Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs

Hartwig Anzt, Stanimire Tomov, Jack Dongarra
Innovative Computing Lab,
University of Tennessee, Knoxville, USA.
Emails: hanzt@icl.utk.edu, tomov@cs.utk.edu, dongarra@eecs.utk.edu

***Abstract*—Numerical methods in sparse linear algebra typically rely on a fast and efficient matrix vector product, as this usually is the backbone of iterative algorithms for solving eigenvalue problems or linear systems. Against the background of a large diversity in the characteristics of high performance computer architectures, it is a challenge to derive a cross-platform efficient storage format along with fast matrix vector kernels. Recently, attention focused on the SELL-C-σ format, a sliced ELLPACK format enhanced by row-sorting to reduce the fill in when padding rows with zeros. In this paper we propose an additional modification resulting in the padded sliced ELLPACK (SELL-P) format, for which we develop a sparse matrix vector CUDA kernel that is able to efficiently exploit the computing power of NVIDIA GPUs. We show that the kernel we developed outperforms straight-forward implementations for the widespread CSR and ELLPACK formats, and is highly competitive to the implementations in the highly optimized CUSPARSE library.**

I. Introduction

In the interest of solving sparse linear systems or eigenvalue problems via iterative methods, there exists a significant need for fast and efficient sparse matrix vector multiplications (SpMVs), as these often dominate the computation. The challenge of implementing fast SpMV kernels is directly connected to the question of the data layout used to store the sparse matrix in memory. With modern hardware platforms often accelerated by GPUs or Intel MIC coprocessors, the storage problem becomes even more challenging, as a certain layout, suitable for one processor type, may fail to allow for efficient matrix vector multiplication on a different architecture. In particular, the overhead-reduced slim formats like CSR (compressed sparse row [3]) that are suitable for cache-aware CPU-dominated platforms, usually render poor performance on streaming processors like GPUs that demand instruction parallelism and coalesced memory access. In the past, different authors proposed data layouts specifically designed for many-core architectures that achieve higher performance, see [4], [13], [17], and significant effort was spent on accelerating sparse matrix vector kernels, see [5], [9], [10], [11], [16]. However, these GPU-optimized formats often fail to allow for efficient sparse matrix vector products on CPU-centric platforms. More recently, Kreutzer et. al. proposed a unified sparse matrix data layout for modern processors with wide SIMD units [12], which allows optimization for a certain hardware architecture via parameters. In the proposed SELL-C-σ format, the block size makes the connection between the slim CSR format and the GPU-friendly ELLPACK [4] format by trading off zero-padding of rows and parallelism against cache usage. Furthermore, they introduce row-sorting to minimize the storage overhead for any parameter choice. They argue that this data layout is suitable for cross-platform usage and show, in runtime experiments, that the format is competitive to the formats optimized for the respective platforms. In this paper we want to support the generality of the format proposed in [12] by showing how minor modifications to the SELL-C/SELL-C-σ format resulting in the padded sliced ELLPACK (SELL-P) format, allow for the efficient implementation of a sparse matrix vector product on GPUs, which is able to achieve outstanding performance. For this purpose, we structure the paper as follows: First we revise some of the most commonly used data layouts when computing sparse matrix vector products on different architectures, particularly on GPUs. We then summarize some kernel characteristics that are key to achieving high performance on GPUs and provide a detailed description of how we implemented a sparse matrix vector kernel for the modified SELL-P format using CUDA for NVIDIA GPUs. We profile the kernel using NVIDIA's NVPROF analysis tool and compare for a set of test matrices taken from the University of Florida matrix collection against sparse matrix vector kernels based on other formats, as well as the highly optimized in-house developed routines provided by NVIDIA. We conclude by summarizing the results and providing ideas for future research.

