponding requirements for these types of formats in general. The only advantage most of these formats have in common is a reduction of the memory demand (Requirement R1).

Hybrid and hierarchical formats also have some possible disadvantages. One issue nearly all hybrid and hierarchical approaches have in common is that they have to execute the SpMV operation multiple times, once for every sub-matrix. Depending on the partitioning this is no problem, but if elements of the same row are stored in different sub-matrices different problems may occur. The calculation of multiple sub-matrices in parallel can result in parallel access on the $\vec{y}$ vector which would require additional synchronization for the write operations (Constraint C3).

The matrix creation for these types of formats is very complex (Constraint C6). The identification of specific matrix structures and sub-structures is a very complex problem. In reasonable time, most often only an approximation of an optimal matrix partitioning can be calculated.

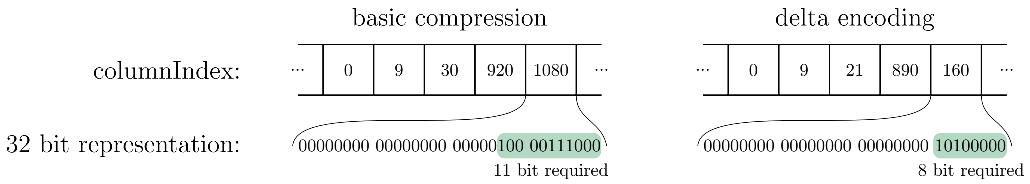


Figure 4.5: Example for a basic bit compression approach (left) and bit compression with delta encoding (right).

## 4.5 Index and Value Compression

The index structures of a matrix typically consists of multiple integer arrays which are stored using 32 or 64 bit integer values. Often smaller data types would be enough for storing the whole index structures or at least parts of it. Additionally there exist different approaches for lossless compressing the index structures. Clearly the aim of the compression is a reduced memory demand (Requirement R1). Figure 4.5 illustrates the possible reduction in memory demand by using a simple compression technique. It also shows the benefit of using delta encoding, which is mentioned later in this section.

A very early use of index compression was in the HiSM format [82]. It stores sparse blocks of a fixed size in the COO format. As the size of the blocks is much smaller, compared to the matrix dimensions, a much smaller data type is sufficient for storing the local indices. A row and column offset is additionally stored for each of the blocks, which allows the calculation of the global position of each element. Very similar techniques are also utilized in the Compressed Sparse Block (CSB) [11] and BCCOO [92] formats. The Bit Level Single Row (BLSI) format [73] uses a very similar technique and compresses the column and row indices of the COO format into a single integer array. It divides the matrix into multiple parts and stores the offset information to every part into an additional array.

More sophisticated approaches were proposed by Buluc et al. [41] with the CSR Delta Unit (CSR-DU) and CSR Value Indexed (CSR-VI) formats. The index compression of the CSR-DU format is based on a delta calculation of the column indices. Each column index is stored as difference to the previous index, which reduces the possible number range that have to be stored. The final index structure is stored in a kind of package structure, where each package contains some header information and bit compressed index information. This structure does not only contain the column information but also the row indices.

The CSR-VI format [41] focuses on a value compression approach. The values of a matrix are typically floating point values, which are much harder to compress, compared to integer values. The reason for this is the more complex bit representation of floating point values and a more efficient use of the available bits to allow maximal possible precision. The CSR-VI format therefore does not try to reduce the memory demand for the floating point values itself and instead tries to exploit the fact that identical values may exist multiple times. A new values array is created which contains only the unique non-zero values. An additional index structure is required, which points to the correct value for each non-zero entry in the matrix.

The Combine Optimized SpMV for CSR (COSC) format [94] is based on the CSR-DU and CSR-VI formats and uses a combination of the index and the value compression. The CSX format [38] is also based on the CSR-DU format, but extends the compression even further. It uses a run-length encoding which stores repeating patterns of delta values as the delta value and the number of occurrences.

Tang et al. [85] proposed a new compression technique called Bit Representation Optimizations (BRO), which is similar to the technique used in CSR-DU. It also uses a delta compression, in the first step, to reduce the number space. The compression is most effective for a sliced ELL format. For each ELL-column and slice the maximum number of bits required for storing all indices in this column are identified. The values are then compressed using these sizes.

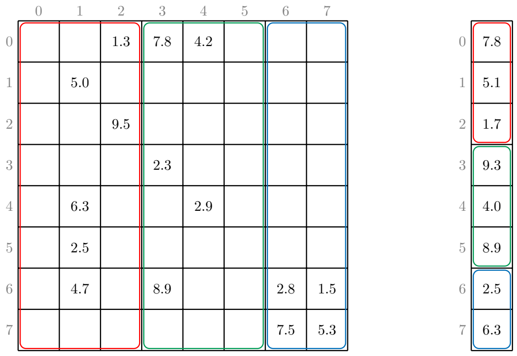
While all compression techniques aim the main issue of the SpMV operation by trying to reduce the memory demand, they also potentially come with major drawbacks. The most effective compression techniques (regarding space savings) require additional meta information for the later decompression. The decompression itself can also be a bottleneck by potentially introducing excessive branching (Constraint C5). The type of compression also has to be selected carefully regarding the possible utilization of vector units (Requirement R5) and memory alignment (Constraint C4).

## 4.6 Vertical Tiling

Yang et al. [93] proposed a vertical partitioning of matrices, later on called vertical tiling or just tiling. If the size of the tiles is selected properly this approach can improve the cache reuse of the SpMV operation (Requirement R2). This is achieved, as each tile contains only a limited number of columns, which relate to a limited number of relevant entries of the $\vec{x}$ vector. Figure 4.6 illustrates this concept and shows the memory accesses of a matrix with vertical tiling.

Yang et al. [93] also proposed the additional use of a column reordering, so that the columns with the most entries are close together. A very similar approach was also used for the BCCOO [92] and Vectorized Hybrid COO+CSR (VHCC) [84] formats. This allows the use of a smaller number of tiles by tiling only the regions of the matrix with a higher non-zero density.

The size of the tiles has to be configured to match the cache size. If the L3 cache of an CPU-based system should be used, the tiles of an matrix can no longer be calculated in parallel. As the L3 cache is shared between the different cores, the accesses in the different tiles would replace cache lines in the L3. Additionally the parallel calculation of multiple tiles has the disadvantage of parallel write accesses on the $\vec{y}$ vector (Constraint C1) which also requires synchronization (Con-



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | | | 1.3 | 7.8 | 4.2 | | | |
| 1 | | 5.0 | | | | | | |
| 2 | | | 9.5 | | | | | |
| 3 | | | | 2.3 | | | | |
| 4 | | 6.3 | | | 2.9 | | | |
| 5 | | 2.5 | | | | | | |
| 6 | | 4.7 | | 8.9 | | | 2.8 | 1.5 |
| 7 | | | | | | | 7.5 | 5.3 |

| | |
|---|---|
| 0 | 7.8 |
| 1 | 5.1 |
| 2 | 1.7 |
| 3 | 9.3 |
| 4 | 4.0 |
| 5 | 8.9 |
| 6 | 2.5 |
| 7 | 6.3 |

Figure 4.6: Memory accesses in a matrix with vertical tiling.

straint C3).

## 4.7 Matrix Reordering

Another technique, that is used for multiple reasons, is matrix reordering. It is often used in combination with ELL based formats and slicing approaches to reduce the amount of required padding, which reduces the memory demand of the format (Requirement R1). The Jagged Diagonal Storage (JDS) format [75] was one of the first formats using this technique. The more recent SELL-C-$\sigma$ format [45] format also uses reordering to reduce the padding of the ELL based format. The difference is, that only a local instead of a global reordering is used.

The main reason for this is, that a matrix reordering can reduce the locality of memory accesses and therefore reduce the caching on the $\vec{x}$ vector (Requirement R2). Doing the reordering only locally can, in theory, preserve the locality.

While reordering can reduce the locality of memory accesses, it is also used to reach the opposite [70]. By reordering the non-zero elements of the matrix in such way, that identical or similar column indexes are grouped together, the locality can be increased. The main drawback of this approach is the complexity of calculating a good reordering. This significantly increases the creating time of the matrix (Constraint C6). Reordering can also be used for creating diagonal structures. The diagonal properties can be utilized as already described in Section 4.2.

## 4.8 Row Grouping and Row Splitting

The matrix structure has a significant impact on the performance of most matrix formats. For formats that use row-based calculations, one major problem often is the varying number of non-zero elements per row of a matrix. The calculation of short rows can be inefficient, because the load of meta information can be slow compared to the actual calculations. Very long rows on the other side require more time to be calculated than other rows, which can lead to load imbalances in case of parallel computations.

One way of solving these problems is by grouping short rows together into single, longer rows and splitting long rows into multiple smaller ones. This allows for smaller calculation overhead and better load balancing (Requirement R6). Averaging the number of non-zeros per row can also have a positive effect on the possibility to use vector units for the calculation (Requirement R5). Disadvantages of these approaches is the additional branching, required to differentiate between the different types of rows (Constraint C5). Splitting one row into multiple rows also may require parallel write accesses to the same elements of the $\vec{y}$ vector which results in additional synchronization (Constraint C3).

Feng et al. [18] proposed the Segmented Interleave Combination (SIC) format, which combines multiple CSR rows into larger rows using an interleaving pattern. This results in an ELL style data structure, which can efficiently be used especially on GPUs. Wong et al. [91] proposed the ELL-Warp format, which also allows splitting long rows into multiple parts using the ELL format. Oberhuber et al. [67] proposed a format called Row Grouped CSR. The name can be misleading, as it uses a splitting technique for very long rows. The Compressed Multirow Storage (CMRS) format [42] also groups multiple rows together and is based on CSR. The format can reuse the original values and column index arrays.

## 4.9 Segmented Sum Algorithm Based Calculations

Most of the established methods use a row based approach for the calculation of the SpMV operation. This has the advantage that no synchronization between multiple threads is required, as each row can be calculated independent of all others (Constraint C3). One big problem with this approach is that the load balance mostly depends on the matrix structure, or more specifically on the number of non-zero element in the rows. Possible solutions have been presented in the previous Section 4.8.

A quite new approach for solving this problem is the use of a segmented sum algorithm [78] for the SpMV calculation. The segmented sum algorithm works very similar to a simple prefix operation. Instead of calculating a single prefix operation for the full number sequence, the sequence is divided into multiple segments. The result of the prefix operation is than calculated for each segment individually. One big advantage of this approach is a very good load balancing, as all elements of the matrix are evenly distributed to the available hardware resources (Requirement R6). This means the load balance is independent of the matrix structure. Another benefit is the possibly high utilization of vector units (Requirement R5), as the row boundaries are not important when iterating over the matrix values. This also allows a proper memory alignment (Constraint C4). The drawback of this method is additional synchronization, that can not be avoided (Constraint C3).

The BCCOO format [92] stores sparse blocks in the COO format. It uses a segmented sum algorithm to efficiently calculate the SpMV operation. The VHCC format [84] also uses a segmented sum instead of a classical row-based operation.

The CSR5 format [51] is based on the CSR format. The segmented sum is used to improve the load balance, but it also provides an efficient SpMV operation for vector processors, like GPUs. The CSR5 format introduces additional data structures, including a bit-string, to allow a very efficient segmented sum calculation. It also has a very low conversion overhead from the CSR format [51].

The Perfect CSR (PCSR) format [23] is also based on CSR and similar to CSR5. Instead of executing the segmented sum directly on the CSR data, they use a temporary array for storing the products of the non-zeros and the $\vec{x}$ vector.

Liu et al. [52] proposed an approach called Speculative Segmented Sum CSR, where the SpMV operation is divided into two phases. In the first phase the segmented sum is calculated without global synchronization, which potentially leads to wrong results. In a second phase all relevant results are corrected. The main advantage is the reduced amount of synchronization (Constraint C3).

## 4.10 Discussion and Summary

The previous sections discussed many different possible techniques for optimizing the SpMV operation on different platforms. Table 4.1 summarizes the requirements and the identified optimization techniques that can be used to fulfill them.

It can be seen, that most optimizations focus on the reduction of the memory demand, data reuse and improved memory accesses, which is reasonable as that the SpMV operation is memory bounded. Still it can be seen that there are many optimization techniques that can be used to improve the use of vector units. Partly this may be motivated by the fact that GPUs are in the focus of many developed formats.

One very big issue with the SpMV operation are the irregular accesses on the $\vec{x}$ vector. No optimizations could be found that focus on the reduction of the memory access latency on the $\vec{x}$ vector, beside the utilization of structural matrix properties.

| **Requirement** | **Optimization Techniques** |
|---|---|
| R1 reduce amount of moved data by reducing the overall memory demand | · Slicing for ELL type formats to reduce padding<br>· Reordering (with ELL / slicing)<br>· Blocking and pattern detection<br>· Exploitation of structural properties (e.g., diagonal, symmetric)<br>· Index and value compression<br>· Hybrid and hierarchical approaches |
| R2 allow / improve data reuse of the $\vec{x}$ vector | · Vertical tiling<br>· Reordering<br>· Blocking and pattern detection<br>· Exploit diagonal properties |
| R3 improve access latency on the $\vec{x}$ vector | · Exploit diagonal properties |
| R4 allow / improve consecutive memory accesses | · Blocking and pattern detection<br>· ELL-style data layout<br>· Row grouping and splitting<br>· Segmented sum instead of row-based calculations |
| R5 allow / improve utilization of vector units | · Blocking and pattern detection<br>· ELL-style data layout<br>· Exploit diagonal properties<br>· Row grouping and splitting<br>· Segmented sum instead of row-based calculations |
| R6 allow / improve load balancing of the SpMV | · Row grouping and splitting<br>· Segmented sum instead of row-based calculations |

Table 4.1: Summary of the requirements and the optimizations techniques that can fulfill them.

# 5 Development of New Matrix Formats

In the two previous chapters the requirements and constraints for developing a proper sparse matrix format and the existing optimization techniques have been identified. In this chapter new efficient sparse matrix formats will be developed, based on the knowledge gained in these chapters. Basically matrix formats are combinations of optimization techniques. Even though, not all combinations of optimization techniques are possible or feasible, the number of possible combinations is high. Additionally the three relevant hardware platforms require different optimizations for reaching sufficient performance. Therefore, multiple matrix formats will be developed with different optimization goals. A reasonable combination of optimization techniques will be utilized for reaching these goals. Focusing on specific requirements in the development process does not imply that other requirements are neglected. If multiple optimization techniques are feasible, the one that satisfies the optimization goals best is used.

In the development process it is not sufficient to select some of these techniques and just combine them to a new format. Instead nearly all optimization techniques are rather basic ideas, which can be implemented in many different ways. For example, bit compression techniques are supposed to reduce the memory amount of a format. This bit compression can be achieved by simply using smaller data type or by storing multiple indexes in a single integer value. Using multiple optimization techniques requires the combination of different basic ideas in an efficient new data structure.

In the following sections the development of the three new formats is described. The sections are thereby structured into the motivation and the theory behind the development, a detailed description of the formats and a discussion.

The rest of the chapter is structured as follows: First the three newly developed formats CSR5 Bit Compressed (CSR5BC), Hybrid Compressed Slice Storage (HCSS) and Local Group Compressed Sparse Row (LGCSR) are presented. Afterwards the optimization of the existing DynB format using a autotuning approach is described. The chapter closes with a summary and discussion of the presented formats.

## 5.1 Development of CSR5 Bit Compressed – CSR5BC

The optimization goal of the new CSR5BC format is the development of a format with a structure independent load balancing (Requirement R6). This means that the calculations of the SpMV operation can be divided into equal parts, regardless of the structure of the matrix and the non-zero distribution. This goal cannot be achieved using traditional row-based approaches, as the occurrence of one row with much more non-zero elements than the other rows can result in significant load imbalances. One solution for this problem is the use of row splitting and row grouping approaches (see Section 4.8). This approach solves the problem only partially, as after the splitting of long and grouping of short rows, still a distribution has to be determined. Another possible approach is the use of a segmented sum based algorithm (see Section 4.9). This allows a calculation of the SpMV independent of row boundaries, which simplifies the work distribution significantly. As the work can be distributed regardless of row boundaries, each thread can work on a fixed number of non-zero elements. Therefore, this approach is used for the CSR5BC format.

The independence of row boundaries also simplifies the efficient utilization of vector units (Requirement R5). When processing elements of a single row, one factor that influences the efficiency of the vector unit utilization is the number of non-zero elements. If the number of elements in the row is not dividable by the vector unit size, the vector unit is not fully utilized in the last iteration. Without the row boundaries, this problem can only occur once for every thread instead of at the end of every row.

The row boundary independent calculation also allows properly aligned vector calculations and consecutive memory accesses (Constraint C4 and Requirement R4). As long as the initial vector operation is executed on a proper memory boundary, all following operations are aligned and consecutive. Again, this is different for most row-based approaches, where it is important that the first element of each row is aligned.

As described in the previous chapter, the SpMV operation is a memory bound operation (Requirement R1). Since the goal of the format is a structural independent load balancing, exploiting structural properties is not suitable for a memory reduction. Two other techniques for reducing the memory demand have been identified in the previous chapter: Index and value compression techniques. Value compression is based on storing unique non-zero values only once, requiring an additional index array for every non-zero element (see Section 4.5). This technique introduces a significant amount of additional indirect memory accesses. Furthermore the efficiency of the compression strongly depends on the non-zero values. Because of this drawbacks a index compression technique is used instead.

### Description of the CSR5 Format

The CSR5 format [51] by Liu et al. considers most of the above motivated aspects. Furthermore, the implementation of the format is available as open source code for all, for this work, relevant hardware platforms. The SpMV of the format is based on an efficient segmented sum algorithm, but it does not implement any index compression techniques. The CSR5BC format is therefore based on the work of Liu et al. and primarily extends the CSR5 format with an additional index compression. In the following, the fundamental concepts of the existing CSR5 format are explained. For a more detailed description, the original publication

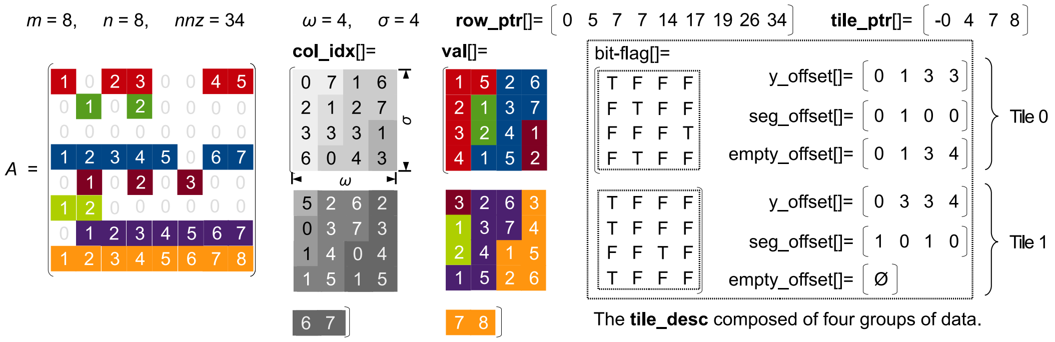


Figure 5.1: Data structures of the CSR5 format [51].

can be consulted [51]. Afterwards the CSR5BC format with the additional index compression is described.

Figure 5.1 presents the complex data structures of the CSR5 format. The CSR5 format stores the non-zero elements in two dimensional blocks of fixed size, called tiles (left half of Figure 5.1). These tiles are used to simplify the data management and the distribution of the non-zero entries to the threads. The tiles should not be mistaken with blocks in blocked storage formats, they only serve as a kind of container for managing the non-zero entries. The size of the tiles is defined by the two tuning parameters $\sigma$ and $\omega$. The width of the tiles, or $\omega$, thereby matches the width of the available vector units, while for the selection of $\sigma$ a simple heuristic is provided. It is also important that the non-zero elements in the tiles are stored in column-major order. The column index and value arrays of the CSR format are used for storing the slices.

To allow an efficient calculation of the segmented sum, additional meta information for each tile is required. The *tile_ptr* stores the matrix row of the first element in the slice, which allows a parallel processing of the tiles. In Addition the *tile_desc* stores 4 additional arrays for each tile, which are necessary for reaching a high segmented sum performance. A bit flag of size $\omega \times \sigma$ is used to mark non-zero elements that are the first elements of a row. The *y_offset* array stores the row of the first element of each column of the tile. This allows the independent processing of each column. Without the offset, the processing depends on all previous columns, because the number of new rows (encoded in $bit\_flag$) is not known beforehand. This is important, as the SpMV operations processes the columns of a tile in parallel using the available vector units. The *seg_offset* array is used to improve the performance of the local segmented sum operation. For each column, it stores the number of new beginning rows in the next column. The *empty_offset* array is used if the matrix contains empty rows, i.e., rows without any non-zero elements. This is necessary, because the bit flag array can only encode the beginning of a new row for a non-zero element. Since the empty rows do not contain any non-zero elements the additional array is required.

