Dynamic Scheduling for Efficient Hierarchical Sparse Matrix Operations on the GPU

Andreas Derler
Graz University of Technology
Graz, Austria
derler@icg.tugraz.at

Rhaleb Zayer
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
rzayer@mpi-inf.mpg.de

Hans-Peter Seidel
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
hpseidel@mpi-inf.mpg.de

Markus Steinberger
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
msteinbe@mpi-inf.mpg.de

ABSTRACT

We introduce a hierarchical sparse matrix representation (*HiSparse*) tailored for the graphics processing unit (GPU). The representation adapts to the local nonzero pattern at all levels of the hierarchy and uses reduced bit length for addressing the entries. This allows a smaller memory footprint than standard formats. Executing algorithms on a hierarchical structure on the GPU usually entails significant synchronization and management overhead or slowdowns due to diverging execution paths and memory access patterns. We address these issues by means of a dynamic scheduling strategy specifically designed for executing algorithms on top of a hierarchical matrix on the GPU. The evaluation of our implementation of basic linear algebra routines, suggests that our hierarchical format is competitive to highly optimized standard libraries and significantly outperforms them in the case of transpose matrix operations. The results point towards the viability of hierarchical matrix formats on massively parallel devices such as the GPU.

CCS CONCEPTS

•**Computing methodologies → Massively parallel algorithms;** •**Mathematics of computing** → *Mathematical software performance;*

KEYWORDS

sparse matrix, hierarchical, GPU, linear algebra

ACM Reference format:
Andreas Derler, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2017. Dynamic Scheduling for Efficient Hierarchical Sparse Matrix Operations on the GPU. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079085

1 INTRODUCTION

Sparse matrices algebra has become an ineluctable workhorse across various scientific computing applications and its performance plays a defining part in the overall algorithmic performance. To attend to the ever demanding performance needs, a variety of sparse matrix formats have been proposed over the years covering virtually all available hardware architectures. While early 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 on them in special settings. In particular, the compressed sparse blocks format (CSB) has been shown to scale well on parallel CPU architectures and to yield steady performance for transpose based operations [1].

Given the rising popularity of the graphics processing unit (GPU), classical formats have been adapted, tuned, or optimized for these low cost parallel architectures [2–7]. Unfortunately, to the best of our knowledge, hierarchical formats have not been explored in this context and are widely considered unviable for GPU execution as they introduce highly dynamic execution paths, which is detrimental for performance on graphics hardware.

In this work, we tackle this issue by providing a hierarchical sparse matrix format suitable for the GPU. Hierarchical matrix representations are not new. Their potential has been recognized in early work on the finite element method [8]. Nonetheless their lack of popularity can be attributed to the burden of accommodating linear algebra primitives to the hierarchical representation. It has been observed in earlier work on vector processors that the use of hierarchical formats can lead to significant memory savings [9]. This storage cost reduction is a key motivation behind our effort to accommodate a hierarchical format on the GPU, where memory resources are much more limited. Furthermore, a hierarchical format offers several additional advantages: it scales well for large matrices, allows for implementing algorithms in a divide an conquer manner, and permits 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 and 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 and thus the overall memory requirements are generally below those of COO and CSR.

- We describe a dynamic GPU scheduling framework that allows implementing algorithms on top of HiSparse. Algorithms can implement specialized routines for the different node types and dynamically adjust the number of threads used for each step of the implementation.
- We provide a typical implementation of SpMV using our dynamic scheduler, which traverses the hierarchical structure of the sparse matrix in parallel, combining the local SpMV results into a global output vector.
- We demonstrate a typical 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.

The remainder of this paper is structured as follows. First, we provide a brief overview of the most common sparse matrix formats and review related work (Section 2). Then, we introduce the HiSparse format and provide an analysis of its 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 we 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 natural format, it consists of three arrays, 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, 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.

2.1 Alternative Sparse Matrix Formats

Besides COO and CSR, several other specialized formats have been proposed. ELLPACK pads rows so that they are all equally sized. For matrices with small row length variation, this approach can significantly boost the performance of algorithms, especially on parallel devices like the GPU. However ELLPACK has severe issues with irregular row lenghts, since much of the data needs to be zero padded. There are several ELLPACK format adoptions, like Hybrid [10], Sliced ELLPACK [3] and ELLPACK-R [4]. However, a common shortcoming of those formats is dealing with highly irregular matrix structures.