II. Sparse Matrix Storage Formats

While for dense matrices it is usually reasonable to store all matrix entries in consecutive order, sparse matrices are characterized by a large number of zero elements, and storing those is not only unnecessary but would also result in significant storage overhead. Different storage layouts exist, that aim for reducing the memory footprint of the sparse matrix by storing only a fraction of the elements explicitly, and anticipating all other elements to be zero, see [3], [18], [6][1]. In the CSR format [3], this idea is taken to extremes, as only nonzero entries of the matrix are stored. In addition to the array **values** containing the nonzero elements, two integer arrays **colind** and **rowptr** are used to locate the elements in the matrix, see

[1]We note that numerous ideas also exist on how to benefit from a certain matrix characteristic like symmetry, tridiagonality, or a special sparsity pattern. In this paper we address unstructured matrices not allowing for any storage reduction due to special matrix properties.

Figure 1. While this storage format is suitable when computing a sparse matrix vector product on processors with a deep cache-hierarchy, as it reduces the memory requirements to a minimum, it fails to allow for high parallelism and coalesced memory access when computing on streaming-processors like GPUs. On those, the ELLPACK-format padding the different rows with zeros for a uniform row-length, coalesced memory access, and instruction parallelism may, depending on the matrix characteristics, outperform the CSR format [4]. This approach removes the need for the **rowptr**, as every row now contains the same number of elements in memory. The storage overhead of ELLPACK is determined by the "longest row," which is the maximum number of nonzero elements in one row of the matrix, see Figure 1. While this is usually compensated for by the more efficient hardware usage when targeting streaming processors, it is not suitable for cache-aware architectures as the explicitly stored zero elements all have to be processed. One idea to reduce the storage overhead is the introduction of the sliced ELLPACK format, splitting the original matrix into blocks of rows, where each slice is then stored using ELLPACK format, see Figure 1. The resulting format is usually abbreviated as SELL or SELL-C, where C denotes the blocksize [13], [12]. As the number of explicitly stored elements in each row is then no longer determined by the maximum of nonzero elements in one row of the matrix but by the "longest row" in this block of rows, some of the slices may have less storage overhead compared to the ELLPACK format. The blocksize becomes the parameter controlling the fill-in, i.e., using a blocksize of 1 (SELL-1), would result in the CSR format, using a blocksize of the matrix dimension n (SELL-n), would result in the basic ELLPACK format. However, an additional integer array pointing to the start of each slice is needed. In [7] the idea of a blocking ELLPACK is applied to block matrices where structured matrices are stored by using a grid of small ELLPACK blocks of auto-tuned size. The idea of the sliced ELLPACK format is enhanced in [12] by adding row sorting such that rows with similar number of nonzero elements are gathered in one block. While this obviously reduces the fill-in furthermore, the authors mention it is important to trade-off the cost of sorting against the acceleration of the sparse matrix vector product. They also propose a trimmed sliced ELLPACK format that arises as hybrid combination of SELL-C-σ and CSR, but refrain from showing performance results for this. In the remainder of their paper they argue that the SELL-C-σ format is suitable for cross-platform usage and show, in benchmark experiments, its capability to compete with other storage layouts on different architectures.

Another approach to reducing not the storage overhead but the computational overhead involves storing the number of nonzero elements in each row in an additional array, and computing the partial products only for the nonzero elements. In [17] this idea was proposed in combination with assigning multiple threads to one row of the matrix, and benchmark results showed the superiority of this idea over the plain ELLPACK format on GPUs.

In the following sections we show how to integrate the idea of assigning multiple threads to one row into an optimized hardware-aware implementation of the sparse matrix vector product for the SELL-C/SELL-C-σ matrix format. For this purpose we modify the SELL-C/SELL-C-σ layout to the SELL-P format (P for "padded"), by padding rows with zeros such that the rowlength of each block becomes a multiple of the number of threads assigned to each row, see Figure 1 for the $t = 2$. We refrain from applying any sorting as we consider this a significant change in the matrix characteristics, and it may be difficult to account for the overhead due to the additional pre-/postprocessing step in a complex algorithm. However, the benchmark results for the SpMV kernel using the non-sorted SELL-P format underpin the cross-platform suitability of the SELL-C/SELL-C-σ storage format for unstructured sparse matrices, and the efficiency of the SELL-P matrix vector product implementation on GPUs.