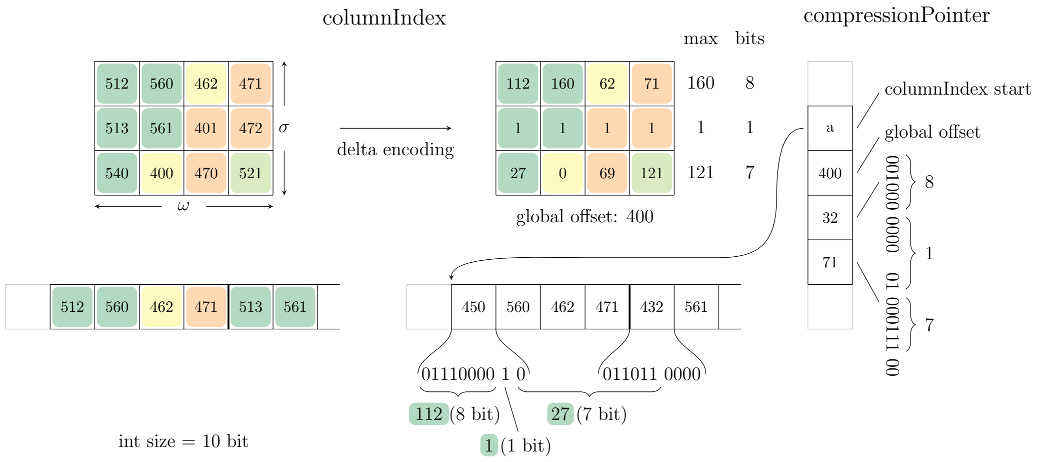


Figure 5.2: Example for the index compression technique used in CSR5BC.

## Description of the Extensions in the CSR5BC Format

Figure 5.2 illustrates the new index compression technique introduced in CSR5BC. The compression reduces the size of the *columnIndex* array and requires one additional array with slice meta information called *compressionPointer*. It can be seen that in a first step a column-wise delta encoding is used to reduce the number range. Before this is actually done, the smallest index in the whole slice is identified and the value is subtracted from all indexes. This value is called global offset, and is stored in the *compressionPointer*. This process further reduces the overall number range of the indexes.

In the next step, the indexes are packed tightly into the *columnIndex* array. The number of stored bits is fixed per row and is therefore determined by the biggest index per row. The number of bits required for every row of the slice is also stored in the *compressionPointer*. As the number of bits can only be between 1 and 32 bits, 6 bits are enough for storing the information. The information is therefore also bit-packed, which reduces the meta information overhead.

It is important at this point, that the delta encoding is done column-wise and that all elements in the same row are encoded using the same number of bits. The first one allows a independent calculation of the elements of every column, which is important as they are processed in parallel by the vector units. The common number of bits ensures consecutive memory accesses (Requirement R4), as the amount of loaded data is identical for each lane of the vector unit.

In the standard CSR5 format, the starting position of the slices in the *values* and *columnIndex* arrays can easily be calculated, because the slices contain a fixed number of entries. In the compressed case of CSR5BC this is no longer true for the *columnIndex* array. The number of entries per slice in the *columnIndex* is variable, as the efficiency of the compression depends on the occurring column indexes. To still allow independent calculation of the slices, the *compressionPointer* array holds an additional pointer to the first element of the slice in the *columnIndex* array.

### Discussion

The presented CSR5BC format does respect many of the identified requirements and constraints. The memory demand is reduced by using index compression, memory accesses are consecutive, the efficient utilization of vector units is possible and a proper load balancing is expected (Requirements R1, R4, R5 and R6). It does not improve the data reuse of the $\vec{x}$ vector (Requirement R2), which is hardly possible without utilizing structural properties of the matrix. The only structural independent technique is a vertical partitioning, of the matrix (see Section 4.6). But this introduces many new problems, like the need of additional synchronization on the $\vec{y}$ vector. Because of this, no tiling approach has not been used. The access latency on the $\vec{x}$ vector is also not improved by the format (Requirement R3), as no structural independent optimization technique could be found.

Each thread processes a big, consecutive partition of the matrix, which prevents interleaved write operations on the $\vec{y}$ vector and allows a NUMA aware implementation (Constraints C1 and C2). The segmented sum algorithm uses small additional data structures to completely prevent synchronized write operations on the $\vec{y}$ vector and the tile layout allows proper memory alignment (Constraints C3 and C4). The creation process of the format does not require any expensive operations like reordering or pattern detection. Even though the bit compression does require multiple iterations over the matrix data, the conversion should be efficient (Constraint C6). The *values* array of the CSR5BC format is created from the *values* arrays of the CSR format by transposing elements. The calculation of the memory location of a matrix element is therefore straightforward, which allows efficient element updates (Constraint C7). Furthermore each tile of the format can be calculated independent of other tiles, which in theory allows the asynchronous computation of the CSR5BC format (Constraint C8). This will not be implemented in practice, because the used software framework (see Chapter 6) does not provide this type of operation. The only tuning parameters of the CSR5BC format are the dimensions of the used tiles, where one is defined by the size of the vector units, and the other one can be determined by using a simple heuristic, which is provided by the CSR5 format (Constraint C8).

Overall the CSR5BC format fulfills most of the identified requirements and constraints. It should therefore be suitable for the efficient calculation of the SpMV operations on all for this work relevant hardware platforms. Nevertheless there are some possible disadvantages. The format requires quite sophisticated and extensive meta information, which increase the overall memory consumption. Furthermore the SpMV operation is quite complex, which may have negative effects on the performance. The complex data layout may result in additional branching (Constraint C5). Another negative aspect is that the data layout depends on the size of the vector units. This could be a disadvantage if a matrix should, for example, be transferred from a CPU to a GPU which has another vector unit size. In this case the matrix has to be recreated with the proper tile dimensions.

The CSR5BC format has been developed to be independent of the matrix structure. Therefore it should be applicable for most matrices and deliver a consistent performance. It is expected, that more specialized formats deliver better performance for matrices with specific properties.

The data structures of the format are very complex and the compression partially depends on the matrix structure. The estimated memory consumption can be calculated by:

$$\begin{aligned} m_{CSR5BC} \approx\ & nnz \times S_{float} && //values \\ & + nnz \times S_{int} \times f_{comp} && //colIndex \\ & + nTiles \times S_{int} && //tilePtr \\ & + nTiles \times (\lceil (\log_2(\omega \times \sigma) + \log_2(\omega) + \sigma)/S_{int} \rceil \times S_{int} \times \omega) && //tileDesc \\ & + nTiles \times (2 \times S_{int} + \lceil (6 \times \sigma)/S_{int} \rceil \times S_{int}) && //comprPtr \end{aligned}$$

The size of the *values* vector is identic to the CSR format. The *columnIndex* array is compressed by the factor $f_{comp}$, compared to the CSR format. Additionally for each tile one additional integer is stored in the *tilePtr*. The required meta information in the *tileDescriptor* array are highly compressed. One bit for every element is required for the *bit-flag* array, the size of the *y_offset* and *seg_offset* arrays depend on the dimensions of the slices. The *empty_offset* is not considered in the equation, as it is not required in most matrices. Furthermore two additional integer values and 6 bit per slices row are stored in the *compressionPtr*.

The equation can be defined more compact, the relation of $nTiles = nnz/(\sigma \times \omega)$ is thereby additionally used:

$$\begin{aligned} m_{CSR5BC} \approx nnz \times (S_{float} + S_{int} \times f_{comp} + \frac{S_{int}}{\omega \times \sigma} \\ \times [3 + \lceil (\log_2(\sigma) + 2\log_2(\omega) + \sigma)/S_{int} \rceil \times \omega + \lceil (6 \times \sigma)/S_{int} \rceil ]) \end{aligned}$$

To allow the actual comparison of the different matrix formats, some assumptions about the matrix structure and used platform have to be done. The average number of non-zero elements per row is important for the estimation of the memory consumption for most formats. By determining the average over a large set of matrices (see Chapter 7), the number of non-zero elements per row was selected to be 64. Vector units are assumed to be 4 elements wide, which is the vector size of the current Intel Haswell architecture. For the CSR5BC format this means $\sigma$ is 64 and $\omega$ is 4. The data size for floating point values, $S_{float}$, is defined as 64 bit and integer values, $S_{int}$, are defined as 32 bit values. For the mentioned set of matrices the average compression of the index data structures has been determined empirically to be about 0.31. Using these assumptions, the average memory consumption of the format per non-zero element can be calculated:

$$m\_nnz_{CSR5BC} \approx nnz \times 77.3$$

Interpreting this number is not possible at this point, as a reference to other formats is required. This comparison is done in the last section of this chapter.

## 5.2 Development of Hybrid Compressed Slice Storage – HCSS

The optimization goal of the HCSS format is on the utilization of vector units (Requirement R5). This should be achieved while preventing the need for complex or large meta information as well as a complex SpMV kernel. It has already been motivated in the previous chapters, that vector units are present in all architectures and getting more important in the future. For reaching a good vector unit utilization the memory accesses have to be consecutive (Requirement R4) and properly aligned (Constraint C4). For a proper alignment, not only the beginning of the arrays have to be aligned to the proper boundaries, but also the accesses to these elements need to be aligned as well. Ignoring the alignment of the data accesses does not prevent vectorization, but can reduce the used width of the vector units or can result in inefficient and therefore slower memory accesses.

Keeping the additional meta information of the format simple and small also keeps the overall memory consumption of the format low. Additional techniques should be used to further reduce the memory consumption (Requirement R1). One possible option is the utilization of matrix properties, like blocking or pattern detection, to reduce the amount of index information. While this can be suitable to reduce the memory consumption of the format, it can also add additional complexity to the format. Additionally the type of used blocks or patterns can have a significant impact on the efficiency of the vectorization. Limiting the format only to suitable block sizes or patterns can reduce the efficiency of the blocking as it can introduce excessive fill-in. The other option is the use of a index compression. In comparison to the CSR5BC format a simpler approach is chosen. The bit compression in the CSR5BC format allows the encoding of column index information of variable length. While this allows a very high reduction of the required memory, it also requires additional meta information, e.g, the number of stored bits. This approach can also have negative effects on the memory alignment, as the alignment of the column index information depends on the compression factor of the previous indexes. For the HCSS format the use of bit compression with a fixed length is therefore more suitable. This requires no additional meta information for the compression itself, and allows proper data alignment for the vector units, depending on the selected bit length. The number of bits used for the compression should be a divider of the used data type size (e.g. 16 bit for a 32 bit data type).

Providing proper load balancing is another important requirement that should be respected by the HCSS format (Requirement R6). The load balance is improved by splitting the matrix into multiple partitions depending on the number of non-zero elements. The partitioning should distribute the non-zero elements as evenly as possible between the threads. The slice structure of the HCSS formats needs to

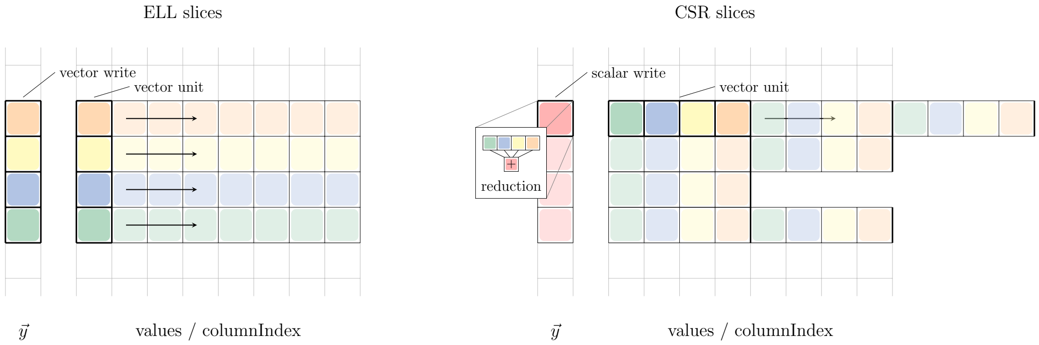


Figure 5.3: Calculation of the ELL (left) and modified CSR (right) slices using vector units in the HCSS format.

be respected as well.

### Description of the HCSS Format

The optimization goal of the HCSS format is to reach a high vector unit utilization. Figure 5.3 shows the usage of vector units with a column-major order formats like ELL and row-major order formats like CSR. In ELL based formats 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 the last elements of the rows, the result of the vector unit is simply written to the $\vec{y}$ vector.

In CSR based formats on the other side, 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 proper memory alignment depends on the number of entries in every row and therefore creating a proper alignment requires additional fill-in.

Overall the ELL format is therefore better suited to reach a high vector unit utilization. But, classic ELL based formats are very sensitive to the matrix structure, which makes them unsuitable in many cases (see Section 2.1.3). The basic idea of the HCSS format is to create a lightweight hybrid format that combines both concepts. The format should be lightweight regarding the required amount of additional meta information and control flow, branching should be avoided whenever possible (Constraint C5).

To avoid creating multiple index structures, both storage types are combined in a single data structure. Therefore the matrix is row-wise partitioned in equally sized slices (every slice contains the same number of rows). Each slice is stored in one of the supported matrix types. One important design principle of the format is, that the number elements in every slice should be a multiple of the vector size. This ensures proper memory alignment, as every slice begins at a proper memory boundary. Furthermore the number of rows in every slices matches the vector unit size. This allows the efficient calculation of slices stored in the ELL format.

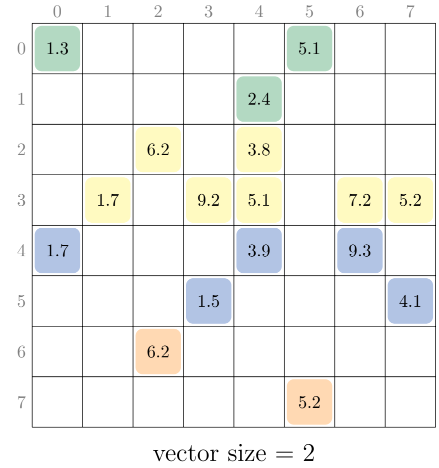
Since the ELL format is the better suited format for the use with vector units, it should be used for most of the slices. Only if the use of the ELL format requires to much fill-in, a modified version of the CSR format is used instead. This is the case for slices that contain rows with strongly varying number of non-zero elements. Storing these slices in ELL leads to excessive padding, which is not the case for CSR. The classic CSR format has the disadvantage that the non-zero elements of the different rows are not properly aligned. Therefore, a padding is introduced that ensures that each row contains a multiple of the vector size number of elements. This leads to fully aligned memory accesses for every row, as illustrated in Figure 5.3. It also ensures that the number of overall stored elements in a CSR slice is a multiple of the vector size.

As described in the beginning of this section, an index compression with a fixed number of bits should be used to further reduce the memory demand of the format. One additional requirement of the compression is that it should have no negative effect on the vector unit utilization. Using a fixed number of bits for the compression does not allow the compression of all slices of the matrix. If the structure of the rows in the slice is disadvantageous, the column indexes can not be stored in a compressed manner. This is the case if the elements of a row are spread out very far from reach other. These structures can result in very large column indexes, even if a delta encoding is used to reduce the number space. As the HCSS format is capable auf handling different formats for each slice anyway, a new slice type is introduced, which is a compressed variation of the ELL format.

The compressed slices have the following structure. In a first step a delta encoding is used to reduce the potential number range (see Section 4.5). After the delta encoding all elements are stored using only 16 bit instead of 32 bit, which halves the memory requirement for the index data. As the first element of each row is not effected by the delta compression, it will often be quite large and would prevent the use of the compression. For this reason the first element is always stored using 32 bit.

In theory more different slice types are possible, as long as the number is not getting to large (e.g., prevent branching). In this work only these three slice types are used, to prove the general concept of the format. The selection of the proper slice type is important for reaching good performance. At this point it is only defined that all slices should be stored in ELL or compressed ELL if possible. Only slices that would require excessive padding should be stored using CSR. The exact process of selecting the slice type is described in more detail in the next chapter on implementation details (see Section 6.2).

Figure 5.4 shows the data structures of the HCSS format with an example matrix. The HCSS format requires 5 arrays for storing the data and meta-information. The non-zero values are stored in an *values* arrays and the corresponding index information are stored in the *columnStart* array. The slices are organized very similar to the rows in CSR. An offset array, called *sliceStart*, is used for storing the start position of every slice in the *values* and *columnStart* array. In difference to CSR two



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1.3 |  |  |  |  | 5.1 |  |  |
| 1 |  |  |  |  | 2.4 |  |  |  |
| 2 |  |  | 6.2 |  | 3.8 |  |  |  |
| 3 |  | 1.7 |  | 9.2 | 5.1 |  | 7.2 | 5.2 |
| 4 | 1.7 |  |  |  | 3.9 |  | 9.3 |  |
| 5 |  |  |  | 1.5 |  |  |  | 4.1 |
| 6 |  |  | 6.2 |  |  |  |  |  |
| 7 |  |  |  |  |  | 5.2 |  |  |

vector size = 2

▨ Slice type encoded in first bits

$$rowPtr = \{0, 2, 3, 5, 10, 13, 15, 16, 17\}$$

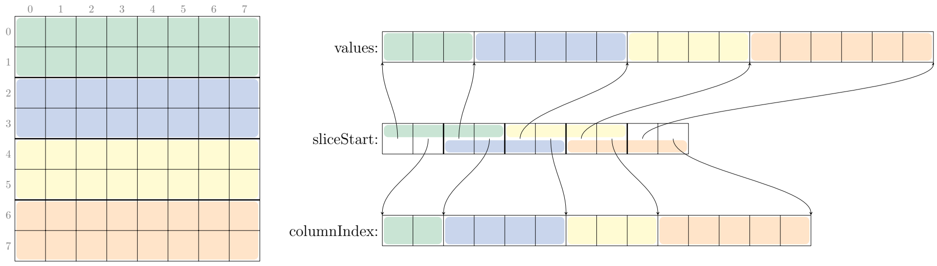
$$tileStart = \{{}^{0}0, 0,\ {}^{2}2, 2,\ {}^{1}6, 6,\ {}^{0}9, 8,\ 10, 9\}$$

$$values = \{1.3, 2.4, 5.1, 0.0, 6.2, 3.8, 1.7, 9.2, 5.1, 7.2, 5.2, 0.0, 1.7, 1.5, 3.9, 4.1, 9.3, 0.0, 6.2, 5.2\}$$

$$columnIndex = \{0, 4, 5, 5, 2, 4, 1, 3, 4, 6, 7, 7, 0, 3, (4, 6), (7, 7), 2, 5\}$$

Simplified structure, (a, b) denotes index a and b compressed to 32 bit

Figure 5.4: Simplified data structures of the HCSS format.



Vector size: 2 number of slices: 4

Figure 5.5: Organization of the slices in the HCSS format.

different offsets for these arrays are required, as the data in the *columnStart* arrays is partially bit-compressed, which results into different offsets. This is additionally illustrated in Figure 5.5. The *rowStart* of the original CSR format is also used by the format, but it is only required for the modified CSR slices. As the *sliceStart* array can only be used for storing the overall number of elements in the slice, the exact number of elements is required for the modified CSR slices. The last arrays is called *blockStart* (not shown in the figures), and is used for improving the load balance of the format (Requirement R6). Since the number of non-zero elements in each slice differs, a static work distribution based on the number of slices is not very effective. The *blockStart* array stores the first and last slice each thread has to process. This allows the use of a more efficient distribution, for example based on the number of overall processed non-zero elements.

Additionally the type of every slice has to be stored as additional meta information. Instead of using an additional array, a small number of bits in the *sliceStart* array is used. This is possible, because the processing is done on a slice level and each slice contains a number on non-zero elements which is a multiple of the vector size. This can be used to reduce all offsets by the factor of the vector size (e.g., factor 4 for a 4 element wide vector unit). By reducing the maximum possible index by the factor 4, the highest 2 bits can be used for storing other information, without limiting the size of the possibly usable matrices. As two integers are stored for each slice this already overall results in 4 bits per slice in the case of a 4 element wide vector unit. These 4 bit are more than enough to encode the three currently used slice types. The used technique reduces the memory demand significantly, compared to using an additional array (Requirement R1).

If the number of rows in the matrix is not a multiple of the vector size, the remaining rows are stored in the normal CSR format. This part of the matrix is called tail partition and is handled separately in the calculation of the SpMV.

### Discussion

The basic idea of combining the ELL format with CSR has been used earlier in other formats. Matam et al. [54] published the HYB format, which also uses both of the formats. The major difference between HCSS and HYB is that HCSS stores the CSR data in the same data structures, while the HYB format divides the matrix into two separate matrices. This reduces the required meta data for the HCSS format. Additionally HCSS is sliced and uses a modified CSR, which improves memory alignment. The HYB format also does not use any index compression techniques.

The design of the HCSS format does respect most of the identified requirements and constraints. It Especially allows the efficient use of the vector units, proper memory alignment and consecutive memory accesses (Requirements R5, R4 and Constraint C4). It also supports a basic load balancing, even though it is not independent of the matrix structure as for the CSR5BC format (Requirement R6). The simple and small meta information as well as the used bit compression reduces the overall memory consumption of the format (Requirement R1).

The sliced layout of the format prevents interleaved and concurrent write operations on the $\vec{y}$ vector (Constraints C1 and C3). As each slice is independent from the other slices, a NUMA-aware implementation is no problem (Constraint C2). Furthermore, this in theory allows the asynchronous computation of the SpMV (Constraint C8). The simple layout of the format reduces the branching to a minimum (Constraint C5).

The creation of the format does not require any time consuming operations like reordering or pattern detection. Because of the required padding and the used index compression the format requires two iterations over the input data. In a first step the size of the required data structures has to be determined, while in a second step the actual conversion can be done. This still should be considerably efficient (Constraint C6).

The complexity of searching and updating an element in the HCSS format is very similar to CSR, as both formats provide row based accesses (Constraint C7). The location of non-zero elements in the HCSS formats can not be calculated easily, because of the introduced padding. This means the location in the CSR format can not be used to update an element in the HCSS format (see Section 3.3).