A common way of dealing with load balancing issues on parallel architectures is splitting matrices into two dimensional blocks. For instance, the CSR5 format [7] arranges the non zeros into tiles (2D blocks) of fixed size. Each tile can be processed individually. Several block formats organize blocks in a two layered hierarchy, where the top layer manages the blocks and the bottom layer stores the non zeros. Blocked CSR (BCSR) [11] divides a sparse submatrix into dense blocks and stores blocks in a CSR format. This approach is inefficient for very sparse matrices [12]. Compressed Sparse Blocks (CSB) [1] uses a COO representation of sparse submatrix blocks. Rows are not favored over columns, which is crucial for transpose multiplication [13]. Since the blocks are always stored in COO format, CSB is inefficient for locally dense matrix subregions. Mixed type formats try to deal with this limitation: Adaptive-blocking hierarchical storage format (ABHSF) [14] stores blocks in a dense, CSR, COO or bitmap format. There are numerous similar formats, like the Cocktail format [5], BRC (Blocked Row-Column) [15], BCOO (Block-based Compressed Common Coordinate) [16], ESB (ELLPACK Sparse Blocks) [17] or JAD (JAgged Diagonal) [18].

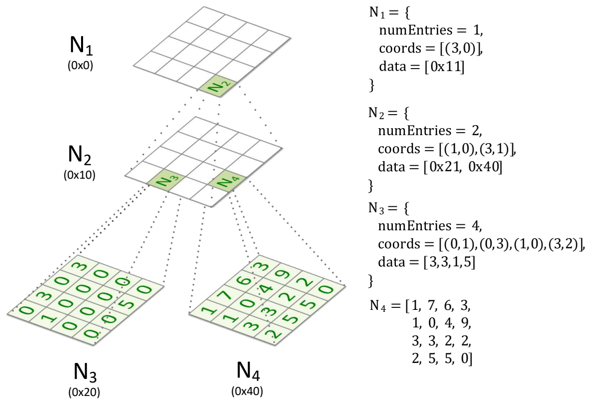
There is no reason to limit block-based formats to two layers. HiSM [9], for instance, builds a full hierarchy of fixed size COO nodes and RCSR [19] stacks CSR nodes of arbitrary size on top of anther. As nodes can be limited in size, the bit length for COO or CSR indices can be reduced and an overall compression compared to standard formats can be achieved. Stathis et al. [9] point out that such a format could be well suited for vector processors, but show results only for a hypothetical architecture. While the processing of individual nodes is potentially efficient on vector processors, a hierarchical format poses a series of challenges for the scheduling on massively parallel devices. Thus, it is not surprising that there exists no implementation of such a hierarchical format for the GPU.

2.2 GPU SpMV

One of the most common, if not the most standard test for sparse matrix formats is sparse matrix-vector multiplication. For CSR matrices, a simple strategy assigns one thread per row. Bell and Garland call this approach the *scalar* CSR kernel [10]. However, it entails severe performance limitations due to inefficient memory access patterns and load balancing issues. Multiple approaches were introduced to counterbalance these issues. One can use multiple threads per row [2], apply grouping and reordering techniques [6, 20], or dynamically choose the number of threads for each row [21, 22]. Load balancing issues can be combated dynamically by using a global row counter [23] or searching for a work load balance considering rows and/or nnz [24, 25]. Liu et al also introduced an SpMV algorithm specifically for heterogeneous processors [26].

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 format has been implemented for the GPU. This is not surprising, as the hierarchical nature of such formats introduce highly dynamical execution paths, performance is likely to degrade on massively parallel environments such as the GPU.



```
N1 = {
  numEntries = 1,
  coords = [(3,0)],
  data = [0x11]
}
N2 = {
  numEntries = 2,
  coords = [(1,0),(3,1)],
  data = [0x21, 0x40]
}
N3 = {
  numEntries = 4,
  coords = [(0,1),(0,3),(1,0),(3,2)],
  data = [3,3,1,5]
}
N4 = [1, 7, 6, 3,
      1, 0, 4, 9,
      3, 3, 2, 2,
      2, 5, 5, 0]
```

Figure 1: Example with $d = 4$. Nodes within the hierarchical HiSparse format can either be dense or sparse. Coordinate pairs are packed together.

3.1 Node Types

Starting from the observation that conceptually working on multiple 2D blocks of data in a hierarchical tree can be efficiently performed on the GPU, we propose and evaluate *HiSparse*, a hierarchical block-based data structure for sparse matrices. Similar to other hierarchical formats, e.g., HiSM [9], HiSparse matrices are constructed by nodes of size $d \times d$, whereas each node's representation depends on the sparsity pattern it describes. In HiSparse, nodes can either be stored in a sparse or dense format, as illustrated with a small example in Figure 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 directly 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 an appropriate sparse type if only a low percentage of entries is non zero. In general, the HiSparse format allows to implement and mix arbitrary node types. Consequently, it would be possible to introduce a local ELLPACK or CSR/CSC format. Sparse types can also be mixed, that is, the best fitting sparse types can be used for nodes with different sparsity patterns. For nodes with one entry only, we always use COO1 as padding would multiply the memory and computation overhead. For two entries COO2 is best suited, and so on.

