Derler Andreas, BSc

HiSparse: A Hierarchical Sparse Matrix Format for the GPU

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr. techn. Dieter Schmalstieg

Institute for Computer Graphics and Vision

Graz, December 2016

AFFIDAVIT

Abstract

Sparse matrices are part of the core of many important scientific computing algorithms. Thus, a variety of sparse matrix formats and implementations for different hardware architectures have been proposed over the years. While simple compressed formats still dominate the computing environment, alternative block-based formats have seen success on the GPU as they scale well for extremely large matrices. However, hierarchical formats have not been explored and are widely considered unviable as they introduce highly dynamic execution paths, which is detrimental for performance on massively parallel devices such as the GPU. Introducing HiSparse, a hierarchical sparse matrix format for the GPU, we show that by applying dynamic scheduling strategies, the issues of varying execution paths can be counteracted. We show that a hierarchical format can adjust itself locally to the present data and thus significantly reduce the memory footprint of the stored matrices in comparison to standard formats. Implementing sparse matrix vector multiplication and sparse matrix addition we show that our hierarchical format is competitive to highly optimized standard libraries for these operations and significantly outperforms them in the case of transpose matrix operations, pointing towards the viability of hierarchical matrix formats on massively parallel devices as the GPU.

Kurzfassung

Dünnbesetzte Matrizen sind ein zentraler Bestandteil vieler wichtiger wissenschaftlicher Algorithmen. Folglich wurden im Laufe der Jahre eine Vielfalt von Formaten und Implementierungen dünnbesetzter Matrizen für verschiedene Hardware-Architekturen vorgeschlagen. Während einfache komprimierte Formate nach wie vor die Rechenwelt dominieren, haben alternative blockbasierte Formate für GPU-basierte Algorithmen immer mehr an Relevanz gewonnen, da diese für sehr große Matrizen äußerst gut skalieren. Hierarchische Formate wurden jedoch noch nicht erforscht, da sie, aufgrund der vielen dynamischen Ausführungspfade, welche die Leistung auf der GPU negativ beeinflussen, großteils als untauglich gelten. Mit der Vorstellung von HiSparse, einem hierarchischen Format für dünnbesetzte Matrizen auf der GPU, zeigen wir, dass mithilfe dynamischer Planungsstrategien das Problem der vielen Ausführungspfade beseitigt werden kann. Wir zeigen, dass sich ein hierarchisches Format lokal an den vorhandenen Daten anpassen kann, wodurch der Speicherverbrauch der Matrizen im Vergleich zu den Standard-Formaten signifikant reduziert werden kann. Mit der Implementierung von dünnbesetzter Matrix-Vektor Multiplikation und Addition von dünnbesetzten Matrizen demonstrieren wir, dass unser hierarchisches Format für diese Operationen im Vergleich zu hoch optimierten Standard-Bibliotheken wettbewerbsfähig ist und im Falle der transponierten Matrizen sogar eine bessere Leistung erzielt wird, wodurch die Realisierbarkeit von hierarchischen Formaten auf massiv parallelen Ausführungsumgebungen, wie die GPU, aufgezeigt wird.

Contents

List of Figures

List of Tables

1 Introduction

Sparse matrices are a key primitive in many fields of scientific computing which require modelling of irregular data, such as physical simulations, graph processing, or economics. Striving to tackle larger problems and delivering results ever faster, a variety of different sparse matrix formats and implementations of core matrix operations for virtually all available hardware architectures have been proposed. While standard formats such as the coordinate list (COO) and compressed sparse rows (CSR) are still predominant across hardware architectures and standard libraries, alternative formats may have the edge over these formats in special cases. For example, the compressed sparse blocks format (CSB) scales well on parallel CPU architectures and its performance stays unchanged when operating on transposed matrices [1]. Many alternative formats on parallel architectures like the graphics processing unit (GPU) are either tuned for single matrix operations, like CSR5 [2] for sparse matrix vector multiplication (SpMV), or only applicable for certain types of matrices, like ELLPACK for matrices with similar number of non-zeros in each row. One reason for this situation stems for the difficulty of deriving efficient implementations for more dynamic data structure on the GPU. For example, dynamic search in a Z-order curve as used in the CSB SpMV implementation [1] will lead to non coherent memory access and execution divergence among threads, greatly reducing the performance of such an implementation on the single instruction, multiple data (SIMD) compute cores found on the GPU.

In this work, we tackle this issue providing a hierarchical, block-based, sparse matrix format suitable for the GPU. A hierarchical format has the advantage that it scales well for ever larger matrices, allows to implement algorithms in a divide an conquer manner, and allows sharing/duplication of sub-matrices between different matrices and within a single matrix. We show that by applying dynamic GPU scheduling strategies to algorithms built on top of this format, competitive performance can be achieved. We make the following contributions:

- We propose the HiSparse sparse matrix format, which supports a combination of various node types and thus can adapt to the local structure of the matrix. Each node is transparent to transpose operations, thus, algorithms built on top of the format achieve similar performance for operations on a transpose matrix. Being hierarchical, the bit length of indices can be reduced, consequently, the overall memory requirements are below COO and CSR.

- We describe a dynamic GPU scheduling framework that allows to implement algorithms on top of HiSparse with little effort. Algorithms can implement specialized routines for the different node types and dynamically adjust the number of threads used for each step of the implementation to ensure high performance.
- We provide an example implementation of SpMV using the previously defined dynamic scheduler, which in parallel traverses the hierarchical structure of the sparse matrix, combining the local SpMV results into a global output vector.
- We show an example implementation of sparse matrix add, which concurrently traverses the hierarchical structure of two input matrices, determines the number of colliding entries in both hierarchies, dynamically allocates and generates the hierarchy of the resulting matrix alongside the non-zero entries.
- We analyze the performance of the SpMV and matrix add implementation for a variety of matrices and compare the performance to state-of-the-art GPU implementations.

The remainder of this paper is structured as follows. At first, we provide a brief introduction to the most common sparse matrix formats and review related work (Section 2). Then, we introduce the HiSparse format and provide an analysis of the format's memory requirements (Section 3). After presenting our scheduling framework (Section 4), we show how SpMV (Section 5) and sparse add (Section 6) can be implemented and analyze their performance (Section 7). We conclude by summarizing our findings and provide an outline of more complex algorithms that could efficiently be implemented on top of HiSparse (Section 8).

2 Background and Related Work

The most common sparse matrix formats are *Coordinate list* (COO) and *Compressed Sparse Row* (CSR). COO is the most trivial format, since it simply consists of three arrays, separately storing the column index, row index and value of each non zero of the sparse matrix. CSR maintains identical arrays for column indices *col_id* and values *val* sorted in row major format. However, in contrast to COO, row indices are compressed such that the entry $row\_ptr[i]$ points to the index of the first entry of the row i within *val* and *col_id*. Whereas the last entry $row\_ptr[m+1] = nnz$, with nnz being the number of non zeros of the matrix. CSR enables an easy way to process individual rows in parallel. Figure 2.1 depicts a simple example of the CSR matrix format. A sequential algorithm for SpMV using the CSR format is shown in algorithm 1.

2.1 GPU SpMV

At first glance, processing CSR matrices on the GPU seems trivial, by simply processing each row in parallel. Using a naive approach, each row is processed by an individual thread. Bell and Garland called this approach the *scalar* CSR kernel [3]. However, this approach entails severe performance limitations. One major problem is the way the memory is accessed. In general, coalesced memory access within threads running on the same SIMD core is necessary in order to efficiently load memory from the global memory of the GPU. However, since the CSR matrix is stored in row-major order, each thread accesses the entries of the corresponding row in a non-coalesced manner. In addition, since the nnz of each row can vary extensively, threads will execute different number of elements, leading to so-called divergence and slower execution on SIMD devices. Multiple approaches were introduced to counterbalance these issues. One can use multiple threads per row [4], apply grouping and reordering techniques [5, 6], or dynamically choose the number of threads for each row [7, 8].

Another major performance limitation are load balancing issues. LightSpMV [9] introduces a simple way to handle this issue by using a global row counter, dynamically scheduling rows to available threads. Recently, an efficient SpMV algorithm using merge path was introduced [10], which handles all previously described performance limitation, although still requiring a slight computational overhead. Liu et al also introduced an SpMV algorithm specifically for heterogeneous processors [11].