The HCSS format does require some simple heuristics to decide which slices should be stored in which format, which could be described as tuning parameters. The next chapter will describe heuristics to allow a simple format usage (Constraint C9).

One drawback of the format is, that the slice size of the format has to be equal to the vector size. This is very similar to the CSR5BC format and has the same disadvantage. When transferring a matrix to another hardware platform, it may be necessary to convert the matrix to the different vector size.

Overall the HCSS format, as CSR5BC, fulfills most of the identified requirements and constraints. But, the overall structure of the format is much simpler and requires much less meta information. This allows the use of a much simpler SpMV implementation, which may have positive effects on the performance. One drawback of the format is the used compression technique. The simpler compression is expected to achieve only smaller memory savings compared to CSR5BC. On the other hand, the compression does not require a sophisticated decompression.

In theory, the HCSS format is applicable for all three relevant hardware platforms, because all three use vector units or can be described as vector processors. Practically, the amount of padding can increase significantly, if the vector size gets much bigger. This is the case for GPUs, which work at 32 elements at a time. For this reason the work will concentrate on the use of the format on CPUs and the Xeon Phi platform.

The HCSS format does not directly utilize specific matrix structures, but may benefit from evenly distributed non-zero structures. Without much variation of the number of non-zero elements per row, a larger fraction of the slices can be stored as ELL, which may improve the performance. In contrast, a very unregularly distribution of the non-zeros can result in most elements being stored in the CSR format. Therefore, the format should deliver a very consistent performance for most general matrices. Performance drops are expected for matrices with very unregularly non-zero distributions.

The memory consumption of the format can be estimated using the following equation:

$$\begin{aligned} m_{HCSS} &\approx (nnz + fillin) \times S_{float} && //values \\ &+ (nnz + fillin) \times S_{int} \times f_{comp} && //columnIndex \\ &+ 2 \times nSlices \times S_{int} && //sliceStart \end{aligned}$$

The calculation of the size for the *values* and *columnIndex* array is very similar to CSR and CSR5BC. In addition to the compression factor, $f_{comp}$, the addition fill-in elements have to be considered. For each slice, two additional integers are stored in the *sliceStart* array, which is very similar to the *rowStart* of the CSR format. This fill-in can also be defined as an additional factor, which increases the number of non-zero elements. The number of slices $nSlices$ can be calculated by dividing the number of rows by the vector size. This allows a more compact definition of the equation:

$$m_{HCSS} \approx nnz \times f_{padding} \times (S_{float} + S_{int} \times f_{comp}) + \frac{nRows}{vectorsize} \times 2S_{int}$$

The average compression factor of the format has been calculated to be about 0.73 for the used set of matrices. Additionally, the average padding was measured to be about 3.4%, which results in the padding factor to be about 1.034. Using these and the previous assumptions, can be used to calculate the average memory consumption per non-zero element:

$$m\_nnz_{HCSS} \approx nnz \times 90.6$$

As for the CSR5BC format, the estimated memory consumption is compared in the last section of this chapter.

## 5.3 Development of Local Group Compressed Sparse Row – LGCSR

The optimization goal of the LGCSR format is on reducing the memory demand of the matrix with advanced index compression and blocking techniques to directly address the primary performance boundary of the SpMV operation (Requirement R1). In the development process of the CSR5BC and HCSS format, techniques for the memory reduction have been applied in a late stage. Because other optimizations have been the focus of the development, the memory savings are expected to be relatively small. For the LGCSR format less tradeoffs have been made, to reduce the memory demand as effective as possible.

Multiple techniques have been identified in the previous chapter for reducing the memory demand of a format. In the LGCSR format two of them should be combined, to reach a high compression factor. The basic idea of the format is to utilize a common matrix structural property for the blocking. Analysis of a large set of matrices (see Section 7.1) showed, that for most non-zero elements there are also elements in neighboring columns of the same row. One example where this property may derive from is the discretization of differential equations. When using finite difference methods, so called stencils occur in the generated matrix. These stencils often have a star-like structure, relating one central mesh element with its neighbors [77, p. 45 ff.]. These star-like structures result in groups of neighboring non-zero elements in a matrix. From these groups one dimensional row blocks can be created. These blocks should be of dynamic length, to prevent the requirement for additional fill-in. For each block the column index and the number of elements have to be stored. These index information should be highly bit compressed, to further reduce the memory consumption.

As no fill-in is required, the number of non-zero values should stay constant, compared to the CSR format. This allows an implementation of the LGCSR format that reuses parts of the CSR data structures, which potentially reduces the conversion time of the format (Constraint C6). Additionally this can also lead to very efficient update operations in applications designed for the CSR format (Constraint C7, see Section 3.3).

Since one-dimensional row blocks are used, it is reasonable to use a row based SpMV operation. This further allows a row based load balancing without negative effect on the compression. The matrix should be split into multiple partitions based on the number of non-zero elements, which is expected to deliver a proper load balance (Requirement R6). The partitioning further allows a NUMA-aware implementation (Constraint C2). It also prevents interleaved and synchronized writes on the $\vec{y}$ vector (Constraints C1 and C3).

### Description of LGCSR Format

Figure 5.6 illustrates the concept of storing one dimensional blocks, or later called local groups, using so called packages. The index information of one group is compressed into a single package, which consists of a header and a variable length payload. The packages encode the starting index of the group as delta to the previous group and the number of non-zero elements in the group. For reaching even higher compression, the number of bits for storing these information can be fitted to the required number of bits to some degree. Both fields can have 4 different sizes, which can be encoded with 2 bit per field in the package header. For the column index 4, 8, 16 or 32 bit can be used, while the number of elements in the group can be encoded using 4, 8, 16 or 27 bit. The 27 bit limit ensures, that the full package consisting of 5 bit header, up to 32 bit index and the 27 bit group size information does not exceed 64 bit.

The 5 bit header consist of 1 bit type information, 2 bit column index size

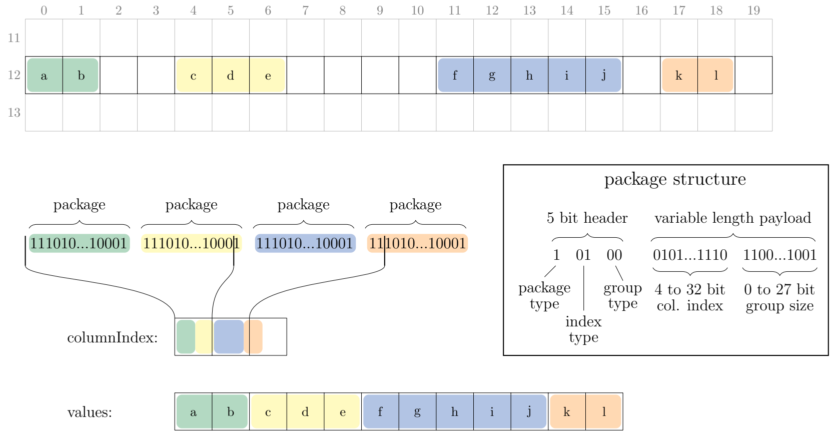
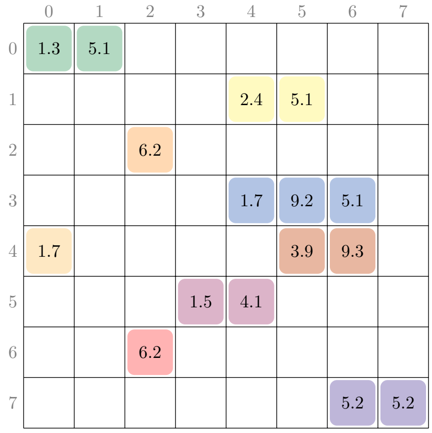


Figure 5.6: Concept of the LGCSR format.



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1.3 | 5.1 |  |  |  |  |  |  |
| 1 |  |  |  |  | 2.4 | 5.1 |  |  |
| 2 |  |  | 6.2 |  |  |  |  |  |
| 3 |  |  |  |  | 1.7 | 9.2 | 5.1 |  |
| 4 | 1.7 |  |  |  |  | 3.9 | 9.3 |  |
| 5 |  |  |  | 1.5 | 4.1 |  |  |  |
| 6 |  |  | 6.2 |  |  |  |  |  |
| 7 |  |  |  |  |  |  | 5.2 | 5.2 |

$$rowPtr = \{0, 2, 4, 5, 8, 11, 13, 14, 16\}$$

$$columnStart = \{0, 1, 2, 3, 4, 6, 7, 8, 9\}$$

$$values = \{1.3, 5.1, 2.4, 5.1, 6.2, 1.7, 9.2, 5.1, 1.7, 3.9, 9.3, 1.5, 4.1, 6.2, 5.2, 5.2\}$$

$$columnIndex = \{(0|2), (4|2), (2|1), (4|3), (0|1), (5|2), (3|2), (2|1), (6|2)\}$$

Simplified package structure ({index}|{length}) with 32 bit length

Figure 5.7: Simplified data structures of the LGCSR format.

information and 2 bit group size information. The package type is used to encode single non-zero elements in the matrix. In this case the group size information in the package is not present, which further reduces the memory demand of the format.

The overall data layout of the LGCSR format is presented in Figure 5.7. As described the format reuses the *values* and *rowStart* array of the CSR format. The packages, containing the column index information, are stored in the *columnIndex* array. As for most compressed formats the reduced size of the *columnIndex* array requires an additional offset array that points to the beginning of the arrays for every row. This array is called *columnStart*. The LGCSR format uses the same technique for improving the load balance (Requirement R6) as the HCSS format. Therefore, it also uses an additional *blockStart* array which manages the rows calculated by each thread (not shown in the figures).

### Discussion

The basic concept of creating dynamic one dimensional blocks is also used in the VBL format [71]. The main difference between VBL and LGCSR is a completely different approach for storing the blocks.

Even though less tradeoffs have been made in the development process of the LGCSR format, many requirements and constraints are fulfilled. The format clearly reduces the memory consumption by using blocking and bit compression techniques (Requirement R1). The developed data structures also allows consecutive memory accesses and the load balancing is expected to be improved by the used partitioning (Requirements R4 and R6).

The partitioning of the format prevents interleaved and synchronized memory accesses to the $\vec{y}$ vector and allows a NUMA-aware implementation (Constraints C1, C2 and C3). As each row of the matrix is independent of the others, it is also in theory possible to calculate the format asynchronously on an accelerator (Constraint C8). The reuse of the *values* and *rowStart* arrays allows an efficient conversion, as well as efficient updates (Constraints C6 and C7). The LGCSR format does not require any additional parameters (Constraint C9).

Even though many of the requirements and constraints are fulfilled, the format has multiple possible drawbacks. The variable length of the blocks makes the efficient utilization of the vector units very difficult (Requirement R5). The format furthermore does not provide proper memory alignment, which further reduces the efficiency of the vector units (Constraint C4). The use of one, instead of two dimensional row blocks has the additional disadvantage, that the cache reuse for elements of the $\vec{x}$ vector is not improved (Requirement R2). This can only be reached by two dimensional blocking or one dimensional column blocks, where the elements of the $\vec{x}$ vector are reused.

Since the format utilizes a specific matrix property to reduce the memory demand of the matrix, the compression factor may vary depending on the matrix structure. Even though the utilized property seems to be very common, there are matrices that do not have any neighboring non-zero elements. This means the format depends stronger on the matrix structure than the other developed formats. The complexity of the compression, and the different possible package types may also introduce issues according to additional branching (Constraint C5).

As described, the LGCSR format only allows a partial vectorization. This makes the format unsuitable for the use on GPUs, which fundamentally rely on the SIMD principle. It is therefore only implemented for the CPU and the Xeon Phi.

The LGCSR is expected to depend much stronger on the matrix structure than the CSR5BC and HCSS format. It should perform best for matrices with large row blocks. The performance is expected to drop, if no or only a small number of row blocks exist, but should still be acceptable. In difference to other blocked formats the overhead for storing the blocks is much lower, and the performance drop is therefore expected to be smaller.

The memory consumption of the LGCSR format can be estimated using the following equation:

$$\begin{aligned} m_{LGCSR} &\approx nnz \times S_{float} && //values \\ &+ nnz \times S_{int} \times f_{comp} && //columnIndex \\ &+ nRows \times S_{int} && //rowStart \\ &+ nRows \times S_{int} && //columnStart \end{aligned}$$

The memory consumption of the *values* and *columnIndex* array is identic to the CSR5BC format. Additionally the format requires two additional offset arrays that have the same memory consumption like the *rowStart* array of the CSR format. The equation can also be written more compact:

$$m_{LGCSR} \approx nnz \times (S_{float} + S_{int} \times f_{comp}) + 2 \times nRows \times S_{int}$$

The average compression factor of the format has been measured to be about 0.12. Using this and the previously made assumptions, the average memory consumption per non-zero element can be estimated:

$$m\_nnz_{LGCSR} \approx nnz \times 68.8$$

The memory consumption is compared with the other formats in the last section of this chapter.

## 5.4 Optimization of the DynB Format

The in the previous sections presented formats have been completely new or have introduced significant extensions to exiting formats. The development presented in this section is based on the existing DynB format [72]. However no significant changes to structure of the format will be done. In fact, the focus of the development is on the optimization of the execution of the SpMV operation itself using an autotuning approach. In general the autotuning approach could be applied to other formats as well, but it at least requires a blocking with reasonable sized blocks. The three newly developed formats are not suitable to be used with this approach. Even though the LGCSR format stores one dimensional blocks, the blocks are expected to be to small. The DynB format has a blocked structure and shows good SpMV performance. Therefore the DynB format has been selected to be optimized using the developed autotuning approach.

Before this optimization is motivated, the existing DynB format will be explained in detail. The structure of this section is different compared to the previous. The motivation for the format is kept much shorter, as the format has not been developed in this work. This will be followed by the detailed explanation of the DynB format. Afterwards the, in this work developed, optimization is explained in detail. Finally, a discussion will show which of the, in this work identified, requirements and constraints have been fulfilled. Furthermore, the possible benefits and problems of the introduced optimization are discussed.

The DynB format [72] is a format utilizing blocked structures within matrices. In comparison to the BCSR format [5], it can store rectangular blocks of variable size. In theory, variable block sizes allow a more general use of blocked formats, because the matrix can consist of a wider verity of blocked structures. By dynamically selecting the size of the blocks, less fill-in is required, compared to fixed size approaches. This approach is very similar to formats based on pattern detection (see Section 4.1). The major difference is that the DynB format stores rectangular blocks only.

Even though the storage of blocked structures requires additional meta information it is overall expected to reduce the memory consumption compared to typical formats like CSR (Requirement R1). Furthermore, the DynB format stores two dimensional blocks, which should improve the cache reuse for accesses on the $\vec{x}$ vector (Requirement R2). Another beneficial effect of the blocked structure is the efficient utilization of vector units (Requirement R5), but this may depend on the used block sizes.

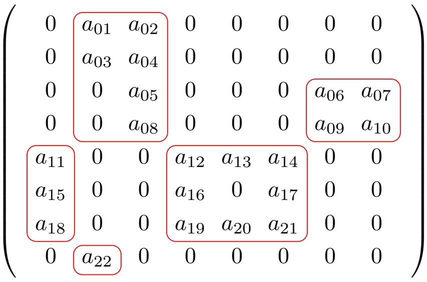
For the efficient parallel execution, the DynB format utilizes a similar distribution approach as the HCSS and LGCSR format. In a first step, the matrix is split into partitions, each containing a similar number of non-zero elements. The block detection is performed on each partition independently, which prevents blocks of one partition to overlap with another. This approach has the benefit, that no synchronization between the different threads is required, as each thread writes only to its own rows (Constraint C3). The partitioning improves the load balancing and prevents interleaved writes to the result vector (Requirement R6 and Constraint C1). It also allows a NUMA-aware implementation (Constraint C2).

### Description of the DynB Format

As already described, the DynB stores rectangular blocks of variable size. In the matrix creation process, the blocks of matrix have to be identified. This can be done using different approaches, but for this work only one, greedy, approach is of interest. First a threshold has to be defined, that describes the minimum non-zero density of the blocks, i.e., the number of non-zero elements in relation to the number of elements in the blocks (including fill-in). The greedy algorithm tries to increase the size of the block in the horizontal and vertical direction in every step. This is repeated as long as the density is above the set threshold. The size of the block is only increased, if additional non-zero elements where found. Additionally the maximum size of the blocks is limited to 64 elements.

Figure 5.8 presents the data structure of the DynB format, which is used for storing the identified blocks. The non-zero values of the blocks are stored in the *values* array in row-major order. The beginning of every block is stored in the *block_start* array, which allows to simply iterate over the blocks. Once for each block the row and the column index is stored in the *row_index* and *column_index* arrays.

As the size of the blocks is variable, the number of rows and columns has also to



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

$$\begin{aligned}
\texttt{values} &= \{a_{01}, a_{02}, a_{03}, a_{04}, 0, a_{05}, 0, a_{08}, \\
&\quad a_{06}, a_{07}, a_{09}, a_{10}, \\
&\quad a_{11}, a_{15}, a_{18}, \\
&\quad a_{12}, a_{13}, a_{14}, a_{16}, 0, a_{17}, a_{19}, a_{20}, a_{21}, \\
&\quad a_{22}\} \\
\texttt{block\_start} &= \{0, 8, 12, 15, 24\} \\
\texttt{row\_index} &= \{0, 2, 4, 4, 7\} \\
\texttt{column\_index} &= \{1, 6, 0, 3, 1\} \\
\texttt{block\_row} &= \{4, 2, 3, 3, 1\} \\
\texttt{block\_column} &= \{2, 2, 1, 3, 1\}
\end{aligned}$$

Figure 5.8: Data structure of the DynB format with a threshold of 0.75 [72].

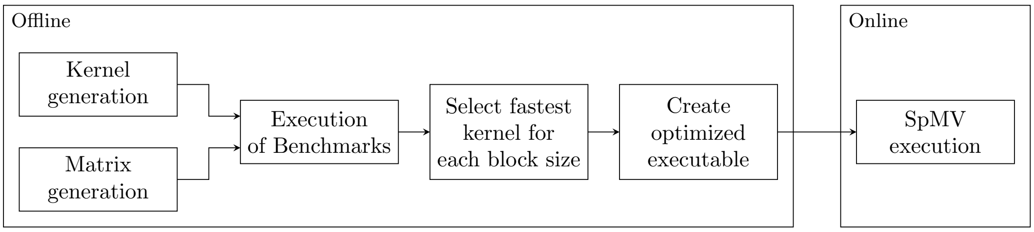


Figure 5.9: Simplified process of the DynB autotuning.

be stored for every block. This is done in the *block_row* and *block_column* arrays. Because the maximum size of a block in one dimension is 64, a smaller data type can be used for storing these information.

In the parallel implementation, each thread stores his partition of the matrix as DynB matrix. The partitioning is therefore done before the block finding step.

### Explanation of the Autotuning Approach

The DynB format supports 280 different block sizes. It is possible to calculate the SpMV using one general kernel that iterates over both dimensions of the blocks. 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 small blocks.

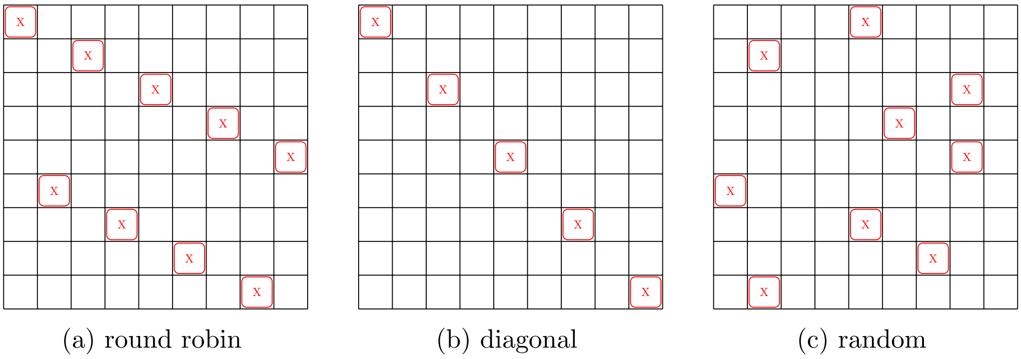
The basic idea of the autotuning approach is to identify the optimal implementation for each block size individually using a large set of possible implementations and synthetic benchmark matrices. The simplified process of the autotuning is presented in Figure 5.9. In a first step, the possible SpMV 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 the kernels and the benchmark matrices, while the execution time is measured. The gathered information can be used to identify the fastest implementation for each specific block size. These implementations are than 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 while the SpMV itself is executed, there is no overhead for the autotuning. In the following, the different steps are explained in more detail.

The kernel generation in the first step generates the required source code for all possible block sizes. Many kernels can be generated automatically, as they follow a fixed pattern. Additionally there are kernels, for example implementations using intrinsics that can not easily be generated. 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 identic 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**: Identic loop implementation as the **loop** kernel. Additionally the *pragma unroll* directive is used to generated unrolled code.
- **novec**: Identic loop implementation as the **loop** kernel. Additionally the *pragma novec* directive is used to prevent a vectorization of the code.
- **simd**: Identic loop implementation as the **loop** kernel. Additionally the *pragma simd* directive is used to enable 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 (see aspects discussed for the HCSS format in Section 5.2).

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



(a) round robin (b) diagonal (c) random

Figure 5.10: 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.

Figure 5.10 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. This creates a pattern that reminds of 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 benchmark execution uses the generated kernels and matrices and measures the SpMV execution time. This should be done with a single thread only, to avoid possible load imbalance problems. 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. This is further discussed, if differences between the different structures can be found.