The interpretation of a node's values depends on its level within the hierarchy. Inner nodes of the hierarchy maintain 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. In fact, addresses will always have zero bits there, and we may as well make use of them to increase memory efficiency. We primarily use them to distinguish between the different node types, by 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 submatrix covered by a child of an inner node contains no non-zeros, the child pointer is not included in sparse nodes and set to a null value for dense nodes. 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 be seamlessly used in a transposed manner. For example, the addressing of rows and columns can simply be altered for dense nodes, for sparse nodes a transpose operation can be carried out in efficient on chip shared memory, as the number of entries per node is relatively small. Even nodes stored in CSR could efficiently be transposed in that manner. Thus, the matrix transpose can be computed on the fly as the algorithm operates on the matrix. Therefore, we propose to decouple the matrix content and its state, such that 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, 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.

3.2 Data Conversion

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 in a top-down approach. To this end, we repeatedly in-place sort the COO entries into $d \times d$ bins, whereas the bins correspond to nodes. Repeating this process until the leaf level is reached, we compute the number of entries for each node and its memory requirements. Once all nodes are processed, the required memory is allocated as one large chunk, 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, this approach offers the possibility for substantial parallel execution.

A hierarchical format offers the ability to perform editing operations efficiently compared to traditional formats. For example, when removing or adding entries to a matrix stored in CSR, all entries that lie behind this value in memory need to be copied. In HiSparse, only the affected nodes need to be altered or generated anew. Thus, a hierarchical format is especially well suited for dynamic scenarios that involve small edits to matrices, inserting or removing data locally. Also note that with HiSparse, matrices can share nodes and thus common sub-parts in matrices. Consider a scenario in which a matrix is edited by adding few entries and subsequent steps require both the original and the edited matrix. While other formats would require copying and editing the first matrix, HiSparse could simply add those nodes that are affected by the insertion operation and share the unchanged part among both matrices, saving both operations and storage. The importance of matrix editing is highlighted by the recent interest in formulating graph algorithms in terms of linear algebra operations [27].

3.3 Storage Analysis

It has been empirically shown that using a hierarchical structure offers the potential to reduce the amount of memory required for storing the matrix [9, 19]. For HiSparse, we analyze the storage gains mathematically and prove that the worst case memory requirements are asymptotically optimal. 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, and 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 number of bits required for storing a single leaf is

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

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 $\frac{L}{d^2} + \frac{L}{d^4} + \frac{L}{d^8} + \ldots$, 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

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

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

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

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 ($\log_2(nnz)$ 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, this switch

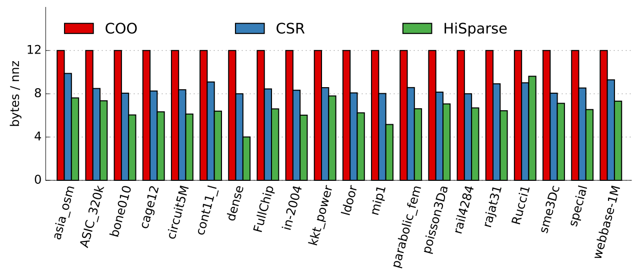


Figure 2: Memory consumption per non-zero of HiSparse compared to the standard formats COO and CSR for matrices stored in single precision floating point (less is better). For double precision, all requirements increase by 4.

reduces 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 $B_{node} = O(d^2) = O(1)$, 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,

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

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 such a configuration, we can add a second non-zero which shares as few nodes as possible with the previously present non-zeros. This corresponds to sharing only the root node, adding $\log_d(m) - 1$ nodes. There are still $d^2 - 2$ non-zeros that can be added to the matrix 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

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

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 has an upper bound in

$$N + L < d^{2^{\lambda+1}} (\log_d(m) + 1).$$

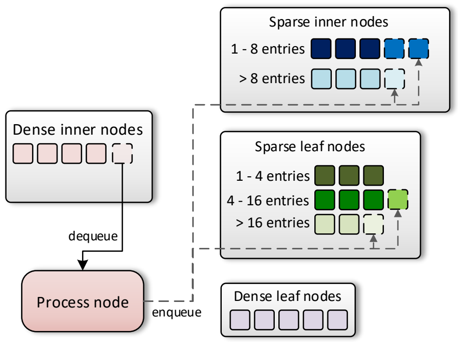


Figure 3: Each node type has at least one dedicated queue. Sparse node types maintain different queues for varying node entry counts and the thread count used to process a node depends on the specific queue. Consequently, nodes with few entries are processed with fewer threads. For example, sparse inner nodes are split into two queues, one containing nodes with one to eight entries, the other containing all nodes with more than eight entries.