2.2 Alternative Formats

Although COO and CSR are the most common formats, multiple additional formats have been introduced. ELLPACK is a common format, which stores a sparse matrix of size $m \times n$ in two $m \times k$ matrices *indices* and *data*, whereas k corresponds to the nnz of the row with the largest nnz. The matrix *indices* stores the column index of the corresponding value entry in *data*. Row indices of entries are stored intrinsically, since each row is padded to the fixed size k. Consequently, ELLPACK performs well with a regular data structure, however has severe issues with irregular row nnz, since much of the data needs to be zero padded. There are several ELLPACK format adoptions, like Hybrid [3], Sliced ELLPACK [12] and ELLPACK-R [13]. However, a common shortcoming of those formats is dealing with highly irregular matrix structures.

A common way of dealing with thread divergence and load balancing issues is splitting the input matrix into two dimensional blocks. For instance, the CSR5 format [2] arranges the non zeros into tiles (2D blocks) of a fixed size. Each tile is processed individually. Several block formats organize the blocks in a two layered hierarchy. The top layer is used to manage the blocks, whereas the bottom layer is the block representation. Blocked CSR (BCSR) [14] divides a sparse submatrix into dense blocks. The blocks are stored in a CSR format. Obviously, this is rather inefficient for very sparse matrices [15]. Compressed Sparse Blocks (CSB) [1] is a COO representation of sparse submatrix blocks. Entries within a block are also stored in COO format (compressed). Consequently, rows are not favored over columns, which is crucial for transpose multiplication [16]. Since the blocks are always stored in COO format, CSB is rather inefficient for dense matrix areas. Some mixed type formats try to deal with this limitation. Adaptive-blocking hierarchical storage format (ABHSF) [17] is a COO representation of mixed type blocks. To this end, blocks can be stored in a dense, CSR, COO or bitmap format. The Cocktail format [18] operates similarly. There are numerous similar formats, like BRC (Blocked Row-Column) [19], BCOO (Block-based Compressed Common Coordinate) [20], ESB (ELLPACK Sparse Blocks) [21] or JAD (JAgged Diagonal) [22]. While many block-based formats were designed to fit massively parallel architecture like the GPU, there exist no multi-level hierarchical formats on such architectures.

$$\mathbf{A} = \begin{bmatrix} 0 & 7 & 0 & 3 \\ 1 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 5 & 5 & 0 \end{bmatrix} \qquad \begin{aligned} val &= \{7,3,1,4,2,5,5\} \\ col\_id &= \{1,3,0,2,0,1,2\} \\ row\_ptr &= \{0,2,4,4,7\} \end{aligned}$$

Figure 2.1: Example of a simple matrix stored in CSR format. Matrix values and column indices are stored explicitly, in addition to non-zero offsets of each row

```
**Sequential CSR SpMV**
Input: CSR Matrix $A$, dense vector $x$
Output: dense vector $y \leftarrow A \cdot x$
for $i \leftarrow 0$ to $A.num\_rows$ do
    $y_i \leftarrow 0$
    for $k \leftarrow A.row\_ptr[i]$ to $A.row\_ptr[i+1]$ do
        $y_i \leftarrow y_i + A.val[k] \cdot x[A.col\_id[k]]$
    end
    $y[i] \leftarrow y_i$
end
```

Algorithm 1: Naïve sequential CSR SpMV implementation. Simple GPU parallelization by row causes thread divergence and load imbalance.

3 HiSparse Format

Previous work on block-based formats for the CPU considered various setups, ranging from two level hierarchies to multi-level hierarchies of sparse, dense and mixed types of blocks. However no multi-level hierarchical formats have been implemented for the GPU. This is not surprising, as the hierarchical nature of such formats introduce highly dynamical execution paths, which supposedly results in inferior performance on massively parallel environments such as the GPU. Undoubtedly, these issues entailed by hierarchical formats can easily appear too big of a challenge to overcome.

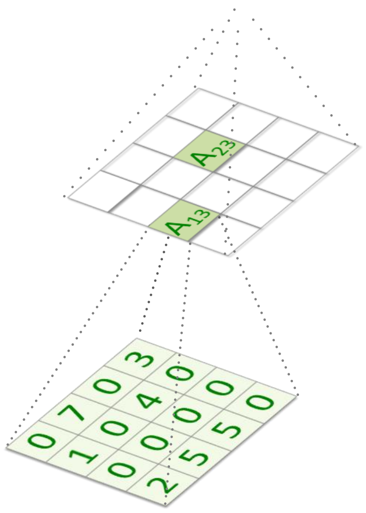
However, working on many 2D blocks of data in a hierarchical tree can in theory be efficient on the GPU. To this end, we propose and evaluate a hierarchical block-based data structure for sparse matrices: HiSparse. HiSparse matrices are constructed by nodes of size $d \times d$, whereas each node's representation depends on the sparsity pattern it describes. Nodes can either be stored in a sparse or dense format, as illustrated with a small example in Figure 3.1. As dense format we use a linear array of d^2 entries stored in row-major order. As sparse format we employ, what we call, the COOX format, which is the same as COO, with the minor difference that it is designed to enable threads to efficiently load X entries at once using efficient vector load instructions on the GPU. For example, COO4 enables each thread to load 4 entries at once. To enable vector loads, it is necessary to add padding if the number of non-zeros within the node is not a multiple of X or if the appropriate memory alignment is not given. As the size of each sparse node depends on the number of non-zeros it holds, we additionally store each node's nnz before the COO arrays.

Using different node formats enables efficient processing of nodes in respect to their sparsity pattern. If a node contains a high percentage of entries, using a dense type not only reduces the required amount of storage, but also reduces the overhead during computation. Obviously, the same applies for using a sparse type if only a low percentage of entries is non zero. In general, the HiSparse format allows to implement arbitrary node types. Consequently, it would be possible to introduce a local ELLPACK or CSR/CSC format. Note, that sparse types can also be mixed, that is, the best fitting sparse types can be used for nodes with different sparsity patterns.

The interpretation of a node's values depend on its level within the hierarchy. Inner nodes of the hierarchy maintain relative pointers to the child nodes as values, whereas leaf nodes store the corresponding matrix value. Each child pointer corresponds to an aligned data array pointer relative to the start address of the matrix. By guaranteeing that nodes are 16 byte aligned, which is necessary for vector loads, we can use the 4 lowest bits to encode additional information or increase the addressable amount of memory, or both. We primarily use them to distinguish between the different node types, simply enumerating them. For example, if there are 4 different node types, say dense, COO1, COO2 and COO4, then we use a zero, one, two and three respectively in the lowest 2 bits of the pointer to identify them. When the subspace of the input matrix covered by a child of an inner node contains no non-zero, the child pointer is set to a null value. As a consequence, data is only stored for nodes, which contain leaves with non zero values in their tree path.

The proposed dense and COOX format can easily be worked on in a transposed manner. For example, the addressing of rows and columns can simply be altered for the dense type, for the sparse type a data transpose operation can be carried out in efficient on chip shared memory, as the number of entries per node is relatively small. Even data formats like CSR could efficiently be transposed in that manner. Thus, the transpose of matrix can usually be efficiently computed on the fly alongside the algorithm run on the matrix. Therefore, we propose to decouple the matrix content and its state. To this end, the matrix content is unchanged when a matrix is transposed. Instead, a boolean in the matrix state is flipped, indicating whether a transposition is to be applied when traversing the nodes of the matrix. Furthermore, we also include a scaling factor for the matrix, which is applied to every leaf value when it is accessed. Consequently, HiSparse operations on the matrix must incorporate this state in the implementation, possibly affecting the traversal order of the matrix or changing its values.

Format conversions are often necessary in practice, thus their cost should not be ignored. Conversion between COO and HiSparse with local COO nodes is simple, since from a local standpoint the formats are similar. Creating a matrix in HiSparse format requires the generation of the tree hierarchy. We perform this conversion in two basic steps. First, we calculate the required memory and create a helper tree skeleton structure in a top-down approach. We repeatedly in-place sort the COO entries into $d \times d$ bins, whereas the bins correspond to nodes. Starting with processing the root node, each child node is stored in a queue. Furthermore, the required node memory is calculated and added to a global sum. Subsequently, this process is repeated for each queue item, until reaching the leaf nodes. Once all nodes are processed, the required memory is allocated as one large chunk of memory, which serves as pool for the allocation of the individual nodes. The second step, filling the nodes, is very efficient, since the tree structure is already known at this point and the entries are assigned to the leaves. We process the tree in a bottom up fashion, allocating memory from the pool and inserting the non-zeros. After creation of the node, we write its position to the helper skeleton, and start the processing of the parent as soon as all children of that node are completed. Obviously,



```
Sparse format:
A11 = {
  numEntries = 7,
  coords = [(0,1),(0,3),
            (1,0),(1,2),
            (1,0),(1,1),(1,2)],
  data = [7, 3, 1, 5, 2, 5, 5]
}

Dense format:
A11 = [0, 7, 0, 3,
       1, 0, 4, 0,
        0, 0, 0, 0,
        2, 5, 5, 0]
```

Figure 3.1: Example with $d = 4$. Nodes within the hierarchical HiSparse format can either be dense or sparse. Coordinate pairs are packed together.

this approach offers the possibility for substantial parallel execution.