III. SpMV kernel for the SELL-P format

Numerous implementation aspects exist, that can influence the efficiency of a SpMV CUDA kernel on NVIDIA GPUs. Among the most important ones are:

- Memory accesses;
- Thread- and instruction-level parallelism;
- Shared memory usage.

To take them into account, our implementation has parametrized the various choices that can influence performance to ease the process of tuning.

The sparse matrix vector product kernel we propose for the SELL-C/SELL-C-σ storage format assigns t threads to each row in one slice with blocksize b. For this purpose it is necessary to convert the SELL-C/SELL-C-σ format into SELL-P by zero-padding the rows to a length divisible by t. While the number of rows in one slice as well as t may be adapted to the respective matrix, we want to ensure that the total number of threads assigned to one block (i.e., $t \cdot b$) is suitable for the GPU architecture used, and enables the usage of shared memory[2]. To allow coalescent memory access, we arrange the threads in a $b \times t$ 2D thread grid, see Figure 3. For each slice, the kernel computes the number, $max\_$, of necessary multiply-add that each thread has to compute, and the threads proceed over the data. We noticed that the performance can be improved by unrolling this loop into chunks of two. Once all data is processed, the partial products are written into shared memory, and a fan-in algorithm with an increment of the thread count computes the sum for each row in shared memory. Accounting for the parameters α and β in the SpMV operation $y = \alpha \cdot Ax + \beta y$, the result is written back into the global memory. The grid necessary to launch the threadblocks has to cover the complete matrix, i.e., the number of blocks is equal to s, the number of slices the matrix is blocked into. As this number is typically large, a 2D grid of thread blocks, with both grid dimensions close to $\sqrt{s}$, is suitable for efficient

[2]We will set $t = b = 8$ for the implementation used in the benchmark section.

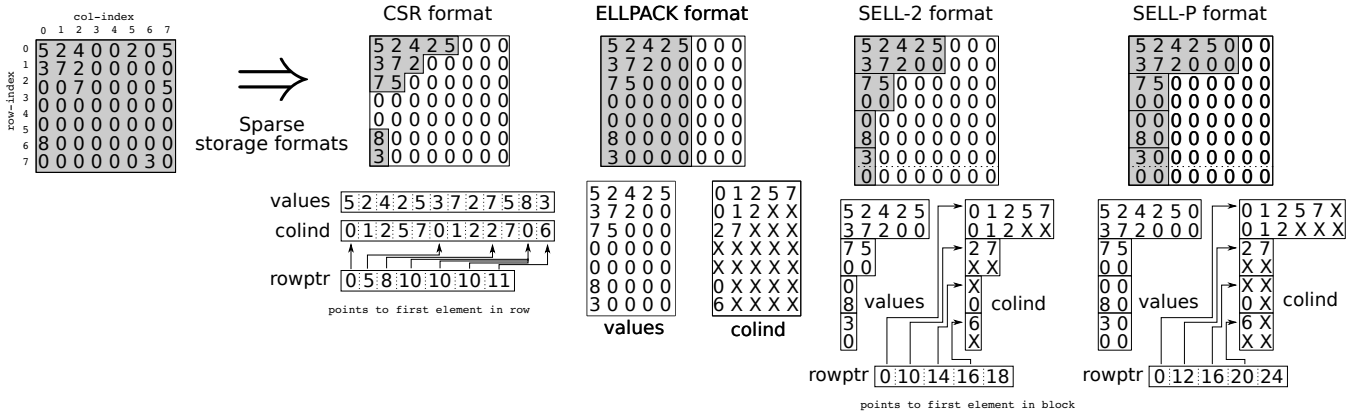


Fig. 1: Dense and sparse matrix storage format representation, partly taken from [2]. The memory demand corresponds to the grey areas. Notice that using SELL-C with the blocksize $b = 2$ (SELL-2), requires adding one row to the original matrix. Furthermore, padding the SELL-P format to a rowlength divisible by 2 ($t = 2$), requires explicit storage of some additional zeros.

processing. While the pseudo-code for the kernel is provided in Figure 2, the underlying data layout, the memory access pattern and the reduction step is visualized in Figure 3.