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 a 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 chapter, where the implementation is described.

### Discussion

The discussion of the DynB format is split into two parts. The first part discusses the format itself, while the second part presents advantages, disadvantages and possible problems of the autotuning approach.

The DynB format fulfills most of the requirements, but only about half of the identified constraints. The format reduces the memory consumption by using a blocking approach (Requirement R1). It is furthermore the only approach that uses a two dimensional blocking, which can improve the cache reuse of the elements loaded from the $\vec{x}$ vector (Requirement R2). The data layout also allows consecutive memory accesses (Requirement R4). Depending on the identified block sizes, the block structure allows a proper vectorization (Requirement R5). Very similar to the HCSS and LGCSR format, the matrix is partitioned based on the number of non-zero elements, which provides a basic load balancing (Requirement R6). The partitioning further prevents interleaved and synchronized writes to the $\vec{x}$ vector and allows a NUMA-aware implementation (Constraints C1, C2 and C3). The blocks of the DynB format are independent from each other, which in theory allows a asynchronous computation of the format (Constraint C8). The simple data structures allow a implementation of the SpMV operation without a lot of branching (Constraint C5). If multiple kernels are used to calculate blocks of different size, branching could be a problem.

The detection of blocks in a matrix is a very time consuming process (Constraint C6). The time is reduced by using a greedy algorithm, but the conversion time is still expected to be much higher compared the three other developed formats. Furthermore, the DynB does not allow the identification of elements in a specific row or column of the matrix. This has negative effects on the efficiency of matrix element updates (Constraint C7). The varying block sizes of the format have a negative effect on the memory alignment (Constraint C4), new blocks do not start at specific memory boundaries. The DynB format uses a threshold parameter, which has to be defined (Constraint C9).

The DynB format is obviously designed for matrices with a blocked structure. It is therefore expected to also perform best for this type of matrices. In comparison to formats with a fixed block size, like BCSR, it is assumed, that DynB is suitable for a wider range of blocked structures. The use of variable block sizes additionally simplifies the usage of the matrix format, as no assumption about the actual block size has to be done before the conversion. The necessity to define the threshold parameter on the other hand makes the conversion more complicated.

The developed autotuning approach for the DynB format has similarities to the pOSKI framework [12]. 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.

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 to 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. 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 the parallel executions. It is possible that these assumptions do not hold true, which would result in wrong findings.

The memory consumption of the DynB format can be estimated using the following equation:

$$\begin{aligned} m_{DynB} \approx\ & (nnz + fillin) \times S_{float} && //values \\ & + nBlocks \times S_{int} && //blockStart \\ & + nBlocks \times 2 \times S_{int} && //columnIndex, rowIndex \\ & + nBlocks \times 2 \times S_{char} && //blockRows, blockColumns \end{aligned}$$

Very similar to the HCSS format, the *values* array contains additional fill-in. For each block additional meta information is required: The offset in the values array, the column and row index and the block dimensions. The variable $S_{char}$ denotes a smaller index type that is used for storing the block dimensions. The number of blocks can be estimated by dividing the number of non-zero elements by the average *blocksize*. The equation can also be written more compact:

$$m_{DynB} \approx nnz \times (f_{padding} \times S_{float} + \frac{3S_{int} + 2S_{char}}{blocksize})$$

The average padding for the DynB format has been determined to be about 3.1%. Therefore, $f_{padding}$ is assumed to be 1.031. Because of the maximum block sizes of the DynB format, $S_{char}$ is assumed to be 8 bit. The memory consumption of the format strongly depends on the average block size. This parameter varies strongly for the different matrices which makes the general estimation very difficult. By analyzing the matrix set, the average block size is assumed to be 4. Using these assumptions the average memory consumption per non-zero element is as following:

$$m\_nnz_{DynB} \approx nnz \times 94$$

| Requirement / Constraint | CSR5BC | HCSS | LGCSR | DynB |
|---|---|---|---|---|
| R1 (reduce memory demand) | ✓ | ✓ | ✓ | ✓ |
| R2 (improve cache usage) | | | | ✓ |
| R3 (improve access latency) | | | | |
| R4 (consecutive memory access) | ✓ | ✓ | ✓ | ✓ |
| R5 (vector unit utilization) | ✓ | ✓ | | ✓ |
| R6 (load balancing) | ✓ | ✓ | ✓ | ✓ |
| C1 (no interleaved writes) | ✓ | ✓ | ✓ | ✓ |
| C2 (NUMA-awareness) | ✓ | ✓ | ✓ | ✓ |
| C3 (no synchronized writes) | ✓ | ✓ | ✓ | ✓ |
| C4 (proper memory alignment) | ✓ | ✓ | | |
| C5 (prevent branching) | | ✓ | | ∽ |
| C6 (efficient format creation) | ✓ | ✓ | ✓ | |
| C7 (efficient matrix update) | ✓ | ✓ | ✓ | |
| C8 (asynchronous computation) | ✓ | ✓ | ✓ | ✓ |
| C9 (no tuning parameters) | ✓ | ✓ | ✓ | |

Table 5.1: Overview over the by the formats fulfilled requirements and constraints.

## 5.5 Summary and Discussion

In the previous sections three new formats have been developed and the existing DynB format has been optimized. Different optimization goals have been in the focus of each of the formats. Furthermore the development resulted in rather different data structures. Table 5.1 summarizes the fulfilled requirements and constraints of the newly developed formats and DynB. It can be seen, that all formats fulfill a large, but different fraction of the requirements and constraints. The HCSS format fulfills the most requirements and constraints of the presented formats, while the DynB format fulfills the least. The bare number of fulfilled requirements and constraints is not expected to allow any assumption about the performance of the SpMV operation. The requirements have different impact on the SpMV performance, some of them also have no impact at all (e.g., the format creation or element updates).

It can be seen, that no format was able to reduce the access latency on the $\vec{x}$ vector (Requirement R3). The reason for this is, that no general usable optimization technique could be identified, which could solve this problem. Furthermore the DynB format is the only format that improves the cache reuse of the elements from the $\vec{x}$ vector. For the CSR5BC, HCSS and LGCSR only vertical partitioning or reordering approaches would have been suitable to fulfill this requirement. No reordering has been used, as it significantly increases the complexity of the matrix creation process. A vertical partitioning of a the matrix introduces other issues, like potential synchronization. It is therefore not assumed to deliver a overall positive effect on the SpMV performance.

| **Matrix Format** | **Memory Estimation Equation** | **Bit per non-zero** |
|---|---|---|
| CSR5BC | $nnz \times (S_{float} + S_{int} \times f_{comp} + \frac{S_{int}}{\omega \times \sigma}$ <br> $\times [3 + \lceil (\log_2(\sigma) + 2\log_2(\omega) + \sigma)/S_{int} \rceil \times \omega + \lceil (6 \times \sigma)/S_{int} \rceil])$ | 77.3 |
| HCSS | $nnz \times f_{padding} \times (S_{float} + S_{int} \times f_{comp}) + \frac{nRows}{vectorsize} \times 2S_{int}$ | 90.6 |
| LGCSR | $nnz \times (S_{float} + S_{int} \times f_{comp}) + 2 \times nRows \times S_{int}$ | 68.8 |
| DynB | $nnz \times (f_{padding} \times S_{float} + \frac{3S_{int} + 2S_{char}}{blocksize})$ | 94.0 |
| CSR | $nnz \times (S_{float} + S_{int}) + nRows \times S_{int}$ | 97.0 |

Table 5.2: Comparison of the memory consumption of the different Formats.

For all formats an estimation of the memory consumption has been done. It is important to mention, that the estimations represent the average case for the in this work used set of matrices. Table 5.2 presents the equations required for estimating the memory consumption of the formats. Additionally it shows the average number of bits required for every non-zero element, using the assumptions presented in the previous sections. For the CSR format the equation from Section 2.1.2 and the assumptions done in this chapter have been used for the estimation. It can be seen that the memory consumption of all presented formats is estimated to be lower than the memory consumption of the CSR format. The LGCSR format thereby has the lowest estimated memory consumption, which matches the optimization goal of the format. The HCSS format has the highest memory consumption of the developed formats. In the evaluation in Chapter 7 the memory consumption for the matrices is analyzed empirically. This will show the accuracy of the calculated estimations.

# 6 Implementation

In this chapter implementation specific details are described. The focus thereby is on the conversion operations of the formats and performance critical details of the SpMV implementation. Only relevant parts of the code are shown, the full source code can be found on the CD attached to this work.

All described implementations have been done inside an existing software environment. The software already offers the basic functionality for managing the vectors, reading in matrix and vector data and checking the correctness of an executed calculation. Using the existing interfaces it is only necessary to provide the proper conversion methods from the CSR format into the new format and the SpMV implementation. In some cases additional minor data handling functions are required, which are used to copy the matrix data to another device, like the Xeon Phi or a GPU. The focus in the following sections is on details of the conversion method and the SpMV operation of the formats only. Additionally the implementation of the autotuning approach is explained in detail.

In the following, the implementation of the three formats CSR5BC, HCSS and LGCSR is described in separate sections. The last section describes the implementation details of the developed autotuning approach.

## 6.1 Implementation Details of the CSR5BC Format

The CSR5BC format is based on the CSR5 format developed by Liu et al. [51]. A reference implementation for the CSR5 format is provided as open-source by the author for all three platforms relevant in this work. The CPU and Xeon Phi implementation is done using OpenMP [68] and AVX2 [31] intrinsics. The GPU implementation is done with Compute Unified Device Architecture (CUDA) [64]. These implementations are used as basis for the development of the CSR5BC format.

The first step is to modify the creation process of the format. As the CSR5 and CSR5BC format share the most data structures, most parts of the CSR5 implementation can be reused. Two additional steps are required at the end of the CSR5 creation process. The first step creates the compressed column index structures in the existing CSR5 *columnIndex* array. It furthermore creates the required block meta information in the *compressionPointer* array. By using the existing *columnIndex* array, the compression of the column data can be done directly, even though the final, compressed size is unknown. After this step the reduced space required for storing the compressed column information is known. In the second step the compressed column information are copied into a new, tightly packed,

```
 1 if(lane_id < compression_pointer_size)
 2     localVal = compression_pointer[lane_id];
 3
 4 globalOffset = __shfl(localVal, 0);
 5
 6 // each entry in the compression_pointer is 6 bit long
 7 packetPos = 1+(6 * lane_id)/32;
 8 bitStart  = (6 * lane_id)%32;
 9 numBits = 0 | ( __shfl(localVal, packetPos) << bitStart) >> (32-6);
10 remainder = __shfl(localVal, packetPos+1) >> (32-6+32-bitStart);
11 if(32-bitStart < 6)
12     numBits |= remainder;
13 ...
14 for (int i = 0; i < sigma; i++)
15 {
16     numBitsCurrent = __shfl(numBits, i);
17     ...
18 }
```

Listing 6.1: Simplified reading of the *compressionPointer* array and unpacking of meta information on CUDA using shuffle functions.

*columnIndex* array. The conversion in the reference implementation is done on the specific device used, which means multiple implementations had to be modified. Nevertheless most part of the code could be reused.

Beside the creation process obviously the SpMV operation itself had to be modified. One important part of the SpMV operation is reading the *compressionPointer* array, which is holding the compression related meta information of the blocks. Listing 6.1 shows the relevant and simplified parts of the GPU kernel. The *compressionPointer* values are read only once, by consecutive threads (line 2). Afterwards the relevant parts of the arrays are exchanged efficiently between the different threads using *shuffle* functions. Each thread thereby determines the number of bits required for one row of the block (lines 6–11). For every row of the block, the currently used number of bits is exchanged between the threads, again using the *shuffle* function (line 13–17). The *shuffle* intrinsics allow the efficient exchange of data between multiple threads without using the shared memory of the GPU [64].

The actual index decompression is shown in Listing 6.2. The decompression only requires the use of some bit shifts, and if necessary the loading of a new element of the *columnIndex* array. One important advantage of the used compression is, that the loading of new array elements is always done concurrently using all available threads (line 6). This is because all threads use always the same number of bits for the compression in the same iteration. This has been an important point in the development of the format. This leads to proper, consecutive memory accesses.

The rest of the SpMV operation is mostly reused from the reference implementation and is therefore not subject of further discussion. The presented code

```
numBitsCurrent = __shfl(numBits, i);

columnIndexTmp = 0 | ((columnIndexLocal << (32-numBitsLeft)) >> (32 - numBitsCurrent));
if(numBitsLeft < numBitsCurrent)
{
    columnIndexLocal = columnIndex[numPacketsLoaded*OMEGA+lane_id];
    numPacketsLoaded++;
    numBitsLeft = 32 - (numBitsCurrent - numBitsLeft);
    columnIndexTmp |= (columnIndexLocal >> numBitsLeft);

} else
    numBitsLeft -= numBitsCurrent;

columnIndex += columnIndexTmp;
```

Listing 6.2: Decompression of the actual column indexes in the innermost loop of the SpMV.

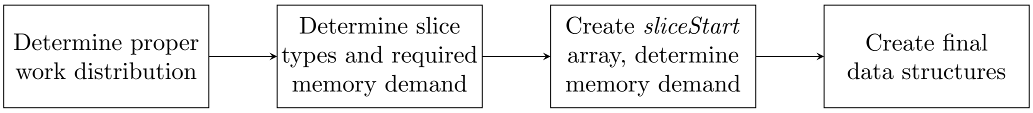

Determine proper work distribution
Determine slice types and required memory demand
Create *sliceStart* array, determine memory demand
Create final data structures

Figure 6.1: The four phases required in the creation method of the HCSS format.

fragments have been taken from the CUDA version of the code. The code for the CPU and Xeon Phi is very similar, but only one thread is used for processing each block, which simplifies the process presented in Listing 6.1. The effective use of the vector units is realized by using compiler intrinsics and manual unrolling. This is not shown here as, especially the unrolling, leads to relatively long code fragments.

## 6.2 Implementation Details of the HCSS Format

The HCSS format is implemented for the CPU and Xeon Phi platform only. It is realized using OpenMP, vectorization annotations and compiler hints. As the implementation for the Xeon Phi platform is essentially the same as the CPU implementation, the relevant parts are only discussed once.

The creation of the format is done in parallel, which reduces the creation time significantly. Even though some time has been spent on implementing an efficient conversion method from CSR to HCSS, it is assumed, that the method can be further optimized. Obviously the focus of this work is the implementation of the SpMV operation.

The creation process is divided into four phases, which are presented in Figure 6.1. In the following, these phases are explained in more detail. In the first step the *blockStart* array is created, that specifies, which thread processes which slices. The distribution is done based on the number of non-zero elements in the slices. The simple greedy algorithm tries to distribute the number of non-zero elements evenly between the threads. Most of this functionality existed already in the software framework, but on a row basis instead of slices. The code has been adapted to distribute the work based on the slices used in the HCSS format. The whole creation process of the matrix works based on this work distribution. This ensures the correct allocation of memory, as each thread processes the rows it is also supposed to process in the SpMV operation (Constraint C2).

In the second step, the existing *rowStart* data from the CSR format is copied to ensure a proper NUMA-aware allocation. Additionally the matrix is traversed and the type of every slice is determined. First it is decided, if the ELL format can be used, or if the required amount of padding is to high. The decision is based on basic structural data of the matrix slices: the number of non-zero elements in the shortest and the longest row. Slices should only be stored as CSR if two conditions are true: The number of non-zero elements exceed a specific threshold and the ratio between the shortest and longest row exceed a given factor.

Empirically the two threshold have been defined to be 16 for the minimum number non-zero elements and the shortest row has to be at least 20% shorter than the longest row. Other heuristic or thresholds may have a positive effect on the performance of the format. The given heuristic is used to proof the general concept of the format and simplifies the use of it (Constraint C9). If the use of ELL is suitable, the compressed version is used, whenever the indexes can be compressed in 16 bit and the rows have at least 3 entries. Part of the second step is the partial creation of the *sliceStart* array, by writing the number of actually stored elements in each slice and the type of the slices.

The third step is the calculation of two prefix sums over the *sliceStart* array for creating the final data structure. As the slice types are also encoded in the first bits of the integers, a modified prefix sum is used, which handles the first two bit separately. The last elements of the *sliceStart* array can afterwards be used to determine the memory demand of the *columnStart* and *values* arrays.

The fourth step is to copy the actual values and required padding to the new *values* array. Additionally the compressed and uncompressed column indexes have to be calculated and required padding has to be added to the *columnIndex* array.

As the data structures of the HCSS format are very simple by intention, the implementation of the basic SpMV operation is straightforward. With the help of the vectorization report [20] of the Intel compiler, the implementation has been extended with additional compiler hints and vectorization directives. Listing 6.3 shows the central part of the compressed ELL SpMV kernel. The innermost loop shows the actual SpMV calculation. Each iteration calculates the partial results of two non-zero elements, whose index data has been compressed. The first step is loading the next 32 bit index (line 4) and extracting the two 16 bit indexes using bit shifting and masking (lines 5 and 6). The `__assume` function is used, to ensure the compiler the proper data alignment for the accesses on *vValues*. Many such hints for the compiler are necessary (not shown here) to force the compiler to generate aligned, more efficient, vector code. Finally the Intel vectorization report confirms,

```
for(index_t j = 0; j < width-1; j++){
  #pragma simd aligned
  for(index_t i = 0; i < VECTOR_SIZE; i++){
    unsigned int index = columnIndex[offsetIndex+(j+1)*VECTOR_SIZE+i];
    index_t index1 = index & 0xFFFF;
    index_t index2 = (index & 0xFFFF0000) >> 16;

    prevIndex[i] += index1;
    __assume(prevIndex[i]%16==0);
    sum[i] += values[offsetValues+...] * vValues[prevIndex[i]];

    prevIndex[i] += index2;
    __assume(prevIndex[i]%16==0);
    sum[i] += values[offsetValues+...] * vValues[prevIndex[i]];
  }
}
```

Listing 6.3: Excerpt of the compressed ELL kernel of the HCSS format.

that the inner loop is properly vectorized and that all memory accesses have been properly aligned.

## 6.3 Implementation Details of the LGCSR Format

The LGCSR format is also implemented for the CPU and the Xeon Phi platform only. The format has been realized using OpenMP parallelism and compiler intrinsics for additional vectorization.

The conversion process from the CSR to the LGCSR format is very similar to the one of the HCSS format and process can be also be divided into the same four steps. In the first step of the creation, the work distribution is determined. In difference to the HCSS format this is done on a row base.

The second step is used to identify the local groups in the rows, which is necessary for determining the final memory demand. In this phase the number of required index elements is stored in the *columnIndexStart* array. In the third phase, the prefix sum over the *columnIndexStart* is calculated, to generate the needed offset array. In comparison to the HCSS format a simple prefix sum operation can be used, as no additional information is encoded in the leading bits of the array elements.

In the last step the space demand is known and the final data structures can be allocated. The data is copied and the compressed data index information are generated.

In the SpMV operation, the process of loading and decoding the index information is a important aspect. Listing 6.4 shows the simplified code of the relevant kernel part. It shows the loading of a 64 bit package, which includes the loading of new elements from the *columnIndex* array if necessary (line 2–10). As the maximum package size is defined to be 64 bit, it is guaranteed, that the 64 bit contain

```
 1  // check if new data has to be loaded
 2  if(bitsLeft < 64){
 3      package = package_current << (64-bitsLeft);
 4
 5      package_last = package_current;
 6      package_current = columnIndexRow[packageNumLoaded++];
 7      package |= package_current >> bitsLeft;
 8
 9      bitsLeft += 64;
10  }else{
11      if(bitsLeft == 64){
12          package = package_current;
13      }else{
14          package = package_last << (64-(bitsLeft-64));
15          package |= package_current >> (bitsLeft-64);
16      }
17  }
```

Listing 6.4: Simplified implementation of the package loading process of LGCSR.

the full package (potentially more). If the read package is smaller than the loaded 64 bit, the additional bits belong to the next package. Two buffers are used for loading the data and preventing multiple reads of the same elements (line 10–17).

Listing 6.5 shows the actual SpMV operation for the LGCSR format. Compiler intrinsics are used for vectorizing the code (line 4–6 and 12–16). The vectorization is thereby only used, when at least 4 elements can be processed at once (the group is at least 4 elements big). If this is not the case, a scalar operation is used (line 8). Like all row-major order formats, the vectorization of the LGCSR format requires an additional reduction at the end of every row (line 12–16).

## 6.4 Implementation Details of the DynB Autotuning

Most parts of the autotuning approach are implemented using Python scripts in combination with the previously described software framework. Not the whole autotuning process has been automated, as this would have been out of scope of this work. The most time consuming steps, e.g., the kernel creation and benchmark execution, have been automated.

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 be written by hand. One example for this are implementations using intrinsics. The kernel creation scripts takes the 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 are stored in a

```
for(index_t jj = rowStart[row]; jj < rowStart[row+1]; jj++){
    // ... loading and decoding of packages
    if(numInGroup > 4){
        x256d = _mm256_loadu_pd(&vValues[currentColumnIndex]);
        value256d = _mm256_loadu_pd(&values[jj]);
        sum256d = _mm256_fmadd_pd(value256d, x256d, sum256d);
    }else{
        sum += values[jj] * vValues[currentColumnIndex];
    }
}
// reduce the vector unit values to a single value
hiQuad = _mm256_extractf128_pd(sum256d, 1);
loQuad = _mm256_castpd256_pd128(sum256d);
hiDual = _mm_add_pd(hiQuad, loQuad);
loDual = _mm_permute_pd(hiDual, 1);
sum128d = _mm_add_sd(hiDual, loDual);

_mm_storel_pd(&sumTmp, sum128d);
uValues[row] = sum+sumTmp;
```

Listing 6.5: The actual SpMV operation of the LGCSR format.

large database. This step is repeated several times for every kernel using different compilers and optimization options.