The complete derivation is shown in the appendix. Using the relationship of λ, the equations boils down to

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

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

In practice, 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 integer, thus, reducing the storage amount significantly compared to the sparse matrix formats COO and CSR. Figure 2 shows a practical comparison of memory consumption between HiSparse, COO and CSR for some example matrices. For further details regarding the test matrices see Figure 4 and Section 7. HiSparse requires on average about 50% less memory than COO and 20% less memory than CSR. For special non-zero distributions as, *e.g.*, in *Rucci1*, CSR can be slightly more efficient then HiSparse. As we switch to dense nodes when there are many entries, the memory overhead of HiSparse for a dense matrix is negligible in comparison to a complete dense storage (4 bytes per non zero). 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, a hierarchical description 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 assigned one thread to each node, each thread may execute a different number of instructions (proportional to the elements in the node), leading to execution divergence and reduced performance. 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 execution paths.

To overcome these issues, we propose to use a dynamic scheduling approach on the GPU. In the ensuing discussion, we will use the terminology of NVIDIA CUDA [28]. In principle, every algorithm running on one or multiple HiSparse matrices, must be able to traverse the node hierarchy of a single or multiple matrices in parallel. Furthermore, it will execute functions on one node or a combination of nodes. One can expect good performance, if

- a suitable number of threads can be assigned to a node, *i.e.*, dense nodes 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.

4.1 Node-based Queuing

To this end, we propose a dynamic scheduler that collects to-be-processed nodes in queues on the GPU. Whenever an inner node is processed and threads should be started for the child nodes, these nodes are put into a queue instead and are stored for later execution. After the number of nodes in a queue reaches a certain threshold, we can launch a kernel of sufficient thread count to process these nodes efficiently. To tackle the requirement of equal workloads, we provide queues for each node type and for different numbers of elements contained per node. As providing specific queues for every possible element count is infeasible, we use a binning strategy, assigning queues to ranges of elements, as shown in Figure 3. Considering operations that involve multiple matrices, like, *e.g.*, matrix-matrix addition, queues can also be set up for combinations of node types and element counts. To provide sufficient flexibility, our CUDA/C++ scheduler implementation takes a specification of the queue setup and information that should be stored in queues as template arguments and generates the appropriate queuing setup during compile time. As underlying queue implementation we use a queue with state flags for each element [29].

While storing nodes in queues solves the issue of small kernel launches, it introduces another problem. Starting kernels from the CPU, we would have to copy the queue fill level from the GPU to the CPU after every step of the algorithm, *i.e.*, after every level of the tree has been processed. This introduces synchronization barriers, leaves the GPU under-utilized, and prohibits load balancing between nodes of different levels of the tree. One solution to this problem would be a so-called persistent threads or megakernel approach [30], launching persistently running blocks which contain code for all supported node types. Such an approach has the downside that it can significantly hurt performance as the resource requirements (registers, shared memory) of the megakernel are determined by the node type with the highest requirements [31].

4.2 GPU-Controlled Dynamic Scheduling

To avoid both a constant back and forth between CPU and GPU and suboptimal resource usage we propose a GPU-controller that starts kernels using dynamic parallelism directly from the GPU. This controller takes over the responsibility of the CPU in terms of determining queue fill levels and launching kernels. Typically, it is sufficient to use a single warp which stays active until the entire algorithm is completed. To this end, the warp keeps iterating the same loop, checking all queue fill levels whether there are enough queued nodes to launch a kernel of appropriate size and initiates its execution. 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 of 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. Note that if dynamic parallelism was used to directly start threads to work on the children of a parent node, the kernel would contain very few threads (possibly a single thread only) leading to all kinds of under-utilization and severe overheads.

To increase performance, the threads of the controller block can, each check a queue in parallel, and all launch kernels at the same time. We starts all kernels into separate streams such that they can execute concurrently. Furthermore, the controller needs to keep track of currently running blocks. To this end, we employ an atomically operated global counter, that the controller increments by the number of launched blocks when it starts a kernel. One thread of each finished block decrements the counter by one. This allows the controller to react on situations when the number of running blocks becomes low by starting kernels which are below the desired size. This situation usually happens at the beginning of the algorithm when only the root node can be used and subsequent nodes should be started immediately; and when the algorithm is about the finish and only few nodes are left in the queues which should be worked on to complete the algorithm. The controller needs to stay active until all queues are empty and the block counter reaches zero, guaranteeing that no new nodes will be enqueued anymore.

Our scheduler guarantees full utilization of threads by adjusting the number of nodes executed within a warp/block. For instance, consider a block size of 256. For small nodes, which require only 4 threads, we combine 64 nodes to be worked on in parallel so that all 256 threads are active. For larger nodes, *e.g.*, 32 threads required, we combine 8 nodes to be worked on in the block, thus keeping all threads active. A heavy node (256 threads required) will be worked on by the entire block as whole.

5 SPARSE MATRIX - DENSE VECTOR MULTIPLICATION

As a first linear algebra application, we implemented SpMV using our scheduler. 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 is not only 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 1. Please note that thanks to our scheduler (sec. 4), writing algorithms does not involve writing kernels but rather individual functions for specific node types in the hierarchy. 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.