IV. Performance analysis of the SELL-P SpMV

In this section we want to analyze the performance of the developed matrix vector kernel. In a first step, we use NVIDIA's in-house profiler, which provides information about compute-intensity, memory usage, occupancy, cache misses etc. Second, we compare against several commonly-used sparse matrix-vector kernels taken from open-source libraries. The hardware we use for our benchmark experiments is a two socket Intel Xeon E5-2670 (Sandy Bridge) platform accelerated by an NVIDIA Tesla K40c GPU with a theoretical peak performance of 1,682 GFlop/s. The host system has a theoretical peak of 333 GFlop/s, main memory size is 64 GB, and theoretical bandwidth is up to 51 GB/s. On the K40 GPU, 12 GB of main memory are accessed at a theoretical bandwidth of 288 GB/s. The GPU implementation of the SELL-P SpMV, and the straight forward CSR and ELLPACK kernels, is realized in CUDA [15], version 5.5 [8], where the reference CUSPARSE-CSR and CUSPARSE-HYB implementations are also taken from [14]. On the CPU, the CSR SpMV is taken from Intel's MKL [1] version 11.0, update 5.

The matrices we use for our benchmarks are taken from the University of Florida Matrix Collection (UFMC)[3]. With some key characteristics collected in Table I, and sparsity plots for selected matrices are shown in Figure 4, we tried to cover a large variety of systems common in scientific computing.

A. Performance Analysis using NVIDIA's NVPROF Profiler

NVIDIA provides a useful tool to analyze the execution characteristics of a CUDA kernel [8]. This enables, to determine whether the objectives listed in Section III have been achieved. In Table II we list some of the most important characteristics we collected using the NVPROF kernel execution analysis.

[3]UFMC; see http://www.cise.ufl.edu/research/ sparse/matrices/

matrix	nonzeros (nnz)	Size (n)	nnz/n
AUDIKW_1	77,651,847	943,645	82.28
BONE_010	47,851,783	986,703	48.50
BONE_S10	40,878,708	914,898	44.68
BMW3_2	11,288,630	227,362	49.65
BMWCRA_1	10,641,602	148,770	71.53
CAGE_10	150,645	11,397	13.22
CANT	4,007,383	62,451	64.17
CRANKSEG_2	14,148,858	63,838	221.63
CUBE_COUP_DT0	124,406,070	2,164,760	57.47
DIELFILTERV2REAL	48,538,952	1,157,456	41.94
F1	26,837,113	343,791	78.06
FAULT_639	27,245,944	638,802	42.65
HOOK_1498	59,374,451	1,498,023	39.64
INLINE_1	36,816,170	503,712	73.09
LDOOR	42,493,817	952,203	44.63
M_T1	9,753,570	97,578	99.96
ML_GEER	110,686,677	1,504,002	73.59
PWTK	11,524,432	217,918	52.88
SHIPSEC1	3,568,176	140,874	25.33
TREFETHEN_20000	554,466	20,000	27.72

TABLE I: Description and properties of the test matrices.

B. Performance comparison between different SpMV kernels

Finally, we compare the performance of the developed kernel against other matrix vector product implementations. All kernels use the same hardware platform. While some libraries feature auto-tuning, the parameters of the developed SELL-P matrix vector kernel are set to the default values $t = b = 8$ (remind that $b = 8$ is equivalent to zero-padded SELL-8). In particular, we compare against the following implementations:

- **MKL-CSR.** Intel's Math Kernel Library provides a very efficient CPU-based sparse matrix vector product based

```
void sellp_spmv(     int n, int b, int t, double alpha,
                     int *rowptr, int *colind, double *values,
                     double *x, double beta, double *y ) {

    // t threads assigned to each row
    int idx = threadIdx.x ;     // thread in row
    int idy = threadIdx.y;      // local row
    int ldx = idy * blocksize + idx;  // first element to be accessed
    int bdx = blockIdx.y * gridDim.x + blockIdx.x; // global block index
    int row = bdx * blocksize + idx;  // global row index

    extern __shared__ double shared[];

    if(row < n ){
        double dot = 0.0;
        int offset = rowptr[ bdx ];
        int block = blocksize * t; // total number of threads

        // number of elements each thread handles
        int max_ = ( rowptr[ bdx+1 ]-offset ) / block;

        // partial product loop unrolled to blocks of two
        int kk, i1, i2;
        double x1, x2, v1, v2;
        d_colind += offset + ldx ;
        d_val += offset + ldx;
        for ( kk = 0; kk < max_-1 ; kk+=2 ){
            i1 = colind[ block*kk ];
            i2 = colind[ block*kk + block ];
            x1 = x[ i1 ];
            x2 = x[ i2 ];
            v1 = values[ block*kk ];
            v2 = values[ block*kk + block ];

            dot += v1 * x1;
            dot += v2 * x2;
        }
        // maybe one additional step
        if (kk<max_){
           x1 = d_x[ d_colind[ block*kk ] ];
           v1 = d_val[ block*kk ];
           dot += v1 * x1;
        }

        // write result to shared memory
        shared[ ldx ]  = dot;

        // reduction
        __syncthreads();
        if( idy < 4 ){
            shared[ ldx ]+=shared[ ldx+blocksize*4 ];
            __syncthreads();
            if( idy < 2 ) shared[ ldx ]+=shared[ ldx+blocksize*2 ];
            __syncthreads();
            if( idy == 0 ) {
                y[ row ] =
                ( shared[ ldx ]+shared[ ldx+blocksize*1 ] ) * alpha
                                                    + beta * y [ row ];
            }

        }

    }
}
```

Fig. 2: SpMV kernel implementation for the SELL-P sparse matrix format for $t = 8$.