The selection of the fastest kernels is done manually and is described in more detail in the evaluation. The creation of the optimized executable is a semi-automated process. The in the previous step selected kernels have to be provided to a script, which than creates the code for the SpMV operation.

A very 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 chapter, 280 different block sizes and kernels are possible, which potentially introduces a lot of additional branching. The use of simple conditions is not suitable, as this would require the evaluation of many expressions for every block calculation. Each time one block is calculated, a series of conditions would have to be checked, until the correct block kernel is identified.

Instead of this another approach has been used. In theory switch case statements can be executed much more efficient than a series of conditional statements. The cases of a switch case statement have to be constant at compile time, which allows for very efficient optimizations of the compiler. When the numerical values of the cases are close to each other, the compiler can create a jump table instead of conditional statements. The n-th row of the table thereby contains a jump directive for the case with the constant value n.

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.

Optimally it would be possible, to use only one large switch case statement, that contains all 280 cases. This would require the transformation of the two dimensional number space of the block sizes into one, consecutive, number space. Because the possible block dimensions are limited by the number of elements in the block, only specific combinations are possible. For example are up to 64 element long or high blocks possible, but only if the other dimension is 1. The transformation is not easily computable, and would introduce significant overhead.

Therefore a more suitable approach is to use nested switch case statements. The first switch case statement is used to identify the correct number of rows, the switch case on the second level identifies the actual kernel by finding the correct number of columns. Using this, two jumps are required for finding the correct implementation for the current block which is expected to be acceptable. The evaluation in the next chapter has to show, if it suitable to provide kernels for all possible block sizes. As already discussed, the amount of program code may have negative effects on the SpMV performance, as the size of the instruction caches may be a bottleneck.

# 7 Evaluation

In this chapter the previously developed formats are evaluated. In the first section, the experimental setup and the methodology is described. Afterwards preliminary investigations are done. Following, the performance of the developed formats is evaluated on the three relevant hardware platforms. In the next section the memory consumption and the conversion effort of the developed formats is evaluated. The performance of the DynB autotuning is evaluated in the following section. The chapter closes with a discussion and summary of the findings.

## 7.1 Experimental Setup and Evaluation Methodology

Evaluating the performance of a matrix format is a complex process [49]. The performance can be influenced by many parameters, e.g, the used hardware platform, the software configuration or format parameters. The work of Langr et al. [49] describes important evaluation criteria for matrix formats, which will partly be used in this work. This section will describe the experimental setup and the used evaluation methodology.

### Experimental Setup

A large set of 73 square matrices is used for all parts of the evaluation. The set consists of matrices from the University of Florida Sparse Matrix Collection[15] and the Comparative Solution Project[80]. All matrices vary in their size and properties, to provide a broad spectrum of different matrix types. Table A.1 in the Appendix presents the full set with a selection of properties. The used set contains many matrices that are commonly used in other publications, which improves the comparability.

Table 7.1 presents selected technical details of the three hardware systems used in the evaluation. The presented CPU system consists of two processors, all performance numbers consider the whole system and not the individual processor performance. The Tesla K80 GPU consists of two separate GPUs on a single PCB, in the benchmarks only one of the GPUs is used. Therefore, the individual performance numbers of each core are listed in the table. Both GPUs of the Tesla K80 behave as individual devices, while the CPUs work as one large system. For this reason both CPUs, but only one of the GPUs is used.

The used software environment can also have a significant impact on the performance. All measurements for the CPU system have been taken using the Intel compiler 2017 [33]. The only exception are the measurements of the CSR5 and

| **Parameter** | **Intel Xeon E5–2680 v3** | **Nvidia Tesla K80** | **Intel Xeon Phi 5110P** |
|---|---|---|---|
| Clock [GHz] (with TurboBoost) | 2.5 (3.3) | 0.560 (0.875) | 1.053 |
| Degree of parallelism | 48[2] | 2 × 2496[3] | 240 |
| Main memory size [GB] | 128 | 2 × 12[3] | 8 |
| Theor. peak performance [GFlops] | 960 | 2 × 935[3] | 1010.8 |
| Memory bandwidth [GB/s][1] | 114.2 | 199.3 | 132.6 |

[1] Measured using the stream benchmark [55].

[2] Including hyperthreading.

[3] Only one GPU is used for the measurements.

Table 7.1: Technical details of the three used hardware platforms.

| **Compiler** | **Version** |
|---|---|
| Intel 2015 | 15.0.1 |
| Intel 2016 | 16.0.2 |
| Intel 2017 | 17.0.0 |
| Nvidia | 7.5.17 |
| GNU | 6.2.0 |

Table 7.2: Overview over the used compilers and versions.

CSR5BC format, which have been measured using the Intel compiler 2016. This is necessary, because the CSR5 implementation is not compatible with the newest version of the Intel compiler. The measurements on the Xeon Phi are taken using the Intel compiler 2015, as the newer versions could not be used because of license restrictions. The CUDA code is compiled using version 7.5 of the Nvidia CUDA compiler [65]. For the measurements of the DynB autotuning the newest Intel compiler 2017 and a recent version of the GNU compiler [19] are used. Table 7.2 shows the exact compiler versions used in the evaluation.

The primary performance factor is the measured runtime of the SpMV operation. The time is measured by the used software framework. Runtime measurements can be influenced by many different factors, e.g, background system processes. The accuracy and consistency of the measurements is increased by executing the SpMV 100 times and using the median runtime as measure in the evaluation.

### Evaluation Methodology

The evaluation is split into multiple sections. Before the actual evaluation, a preliminary investigation is done in Section 7.2 to identify the, for the evaluation, relevant matrix formats and to define important parameters. In the Sections 7.3–7.5 the developed matrix formats are evaluated on the three hardware platforms.

Each platform is discussed in a separate section allowing a direct comparison of the developed formats. In Section 7.6, the memory consumption and the conversion effort of the developed formats is evaluated. The performance of the DynB autotuning is evaluated in Section 7.7. Finally in Section 7.8, the findings are discussed and summarized.

The performance analysis of the developed formats on the different platforms is primarily based on two specific performance comparisons. First, the performance of the formats is compared to the performance of the CSR format. The comparison to CSR is also suggested by Langr et al. [49], because it allows a better comparability to other publications. This is because the CSR format is the most commonly used matrix format and by calculating the speedup to CSR, the performance is abstracted from the specific hardware.

In a second step, the performance of the developed formats is compared to a set of relevant matrix formats (see Section 7.2). First the speedup is calculated by identifying the fastest format for every matrix from the list of relevant formats. Then the runtime of the developed format is compared with the previously identified runtimes. Depending on the findings of these two comparisons, additional investigations may be done individually for every hardware platform.

It is important to mention, that all these performance comparisons are done against other, highly optimized, matrix formats. This means that performance improvements of a few percent are already very good. Since the SpMV operation is often the most time consuming operation in applications, small performance improvements can have a significant impact on the overall execution time. In the second comparison against the best formats for every individual matrix, it is important to know that the performance of most formats depend on the matrix structure. Additionally some formats are specialized to achieve very high performance for very specific matrix properties, e.g., for blocked matrices. Therefore, it is very unlikely, or even impossible, that a single matrix format can outperform all other formats for any given matrix. Furthermore it is even unlikely, that a single format can reach a median speedup above 1, which would mean it is the fastest format for at least 50% of the matrix set. The comparison to the fastest formats allows to answer three important questions:

1. For how many matrices a speedup can be achieved? For these matrices the new format outperformed all existing formats and its the new fastest format.

2. How much performance is lost in median? This is the median performance loss if always the new format is used, instead the optimal format.

3. What is the highest performance loss? This shows the maximum performance loss if always the new format is used, instead the optimal format.

Thereby, it is very unlikely that the best format for every matrix can be determined at runtime. Thus all numbers present the worst-case.

In the following evaluation primarily boxplots are used for the presentation of the results. Since multiple variations of boxplots exist, the used representation is

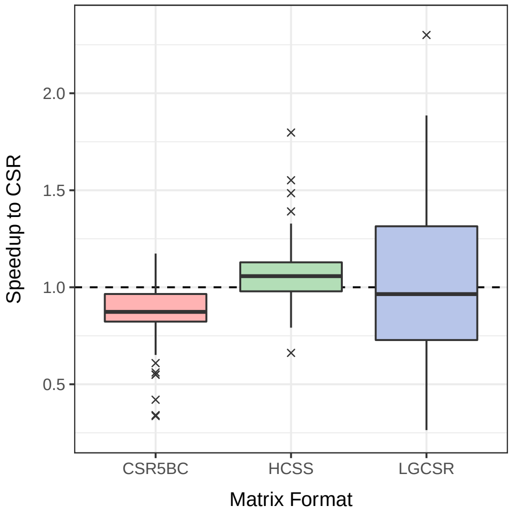


Figure 7.1: Example for the boxplots used in this work.

explained in more detail. Figure 7.1 shows an example of the used boxplots. The colored box thereby marks the area in which 50% of the data reside. Therefore, the upper and lower bounds are the 25th and 75th percentiles. The line in the box additionally marks the median.

The vertical lines on top and below the box are called whisker. The maximum length of the whisker is defined as the 1.5 of the Interquartile Range (IQR). The IQR is calculated by the difference between the 75th and 25th percentiles, which matches the hight of the colored box. The whiskers reach to the lowest and highest value still without the maximum length starting form the box. All values below or above the whiskers are considered as outliers. They are shown as individual values in the figure.

## 7.2 Preliminary Investigation

As previously described, the first step of the evaluation is the identification of the relevant matrix formats for the performance comparisons in the following sections. This is done by identifying the fastest formats for the different matrices on the relevant platforms. The analysis is thereby made on a large dataset. The dataset contains 13 different matrix formats on the CPU, 7 on the GPU and 5 formats on the Xeon Phi. Furthermore, it also contains various different configurations for every matrix format. This includes different degrees of parallelism, format specific parameters and multiple different SpMV implementations.

From this dataset the matrix format with the fastest SpMV execution is identified for each individual matrix. For some matrices and formats the runtimes are very similar, consequently not a single format could clearly be determined as fastest. This is additionally problematic because of possible measurement inaccuracies. For

| Format | CPU |
|---|---|
| CSR | 37 |
| DynB | 29 |
| BCSR | 14 |
| CSR5 | 9 |
| Others | 3 |

| Format | GPU |
|---|---|
| ELL-BRO | 34 |
| ELL | 28 |
| CSR5 | 26 |
| HYB | 17 |
| CSR | 12 |
| SELL-C-$\sigma$ | 2 |

| Format | Phi |
|---|---|
| CSR | 57 |
| CSR5 | 23 |

Table 7.3: Number of occurrences of the different formats in the list of fastest formats on the relevant platforms.

these reasons all formats within 5% from the fastest runtime are assumed to be equal and are thus all considered as fastest. The relevant matrices are determined by counting the number of occurrences of the different formats over the full set of matrices. The resulting ranking gives a general view on the relevance of the different formats. Because of the 5% range, the list can contain more occurring formats than the number of benchmark matrices.

Table 7.3 shows the number of occurrences of the different matrix formats in the rankings. On the CPU overall only 4 formats are relevant. The CSR format occurs most often, followed by the DynB format, the BCSR format and CSR5. On the GPU system, the distribution is very different. Here are 5 relevant matrix formats. Both ELL based formats ELLpack Bit Representation Optimization (ELL-BRO) [85] and basic ELL occur most often, closely followed by the CSR5 format, the HYB format and CSR. The SELL-C-$\sigma$ format occurs only twice, which makes it irrelevant for the later comparisons. On the Xeon Phi, only 2 of the 5 measured formats are relevant. For most matrices the CSR format delivers the best performance, followed by the CSR5 format.

For some of the formats, multiple versions of the SpMV operation exist, e.g., using intrinsics or *simd* compiler directives. This is only relevant for the CPU implementation. By analyzing the occurring versions in the list of fastest matrix formats, the used implementations have been selected. For the CSR5BC and the LGCSR format most often the intrinsic implementation performed best and is therefore used in the evaluation. For all other formats the default implementation is used.

In the next step, proper parameters for the relevant formats have to be identified. For all CPU and Phi implementations one important parameter is the number of used threads. Based on the measurements of all different thread configurations, the best number of threads has been identified for every format. This was done by first identifying the runtime with the optimal number of threads for each individual matrix. This runtime is then compared with the runtime using a fix number of threads for all matrices. By comparing the average slowdown over all matrices using the fixed numbers of threads, the best general setting has been identified.

The analysis showed that the use of the full parallelism including hyper-threading results in the best average performance for all formats and platforms. On the CPU all 48 threads are used, which results in an average performance loss of 1–5%. On the Xeon Phi, the average performance loss by using all 236 threads is 1–8%. Even though, the fix selection may result in a performance loss, this is a very practical approach, as in a real use-case the optimal number of threads is also unknown.

In addition to the thread mapping, the two formats DynB and BCSR require the definition of further parameters. For the DynB format this is the threshold value, which defines the maximal allowed fill-in in the blocks. The optimal value has been identified in the original publication [21] to be 0.85 and is therefore also used in this work. The performance of the BCSR format highly depends on the selected block size, which again depends on the matrix structure. It is unlikely that the selection of one fixed block size results in a practical relevant performance. Therefore, always the best found block size is used for all mentioned runtimes of the BCSR format.

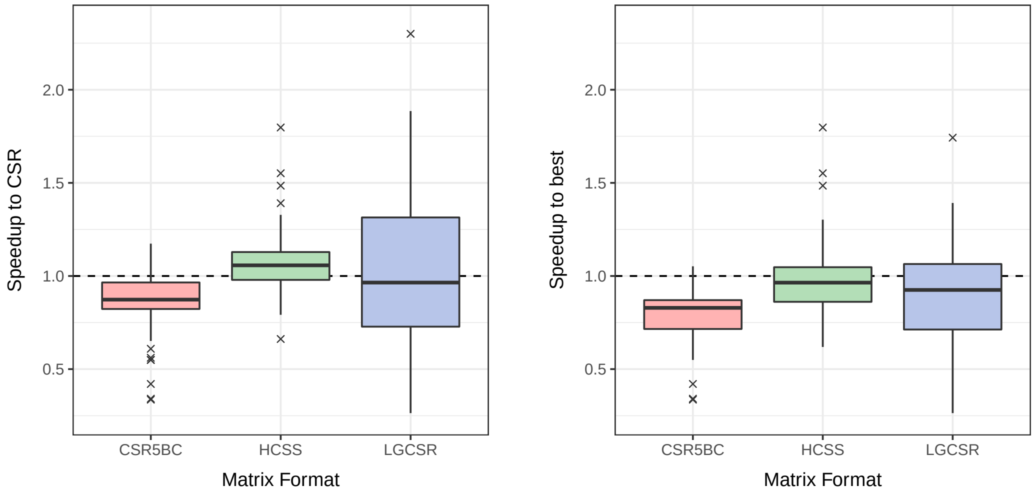
Determining the proper amount of parallelism for GPU based systems is much more complicated. This problem has already been analyzed in an earlier publication [40]. The work presents heuristics for the proper thread selection, which are used for the measurements in this work. The ELL-BRO format has one relevant additional parameter, which has also been analyzed in [40]. The parameter defines the size of the used slices and has been defined to be 128.

In the following sections, the performance of the newly developed formats is evaluated. The evaluation process is thereby based on the, in this section defined, parameters.

## 7.3 Performance Evaluation on the CPU

In this section, the performance of all newly developed formats is evaluated on the CPU-system. In a first step, the performance of the different formats is compared to the CSR format (see Figure 7.2a). It can be seen that HCSS is the only format, that reaches a speedup in the median case. The CSR5BC format has the worst performance and only for a small number of matrices a speedup can be reached at all. The performance of the LGCSR format varies much more, compared to the other two formats. For both the CSR5BC and LGCSR format the median speedup is below 1, which means it is slower than CSR. In contrast to CSR5BC, the LGCSR format can reach a speedup for a significant number of matrices.

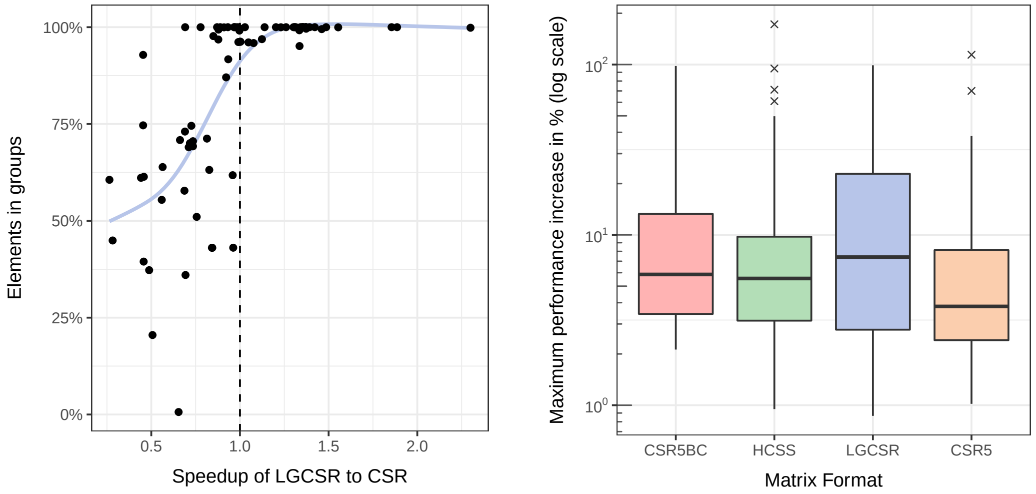
The high variability of the LGCSR format can be explained by the layout of the format. It utilizes block-like structures of the matrix to reduce the memory consumption. A more detailed analysis showed a clear correlation between the blocked structures and the reached speedups compared to CSR. A new simple measure has been developed for evaluation the amount of blocking of matrices, specific for LGCSR. The measure counts the number of elements that are part of vertical line block (or local group), as they are used by LGCSR. Afterwards the number of elements, that are part of a group is compared to the overall number



(a) Speedup to CSR

(b) Speedup to the best format

Figure 7.2: Speedup of the developed formats compared to CSR and the best format for every matrix from the relevant matrix formats on the CPU.



(a) Structural dependency of LGCSR

(b) Load balance

Figure 7.3: Structural dependency of the LGCSR format (left) and the load balance of the CSR5BC, HCSS, LGCSR and CSR5 format (right) and the CPU.

of non-zero elements. The result is the percentage of non-zero elements which are part of a local group.

Figure 7.3a presents the new measure in relation to the speedup of LGCSR compared to CSR. The additional line presents the smoothened mean of the present data. It can be seen, that high speedups are only achieved when the majority of elements are part of a group. On the other hand, a high amount of elements in groups does not guarantee a speedup compared to CSR. Furthermore, there is no clear relation between the amount of elements in groups and the reached speedup. Nevertheless, the analysis clearly shows the structural dependency of the LGCSR format.

The HCSS format shows a very consistent performance. The performance of the format is less dependent on the matrix structure than LGCSR. But it can also be seen that the maximum speedup and the 75th percentile is much lower compared to LGCSR. Here, the LGCSR format can reach higher speedups by utilizing the matrix structure. On the other hand, the maximum performance loss of the HCSS significantly lower than for LGCSR.

The CSR5BC format is unsuitable for the use on CPU-based systems. Only for a small number of matrices a speedup is reached and the median performance loss is about 13%. Further analysis have been made, to explain this poor performance, but no definite reason could be identified. The hardware performance counters have been analyzed to identify issues related to excessive branching, but no issues could be found. One possible reason for the poor performance of the CSR5BC format could be the complexity of the SpMV calculation. It is the most complex implementation and uses intrinsics, which allows less optimization by the compiler.

Another explanation for the poor performance would be an insufficient load balance. The load balancing of the formats has been analyzed by measuring the execution time of every individual thread. Afterwards, the maximum performance improvement with optimal load balancing was calculated. Figure 7.3b presents the result for the three newly developed formats and CSR5. It can be seen that the median load balance of all formats is considerably good, with maximal median improvements of 4–7%. But it can also be seen that the load balance of the CSR5BC format is very similar to the load balance of the HCSS format. The load balance of the LGCSR format varies much more as the other formats, which can be explained by the structural dependency of the format.

Compared to the CSR5 format, CSR5BC has higher load imbalances. Since the CSR5BC format only adds an additional index compression to the CSR5 format, this optimization has to be the reason for the increased load imbalances. One explanation for this is the variable compression technique, which reaches different compression factors for different parts of the matrix. Therefore, the calculation of some parts of the matrix require a higher memory bandwidth than other parts, which can lead to load imbalances. Overall it could not be shown, that the optimization goal of the CSR5BC is fulfilled on the CPU. The load balance could not be improved in comparison to the other developed formats.

Figure 7.2b shows the speedup of the developed formats compared to the fastest

| **Format** | **CPU**<br>rank (diff) |
|---|---|
| HCSS | 30 |
| LGCSR | 30 |
| CSR | 24 (-13) |
| DynB | 15 (-14) |
| BCSR | 12 (-2) |
| Others | 5 (-7) |

Table 7.4: Number of occurrences of the formats in the list of fastest formats including the newly developed formats, with the change to the base ranking on the CPU.

format for every matrix. The fastest formats are thereby selected form the list of relevant matrices, introduced in the previous section. It can be seen that the overall performance of all formats is reduced, which is reasonable. The maximum speedup of all formats has been reduced, especially for the LGCSR format.