3.1 Storage analysis

Using a hierarchical structure offers the potential to reduce the amount of memory required for storage of the matrix, while at the same time leading to the same worst case memory consumption as standard compressed formats. Without loss of generality, consider a matrix M of size $m \times m$, stored in HiSparse with a node size of d, a tree depth of $D = \log_d(m)$, L leaf nodes, N inner nodes. Let nnz_{L_i} be the number of non zero entries within a sparse leaf L_i. Additionally, let b be the number of bits required to store a single matrix element. Each node stores the number of non zeroes in a 32 bit unsigned integer. Since the coordinates within a node are limited in size by $\log_2(d)$, they can be stored in a packed storage format. Thus, the amount of bytes required for storing a single leaf

$$B_i = nnz_{L_i} \cdot (2 \cdot \log_2(d) + b) + 32.$$

Since the sum over all leaves $\sum_{i=0}^{L} nnz_{L_i} = nnz$ (with nnz being the number of non zeroes of the input matrix), the amount of storage required to store all leaves is $nnz \cdot (2 \cdot \log_2(d) + b) + 32 \cdot L$.

In case the tree is close to a full hierarchy, the number of inner nodes can be described as $\frac{L}{d^2} + \frac{L}{d^4} + \frac{L}{d^8} + ...$, which is $L \cdot \sum_i \frac{1}{d^{2^i}} < L$, leading to the memory requirements of the inner nodes being smaller than the leaf nodes. In any case, the overall memory requirements (in bits) is

$$B_{HiSparse} \approx (1 + N/L) \cdot (nnz \cdot (2 \cdot \log_2(d) + b) + 32 \cdot L)\,,$$

when the number of bits required for the pointer/offset to a child node is also b.

The storage requirements of the COO and CSR formats are

$$\begin{aligned} B_{COO} &= nnz \cdot (2 \cdot \log_2(m) + b), \\ B_{CSR} &= nnz \cdot (\log_2(m) + b) + m \cdot \log_2(nnz). \end{aligned}$$

COO stores two coordinates that must be able to hold m ($\log_2(m)$ bits) and the values themselves. The CSR format stores only one coordinate, but requires m row offsets that can point to an arbitrary matrix element of the matrix ($\log_2(m^2)$ bits). Usually, 32 bit unsigned integers are used for coordinates and offsets. This also makes it apparent that HiSparse can reduce the memory requirements if d is significantly smaller than m, such that the memory saved by the bit reduction is higher than the overhead of the inner nodes. Also, note that nodes are stored in dense form, if the memory costs of storing it in sparse format are higher. Consequently, reducing the memory requirements for denser matrices and for inner nodes in general, which are more likely to contain a higher number of entries.

In terms of asymptotic memory consumption, the HiSparse format is equal to CSR and COO. Even ignoring dense nodes, the memory consumption of a single node is always bounded by $B_{node} = \mathcal{O}(d^2) = \mathcal{O}(1)$, as it can hold a maximum of d^2 elements and their local coordinates. Thus, to evaluate the asymptotic memory consumption, we are only interested in the overall number of nodes. In the extreme case of a full matrix, $L = nnz/d^2$ and as mentioned before $N < L$. Thus,

$$B_{full} = B_{node} \cdot \mathcal{O}(nnz/d^2) = \mathcal{O}(nnz).$$

However, when there are fewer non-zeros the memory analysis becomes more complicated. In the extreme case, when $nnz = 1$, an inner node for each level of the hierarchy is needed to reach the leaf level, thus $N + L = \log_d(m)$. Starting from that situation, we can add another non-zero that shares as few nodes with the previously present non-zeros as possible. In case of adding a second non-zero this corresponds to sharing only the root node, adding $\log_d(m) - 1$ nodes. There are still $d^2 - 2$ non-zeros that be added to the matrix so that each require $\log_d(m) - 1$ additional nodes, *i. e.*, until all nodes are present on the second level of the hierarchy. This idea can be generalized to

$$\begin{aligned} N + L \leq & 1 \cdot \log_d(m) + (d^2 - 1) \cdot (\log_d(m) - 1) + \\ & + ((d^2)^2 - d^2) \cdot (\log_d(m) - 2) + \\ & + ((d^2)^3 - (d^2)^2) \cdot (\log_d(m) - 3) \ldots, \end{aligned}$$

essentially counting the nodes that are added when as many non-zeros are added in such a way that the entire level is fully filled with nodes. The number of such fully filled levels can be computed with the log over the number of non-zeros. Thus, for simplicity, let λ be $\lceil \log_{d^2}(nnz) \rceil$. The previous formula can then be written as a sum:

$$N + L \leq \sum_{i=0}^{\lambda} (d^{2^i} - \lfloor d^{2^{i-1}} \rfloor) \cdot (\log_d(m) - i)$$

and simplified as follows.

$$
\begin{aligned}
&\sum_{i=0}^{\lambda}(d^{2^i} - \lfloor d^{2^{i-1}} \rfloor) \cdot (\log_d(m) - i) \\
&< \sum_{i=0}^{\lambda} d^{2^i} \cdot (\log_d(m) - i) \\
&< \log_d(m) \cdot \sum_{i=0}^{\lambda} d^{2^i} - \sum_{i=0}^{\lambda} i \cdot d^{2^i} \\
&= \log_d(m) \frac{d^2(d^{2^\lambda} - 1)}{d^2 - 1} - \frac{d^2(-\lambda d^{2^\lambda} - d^{2^\lambda} + \lambda d^{2^{\lambda+1}} + 1)}{d^2 - 1} \\
&< \log_d(m)(d^{2^{\lambda+1}} - d^2) + \lambda d^{2^{\lambda+1}} + d^{2^{\lambda+1}} - d^2 \lambda d^{2^{\lambda+1}} - d^2 \\
&= d^{2^{\lambda+1}}(\log_d(m) + 1) + \lambda d^{2^{\lambda+1}}(1 - d^2) - d^2(\log_d(m) + 1) \\
&< d^{2^{\lambda+1}}(\log_d(m) + 1).
\end{aligned}
$$

Using the relationship of λ, the equations boils down to

$$
\begin{aligned}
N + L < nnz \cdot d^2(\log_d(m) + 1) \text{ and} \\
B = B_{node} \cdot \mathcal{O}(nnz \cdot \log_d(m) + nnz) \\
B = \mathcal{O}(nnz \cdot \log_d(m))
\end{aligned}
$$

Comparing this result to B_{COO}, it becomes apparent, that asymptotically they are identical.

In praxis, the constants do matter and efficient memory access requires to use one of the supported memory types to store coordinates. Using node dimensions $d <= 256$ enables to store the coordinates within a single byte unsigned integer, thus, reducing the storage amount significantly compared to the sparse matrix formats COO and CSR.

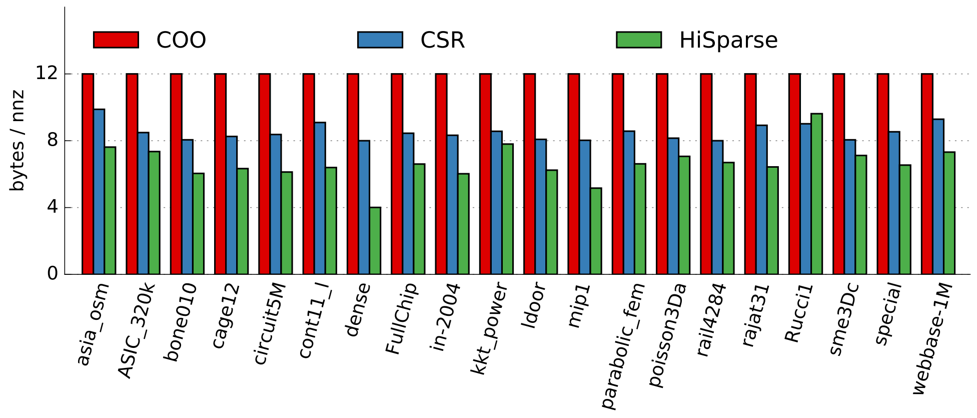


Figure 3.2: Memory consumption per non-zero of HiSparse compared to the standard formats COO and CSR for matrices stored in single precision floating point (less is better). For double precision, all memory requirements increase by 4 bytes.