HiSparse Leaf SpMV

```
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 1: HiSparse implementation for processing a leaf node for SpMV. After dequeue, the global offsets w.r.t the input matrix A of 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 a second application, 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 an additional queue and copy function. Since the size of the output matrix is not known in advance, a memory pool of sufficient size is allocated before the addition is executed. Since nodes are processed independently, a memory allocator is required to allocate memory from the pool. We use a simple allocator using an atomic counter.

When two 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 pool. In case of a dense output node this step is trivial, since the output size is statically known. For sparse 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 and subtract it from the sum of total entries. 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 the child. Consequently, during processing of the child, we set the pointer in the parent.

Sparse nodes. For simplicity, we do not distinguish between different COO*X* nodes as they only differ in terms of padding. For any combination of COO*X* nodes, we provide two queues, distinguishing between nodes that can efficiently be processed in shared memory and those that exceed the shared memory requirements. In the second case, we work on batches that fit in shared memory at once, generating the output in batches. In any case, we start by fetching the coordinates of both nodes and sort them in a combined manner, allowing to easily identify and count the common entries. Next, we allocate the required storage for the output, compute a prefix sum over the unique entries to calculate the output positions and write the entries to the new node or enqueue the child nodes for processing. To choose the appropriate queue for enqueue, we rely on the node type stored alongside the pointer and we fetch the entry count from the child nodes to determine whether the data fits in shared memory. As (according to our definition) nodes are stored in row-major order, we can avoid sorting and simply merge the coordinate arrays if neither of the matrices is transposed, yielding slightly better performance in this cases.

Dense nodes. Handling two dense nodes is trivial, as both can be jointly traversed. 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

We compare the performance of our HiSparse implementations to that of popular libraries on The University of Florida Sparse Matrix Collection [32]. Statistics related to our format are shown on typical matrices in Figure 2. We also include a dense matrix and a special matrix which has a very localized non zero pattern.

Our pretests have shown that HiSparse performs well with $d = 128$ and mixed sparse nodes of types COO1, COO2 and COO4. Whereas COO1 is used for nodes with only a single non zero entry,

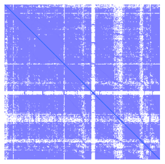
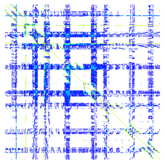
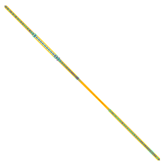
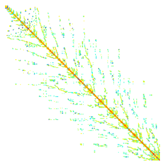
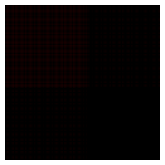
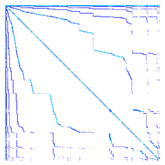
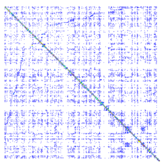
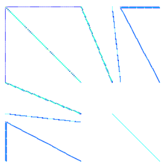
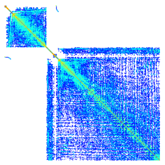
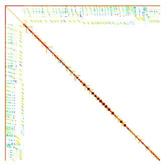
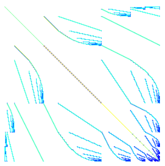
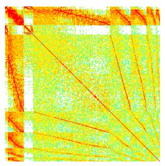
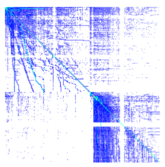
Matrix name	rows × columns	non-zeros	CSR: row mean (std-dev), max	HiSparse nodes: #sparse, #dense, mean entries	HiSparse leaves: #sparse, #dense, mean nnz	Structure
asia_osm	12.0M×12.0M	25.4M	2.1 (0.5), 9	236.5k, 3, 6.6	1.6M, 0, 15.2	
ASIC_320k	321.8k×321.8k	2.6M	8.2 (503.0), 203.8k	387, 0, 363.1	140.5k, 0, 17.8	
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	
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	
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	
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	
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	
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 4: Overview of some matrices used in performance evaluation alongside statistics about storing them in CSR or HiSparse.

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.