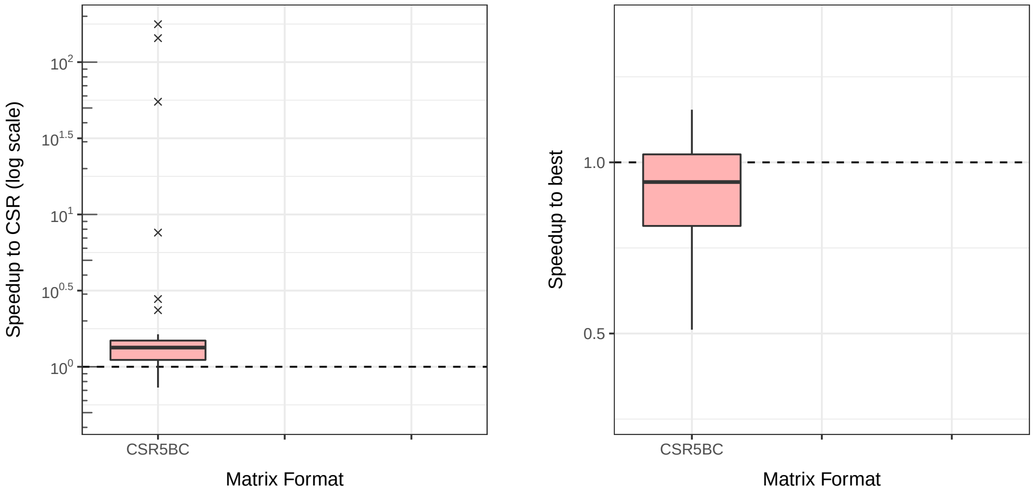
Two of the relevant formats also utilize blocked structures, which explains the reduced maximum speedup of the LGCSR format. The HCSS and LGCSR formats achieve the fastest runtime for over 25% of the matrices. In the median the HCSS format looses only about 4% and LGCSR about 7% performance compared to the best formats. Considering that the formats are compared to the best runtimes of a large set of formats, that are highly optimized these are very good results.

Table 7.4 presents the ranking of the fastest formats on the CPU taking the newly developed formats into consideration. It can be seen that the HCSS and LGCSR format are on the top of the ranking, which shows that both perform very well, at least for a subset of the matrices. As expected, because of the low speedup numbers, the CSR5BC format is irrelevant on the CPU. The number of occurrences of the CSR and DynB format is reduced significantly. A more detailed analysis showed, that the CSR format is primarily displaced by the HCSS format and DynB by LGCSR. This matches the findings, that the LGCSR format performs well, especially for blocked matrices and that the HCSS has a very consistent performance.

Overall the HCSS and LGCSR format showed high performance on the CPU-system. The HCSS format thereby shows the more consistent performance over all matrices, while LGCSR reaches very high speedups for blocked matrices. But LGCSR has also high performance drops for non-blocked matrices. The CSR5BC format is not suitable to be used on the CPU.

## 7.4 Performance Evaluation on the GPU

In this section, the performance of the CSR5BC format is evaluated on the GPU-system. Among the newly developed formats, only CSR5BC has been implemented for this platform. Figure 7.4a shows the speedup of CSR5BC to the CSR format.



(a) Speedup to the CSR

(b) Speedup to the best format

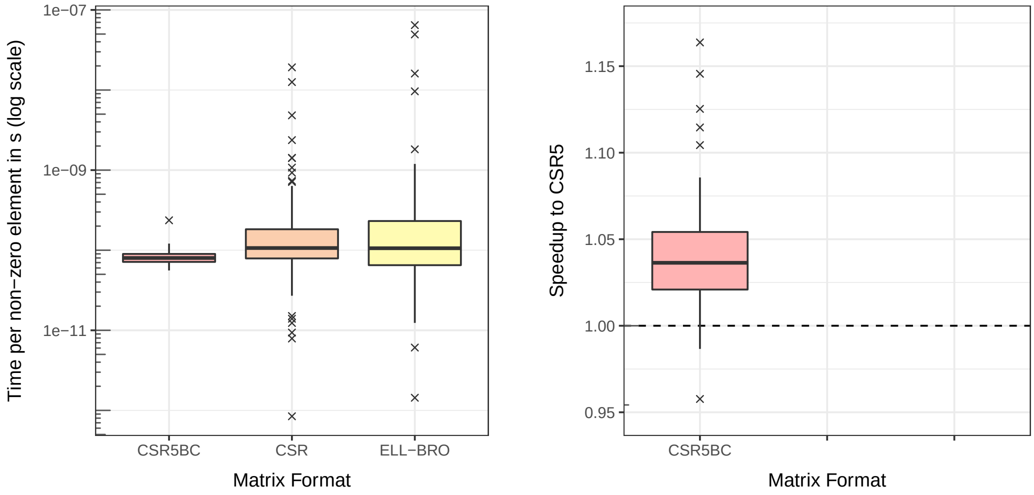
Figure 7.4: Speedup of the CSR5BC format to CSR and the best format for every matrix from the relevant matrix formats on the Tesla K80.

It can be seen, that CSR5BC outperforms CSR for most matrices with a median speedup of about 1.34. Even higher speedups are achieved for a small number of matrices.

Figure 7.4b presents the speedup of CSR5BC to the best formats on the GPU. It can be seen, that for more than 25% of the matrices CSR5BC achieves the best performance. The median performance loss compared to the fastest formats is only about 6%. A more detailed analysis showed, that the highest speeddowns are achieved for matrices where the ELL-BRO format delivered the best runtimes.

While the ELL-based formats deliver a very high performance for some matrices, the CSR5BC format achieves a much more consistent performance for all matrices. One reason for this could be the structural independent load balancing of CSR5BC. Because of the execution model of the GPU it is not possible to directly measure the load balance, or at least not with acceptable effort. Therefore, a new measure has been developed, to at least show the dependency of a format to the structure of the matrices. The measure is the SpMV execution time per non-zero element. It is calculated by dividing the overall execution time by the number of non-zero elements of the matrix. In theory, the calculation time per non-zero element should be very similar for multiple matrices, when the SpMV performance is independent of the matrix structure. Thus by analyzing the variation of this measure for multiple matrices, the structural dependency of a format can be evaluated.

Figure 7.5a shows this new measure for the CSR5BC format as well as for the common CSR and the most relevant format on the GPU, the ELL-BRO format. It can be seen, that CSR5BC has the by far lowest variation for the per non-zero runtime. The variation for the both row-based approaches CSR and ELL-



(a) Calculation time per non-zero element

(b) Speedup of CSR5BC to CSR5

Figure 7.5: Calculation time per non-zero element of CSR5BC, CSR and ELL-BRO (left) and the speedup of CSR5BC over CSR5 (right) on the Tesla K80.

BRO is much higher, which means they have a much stronger dependency on the matrix structure. Even though this measure can not certainly proof the proper and structural independent load balance of the CSR5BC format, it is the most likely explanation for the much lower variation. Therefore, it is assumed that the optimization goal of the CSR5BC format has been fulfilled on the GPU.

The CSR5BC format has been developed based on the existing CSR5 format. A further analysis has been done, to show the performance differences between the two formats. Figure 7.5b presents the speedup of the newly developed CSR5BC format compared to CSR5. It can be seen, that CSR5BC outperforms CSR5 for nearly all matrices. A slowdown is only achieved for 2 matrices with a maximum performance loss of about 4%.

The median speedup is 4%, which is significant when considering that the CSR5 implementation is already highly optimized. The analysis clearly shows, that CSR5BC is a reasonable extension of the CSR5 format. It can significantly increase the performance without any significant drawbacks.

Table 7.5 presents the ranking of the fastest formats including CSR5BC. It can be seen, that CSR5BC is the fastest format for most matrices. Furthermore the number of occurrences of the base formats is not reduced significantly, which shows that the CSR5BC format can not significantly improve the performance for most matrices. This matches with the previous findings in Figure 7.4b. Nevertheless, the CSR5BC format delivers a very consistent performance for a wide range of matrices, which makes it much more general applicable than the other matrix formats.

Overall the CSR5BC format performs very well on the GPU. The results indicate, that the optimization goal of the CSR5BC format, a structural independent load

| **Format** | **GPU**<br>rank (diff) |
|---|---|
| CSR5BC | 37 |
| ELL-BRO | 33 (-1) |
| ELL | 25 (-3) |
| CSR5 | 22 (-4) |
| HYB | 16 (-1) |
| Other | 9 (-5) |

Table 7.5: Number of occurrences of the formats in the list of fastest formats including CSR5BC, with the change to the base ranking on the GPU.

balancing, has been achieved. It shows a very consistent performance and allows a very efficient calculation for most matrices. Because of the improved load balance and no required padding, it can outperform ELL-based formats for many matrices. Furthermore it outperforms the CSR5 format on which it is based.

## 7.5 Performance Evaluation on the Xeon Phi

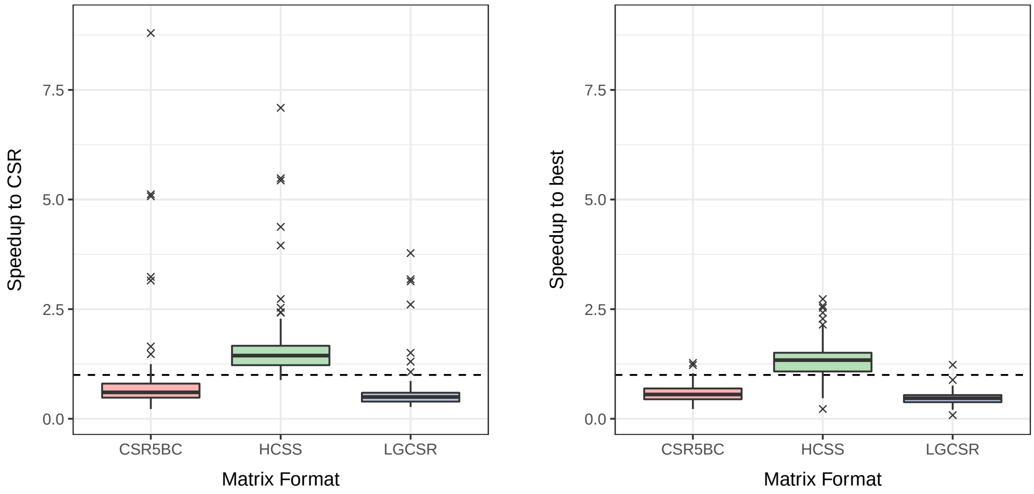
In this section, the performance of the developed formats is evaluated on the Xeon Phi. All three formats have been implemented for this platform. Figure 7.6a shows the speedup of the formats compared to CSR. It can be seen that the HCSS format performs much better than CSR and reaches a median speedup of about 1.4. Additionally, only for a very small number of matrices the speedup is below 1.

The very good performance can be explained by the data layout of the format. All elements are properly aligned and padded to allow an efficient vectorization. Furthermore, the SpMV implementation is very simple, which is advantageous for the simplistic cores of the Xeon Phi.

In contrast, CSR5BC and LGCSR are much slower than CSR. They achieve a speedup only for a very small number of matrices. One reason for the poor performance could be the complexity of the formats and their index decompression. Both formats require very complex SpMV implementations. This is especially problematic, as the cores of the Xeon Phi only support in-order execution.

Additionally, the data layout of the LGCSR is not optimized for the utilization of vector units. Hence, only parts of the calculation can be done using vector units. Since the Xeon Phi heavily relies on the utilization of vector units, this explains the even worse performance of LGCSR compared to CSR5BC.

Figure 7.6b presents the speedup of the newly developed formats to the best format for every matrix. It can be seen that the speedups of all formats are lower compared to the CSR speedups. Furthermore, the extreme outliers are no longer present. The number of matrices with a speedup above 1 for CSR5BC and LGCSR is even smaller than before.



(a) Speedup to the CSR

(b) Speedup to the best format

Figure 7.6: Speedup of developed formats to CSR and the best format for every matrix from the relevant matrix formats on the Xeon Phi.

| **Format** | **Phi**<br>rank (diff) |
|---|---|
| HCSS | 68 |
| CSR5 | 8 (-15) |
| CSR | 6 (-51) |

Table 7.6: Number of occurrences of the formats in the list of fastest formats including all newly developed formats, with the change to the base ranking on the Xeon Phi.

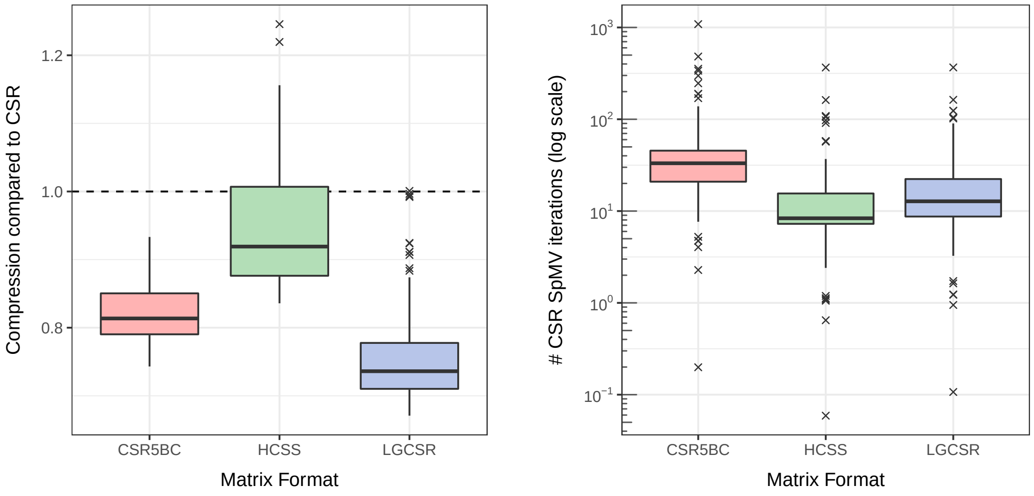
The performance of the HCSS format is very good and it reaches a median speedup of about 1.3. Even the 25th percentile is above 1, which means that the format outperforms the other matrix formats on at least 75% matrices of the matrix set. Only for a small number of matrices the HCSS format reaches very low speedups. This shows that the HCSS format is by far the best format on the Xeon Phi for most matrices.

Table 7.6 shows the ranking of the fastest formats on the Xeon Phi platform, taking the new developed formats into consideration. As expected, the CSR5BC and LGCSR format are not relevant. However, the HCSS format nearly replaces the existing formats. It delivers the best performance for the majority of matrices.

Overall, HCSS is the only suitable format for the Xeon Phi, among the newly developed formats. One reason for this is that the optimization goal of the HCSS format matches very well with the requirements of the Intel MIC architecture. The format allows a very high vector unit utilization and aligned memory accesses.

| **Format** | **Memory Consumption (Bit per non-zero)** | |
|---|---|---|
| | Measured | Estimated |
| CSR5BC | 78.5 | 77.3 |
| HCSS | 90.0 | 90.6 |
| LGCSR | 71.1 | 68.8 |

Table 7.7: Median memory consumption and the estimated consumption of the developed formats for the used set of matrices.



(a) Compression factor

(b) Conversion effort

Figure 7.7: Amount of compression of the developed formats compared to CSR and the conversion effort.

Both of which are very important on the Phi.

## 7.6 Evaluation of the Memory Consumption and Conversion Effort

In this section, the memory consumption and the conversion effort of the developed formats is evaluated. Table 7.7 presents the measured median memory consumption and the memory estimation for the used set of matrices (see Section 5.5). It can be seen that the memory estimations are quite accurate. Thereby, it should be mentioned that the estimations are partially based on empirical measurements based of the same set of matrices.

Figure 7.7a presents the compression of the matrices in the different formats compared to CSR. It can be seen that the LGCSR format has the lowest memory consumption for most matrices with a median reduction of about 26%. But there are also some outliers for which no memory reduction could be reached compared to CSR. These outliers can be explained by the matrix structure. The matrices in question do not have a blocked structure, which is utilized by the LGCSR format.

The HCSS format has the lowest compression compared to CSR with a median memory reduction of only about 8%. It can be seen that for over 25% of the matrices the memory consumption is similar or even increased compared to CSR. Just as for the LGCSR format this can be explained by structural properties of the matrices. For matrices with a small number of rows with a large number of non-zero elements the required padding can increase the memory consumption. Compared to classic ELL based formats, the effect is significantly reduced by the data layout of HCSS.

The CSR5BC format shows a very consistent memory reduction without any extreme outliers. The achieved median reduction is about 19%, placing it between the other two formats. The very consistent memory consumption can be explained by the layout of the format. It does not require any padding and the compression depends only to a small degree on the matrix structure.

Figure 7.7b shows the conversion effort of the three developed formats in number of CSR SpMV iterations that could be done in the same time. It can be seen that the HCSS format has the most efficient conversion and it requires only about 8 iterations. The conversion effort of LGCSR is marginally higher with a median of 13 iterations. CSR5BC has the most complex conversion and it needs in median about 33 SpMV iterations.

The number of CSR SpMV iterations, $i_{convert}$ can be used together with the assumed speedup to calculate the required number of SpMV iterations to amortize the format conversion:

$$i_{spmv} \times t_{CSR} > i_{spmv} \times t_{other} + t_{CSR} \times i_{convert}$$

Where $i_{spmv}$ is the number of required SpMV iterations to amortize the conversion cost. The variables $t_{CSR}$ and $t_{other}$ denote the time required for one SpMV iteration using the different formats. The *speedup* expresses the relation between the two runtimes as $speedup = t_{CSR}/t_{other}$. The inequation can be simplified and transformed, to allow a simple calculation of the required SpMV iterations:

$$\begin{aligned}
& i_{spmv} \times t_{other} \times speedup > i_{spmv} \times t_{other} + t_{other} \times speedup \times i_{convert} \\
\Leftrightarrow\quad & i_{spmv} \times t_{other} \times (speedup - 1) > t_{other} \times speedup \times i_{convert} \\
\Leftrightarrow\quad & i_{spmv} > \frac{speedup \times i_{convert}}{speedup - 1}
\end{aligned}$$

The HCSS format reached a median speedup of about 1.06 on the CPU. Using the presented inequation the number of iterations for amortizing the conversion effort is about 141. It should thereby be mentioned that these are only rough estimations, as the reached speedup and the conversion effort differs for every matrix.

| Block size | Kernel |
|---|---|
| 1x1 | inlined simd |
| 2x1 | transposed inlined simd |
| 2x2 | transposed inlined simd |
| 3x1 | transposed inlined simd |
| 3x3 | inlined simd |
| 4x4 | transpoed inlined intrinsic |
| 7x7 | inlined normal |
| 8x4 | transposed inlined intrinsic |
| 9x3 | transposed inlined partial simd |

Table 7.8: List of the kernel implementations selected by the DynB autotuning.

Whether the conversion effort is relevant or not strongly depends on the application. The same is true for the number of acceptable iterations until the conversion is amortized. Therefore no general evaluation of the conversion effort can be done in this work.

## 7.7 Evaluation of the DynB Autotuning

In this section, the performance of the DynB autotuning approach is evaluated. In a first step, the performance of the different kernels is analyzed. Afterwards, the best kernels for the different block sizes are identified and used for the final performance evaluation.

All measurements for the autotuning are done on the CPU. Overall, 9 out of the 280 possible block sizes have been analyzed: 1x1, 2x1, 2x2, 3x1, 3x3, 4x4, 7x7, 8x4 and 9x3. The block sizes have been selected based on the number of occurrences in the used set of matrices. A relative small set of block sizes has been selected. For each block, the more specialized kernels have to be developed and the overall required benchmark time increases rapidly. Furthermore, the used set of block sizes should be large enough to evaluate the general feasibility of the approach.

The selection of the optimal kernels for the different block sizes is a manual process. For each block size and synthetic benchmark matrix, all kernel variants are ordered by their runtime. By comparing the resulting rankings of the different benchmark matrices the fastest kernel is selected. The selection is thereby based on the ranking of the kernels, but also by their speeddown compared to the best possible kernel.

Table 7.8 shows the finally selected kernels. For most of the kernels the selection was very clear, as the selected kernel ranked first for all, or nearly all benchmark matrices. The selection for the block sizes 1x1, 7x7 and 8x4 was more complicated. No single kernel performed best on the majority of matrices. The kernels have been selected by comparing the maximum performance loss of multiple kernels.

For the block size 7x7 no ideal kernel could be found at all and therefore the default implementation is used.

The analysis of the kernel runtimes showed, that a structural dependency exists at least for some block sizes. The performance of the kernels differs strongly between the different benchmark matrices for the 7x7 kernel. For the other block sizes only minor performance differences between the benchmark matrices could be found.

Overall, it can be seen that all selected kernels are inlined into the SpMV operation instead of using external function calls. This also means that all kernels are compiled using the Intel compiler and using the default optimization level, as no different compiler or optimization level can be used for the inlined code. Especially for the smaller block sizes the *simd* and transposed *simd* implementations performed best. For the larger kernels also other kernels like the intrinsic implementation is more efficient. The *partial simd* implementation for the 9x3 blocks calculates 8x3 values using a simd loop, while the remaining 3 elements are calculated separately. The benchmarks also show that the use of transposed, column-major order blocks is often advantageous over row-major order.