Figure 3.2 shows a practical comparison of memory consumption between HiSparse, COO and CSR using a set of example matrices. Further details regarding the test matrices can be taken from Figure 7.1 and Section 7. As can be seen, HiSparse requires on average about 50% less memory than COO and 20% less memory than CSR. Also, note that COO always requires 12 bytes per non zero, since three 4 byte integers are stored for each non zero.

4 Dynamic Scheduling

After overcoming the first levels of a tree structure, it potentially offers large amounts of parallel workloads, as nodes on the same level and nodes on different levels can be worked on in parallel. However, traversing non-complete trees poses a scheduling problem. From the perspective of a single node, the number of threads that can be launched to process its children varies strongly between nodes. As threads on the GPU must be launched in form of kernels, which should at least contain hundreds of threads, this becomes difficult on the GPU. At the same time, the number of elements per node and thus the workload per node may vary greatly throughout the tree structure. Thus, if we simply assign one thread to each node, they would execute vastly different number of instructions (usually proportional to the number of elements per node), leading to execution divergence, slowing down the execution. The issue becomes worse when considering different node types, *e. g.*, sparse and dense, not only leading to different number of executed instructions, but completely different instructions.

To overcome these issues, we propose to use a dynamic scheduling approach on the GPU. For further discussion, we rely on NVIDIA CUDA terminology. For details of the processing model, see [**?**]. In principle, every algorithm running on one or multiple HiSparse matrices, must traverse the node hierarchy of one matrix or multiple matrices in parallel. It will execute functions which take one node or a combination of nodes as input. One can expect good performance, if

- a suitable number of threads can be assigned to a node, *i. e.*, dense nodes can provide parallel workloads for multiple threads, while sparse nodes that only hold a single element will only need one thread for processing.
- only threads working on the same node type and facing equal workloads end up in the same warp, *i. e.*, thread divergence is avoided.

To this end, we propose a dynamic scheduler that collects to-be-processed nodes in queues on the GPU. We provide queues for each node type and for different number of element contained per node. As providing specific queues for every possible element count is infeasible, we assign queues to ranges of elements, as shown in Figure 4.1. Considering operations that involve multiple matrices, like *e. g.* matrix matrix multiplication, queues can also be setup for combination of node types and element counts. To provide sufficient flexibility, our scheduler implementation takes a specification of the queue setup and information that should be stored in queues as template argument and generates the appropriate queuing setup during compile time.

To execute the appropriate code for the elements stored in the queues, we use a strategy similar to hybrid dynamic parallelism, as described in the Whippletree scheduler [24]: A controller block iterates over all queues and checks their fill level. If there is a sufficient number of elements in any queue (or if the number of currently executing blocks becomes low), using dynamic parallelism the controller starts a kernel that will process the element of this queue. Each block of this kernel is assigned to a region of the queue and draws as many elements from the queue such that all threads can execute coherently, *e.g.*, a block of 128 threads that is assigned to a queue of nodes that should be processed by 32 threads each, will dequeue four nodes. The entirety of this process leads to all threads running in a block working on the same node type and having similar workloads, consequently, fulfilling the previously mentioned requirements. At the same time, the kernels launched using dynamic parallelism are large enough to achieve good performance. If dynamic parallelism was used to directly start threads to work on the children of a parent node, the kernel in an extreme case might contain a single thread only, leading to all kinds of underutilization and extreme overhead issues.

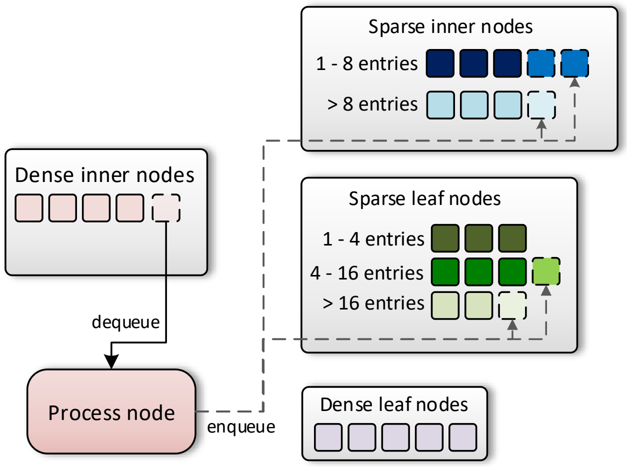


Figure 4.1: Each node type has at least one dedicated queue processed by the corresponding kernel. Sparse node types maintain different queues for varying node entry counts, whereas the thread count used to process an entry depends on the specific queue. Consequently, nodes with few entries are processed with less threads than nodes with a higher amount of entries, thus, reducing thread divergence. For example, sparse inner nodes are split into two queues, one containing nodes with one to eight entries, the other containing all nodes with more than eight entries. Each node is processed by all assigned threads after dequeuing. For inner nodes, each child node is enqueued in the node type specific queue.

5 Sparse Matrix - Dense Vector Multiplication

As a first example, we implemented SpMV using our scheduler. This is extremely challenging, as SpMV has received a lot of attention and a hierarchical format entails significant scheduling overhead (traversing the hierarchy) compared to simply processing the non-zeros in a bulk-like fashion. In any case, multiplying a HiSparse matrix with a dense vector corresponds to a top-down parallel tree traversal. We simply use queues for each node type and distinguish between nodes that only contain a single entry, few entries (up to 32), and many entries (more than 32). Each queue entry not only is associated with a node, but also holds a row and column offset, as well as the depth of the node in the hierarchy. In this way, the input and output indices of the corresponding dense vectors can be computed when processing leaf nodes.

Initially, the queues are filled by processing the root node of the input matrix. While traversing the node hierarchy, the lower bits of the node pointers are used to identify each nodes type and the corresponding queues are used for enqueue. When reaching leaf nodes, a simple local SpMV algorithm is executed as shown in algorithm 2. Transposition is trivially handled by swapping the coordinates. As the different threads might access the same output element, we use atomic operations to write the result.

```
1 HiSparse process leaf
2 Input: filled Queue $Q$, matrix state $State$, dense vector $x$
3 Output: dense vector $y$
 1: $w \leftarrow dequeue(Q)$
 2: $node \leftarrow getNode(w.node\_ptr)$
 3: $rowOffset \leftarrow w.rowOffset \cdot d$
 4: $colOffset \leftarrow w.columnOffset \cdot d$
 5: for all $i \leftarrow 0$ to $node.numEntries$ do in parallel
 6:     $coord \leftarrow node.coords[i]$
 7:     if $State.transposed == true$ then
 8:         $swapCoord(coord)$
 9:     end if
10:     $column\_idx \leftarrow colOffset + coord.column$
11:     $row\_idx \leftarrow rowOffset + coord.row$
12:     $v \leftarrow x[column\_idx] \cdot node.values()[i] \cdot State.scale$
13:     $atomicAdd(y[row\_idx], v)$
14: end for
```
Algorithm 2: HiSparse implementation of processing a leaf node. After dequeuing the work package, the global offsets in respect to the input matrix A with node dimension d are calculated. Subsequently, each node entry is multiplied with the input vector x and atomically written to the output vector y.

6 Sparse Matrix Addition

As second example, we implemented addition of two HiSparse matrices, which requires a dual top-down parallel tree traversal. To this end, the addition follows the same basic concept as HiSparse matrix-vector multiplication. We set up queues for each combination of node types and maximum number of elements in either node. Starting from the root nodes, common nodes of both input matrices are traversed together. If a specific node within the tree hierarchy only exists in one matrix, it is simply copied into the output matrix, for which we set up additional queues and copy functions.

Since the size of the output matrix is not known in advance, sufficient memory is allocated before the addition is executed. This is done by allocating a memory pool to the amount of the sum of required memory of both input matrices. However, since nodes are processed independently, a memory allocator is required to allocate memory from the memory pool. We use a simple memory allocator using an atomic counter.