SpMV. For evaluating SpMV, we compare HiSparse to cuSparse CSR [33], cusp CSR [34], Naive SpMV [35], bhSparse [7], clSpMV [5], and yaSpMV [16]. bhSparse, clSpMV and yaSpMV do not provide an implementation for a transposed SpMV. clSpMV and yaSpMV only work with single precision floating point. As cusp does not provide 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 [36] (compute capability 6.1).

Prior to performance measurements each operation was executed with 20 warm-up iterations. The performance measurement itself is the average time of executions measured over 100 iterations. The performance in the evaluation is reported as the throughput in $GFLOPS = nnz \cdot 3/t_s \cdot 10^{-9}$. The factor 3 comes from multiplication of the input vector with the non-zero, the multiplication with a scaling factor and the addition to the output vector.

Figure 5 provides the performance comparison for SpMV and SpMVT in double precision floating point, average number for single and double precision are shown in Table 1. Comparing performance to state-of-the-art approaches and formats that are specifically designed to reach high SpMV throughput on GPUs is challenging as SpMV has received a lot of attention and a hierarchical format entails significant scheduling overhead compared to simply processing the non-zeros in a bulk-like fashion. As can be seen, HiSparse is in general not outperforming the 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. Especially when comparing to the formats specifically designed for SpMV (bhSparse, clSparse, and yaSpMV), which are on average up to 3× faster than HiSparse, it becomes apparent that traversing a hierarchy for multiplying every entry of a matrix once is not the most efficient. However, comparing to standard libraries, the performance of HiSparse is competitive. Since cuSparse and cusp show exceedingly bad performance with the matrices *Fullchip* and *circuit5M* (up to about 50× slower than Hisparse), their average performance is even below HiSparse. This indicates that HiSparse is a consistent performer with low standard deviation.

In contrast to SpMV, HiSparse achieves the best performance with transposed matrices. It outperforms both cuSparse and cusp as well as a naive SpMVT implementation based on atomic addition. For single precision SpMVT HiSparse performs on average 7× better than cuSparse, 19× better than cusp and 5× faster than naive and for double precision the average improvement is 6× compared to cuSparse, 13× to cusp and 5× better than naive. The performance of HiSparse hardly changes from SpMV to SpMVT. The only real difference is the dense matrix, which is due to a different memory access pattern for all dense nodes. Again, HiSparse shows very little variance in terms of execution speed, since it is less dependent on the sparsity pattern of the matrix.

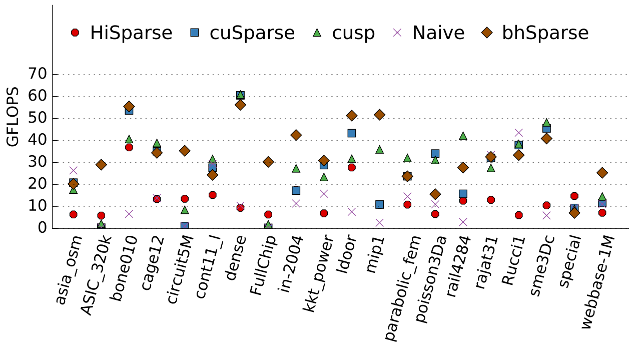
Library	SpMV mean	SpMV s.dev	SpMVT mean	SpMVT s.dev
HiSparse	3.45	3.45	3.45	3.88
cuSparse	23.51	67.27	24.16	37.78
cusp	4.28	8.65	67.31	64.85
Naive	22.51	57.80	18.11	42.01
bhSparse	1.52	2.51	-	-
clSpMV	1.31	1.34	-	-
yaSpMV	0.77	0.76	-	-

(a) single precision

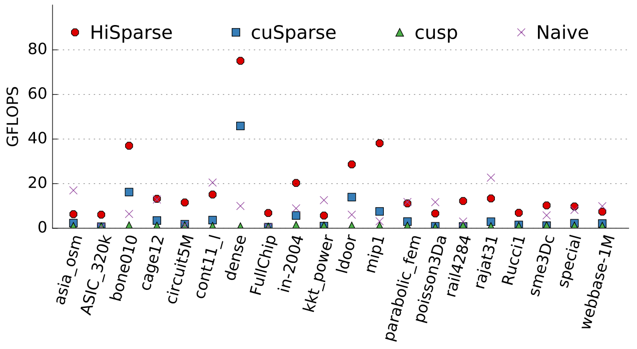
SpMV mean	SpMV s.dev	SpMVT mean	SpMVT s.dev
4.68	4.13	4.47	4.62
27.11	76.00	27.90	42.76
5.29	10.40	61.01	58.19
30.71	74.65	23.92	51.92
2.16	3.23	-	-
-	-	-	-
-	-	-	-

(b) double precision

Table 1: Average and standard deviation of the execution time (ms) for SpMV.



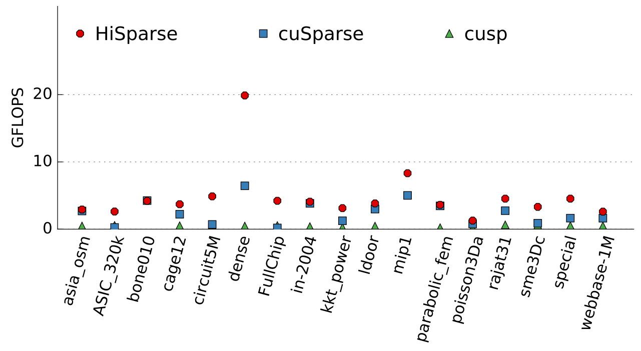
(a) SpMV comparison using double precision.



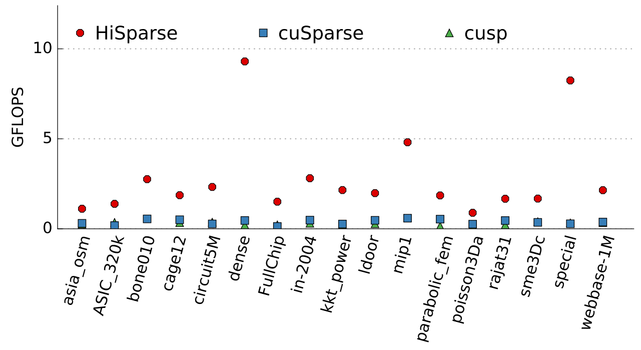
(b) SpMVT comparison using double precision.

Figure 5: SpMV performance in GFLOPS (higher is better).

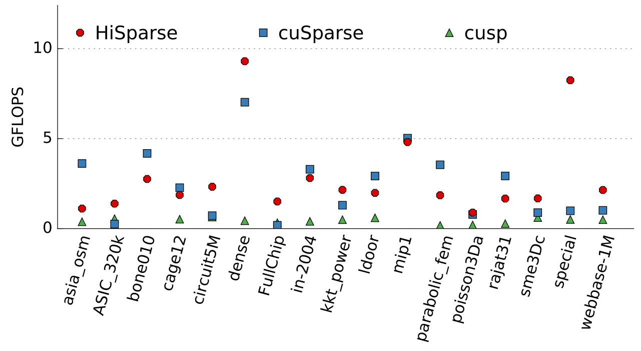
Addition. Figure 6 provides performance comparisons for sparse matrix addition ($C = A + B$ and $C = A^\top + B^\top$) and Table 2b shows averaged results, where A is the input matrix and B is generated from A by randomly moving each entry within its local neighbourhood for a normal distribution with zero mean and a standard deviation of 10, generating a different, but similar non-zero pattern. Transposed addition for cuSparse and cusp is done by converting the input matrix from CSR to CSC, since this is the only / most efficient solution. Still, this conversion adds a significant overhead. In practice, it may amortize over multiple additions if the matrix is used more than once. Thus, Figure 6c ignores transposition cost.