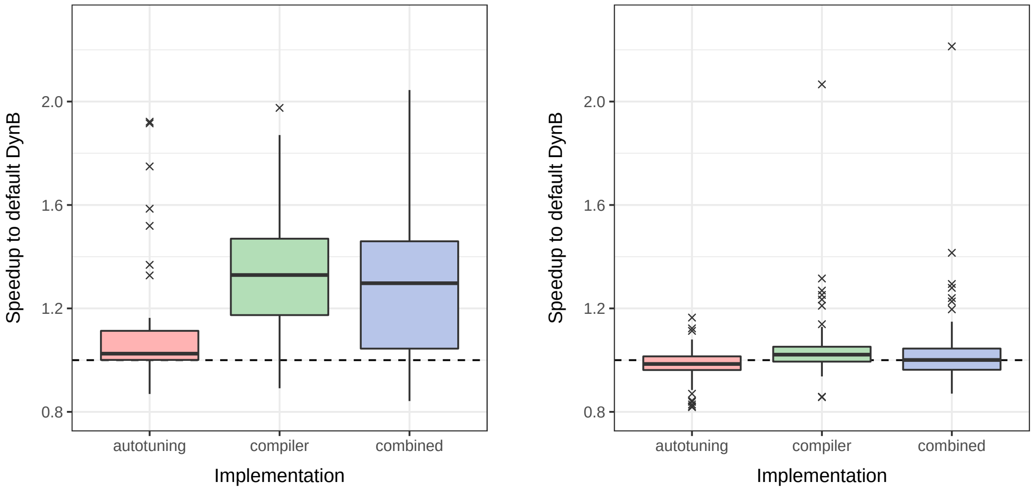
For the evaluation of the autotuning approach three different SpMV implementations have been developed and measured. The first version uses only specialized kernels for the 9 block sizes, that have been optimized by the autotuning. The blocks of all other block sizes are calculated using the default block implementation. In the following evaluation this version is called *autotuning*.

The second implementation is used to demonstrate the optimization potential of recent compilers. In this version a separate function for every block size is generated. Every function thereby contains the general block implementation. Because of the used nested switch case statements the compiler knows the dimensions of every block and can optimize the general implementation according to it. This implementation allows the compiler to optimize every possible block size individually. The individual block kernels can be compared to the *loop* kernel described in Section 5.4. In the following this version is called *compiler*.

The third implementation is a combination of both previously presented versions. Individual functions for all possible block sizes are generated using the default implementation. For the 9 optimized block sizes the selected kernels are used instead of the default kernel. This version is called *combined*.

### Performance Evaluation of the Autotuning

Figure 7.8a presents the sequential performance of the different DynB implementations compared to the unoptimized implementation. It can be seen, that the performance is improved by using the *autotuning* version. The median improvement is only about 3%, but very high speedups could be reached for some matrices. This result can be explained by the small number of optimized kernels. Speedups can only be seen for matrices that mostly consist of blocks of the corresponding size. Because of this, the runtime of many matrices stays the same. It can be seen



(a) Sequential

(b) OpenMP

Figure 7.8: Performance of the three optimized versions of the DynB format compared to the unoptimized version.

that for about 25% of the matrices the performance is decreased compared to the unoptimized implementation

The performance of the *compiler* optimized implementation is much better than the default implementation, with a median speedup of about 1.33. It also outperforms the *autotuning* implementation. Since all block sizes are optimized in this implementation, speedups for most matrices are reasonable. In contrast to the *autotuning* implementation the performance is only decreased for a small number of matrices.

The *combined* implementation is slower than the *compiler* optimized version. This means that the by the autotuning selected kernels are slower than the ones optimized by the compiler. This shows the high efficiency of the optimizations done by recent compilers.

The analysis of the autotuning results showed that the *loop* implementation, which is assumed to be very equal to the compiler optimized version, also performed very good in the benchmarks. But for all block sizes other kernels delivered better performance on the synthetic matrices. This shows that there may be additional important aspects that are currently not respected in the autotuning approach. Furthermore, this shows that the developed simple autotuning approach can not be used to outperform the compiler optimizations.

Figure 7.8b presents the parallel performance of the three different implementations compared to the unoptimized implementation. It can be seen that only small speedups can be reached for most matrices using the *compiler* optimized implementation. The *autotuning* version has a median speedup of 0.99 and has some positive and negative outliers. The median performance of the *combined* version

| Format | CPU | GPU | MIC |
|---|---|---|---|
| CSR5BC | | ✓ | |
| HCSS | ✓ | ○[1] | ✓ |
| LGCSR | ✓ | ○[1] | |

[1] Not implemented and evaluated.

Table 7.9: Suitability of the newly developed formats on the three different hardware platforms.

does not change compared to the unoptimized version.

Overall only a small speedup of the *compiler* optimized version can be seen. This seams unreasonable when comparing the speedups to the sequential speedups with a median of 1.33. One possible explanation for this is the available memory bandwidth. In the parallel execution all threads share the overall available memory bandwidth. The single thread in the sequential execution cannot utilize the complete memory bandwidth [8]. But the single thread can use a much lager fraction of the bandwidth, compared to a single thread in an application with parallel memory accesses. Because of this, the sequential implementation is much more limited by the computation as the parallel implementation. The assumption is that the optimized computation in the sequential implementation can increase the overall performance, while the parallel implementation is memory bound. Therefore, the optimization of the computational part of the execution has a much smaller impact on the performance of the parallel implementation.

Overall, the evaluation shows that the developed autotuning can only reach small performance improvements. It is outperformed by the optimizations done by recent compilers. To allow efficient optimizations, the compiler needs as many additional information as possible. The developed nested switch case implementation of the SpMV operation, allows the compiler to optimize every individual block kernel. Using this the performance of the sequential implementation was improved by over 30%. Only small improvements for the parallel implementation could be reached.

## 7.8 Discussion and Summary

In the previous sections, the performance of the newly developed formats and the DynB autotuning has been evaluated. Table 7.9 summarizes the suitability of the developed formats for the three relevant hardware platforms. It can be seen that the CSR5BC format showed good performance only on the GPU. There it delivered a very consistent performance and outperformed the ELL-based formats for many matrices.

The CSR5BC format has been developed with the focus on a structural independent load balancing. The evaluation on the GPU indicates, that this optimization goal has been reached. Some of the very high speedups of the format can be explained by a better load balancing of the CSR5BC format. On the CPU and on the Xeon Phi these findings could not be confirmed. The measured load balancing on the CPU showed that the CSR5BC format had no improved load balance compared to the other developed formats. Therefore, the optimization goal of the CSR5BC format was achieved only partially. The developed index compression efficiently reduced the memory consumption of the format. An important disadvantage of CSR5BC is the high conversion time, compared to the other developed formats.

The HCSS format delivered very good performance on the CPU and outperformed all formats on the Xeon Phi. It delivers a very consistent performance, mostly independent of the matrix structure. Because of the required padding and the row-based organization it is not completely independent of the matrix structure. In the implementation process of the format, the vectorization report [20] of the Intel compiler has been used (see Section 6.2). It showed that the implementation can be highly vectorized. Additionally the achieved high performance numbers on the Xeon Phi also indicate a high vector unit utilization. Therefore, the optimization goal of the HCSS format has been achieved. An important advantage of the HCSS format is its simplicity over the other two formats. This also results in the fastest conversion of all developed formats.

The LGCSR format delivered high performance only on the CPU. It is primarily suited for matrices with blocked structures. Because of the utilization of block structures the achieved performance also depends on the matrix structure. Significant performance drops occur for matrices without any blocked structures. But for matrices with block structures the lowest memory consumption of all developed formats was achieved. It achieves significant memory savings compared to CSR which shows that the optimization goal has been reached. The LGCSR format requires a quite sophisticated SpMV implementation and the conversion is slightly slower compared to the HCSS format.

The developed autotuning approach for the DynB format did achieve a performance increase for the sequential SpMV implementation. The evaluation showed that the compiler is capable of creating highly optimized implementations that outperform the autotuned version. For achieving this, the compiler has to be provided with enough additional information for the optimization. Even though, the developed autotuning approach has been outperformed, the sequential performance of the DynB format could be improved significantly using the compiler optimizations. In the parallel implementation a signficant performance increase could only be achieved for a small number of matrices.

Overall, it could be shown that the used theoretical development approach is suitable for developing efficient sparse matrix formats. The developed formats did not deliver high performance on all relevant hardware platforms. This indicates, that additional requirements and constraints may exist that have not been considered in this work.

# 8 Summary

The target of this work was the development of new and efficient sparse matrix formats, based on a comprehensive requirements analysis. It should be researched whether it is possible to develop new and efficient sparse matrix formats using such a theoretical approach. Furthermore, the formats should be developed for three different hardware platforms: Intel CPUs, Nvidia GPUs and the Intel MIC architecture.

The first chapter of this work introduced the terminology and and basic concepts that are important for the understanding of this work. This included the definition of sparse matrices and basic sparse matrix formats. Furthermore, the SpMV operation was introduced and the relevant hardware platforms have been described. Additionally, related work has been presented.

In Chapter 3, a comprehensive requirements analysis has been done. Based on the analysis of the SpMV operation, a large number of requirements and constraints for the development of efficient sparse matrix formats have been identified. The analysis showed that the SpMV operation is a memory bounded problem. Therefore, many requirements are related to the memory subsystem, but also compute and application related requirements were identified.

Subsequently, in Chapter 4 an extensive analysis of the optimization techniques that are used in existing formats has been done. Each of the identified optimization techniques has been presented in detail with its advantages and disadvantages. Additionally, the relation between the identified requirements and the optimization techniques was discussed.

In Chapter 5, the development process of the newly developed formats and the autotuning approach is described. The development is thereby based on the findings of the requirements analysis and the analysis of the existing optimization techniques. The focus of the format development was on different optimization goals.

The following Chapter 6 described the implementation of the developed formats and the autotuning. The focus thereby was on the description of the conversion operations and performance critical aspects of the SpMV implementation.

In Chapter 7, the performance of the developed formats and the autotuning has been evaluated. A comprehensive evaluation of the SpMV performance for the different formats and hardware platforms has been done. Moreover, the memory consumption and the conversion effort of the formats was analyzed. Furthermore, the performance of the developed autotuning approach was evaluated.

In this work, three new sparse matrix formats and an autoutuning approach have been developed. It could be shown that the used theoretical development approach can be used for the development of efficient sparse matrix formats. The initial assumption that a single format can not achieve optimal performance all different hardware platforms was partially approved.

Based on the identified requirements and constraints three new sparse matrix formats have been developed: CSR5BC, HCSS and LGCSR. For all three relevant platforms at least one of the developed formats achieved a high SpMV performance. The CSR5BC format is based on the existing CSR5 format and its optimization goal was the development of a format with a structural independent load balancing. In the evaluation it could be shown, that the format delivers a decent performance on the GPU, were it outperformed the classic CSR format, but also ELL-based formats for many matrices. The main drawbacks of CSR5BC are the complexity of the SpMV implementation and the high conversion cost. Only on the GPU it could be shown, that the optimization goal of the format is reached.

The HCSS format has been optimized for the utilization of the available vector units. It performed very well on the CPU, where it achieves very consistent performance mostly independent of the structural properties of the matrix. On the Xeon Phi it outperformed all existing formats and reached significant speedups. Additionally HCSS has the simplest structure and the shortest conversion time of the developed formats. It could be shown that the format can be highly vectorized, which means its optimization goal has been achieved.

The focus of the development for the LGCSR format was on the reduction of the memory demand. It reached very high performance on the CPU, especially for matrices with blocked structures. According to its development goal it reached the highest memory reduction compared to the other developed formats. The main drawbacks of the format are the complexity of the data decompression and its structural dependency.

The developed autotuning approach for the exiting DynB format only delivered small speedups. By providing the compiler with many additional information a much faster implementation could be developed. This implementation reached significant speedups for the sequential implementation of the DynB format. With no implementation significant speedups for the parallel DynB implementation could achieved.

In the evaluation, new metrics have been developed, to allow a proper evaluation of specific performance aspects of the developed formats. This includes a measure for the amount of blocked structures in sparse matrices. Furthermore, an indirect measure for the load balance on the GPU has been developed.

Considering the high performance of the developed formats, further improvements on their implementations may be reasonable. Furthermore, it may be of interest to analyze the performance of the HCSS format on the GPU. Even though it has some disadvantages on this platform, it may still deliver considerable performance. Additionally, this work could not find the definite reason for the low performance of the CSR5BC format on the CPU and the Xeon Phi. An extensive analysis may reveal additional requirements or constraints for the development of effective sparse matrix formats.

# A Appendix

```
$lastRowPtr = rowPtr[0]$
for $i = 0$ to $nRows$ do
    $nextRowPtr = rowPtr[i + 1]$
    $tmp = 0$
    for $j = lastRowPtr$ to $nextRowPtr$ do
        $tmp \mathrel{+}= values[j] \times x[columnIndexes[j]]$
    end
    $y[i] = tmp$
    $lastRowPtr = nextRowPtr$
end
```

**Algorithm 2:** SpMV algorithm for the CSR format with reduced memory bandwidth.

| **Matrix** | **Dimension** | **NNZ** | **NNZ per row** | | | |
|---|---|---|---|---|---|---|
| | | | **min** | **max** | **avg** | **dispersion** |
| af_0_k101 | 503625 | 17550675 | 15 | 35 | 34.85 | 1.26 |
| af_shell1 | 504855 | 17588875 | 20 | 40 | 34.84 | 1.28 |
| atmosmodd | 1270432 | 8814880 | 4 | 7 | 6.94 | 0.24 |
| atmosmodl | 1489752 | 10319760 | 4 | 7 | 6.93 | 0.26 |
| audikw_1 | 943695 | 77651847 | 21 | 345 | 82.28 | 42.45 |
| bmw3_2 | 227362 | 11288630 | 2 | 336 | 49.65 | 13.81 |
| bmw7st_1 | 141347 | 7339667 | 1 | 435 | 51.93 | 12.72 |
| bmwcra_1 | 148770 | 10644002 | 24 | 351 | 71.55 | 18.51 |
| bone010 | 986703 | 71666325 | 12 | 81 | 72.63 | 15.81 |
| boneS10 | 914898 | 55468422 | 12 | 81 | 60.63 | 20.37 |
| Chebyshev4 | 68121 | 5377761 | 9 | 68121 | 78.94 | 1061.44 |
| circuit5M | 5558326 | 59524291 | 1 | 1290501 | 10.71 | 1356.62 |
| circuit5M_dc | 3523317 | 19194193 | 1 | 27 | 5.45 | 2.09 |
| CO | 221119 | 7666057 | 15 | 313 | 34.67 | 7.04 |
| consph | 83334 | 6010480 | 1 | 81 | 72.13 | 19.08 |
| crankseg_2 | 63838 | 14148858 | 48 | 3423 | 221.64 | 95.88 |
| Cube_Coup_dt0 | 2164760 | 127206144 | 24 | 68 | 58.76 | 4.47 |
| dielFilterV2real | 1157456 | 48538952 | 6 | 110 | 41.94 | 16.15 |
| dielFilterV3real | 1102824 | 89306020 | 9 | 270 | 80.98 | 36.55 |
| Emilia_923 | 923136 | 41005206 | 15 | 57 | 44.42 | 3.72 |
| F1 | 343791 | 26837113 | 24 | 435 | 78.06 | 40.81 |
| Fault_639 | 638802 | 28614564 | 15 | 318 | 44.79 | 5.16 |
| Flan_1565 | 1564794 | 117406044 | 24 | 81 | 75.03 | 11.43 |
| FullChip | 2987012 | 26621990 | 1 | 2312481 | 8.91 | 1806.8 |
| G3_circuit | 1585478 | 7660826 | 2 | 6 | 4.83 | 0.64 |
| Ga10As10H30 | 113081 | 6115633 | 7 | 698 | 54.08 | 82.38 |

Continued on next page

| **Matrix** | **Dimension** | **NNZ** | **NNZ per row** | | | |
|---|---|---|---|---|---|---|
| | | | **min** | **max** | **avg** | **dispersion** |
| Ga19As19H42 | 133123 | 8884839 | 15 | 697 | 66.74 | 103.18 |
| Ga41As41H72 | 268096 | 18488476 | 18 | 702 | 68.96 | 105.39 |
| Ge87H76 | 112985 | 7892195 | 7 | 469 | 69.85 | 90.15 |
| Ge99H100 | 112985 | 8451395 | 7 | 469 | 74.8 | 94.54 |
| Geo_1438 | 1437960 | 63156690 | 15 | 57 | 43.92 | 4.4 |
| gsm_106857 | 589446 | 21758924 | 12 | 106 | 36.91 | 15.63 |
| hood | 220542 | 10768436 | 21 | 77 | 48.83 | 12.82 |
| Hook_1498 | 1498023 | 60917445 | 15 | 93 | 40.67 | 13.95 |
| HV15R | 2017169 | 283073458 | 1 | 484 | 140.33 | 53.95 |
| inline_1 | 503712 | 36816342 | 18 | 843 | 73.09 | 35.63 |
| kkt_power | 2063494 | 14612663 | 1 | 96 | 7.08 | 7.4 |
| largebasis | 440020 | 5560100 | 4 | 14 | 12.64 | 1.15 |
| ldoor | 952203 | 46522475 | 28 | 77 | 48.86 | 11.95 |
| memchip | 2707524 | 14810202 | 2 | 27 | 5.47 | 2.06 |
| ML_Geer | 1504002 | 110879972 | 26 | 74 | 73.72 | 2.51 |
| msdoor | 415863 | 20240935 | 28 | 77 | 48.67 | 11.71 |
| m_t1 | 97578 | 9753570 | 48 | 237 | 99.96 | 28.56 |
| nd24k | 72000 | 28715634 | 110 | 520 | 398.83 | 76.94 |
| nd6k | 18000 | 6897316 | 130 | 514 | 383.18 | 89.17 |
| nlpkkt80 | 1062400 | 28704672 | 5 | 28 | 27.02 | 3.74 |
| nlpkkt120 | 3542400 | 96845792 | 5 | 28 | 27.34 | 3.09 |
| nlpkkt160 | 8345600 | 229518112 | 5 | 28 | 27.5 | 2.7 |
| nlpkkt200 | 16240000 | 448225632 | 5 | 28 | 27.6 | 2.42 |
| nlpkkt240 | 27993600 | 774472352 | 5 | 28 | 27.67 | 2.22 |
| ohne2 | 181343 | 11063545 | 15 | 3441 | 61.01 | 21.09 |
| PR02R | 161070 | 8185136 | 1 | 92 | 50.82 | 19.7 |
| pre2 | 659033 | 5959282 | 1 | 628 | 9.04 | 22.12 |
| pwtk | 217918 | 11634424 | 2 | 180 | 53.39 | 4.74 |
| rajat30 | 643994 | 6175377 | 1 | 454746 | 9.59 | 784.58 |
| rajat31 | 4690002 | 20316253 | 1 | 1252 | 4.33 | 1.11 |
| RM07R | 381689 | 37464962 | 1 | 295 | 98.16 | 68.68 |
| Serena | 1391349 | 64531701 | 15 | 249 | 46.38 | 9.25 |
| Si34H36 | 97569 | 5156379 | 17 | 494 | 52.85 | 68.93 |
| Si41Ge41H72 | 185639 | 15011265 | 13 | 662 | 80.86 | 126.97 |
| Si87H76 | 240369 | 10661631 | 17 | 361 | 44.36 | 39.69 |
| SiO2 | 155331 | 11283503 | 15 | 2749 | 72.64 | 294.17 |
| thermal2 | 1228045 | 8580313 | 1 | 11 | 6.99 | 0.81 |
| tmt_sym | 726713 | 5080961 | 3 | 9 | 6.99 | 1.01 |
| torso1 | 116158 | 8516500 | 9 | 3263 | 73.32 | 419.59 |
| TSOPF_FS_b300_c3 | 84414 | 13135930 | 1 | 41542 | 155.61 | 1181.18 |
| TSOPF_RS_b2383 | 38120 | 16171169 | 2 | 983 | 424.22 | 484.24 |
| x104 | 108384 | 10167624 | 30 | 324 | 93.81 | 29.2 |
| matrix_spe1Ref_a | 900000 | 18612000 | 12 | 21 | 20.68 | 0.96 |
| matrix_spe1RefDPDP_a | 1800000 | 42624000 | 15 | 24 | 23.68 | 0.96 |
| matrix_spe5Ref_a | 1029000 | 49529200 | 28 | 49 | 48.13 | 2.4 |
| matrix_spe5Ref_dpdp_a | 2058000 | 113464400 | 35 | 56 | 55.13 | 2.4 |
| matrix_spe5Ref_dpdp_b | 2058000 | 113464400 | 35 | 56 | 55.13 | 2.4 |
| matrix_spe5Ref_dpdp_c | 2058000 | 113464400 | 35 | 56 | 55.13 | 2.4 |
| matrix_spe5Ref_dpdp_d | 2058000 | 113464400 | 35 | 56 | 55.13 | 2.4 |
| matrix_spe5Ref_dpdp_e | 2058000 | 113464400 | 35 | 56 | 55.13 | 2.4 |
| matrix_spe10_a | 2153544 | 29192160 | 2 | 14 | 13.56 | 1.11 |
| matrix_spe10_dpdp_a | 3506080 | 50928264 | 2 | 16 | 14.53 | 1.96 |

Table A.1: List of matrices used in the evaluation with additional structural properties.