When two common nodes are processed jointly, it is necessary to first compute the size of the output node in order to allocate the corresponding amount of memory from the memory pool. In case of a dense output node, this step is trivial, since the output size is statically known with $d^2 \cdot b$ bytes. For sparse output nodes, it is necessary to compute the number of union entries of both nodes. Since entries of nodes are stored as arrays in a COO format it is not possible to directly access two corresponding values at a specific coordinate. The general approach is to first count the number of common entries. Subtracting the number of common entries from the sum of total entries provides the entry count of the output node, which is required to determine the type of output node as well as its memory requirements. After allocating the memory, the input nodes are processed and the output node can be filled.

For inner nodes the value corresponds to a pointer to a child node. Since the location of the children is not known when processing the parent, we store a pointer to the value in the queue entry of a child. Consequently, during processing of the child, we set the pointer in the parent. In the following, we distinguish between different node types and different node workloads. For simplicity, we limit the discussion to local COO1 nodes. COO2 and COO4 nodes are a simple extension which only require appropriate padding and are generated as soon as the entry count surpasses a given threshold.

A first queue is used for nodes, whose number of entries is small enough for computation in shared memory by a block of threads. We at first load the entries of both nodes into shared memory before sorting the combined array of coordinates. As (according to our definition) nodes are stored in row-major order, we can simply merge the coordinate arrays if neither of the matrices is transposed. Contrary, if either input matrix is transposed and the other matrix is non-transposed, we swap row and column index accordingly while loading the data to shared memory and sort them with merge sort. In each case we sort a permutation vector alongside the coordinates, such that the original values can be fetched after sorting. After sorting, we check neighbouring coordinates for equality, counting entries that are present in both nodes. The resulting count is used to allocate the appropriate output node. In case the output is a sparse node, we use a prefix sum to remove duplicated entries from the sorted coordinate array and write it to the allocated node. Finally, we fill the node (enqueue child nodes), by using the permutation array to look up the values from the original nodes and compute the sum (generate the appropriate queue entry).

A second queue is used for node combination that cannot be processed in shared memory. In that case, we buffer the coordinates array of the first node in a batch-wise manner in shared memory. By iteration over all batches of the first node and comparing them to all the entires of the second node, we identify identical entries and can compute the output node's size. To fill the output node, we again buffer node data in shared memory. However, we now work on both nodes concurrently, loading n values with the lowest combined coordinates from both nodes and sort them. As we loaded the lowest coordinates from both nodes, we know that the lower of the two highest coordinates gives us the number of completed entries. After writing them to the output matrix (enqueuing them), we again load entries from both matrices, replacing the completed entries and perform the same steps. Loading more entries from the node that yielded the lower maximum combined coordinate guarantees that we at least generate n output values in every step.

A third and forth queue is set up for cases with one or two dense input nodes. Handling two dense nodes is trivial, as both can simply be jointly iterated. In case of sparse/dense mixed case, we already know that a dense output node will be generated. To fill the node, we iterate over the dense array, buffering it in shared memory. Concurrently, we fetch the entries from the sparse node and use each values coordinates to attach the entry to the value stored for the dense node. Again using one thread for each dense entry, we combine the non-zeros of the dense node with the attached entries from the sparse node and fill the output node accordingly (enqueue the child nodes).

7 Evaluation

In this section we provide a comparison of our HiSparse implementations with popular libraries on selected matrices. The test matrices were taken from The University of Florida Sparse Matrix Collection [23] and were chosen to show different characteristics. They are shown in Figure 3.2. Each matrix is described with key statistics for CSR and HiSparse matrix formats. CSR statistics consist of the mean non-zeros per row and standard deviation, as well as the maximum. For HiSparse, statistics are provided for inner nodes and leaf nodes separately. The statistics include the total number of sparse and dense nodes, in addition to the mean number of non zero entries within those nodes. We also include a dense matrix and a special matrix which shows very localized behavior and should thus favor a hierarchical format.

For evaluating SpMV, HiSparse is compared to cuSparse [25], cusp [26], MKL [27] and bhSparse with the CSR5 format [2]. Since bhSparse does not provide an implementation for a transposed SpMV, it is not used in the second part of the evaluation. Since cusp does not provide

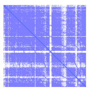
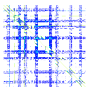
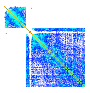
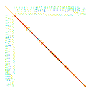
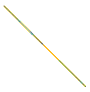
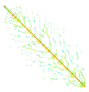
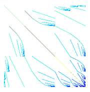
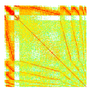
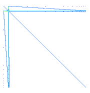
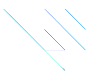
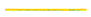
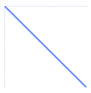
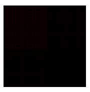
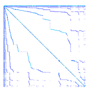
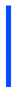
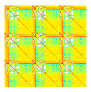
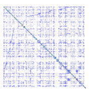
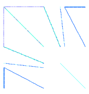
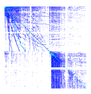
Matrix name	rows × columns	non-zeros	CSR: mean (std-dev), max	HiSparse: sparse, dense, mean nodes	HiSparse: sparse, dense, mean leaves	Structure
asia_osm	12.0M×12.0M	25.4M	2.1 (0.5), 9	236.5k, 3, 6.6	1.6M, 0, 15.2	
ASIC_320k	321.8k×321.8k	2.6M	8.2 (503.0), 203.8k	387, 0, 363.1	140.5k, 0, 17.8	
ldoor	952.2k×952.2k	46.5M	48.9 (11.9), 77	2.1k, 0, 130.8	278.6k, 0, 166.0	
mip1	66.5k×66.5k	10.4M	155.8 (350.7), 66.4k	20, 0, 15843.4	6.0k, 575, 425.5	
bone010	986.7k×986.7k	71.7M	72.6 (15.8), 81	182, 0, 496.0	90.3k, 0, 792.8	
cage12	130.2k×130.2k	2.0M	15.6 (4.7), 33	49, 0, 461.1	22.6k, 0, 89.0	
parabolic_fem	525.8k×525.8k	3.7M	7 (0.2), 7	448, 0, 170.6	76.4k, 0, 47.1	
poisson3Da	13.5k×13.5k	352.8k	26.1 (13.8), 110	1, 0, 10793.0	10.8k, 0, 31.7	
circuit5M	5.6M×5.6M	59.5M	10.7 (1356.6), 1.3M	2.5k, 0, 210.9	327.3k, 36, 179.7	
cont11_l	1.5M×2.0M	5.4M	3.7 (0.9), 5	390, 0, 158.8	61.9k, 0, 85.9	
rail4284	4.3k×1.1M	11.3M	2634 (4209.3), 56.2k	68, 0, 3474.8	236.3k, 0, 46.8	
rajat31	4.7M×4.7M	20.3M	4.3 (1.1), 1.3k	2.3k, 0, 120.3	271.0k, 0, 74.0	
dense	5.0k×5.0k	25.0M	5000 (0.0), 5.0k	1, 0, 16373.3	79, 1521, 1010.9	
FullChip	3.0M×3.0M	26.6M	8.9 (1806.8), 2.3M	2.8k, 0, 182.0	510.8k, 0, 51.1	
Rucci1	2.0M×109.9k	7.8M	3.9 (0.3), 4	848, 0, 1049.1	889.6k, 0, 7.8	
sme3Dc	42.9k×42.9k	3.1M	73.3 (37.0), 405	6, 4, 10344.7	103.4k, 0, 29.4	
in_2004	1.4M×1.4M	16.9M	12.2 (37.2), 7.8k	4.3k, 0, 787.9	143.3k, 209, 96.7	
kkt_power	2.1M×2.1M	14.6M	7.1 (7.4), 96	920, 0, 829.9	763.5k, 0, 18.1	
special	7.0M×7.0M	52.3M	7.5 (125.3), 6.3k	183, 9, 4199.9	806.4k, 0, 63.8	
webbase	1.0M×1.0M	3.1M	3.1 (25.3), 4.7k	2.3k, 0, 65.5	151.5k, 0, 19.5	

Figure 7.1: Overview of matrices used in performance evaluation. For each matrix key statistics for CSR and HiSparse format are provided. CSR statistics include the mean, standard deviation and maximal number of non zeros for the matrix rows. HiSparse statistics are separated between inner nodes and leaves. For each node type, the total number of nodes and the average number of entries within those nodes are provided. Next to the statistics are pictures illustrating the sparsity layout of the matrices in form of a heat map.

means to directly multiply a matrix as transposed, the costs of computing the transpose are included in the performance measurement unless explicitly stated otherwise. The used hardware for performance measurements consists of a i7-4790k 4x4.4 GHz CPU and a NVIDIA GeForce GTX 1080 graphics card [28] (compute capability 6.1).