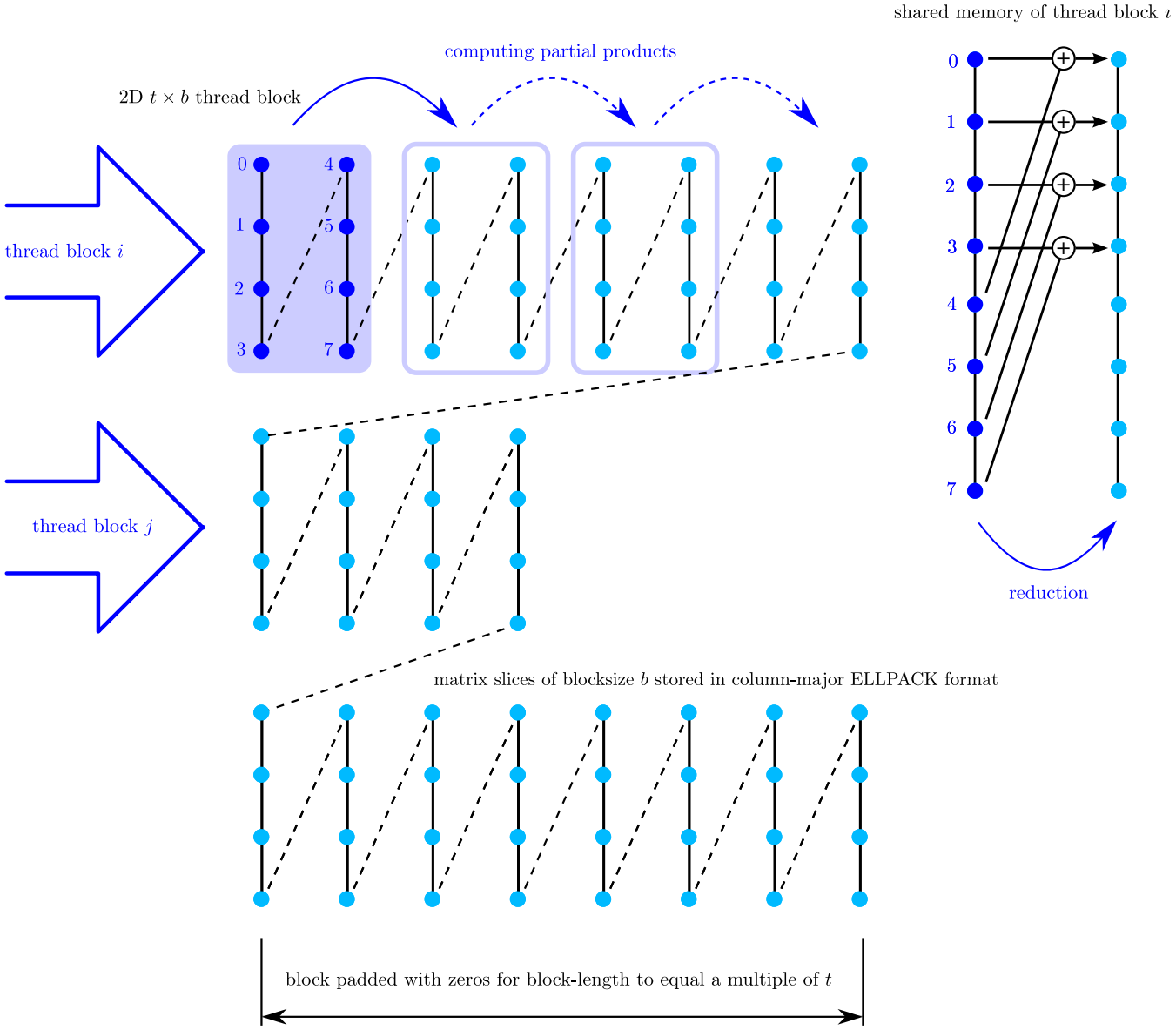
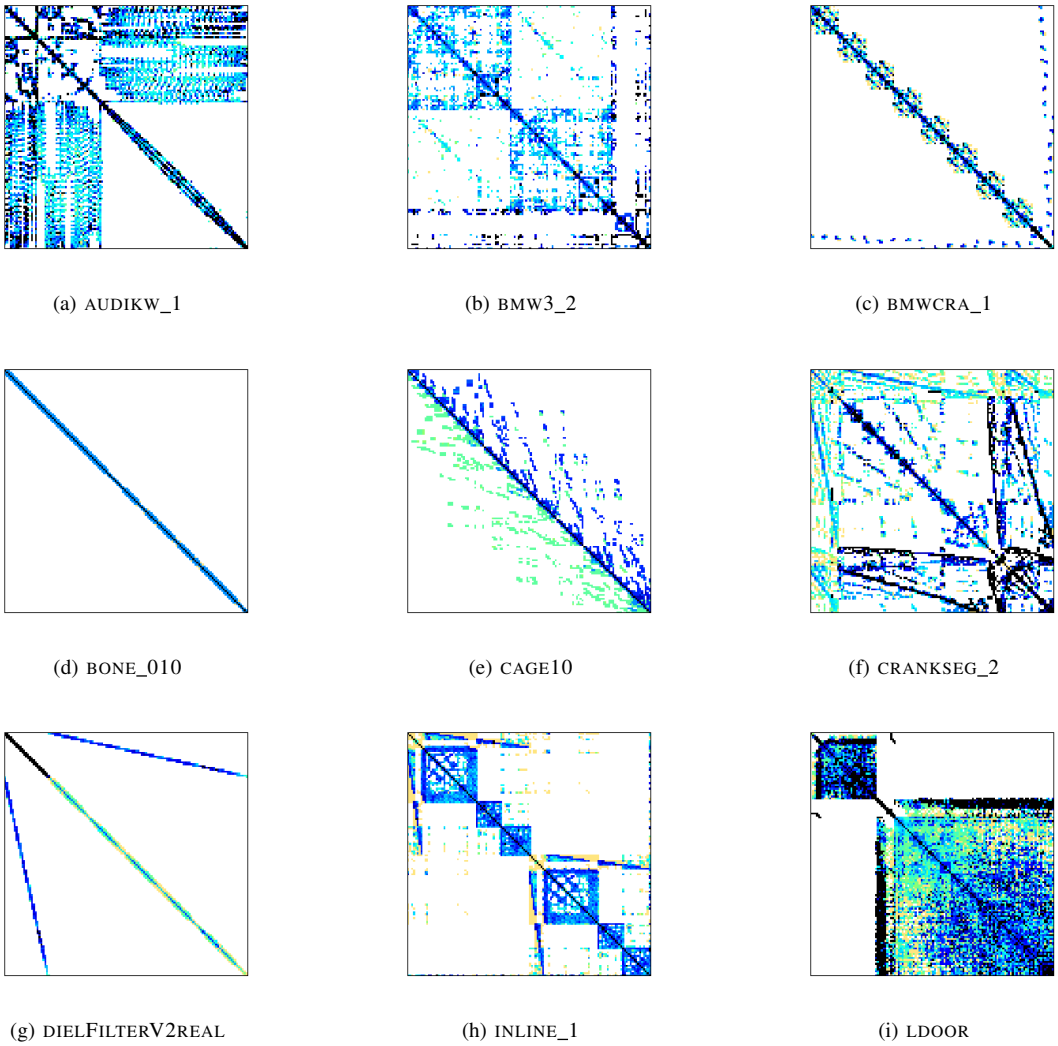