(a) Sparse matrix addition ($C = A + B$) using double precision.



(b) Transposed matrix addition ($C = A^\top + B^\top$) using double.



(c) Transposed matrix addition ($C = A^\top + B^\top$) comparison using double precision without conversion costs.

Figure 6: Sparse Matrix addition performance comparisons in GFLOPS (higher is better). A is the given matrix, B is generated from A by randomly moving each entry from A within its local neighbourhood using a normal distribution with $\mu = 0$ and $\sigma = 10$.

	non-transpose		transpose		transpose w/o conv.	
Library	mean	std-dev	mean	std-dev	mean	std-dev
HiSparse	12.79	12.99	26.42	28.23	26.43	28.24
cuSparse	52.08	99.65	165.35	175.95	55.43	100.78
cusp	86.08	86.06	144.29	140.72	79.54	78.30

(a) single precision

	non-transpose		transpose		transpose w/o conv.	
Library	mean	std-dev	mean	std-dev	mean	std-dev
HiSparse	14.94	15.06	27.91	29.20	27.91	29.20
cuSparse	59.73	112.35	195.50	206.95	61.64	112.43
cusp	119.08	101.63	208.03	174.96	126.07	100.07

(b) double precision

Table 2: Average and standard deviation of the execution time (ms) for sparse matrix addition.

As the results show, HiSparse achieves the best behavior for non-transposed matrices, completing the addition in the shortest time in all test cases. On average HiSparse is approximately 4× faster than cuSparse and almost 8× faster than cusp for double precision sparse non-transposed addition. Again, a large portion of the performance gain is due to cuSparse having issues with *Fullchip* and *circiut5M*. In the transpose case, when including the cost for transposition, the performance difference becomes even more pronounced, with HiSparse being approximately 7× faster than cuSparse. However, HiSparse also shows a reduction in performance for transposed matrices compared to non-transposed addition, as nodes cannot simply be merged anymore and require sorting. Nevertheless, the performance for the transpose case can be extrapolated from the the non-transpose case. Even when the cost for transposition is ignored and cuSparse and cusp only need to merge rows, while HiSparse still traverses two tree structures in an orthogonal manner, HiSparse is on average about 2× faster than cuSparse and 4.5× faster than cusp. In all cases, HiSparse has the lowest standard deviation.

Discussion. By shifting the burden to scheduling, the use of hierarchical formats is not much more involved than that of other existing formats. For instance, the core of our SpMV implementation covering transpose handling and 4 node types spans about 600 lines of code, which is on par with BhSparse and yaSpMV, though they do not address the transpose case. Clearly, the effort involved in our method increases with the number of node types but the code for different nodes remains similar and of local scope. By performing transposition locally at the node level, our approach is memory efficient and can accommodate a wealth of algorithms where the transpose is not explicitly required by a simple coordinate switch.

8 CONCLUSION

In this paper, we introduced HiSparse, a hierarchical format for sparse matrices on the GPU. In terms of storage requirements, HiSparse is more economic than the common CSR and COO formats. In terms of performance, HiSparse proved to be a efficient when endowed with the right dynamic scheduling capabilities. We evaluated HiSparse by comparing our SpMV and sparse addition implementations to various other libraries. As HiSparse does not favor rows over columns, a steady performance for non-transposed and transposed matrix computations is observed. Consequently, HiSparse SpMV achieved the best performance for the transpose case, while still being highly competitive for the direct case. We showed that our format achieves the best overall performance for both direct and transpose sparse addition.

We foresee that the strength of a hierarchical format with dynamic scheduling becomes more apparent when scaling to multi GPU and multi node cluster setups, where load balancing across different nodes is of utter importance. Also, when sequences of complex operations are involved, the hierarchical format may be more efficient, as sequences of operations on nodes can be grouped and different matrices can share sub trees. Our matrix addition results lead us the believe that HiSparse could also be a good candidate for implementing SpGEMM which also requires a dual traversal of two matrices and a dynamic construction of the output matrix. Moreover, thanks to our dynamic scheduler the effort needed for incorporating new operations reduces to operations for each node type while the scheduler takes care of scheduling them efficiently.