Our pretests have shown that HiSparse achieves the best results with $d = 128$, Furthermore, the most stable performance over the matrix set was achieved using mixed sparse nodes of types COO1, COO2 and COO4. Whereas COO1 is used for nodes with only a single non zero entry, COO2 is used for nodes with two entries and COO4 is used for nodes with three or more entries. Note, that nodes with a high number of entries are still stored in a dense format. For more information about those node types see Section 3.

Prior to performance measurements each operation was executed with 20 warm-up iterations. The performance measurement itself is the average time of execution measured over 100 iterations. The performance in the evaluation is measured as the throughput in $GFLOPS = \frac{nnz*3}{t_{ms}*10^{-3}}$, where t_{ms} is the execution time in milliseconds. The factor 3 comes from multiplication of the input vector with the non-zero, the multiplication with the scaling factor and the addition to the output vector.

Figure 7.2a provides the performance comparison of Sparse Matrix Dense Vector multiplication ($y = A \cdot x$) with single precision for non-transposed matrices. Figure 7.2b contains the performance of transposed matrices ($y = A^T \cdot x$). Each marker represents a library as depicted in the legend. As can be seen, HiSparse is slower than other libraries in the non-transpose case, with the exception of the matrix special, which was specifically constructed to illustrate that for certain sparsity types HiSparse delivers best results. In general, bhSparse delivers best performance for non-transposed SpMV in our tests. In contrast, HiSparse always achieves best performance with transposed matrices. Table 7.1 provides the standard deviation and average measured execution times in milliseconds over the matrix test set. Since cuSparse and cusp show exceedingly bad performance with matrices *Fullchip* and *circuit5M* (up to about 50× slower than Hisparse), their averages are below HiSparse. This also shows that HiSparse, compared to cuSparse and cusp, is more stable in execution, since it is less depended on the sparsity layout of a matrix. In contrast, the matrices *Rucci* and *poisson3D* show about 7× better performance with cuSparse and cusp compared to HiSparse. Using transposed matrices it can be seen that HiSparse is significantly faster (on average about 7× faster than cuSparse). At worst, HiSparse still achieves 2× faster performance than cuSparse, while at best, HiSparse is about 18.5× faster. Figure 7.2c, 7.2d show similar results for double precision performance. Obviously, the overall performance for double precision is lower, however, the relative performance between the libraries is very similar.

Figure 7.3a provides the performance comparison of non-transposed

	Non-Transpose		Transpose	
Library	mean	std-dev	mean	std-dev
HiSparse	3.731	3.504	3.589	3.971
cuSparse	23.648	67.23	25.058	38.305
cusp	4.242	8.619	78.708	73.204
MKL	14.204	15.398	21.738	26.704
bhSparse	1.4012	2.012	-	-

Table 7.1: Average and standard deviation of SpMV single precision performance of each library over the selected matrices in milliseconds.

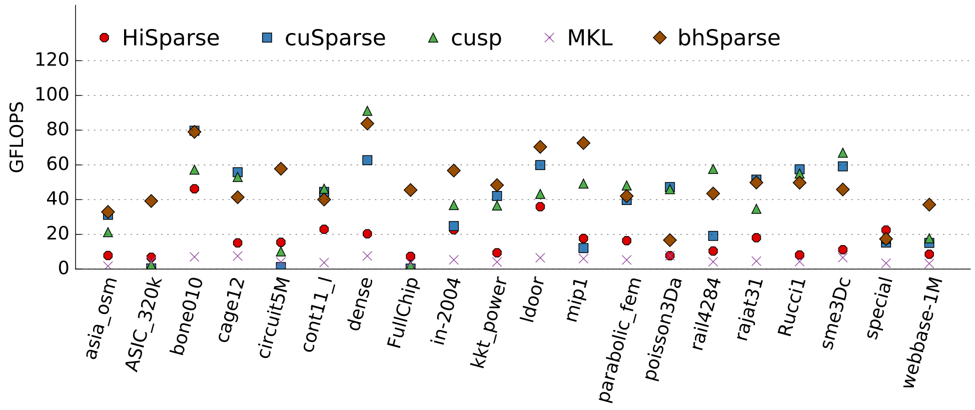
sparse matrix addition ($B = A + A$) using single precision. In contrast, Figure 7.3b shows results for transposed sparse matrix addition ($B = A + A^T$). Transposed addition for cuSparse, cusp and MKL is done by first converting the input matrix from CSR to CSC, since this is the only or at least most efficient solution. This conversion consumes a significant proportion of the execution time. In praxis, this step is only necessary once for each matrix. Each following transposed addition with the same transposed matrix no longer requires the conversion. As a consequence, Figure 7.3c provides an unbiased comparison, where conversion costs are neglected.

As the results show, HiSparse and cuSparse show alternating best performances for non-transposed matrices. The same applies to transposed addition without conversion costs. In contrast, considering CSR to CSC conversion costs for transposed addition leads to significantly better results for HiSparse, since it does not require any conversion for transposed addition. Furthermore, the results show that cusp provides notably poor performance for sparse addition in general. Table 7.2 provides average and median execution times of each library over all matrices. Again, we consider transposed addition with and without conversion costs. As can be seen, HiSparse is on average significantly faster than the compared libraries for non-transposed as well as transposed addition. However, the median of the measured HiSparse performances is similar to that of cuSparse. This applies for non-transposed as well as transposed matrices, when cuSparse conversions costs from CSR to CSC are not considered. That cuSparse is on average significantly slower is a result of cuSparse showing distinctly bad performance with the matrices *Fullchip* and *circiut5M*, which was also the case for cuSparse SpMV. Consequently, HiSparse is at worst 2× slower than cuSparse, at best it is 28.5× faster for non-transposed matrices. For transposed matrices HiSparse is at worst 6.4× slower, at best 8× faster than cuSparse without considering cuSparse conversion costs. When the conversion costs are included, HiSparse is at minimum 1.8× and up to 63.2× faster than cuSparse. As already mentioned, cusp shows remarkably bad performance for sparse addition, at best being 9.5× slower than HiSparse, at worst up to 305× slower for non-transposed matrices. Figures 7.3d - 7.3f

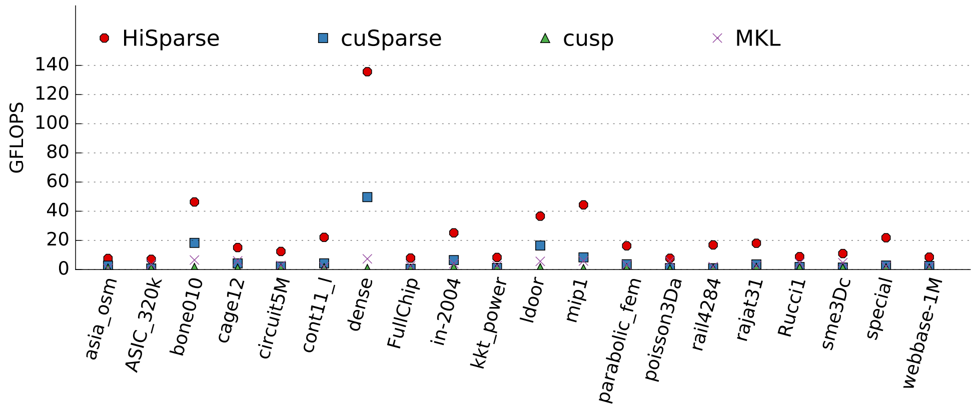
	Non-Transpose		Transpose			
			full		without conversion	
Library	mean	std-dev	mean	std-dev	mean	std-dev
HiSparse	8.656	8.449	23.382	24.874	23.382	24.874
cuSparse	42.563	92.844	179.349	189.016	46.973	101.923
cusp	326.391	369.315	384.381	411.438	277.967	276.023
MKL	109.237	104.462	280.634	255.756	126.597	120.909

Table 7.2: Average and standard deviation of the single precision sparse matrix addition performance of each library over the selected matrices in milliseconds. Times are provided for non-transpose and transpose addition. Furthermore, times for transposed addition without conversion costs are provided. Since HiSparse does not require any conversion, times are equal for transposed addition with or without conversion.