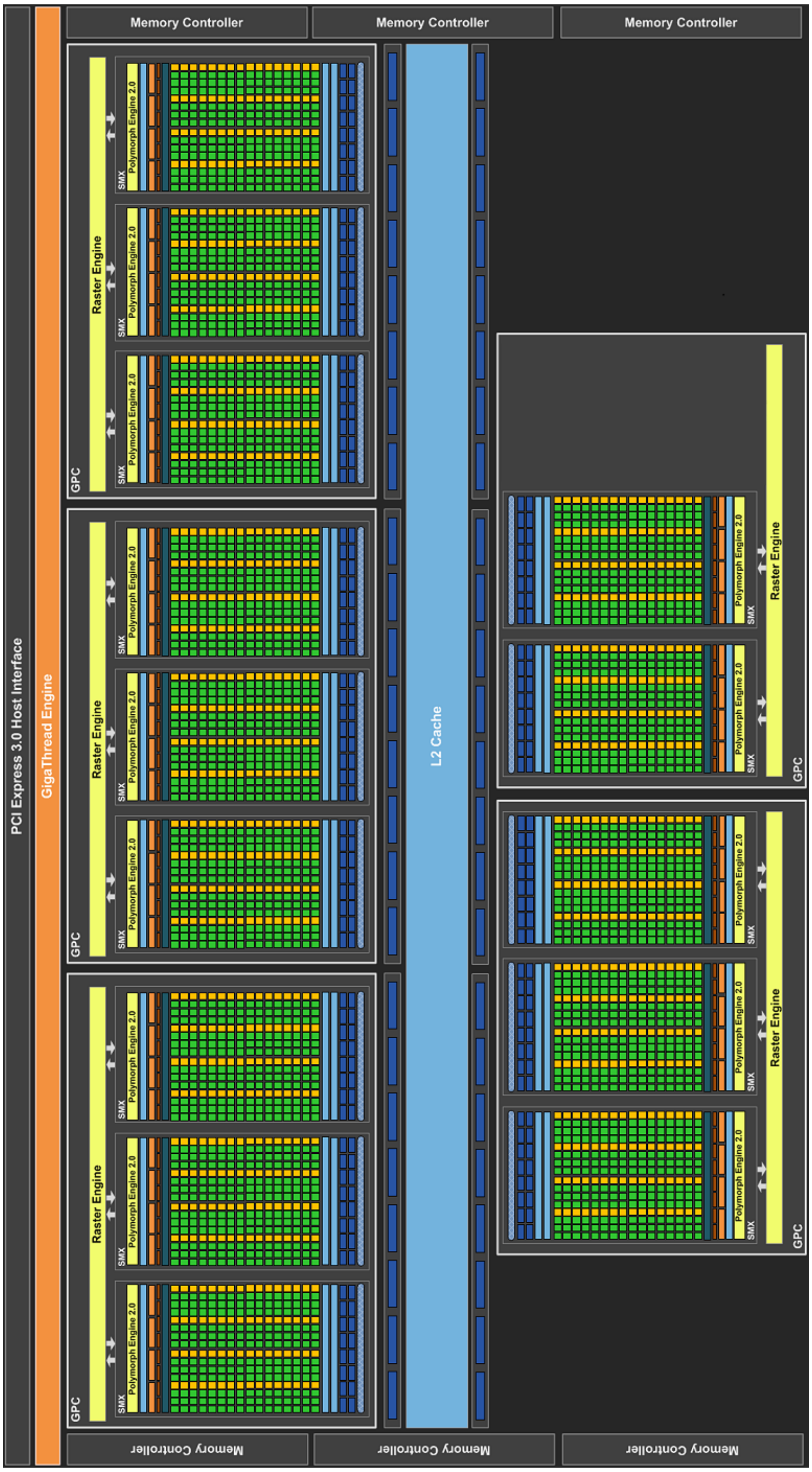


Figure A.1: Block diagram of the Nvidia Kepler architecture [1].

# List of Figures

# List of Tables

# Listings

# Acronyms

**AVX2** Advanced Vector Extensions 2

**AVX-512** Advanced Vector Extensions 512

**BCCOO** Blocked Compressed COO

**BCSR** Blocked Compressed Sparse Row

**BCSR-DEC** BCSR Decomposed

**BDIA** Banded Diagonal

**BELLpack** Blocked ELLpack

**BLSI** Bit Level Single Row

**BRO** Bit Representation Optimizations

**CMRS** Compressed Multirow Storage

**COO** Coordinate

**COSC** Combine Optimized SpMV for CSR

**CPU** Central Processing Unit

**CSB** Compressed Sparse Block

**CSC** Compressed Sparse Column

**CSR** Compressed Sparse Row

**CSR5BC** CSR5 Bit Compressed

**CSR-DU** CSR Delta Unit

**CSR-VI** CSR Value Indexed

**CSX** Compressed Sparse eXtended

**CUDA** Compute Unified Device Architecture

**DIA** Diagonal

**DynB** Dynamic Block

**ELL** ELLpack

**ELL-BRO** ELLpack Bit Representation Optimization

**ELL-R** ELLpack-R

**ESB** ELLpack Sparse Block

**FLOP** Floating Point Operation

**FLOPS** Floating Point Operations Per Second

**FMA** Fused Multiply-Add

**FMA3** Fused Multiply-Add 3

**GPU** Graphics Processing Unit

**HBM2** High Bandwidth Memory 2

**HCSS** Hybrid Compressed Slice Storage

**HiSM** Hierarchical Sparse Matrix Storage

**HPC** High-performance computing

**HYB** Hybrid

**IQR** Interquartile Range

**JDS** Jagged Diagonal Storage

**LGCSR** Local Group Compressed Sparse Row

**MCDRAM** Multi-Channel DRAM

**MIC** Intel Many Integrated Core architecture

**NUMA** Non-Uniform Memory Access

**PBR** Pattern-based Representation

**PCSR** Perfect CSR

**QPI** Intel QuickPath Interconnect

**SELL** Sliced ELLpack

**SELLR-T** Sliced ELLR-T

**SFU** Special Function Unit

**SIC** Segmented Interleave Combination

**SIMD** Single Introduction Multiple Data

**SMX** Next Generation Streaming Multiprocessor

**SpMV** Sparse Matrix-Vector Multiplication

**UBCSR** Unaligned BCSR

**VBL** Variable Block Length

**VBR** Variable Block Row

**VHCC** Vectorized Hybrid COO+CSR

**VPU** Vector Processing Unit

# Bibliography

[1] 3DCenter, http://www.3dcenter.org/artikel/launch-analyse-nvidia-geforce-gtx-titan. *Launch-Analyse: nVidia GeForce GTX Titan*, 2013. (in german), retrieved: May 2016.

[2] AMD, https://www.amd.com/Documents/Opteron_6300_QRG.pdf. *AMD Opteron$^{TM}$6300 Series processor Quick Reference Guide*, 2012. retrieved: March 2016.

[3] AMD, http://www.amd.com/Documents/AMD-FirePro-S9170-Datasheet.pdf. *AMD FirePro$^{TM}$S9170 Server GPU*, 2015. whitepaper, retrieved: March 2016.

[4] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. Implementing a sparse matrix vector product for the SELL-C/SELL-C-$\sigma$ formats on NVIDIA GPUs. Technical Report ut-eecs-14-727, University of Tennessee, EECS Department, 2014.

[5] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, second edition, 1994.

[6] Mehmet Belgin, Godmar Back, and Calvin J. Ribbens. Pattern-based sparse matrix representation for memory-efficient smvm kernels. In *Proc. 23rd International Conference on Supercomputing (SC'09)*, ICS '09, pages 100–109, New York, NY, USA, 2009. ACM.

[7] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, Nvidia Corp., December 2008.

[8] R. Berrendorf, J.P. Ecker, J. Razzaq, S.E. Scholl, and F. Mannuss. Using application oriented micro-benchmarks to characterize the performance of single-node architectures. In Claus-Peter Rueckemann, editor, *Proc. Ninth International Conference on Advanced Engeneering Computing and Applications in Sciences (ADVCOMP 2015)*, pages 31–38. IARIA, 2015.

[9] Rudolf Berrendorf, Jan Ecker, Javed Razzaq, and Simon Scholl. Project next generation reservoir simulation, report on deliverable 5.1. Technical report,

Bonn-Rhein-Sieg University, Computer Science Department, November 2014. confidental.

[10] BLAS Technical Forum. *Basic Linear Algebra Subprograms Technical Forum Standard*, August 2001.

[11] Aydin Buluc, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles 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)*, pages 233–244, 2009.

[12] Jong-Ho Byun, Richard Lin, Katherine A. Yelick, and James Demmel. Autotuning sparse matrix-vector multiplication for multicore. Technical Report UCB/EECS-2012-215, EECS Department, University of California at Berkeley, November 2012.

[13] Sebastian Böckem. *A Survey on Sparse Matrix Formats.* Bachelor's thesis, Bonn-Rhein-Sieg University of Applied Sciences, 2016.

[14] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proc. Principles and Practices of Parallel Programming (PPoPP'10)*, pages 115–125. ACM, January 2010.

[15] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, November 2010.

[16] A. Dziekonski, A. Lamecki, and M. Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress In Electromagnetics Research*, 116:49–63, 2011.

[17] Athena Elafrou, Georgios I. Goumas, and Nectarios Koziris. A lightweight optimization selection method for sparse matrix-vector multiplication. *arXiv.org*, abs/1511.0249, Dec 2015.

[18] X. Feng, H. Jin, K. Hu, J. Zeng, and Z. Schao. Optimization of sparse matrix-vector multiplication with variant CSR on GPUs. In *Proc. 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 165–172. IEEE, 2011.

[19] Free Software Foundation. *GCC, the GNU Compiler Collection.* retrieved: November 2016.

[20] Ronald W. Green. *Vectorization and Optimization Reports.* Intel, https://software.intel.com/en-us/articles/vectorization-and-optimization-reports. retrieved: October 2016.

[21] Sönke Hack. *Survey, Realization and Evaluation of Blocked Storage Formats for Sparse Matrices with SpMV-Operations.* Bachelor's thesis, Bonn-Rhein-Sieg University of Applied Sciences, 2015.

[22] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, Boca Raton, FL, 2011.

[23] Guixia He and Jiaquan Gao. A novel CSR-based sparse matrix-vector multiplication on GPUs. *Mathematical Problems in Engineering*, 2016, 2016. Article ID 8471283.

[24] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 5. edition, 2012.

[25] IBM, http://www.redbooks.ibm.com/abstracts/sg248171.html?Open. *Performance Optimization and Tuning Techniques for IBM Processors, including IBM POWER8.* retrieved: November 2014.

[26] Intel, https://software.intel.com/en-us/blogs/2013/avx-512-instructions. *AVX-512 instructions.* retrieved: May 2016.

[27] Intel, https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors. *Disclosure of H/W prefetcher control on some Intel processors.* retrieved: June 2016.

[28] Intel, http://ark.intel.com/products/codename/42174/Haswell. *Intel® Haswell.* retrieved: November 2014.

[29] Intel, http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. *Intel® Many Integrated Core Architecture - Advanced.* retrieved: May 2016.

[30] Intel, http://www.intel.de/content/www/de/de/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html. *Intel® Omni-Path Architecture: The next generation Fabric.* retrieved: November 2016.

[31] Intel, https://software.intel.com/en-us/isa-extensions/intel-avx. *ISA Extensions.* retrieved: May 2016.

[32] Intel, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, January 2016. retrieved: March 2016.

[33] Intel Corporation. *Intel C++ Compiler 17.0 Developer Guide and Reference*, https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide edition, 2016. retrieved: November 2016.

[34] Ankit Jain. pOSKI: An Extensible Autotuning Framework to Perform Optimized SpMVs on Multicore Architectures. Technical report, Computer Science Division, University of California, Berkeley, 2008.

[35] Jim Jeffers and James Reinders. *Intel® Xeon Phi$^{TM}$Coprocessor High-Performance Programming.* Morgan Kaufmann, 2013.

[36] Jim Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming, Knights Landing Edition.* Morgan Kaufman Publishers, Inc., Cambridge, 2016.

[37] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Performance models for blocked sparse matrix-vector multiplication kernels. In *Proc. 38th Intl. Conference on Parallel Processing (ICPP'09)*, 2009.

[38] Vasileios K. Karakasis. *Optimizing the Sparse Matrix-Vector Multiplication Kernel for Modern Multicore Computer Architectures.* PhD thesis, National Technical University of Athens, Greece, 2012.

[39] David Kirk and Wen mei W. Hwu. *Programming massively parallel processors.* Morgan Kaufmann, 2nd edition, 2013. ISBN 978-0-12-415992-1.

[40] Hendrik Klohn. Entwicklung von heuristiken zur parameterbelegung bei der optimierung der spmv-operation auf grafikprozessoren. Bachelor's thesis, Bonn-Rhein-Sieg University of Applied Sciences, April 2016.

[41] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Proc. 5th Conference on Computing Frontiers (CF'08)*, pages 87–96. ACM, 2008.

[42] Z. Koza, M. Matyka, S. Szkoda, and L. Miroslaw. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM J. Sci. Comput*, 36(2):219–239, 2014.

[43] Zbigniew Koza, Maciej Matyka, Łukasz Mirosław, and Jakub Poła. Sparse matrix-vector product. In Volodymyr Kindratenko, editor, *Numerical Computations with GPUs*, pages 103–121. Springer International Publishing, Cham, 2014.

[44] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R. Bishop. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Proc. 26th Intl. Parallel and Distributed Processing Symposium (IPDP2012)*, pages 1696–1702. IEEE, 2012.

[45] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Ailan R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiply on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing*, 26(5):C401–423, 2014.

[46] M. Krotkiewski and M. Dabrowski. Parallel symmetric sparse matrix–vector product on scalar multi-core CPUs. *Parallel Computing*, 36(4):181–198, 2010.

[47] Yuji Kubota and Daisuke Takahashi. Optimization of sparse matrix-vector multiplication by auto selecting storage schemes on GPU. In *Proc. Computational Science and Its Applications - ICCSA 2011*, volume 6783, pages 547–561. Springer-Verlag, 2011.

[48] Daniel Langr. *Algorithms and Data Structures for Very Large Sparse Matrices*. PhD thesis, Czech Technical University in Prague Faculty of Information Technology Department of Computer Systems, 2014.

[49] Daniel Langr and Pavel Tvrdík. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):428–440, 2016.

[50] Christoph Lehnert. Reliable performance ranking mechanisms for SpMV kernels on GPU architectures. Master's thesis, Bonn-Rhein-Sieg University of Applied Sciences, Sankt Augustin, Germany, December 2015.

[51] Weifeng Liu and Brian Vinter. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proc. 29th Intl. Conference on Supercomputing (ICS'15)*, pages 339–350. ACM, 2015.

[52] Weifeng Liu and Brian Vinter. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing*, 49:179–193, 2015.

[53] Xing Liu, Edmond Chow, Mikhail Smelyanskiy, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-code processors. In *Proc. Intl. Conference on Supercomputing (ICS'13)*, pages 273–282. ACM, 2013.

[54] Kiran Kumar Matam and Kishore Kothapalli. Accelerating sparse matrix vector multiplication in iterative methods using GPU. In *Proc. Intl. Conference on Parallel Processing*, pages 612–621. IEEE, 2011.

[55] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical Report TM-88, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. retrieved: November 2014 http://www.cs.virginia.edu/stream/.

[56] Media and Entertainment Technologies, http://mandetech.com/2012/05/20/nvidia-new-gpu-and-visualization/. *Nvidia new GPU and visualization*. retrieved: May 2016.

[57] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In

*Proc. HIPEAC 2010*, number 5952 in LNCS, pages 111–125. Springer-Verlag, 2010.

[58] Hassan Mujtaba. *Intel 14nm Skylake-EP and 10nm Cannonlake-EP Supported on Purley Platform – Up To 160W TDP, Arriving in 1H 2017*. wccftech, http://wccftech.com/intel-14nm-skylake-ep-10nm-cannonlake-ep-supported-purley-platform-160w-tdp-48-pcie-lanes-6-channel-ddr4/. retrieved: October 2016.

[59] Nvidia Corp., http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf. *NVIDIA's Next Generation CUDA® Compute Architecture: Kepler® GK110*, 2013. whitepaper, retrieved: November 2014.

[60] Nvidia Corp., http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF. *NVIDIA GeForce GTX 980*, 2014. whitepaper, retrieved: November 2014.

[61] Nvidia Corp., http://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf. *NVIDIA® NVLink TM High-Speed Interconnect: Application Performance*, 2014. whitepaper, retrieved: May 2016.

[62] Nvidia Corp. *CUDA C best practices guide*. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, dg-05603-001_v7.5 edition, March 2015. retrieved: June 2016.

[63] Nvidia Corp., http://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf. *TESLA K80 GPU ACCELERATOR*, 2015. whitepaper, retrieved: December 2015.

[64] Nvidia Corp. *CUDA C Programming Guide*. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, pg-02829-001_v8.0 edition, March 2016. retrieved: November 2016.

[65] Nvidia Corp. *CUDA COMPILER DRIVER NVCC*. http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf, v8.0 edition, August 2016. retrieved: November 2016.

[66] Nvidia Corp., http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. *NVIDIA Tesla P100 Whitepaper*, 2016. whitepaper, retrieved: October 2016.

[67] Tomas Oberhuber and Martin Heller. Improved row-grouped CSR format for storing of sparse matrices on GPU. In *Proc. ALGORITMY*, pages 282–290, 2012.

[68] OpenMP Architecture Review Board, http://www.openmp.org/. *OpenMP Application Program Interface*, 4.5 edition, November 2015. retrieved: July 2016.

[69] Juan C. Pichel, Francisco F. Rivera, Marcos Fernandez, and Aurelio Rodriguez. Optimization of sparse matrix-vector multiplication using reordering techniques on GPU's. *Microprocessors and Microsystems*, 36:65–77, 2012.

[70] Juan Carlos Pichel, Dora Blanco Heras, José Carlos Cabaleiro, and Francisco F. Rivera. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *Parallel, Distributed and Network-Based Processing, 2004. Proceedings. 12th Euromicro Conference on*, pages 66–71. IEEE, 2004.

[71] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proc. ACM/IEEE Conference on Supercomputing (SC'99)*. IEEE, November 1999.

[72] Javed Razzaq, Rudolf Berrendorf, Soenke Hack, Max Weierstall, and Florian 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)*, pages 84–90, 2016.

[73] Neelima Reddy, Raghavendra Prakash, and Ram Mohana Reddy. New sparse matrix storage format to improve the performance of total SPMV time. *Scalable Computing: Practice and Experience*, 13(2):159–172, 2012.

[74] István Reguly and Mike Giles. Efficient sparse matrix-vector multiplication on cache-based GPUs. In *Proc. Innovative Parallel Computing (InPar)*, pages 1–12. IEEE, 2012.

[75] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989.

[76] Youcef Saad. Sparskit: A basic tool kit for sparse matrix computations. 1994. Version 2.

[77] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, second edition, 2003.

[78] Jacob T. Schwartz. Ultracomputers. *ACM Trans. Program. Lang. Syst.*, 2(4):484–521, October 1980.

[79] R. Shahnaz, A. Usman, and I.R. Chughtai. Review of Storage Techniques for Sparse Matrices. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1–7. IEEE, 2005.

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

[81] *SPARC International, Inc.* http://sparc.org/. retrieved: March 2016.

[82] P. Stathis, S. Vassiliadis, and S. Cotofana. A hierarchical sparse matrix storage format for vector processors. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., 2003.

[83] Bor-Yiing Su and Kurt Keutzer. clSpMV: A cross-platform opencl spmv framework on GPUs. In *Proc. 26th ACM International Conference on Supercomputing*, ICS '12, pages 353–364. ACM, 2012.

[84] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh. Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on intel xeon phi. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 136–145, 2015.

[85] W.T. Tang, W.J. Tan, R. Ray, Y.W. Wong, W. Chen, S. Kuo, R.S.M. Goh, S.J. Turner, and W. Wong. Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes. In *Proc. Intl. Conference on High Performance Computing, Networking, Storage and Analysis (SC'13).* ACM, 2013. article no. 26.

[86] *Top 500 List.* http://www.top500.org/. retrieved: November 2016.

[87] F. Vazquez, E. M. Garzon, J. A. Martinez, and J. J. Fernandez. The sparse matrix vector product on GPUs. In *Proc. 2009 Intl. Conference on Computational and Mathematical Methods in Science and Engineering*, volume 2, pages 1081–1092, 2009.

[88] Richard W. Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *Proc. First Intl. Conference on High Performance Computing and Communications (HPCC'05)*, pages 807–816, 2005.

[89] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels.* PhD thesis, University of California, Berkeley, 2003.

[90] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. ACM/IEEE Supercomputing 2007 (SC'07)*, pages 1–12. IEEE, 2007.

[91] J. Wong, E. Kuhl, and E. Darve. A new sparse matrix vector multiplication GPU algorithm designed for finite element problems. *Intl. Journal for Numerical Methods in Engineering*, 102(12):1784–1814, June 2015.

[92] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. yaSpMV: yet another SpMV framework on GPUs. In *Proc. 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*, pages 107–118, 2014.

[93] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining. In *Proc. VLDB Endowment (PVLDB)*, volume 4, pages 231–242, January 2011.

[94] Ji-Lin Zhang, Li Zhuang, Jian Wan, Xiang-Hua Xu, Cong-Feng Jiang, and Yong-Jian Ren. COSC: Combine optimized sparse matrix-vector multiplication for CSR format. In *Proc. Sixth Annual ChinaGrid Conference*, 124–129, 2011.

# Statutory Declaration

I declare that I have authored this work independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Sankt Augustin, November 15th, 2016