Fig. 3: Visualization of the SELL-P memory layout and the SELL-P SpMV kernel procedure including the reduction step using the blocksize $b = 4$ (corresponding to SELL-2), and $t = 2$.

on the CPU-optimal CSR format. We use the library's version 11.0, update 5.

- **CSR.** For the CSR format we include a straight-forward GPU implementation in the comparison. This reveals that the CSR format usually results in poor performance on GPUs.
- **ELLPACK.** We also compare against a straight-forward SpMV implementation of the ELLPACK format which is, in general, more suitable for GPU computing. The implementation is according to [4].
- **SELL-P.** The SPMV kernel developed in Section III may be used in combination with row-sorting like proposed in [12] and be optimized for a certain matrix test-case. We, however, refrain from applying these optimizations and assume no information about the matrix characteristics. As the default configuration, we set the blocksize and the number of threads assigned to each row to $b = t = 8$.
- **CUSPARSE-CSR.** Within the CUSPARSE library, NVIDIA provides a highly optimized SpMV kernel for CSR format. This kernel assigns multiple threads to each row and uses static as well as dynamic shared memory.
- **CUSPARSE-HYB.** NVIDIA's CUSPARSE library also features a sparse matrix vector kernel for a hybrid format based on the idea of storing the major part of the matrix in ELLPACK format and using CSR for the remaining part.
- **ELLRT.** The open source library SpMVELLRT[4] pro-

[4]SpMVELLRT; see https://sites.google.com/site/mcfastsparse/



(a) AUDIKW_1 (b) BMW3_2 (c) BMWCRA_1

(d) BONE_010 (e) CAGE10 (f) CRANKSEG_2

(g) DIELFILTERV2REAL (h) INLINE_1 (i) LDOOR

Fig. 4: Sparsity plots of selected test matrices.

execution time	7.02 ms
grid size	$343 \times 344 \times 1$
block size	$8 \times 8 \times 1$
registers/thread	40
shared memory	512 kB
device memory throughput (read)	212.59 GB/s
device memory throughput (write)	1.34 GB/s
shared memory throughput (load)	25.78 GB/s
shared memory throughput (store)	17.19 GB/s
L2 throughput (read)	24.36 GB/s
L2 throughput (write)	1.07 GB/s
L2 hit rate	30.34 %
shared memory efficiency	100%
warp execution efficiency	100 %
instructions per warp	250,053
issued instructions per cycle (IPC)	1.30
executed instructions per cycle (IPC)	0.75
achieved occupancy	0.47
multiprocessor activity	99.8 %

TABLE II: Some of the important kernel execution characteristics of the SELL-P SpMV kernel applied to the test matrix AUDIKW_1.

vides an optimized version of a sparse matrix vector kernel based on a modified ELLPACK format. The idea is to store the length of each row in addition to the values and the column-indices, and to assign multiple threads to each row [17]. Although publications exist showing performance superior to the CUSPARSE SpMV kernels, we were unable to generate correct results using the SpmvELLRT library, and therefore refrain from showing results for this library.