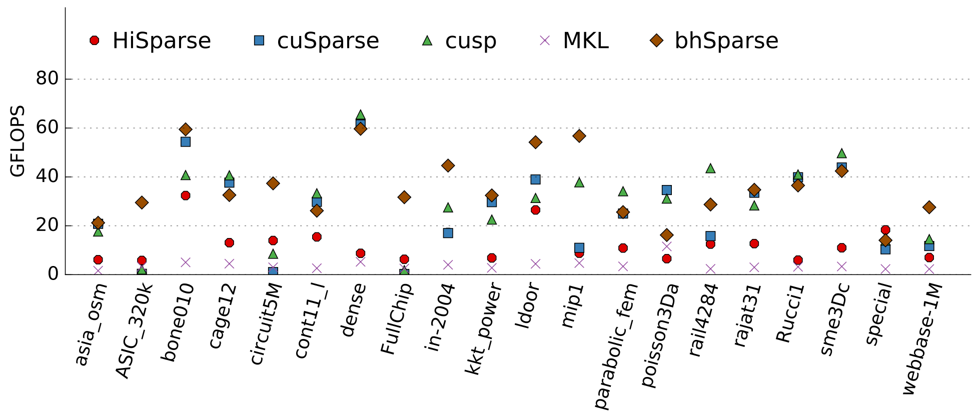
provide results for double precision. Despite the overall performance being naturally slower, the relative performance between the libraries is similar to single precision.



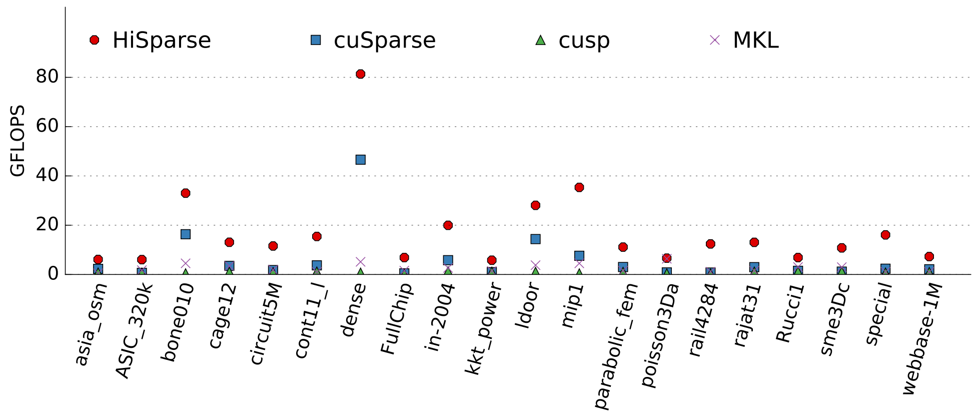
(a) Non-Transposed SpMV ($y = A \cdot x$) comparison using single precision.



(b) Transposed SpMV ($y = A^T \cdot x$) comparison using single precision.

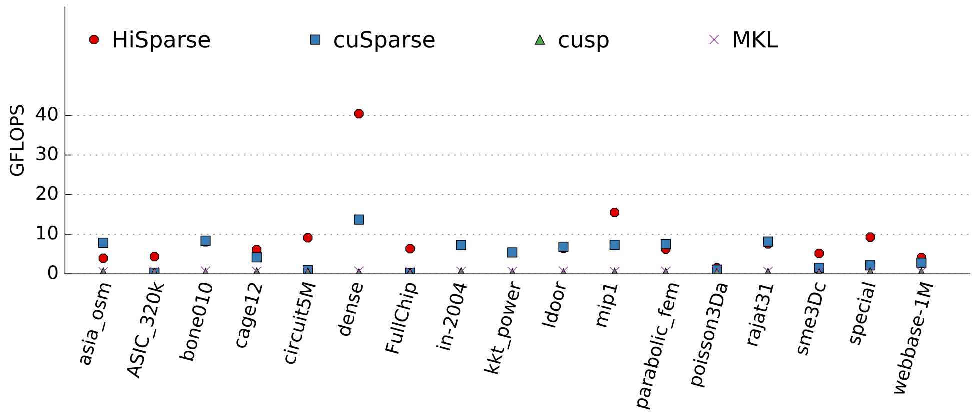


(c) Non-Transposed SpMV ($y = A \cdot x$) comparison using double precision.

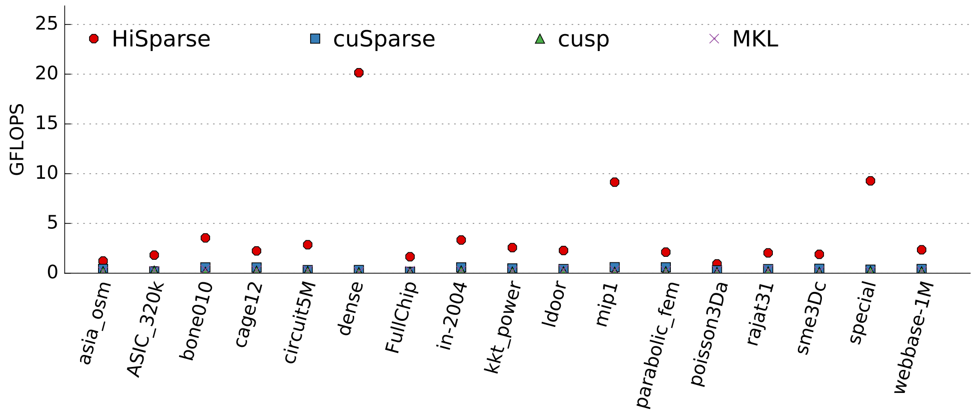


(d) Transposed SpMV ($y = A^T \cdot x$) comparison using double precision.

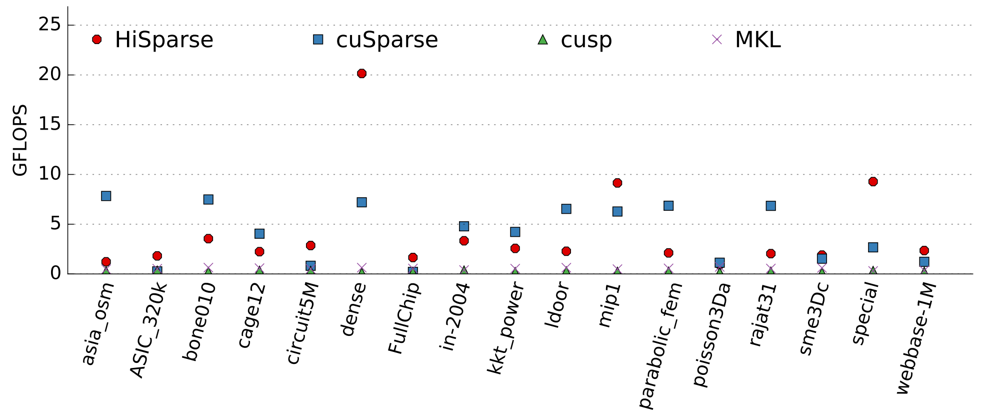
Figure 7.2: SpMV performance comparisons. Each marker corresponds to the library indicated in the legend on top of the plot. The plot shows the achieved performance in GFLOPS (higher is better). The performance does not include conversion costs. Note, that since cusp does not provide direct means to perform transposed multiplication, the performance includes costs for explicitly transposing the matrix. Also, note that bhSparse does not support transposed matrices.



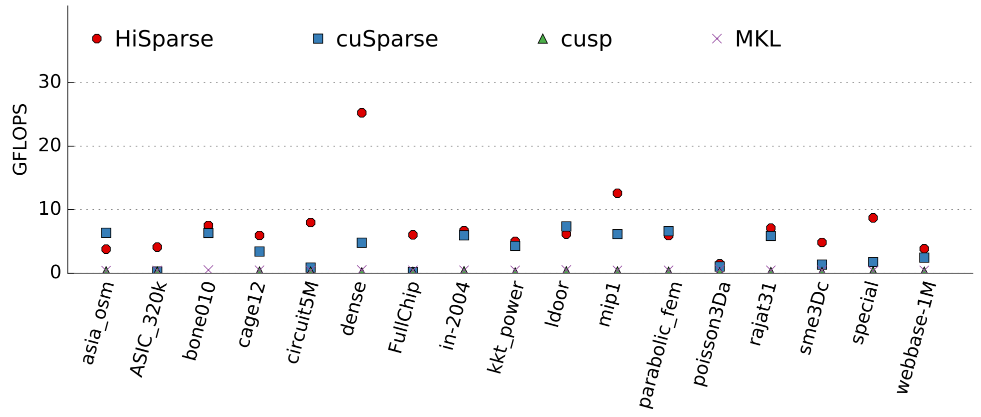
(a) Non-Transposed matrix addition ($B = A + A$) comparison using single precision.