Acknowledgments

This research was supported by the Max Planck Center for Visual Computing and Communication, DFG grant STE 2565/1-1 and the Austrian Science Fund (FWF) I 3007.

REFERENCES

[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 *Proc. SPAA '09*. ACM, 2009, pp. 233–244.

[2] 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.

[3] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *Proc. HiPEAC'10*. Springer-Verlag, 2010, pp. 111–125.

[4] 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.

[5] B.-Y. Su and K. Keutzer, "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *Proc. ICS '12*. ACM, 2012, pp. 353–364.

[6] 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.

[7] W. Liu and B. Vinter, "CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proc. ICS '15*. ACM, 2015, pp. 339–350.

[8] E. Schrem, "Computer implementation of the finite-element procedure," in *Numerical and Computer Methods in Structural Mechanics*. Academic Press, 1973, pp. 79 – 121.

[9] P. Stathis, S. Vassiliadis, and S. Cotofana, "A hierarchical sparse matrix storage format for vector processors," in *Proc. IPDPS '03*, April 2003, p. 8 pp.

[10] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. SC '09*. ACM, 2009, pp. 1–11.

[11] 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.

[12] 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.

[13] 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.

[14] D. Langr, I. imecek, and P. Tvrdk, "Storing sparse matrices to files in the adaptive-blocking hierarchical storage format," in *FedCSIS 2013*, Sept 2013, pp. 479–486.

[15] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs," in *Proc. ICS '14*. ACM, 2014, pp. 273–282.

[16] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: Yet another spmv framework on gpus," in *Proc. PPoPP '14*. ACM, 2014, pp. 107–118.

[17] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proc. ICS '13*. ACM, 2013, pp. 273–282.

[18] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[19] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, "On the usage of 16 bit indices in recursively stored sparse matrices," in *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Sept 2010, pp. 57–64.

[20] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proc. SC '14*. IEEE Press, 2014, pp. 769–780.

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

[22] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs for graph applications," in *Proc. SC '14*. IEEE Press, 2014, pp. 781–792.

[23] Y. Liu and B. Schmidt, "LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs," in *ASAP 2015*, July 2015, pp. 82–89.

[24] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proc. SC '16*. IEEE Press, 2016, pp. 58:1–58:12.

[25] M. Steinberger, R. Zayer, and H.-P. Seidel, "Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU," in *Proc. ICS '17*. ACM, 2017.

[26] 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.

[27] J. Kepner and J. Gilbert, Eds., *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[28] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2016.

[29] 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.

[30] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proc. HPG '09*. ACM, 2009, pp. 145–149.

[31] S. Laine, T. Karras, and T. Aila, "Megakernels considered harmful: Wavefront path tracing on GPUs," in *Proc. HPG '13*. ACM, 2013, pp. 137–143.

[32] 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.

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

[34] 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/

[35] M. Steinberger, A. Derler, R. Zayer, and H. P. Seidel, "How naive is naive SpMV on the GPU?" in *Proc. IEEE HPEC*, Sept 2016, pp. 1–8.

[36] Nvidia, "NVIDIA geforce gtx 1080 whitepaper," 2016.

A ASYMPTOTIC STORAGE ANALYSIS

The upper bound for the maximum number of nodes required for a HiSparse matirx (equation 1, Section 3.3) can be reformulated as

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

and simplified as follows

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