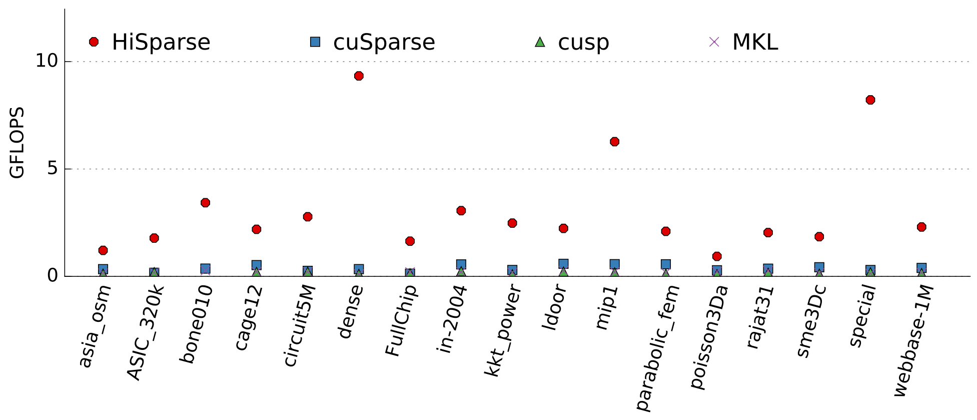
(b) Transposed matrix addition ($B = A + A^T$) comparison using single precision including conversion costs.



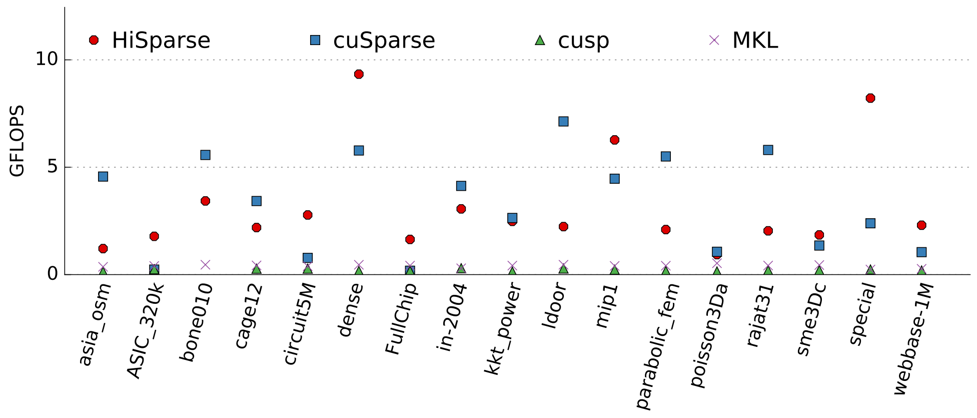
(c) Transposed matrix addition ($B = A + A^T$) comparison using single precision without conversion costs.



(d) Non-Transposed matrix addition ($B = A + A$) comparison using double precision.



(e) Transposed matrix addition ($B = A + A^T$) comparison using double precision including conversion costs.



(f) Transposed matrix addition ($B = A + A^T$) comparison using double precision without conversion costs.

Figure 7.3: Matrix addition performance comparisons. For non-transposed matrices the performance does not include conversion costs. Also, note that since the matrix A is added to itself, the matrix structure is not changing. We distinguish two cases for the performance comparison plots using transposed matrices. For one, the performance includes conversions between CSR and Compressed Sparse Column (CSC) for cuSparse, cusp and MKL. For those libraries, the matrix is first converted to CSC and subsequently added to the original matrix. For the second case, those conversion costs are not included in the performance.

8 Conclusion and future work

In this work, we introduced HiSparse, a hierarchical storage format for sparse matrices on the GPU. We showed that using a hierarchical matrix structure as a sparse matrix format is in praxis more storage efficient than the common CSR and COO formats. Furthermore, we were also able to show that using such a hierarchical sparse matrix format is viable for efficient computation on the GPU when paired with dynamic scheduling capabilities. We evaluated HiSparse by comparing our SpMV and sparse addition implementation to cuSparse, cusp and MKL. Since our format does not prefer rows over columns, similar performance results for non-transposed and transposed matrix computations are achieved. Consequently, HiSparse SpMV provides the fastest performance for transposed matrices, while not being significantly slower for non-transposed matrices. We showed that our format provides overall best performance for sparse addition of non-transposed and transposed sparse matrices.

We believe that the strength of a hierarchical format becomes especially apparent when scaling to multi GPU and multi node cluster setups. Also, when sequences of more complex operations are involved, the hierarchical format can be more efficient, as sequences of operations on nodes can potentially be grouped and different matrices can potentially share sub trees. We do not think that a consistent back and forth between a hierarchical format and a standard format, like CSR, during an algorithm makes sense. Thus, we only see success of the hierarchical format, if all necessary operations are provided. With our dynamic scheduler, adding new operations is as simple as writing operations for each node type and letting the scheduler take care of scheduling them efficiently. We hope that this encourages the community to pick up our format and extend it. In any case, in the near future, we will look into more complex operations like Sparse Matrix Multiplication ($C = A \cdot B$), which can be realized very efficiently in a hierarchical format as the collisions of non-zeros from both matrices can be determined easily.

Bibliography

[1] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 233–244.

[2] W. Liu and B. Vinter, "CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication," *CoRR*, vol. abs/1503.05032, 2015. [Online]. Available: http://arxiv.org/abs/1503.05032

[3] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[4] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies," *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[5] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, "Optimization of sparse matrix-vector multiplication using reordering techniques on gpus," *Microprocess. Microsyst.*, vol. 36, no. 2, pp. 65–77, Mar. 2012.

[6] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 769–780. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.68

[7] H. Yoshizawa and D. Takahashi, "Automatic tuning of sparse matrix-vector multiplication for crs format on gpus," in *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, Dec 2012, pp. 130–136.

[8] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA:

IEEE Press, 2014, pp. 781–792. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.69

[9] Y. Liu and B. Schmidt, "Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 82–89.

[10] S. Dalton, S. Baxter, D. Merrill, L. N. Olson, and M. Garland, "Optimizing sparse matrix operations on gpus using merge path," in *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 2015, pp. 407–416. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2015.98

[11] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Comput.*, vol. 49, no. C, pp. 179–193, Nov. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2015.04.004

[12] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–125.

[13] F. Vázquez, J. J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on nvidia gpus," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 8, pp. 815–826, Jun. 2011.

[14] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1177/1094342004041296

[15] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *SIGPLAN Not.*, vol. 45, no. 5, pp. 115–126, Jan. 2010.

[16] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007. [Online]. Available: http://dx.doi.org/10.1007/s00200-007-0038-9

[17] D. Langr, I. Šimecek, and P. Tvrdík, "Storing sparse matrices to files in the adaptive-blocking hierarchical storage format," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, Sept 2013, pp. 479–486.

[18] B.-Y. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New

York, NY, USA: ACM, 2012, pp. 353–364. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304624

[19] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597678

[20] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: Yet another spmv framework on gpus," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 107–118. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555255

[21] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465013

[22] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11227-012-0825-3

[23] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[24] M. Steinberger, M. Kenzel, P. Boechat, B. Kerbl, M. Dokter, and D. Schmalstieg, “Whippletree: Task-based scheduling of dynamic workloads on the gpu,” *ACM Trans. Graph.*, vol. 33, no. 6, pp. 228:1–228:11, Nov. 2014. [Online]. Available: http://doi.acm.org/10.1145/2661229.2661250

[25] NVIDIA, *The API reference guide for cuSPARSE, the CUDA sparse matrix library.*, v7.5 ed., NVIDIA, October 2016.

[26] S. Dalton, N. Bell, L. Olson, and M. Garland, “Cusp: Generic parallel algorithms for sparse matrix and graph computations,” 2014, version 0.5.0. [Online]. Available: http://cusplibrary.github.io/

[27] Intel Corporation, *Intel math kernel library*, Intel Corporation, October 2016, see http://software.intel.com/en-us/intel-mkl/.

[28] Nvidia, “Nvidia geforce gtx 1080 whitepaper,” 2016.