The runtime results, averaged over 1000 kernel executions, are listed in Table III with their respective performance visualized in Figure 5.

As expected, the CSR format achieves low FLOP rates when used on GPUs. On the other hand, Intel's MKL is able to exploit the computing power of the 16 cores of Intel's Sandy Bridge processor very efficiently, sometimes even outperforms the ELLPACK kernel on the GPU, see results for BMW3_2, CRANKSEG_2, F1, and INLINE_1. This

matrix	MKL-CSR	CSR	ELLPACK	SELL-P	CUSPARSE-CSR	CUSPARSE-HYB
AUDIKW_1	4.55e-02	8.94e-02	3.56e-02	7.02e-03	7.08e-03	8.79e-03
BONE_010	2.45e-02	5.10e-02	5.50e-03	4.28e-03	6.15e-03	3.80e-03
BONE_S10	1.32e-02	4.16e-02	5.45e-03	3.93e-03	5.21e-03	3.43e-03
BMW3_2	4.70e-03	1.19e-02	6.62e-03	9.93e-04	1.50e-03	9.86e-04
BMWCRA_1	4.60e-03	1.19e-02	4.63e-03	9.01e-04	9.53e-04	8.79e-04
CAGE_10	4.72e-05	7.45e-05	3.72e-05	3.35e-05	3.45e-05	6.44e-05
CANT	1.48e-03	4.61e-03	4.56e-04	3.44e-04	5.51e-04	3.94e-04
CRANKSEG_2	5.25e-03	1.60e-02	2.23e-02	1.12e-03	1.23e-03	1.48e-03
CUBE_COUP_DT0	4.87e-02	1.39e-01	1.22e-02	9.77e-03	1.68e-02	9.18e-03
DIELFILTERV2REAL	2.06e-02	4.45e-02	1.05e-02	6.14e-03	6.46e-03	5.99e-03
F1	1.31e-02	3.08e-02	1.63e-02	2.73e-03	2.77e-03	3.18e-03
FAULT_639	1.08e-02	2.92e-02	1.51e-02	2.53e-03	3.48e-03	2.12e-03
HOOK_1498	2.41e-02	5.71e-02	1.25e-02	6.60e-03	7.52e-03	6.46e-03
INLINE_1	1.41e-02	4.15e-02	3.01e-02	3.49e-03	3.55e-03	3.85e-03
LDOOR	1.78e-02	4.43e-02	6.75e-03	4.11e-03	5.71e-03	4.39e-03
M_T1	3.67e-03	1.03e-02	2.10e-03	8.27e-04	8.64e-04	1.02e-03
ML_GEER	4.23e-02	1.29e-01	9.62e-03	8.60e-03	9.62e-03	8.19e-03
PWTK	4.47e-03	1.30e-02	3.37e-03	9.31e-04	1.53e-03	8.94e-04
SHIPSEC1	1.15e-03	2.49e-03	8.74e-04	4.74e-04	4.52e-04	4.48e-04
TREFETHEN_20000	1.15e-04	5.12e-04	5.29e-05	6.47e-05	8.12e-05	6.71e-05

TABLE III: Runtime in seconds of the different matrix vector kernels.

happens, in particular, for unstructured matrices where large variances in the number of nonzeros in the distinct rows results in significant zero fill-in when using ELLPACK (compare sparsity plots in Figure 4). For other matrices, the computing power of the GPU compensates for the additional zero entries, and the ELLPACK matrix vector routine is superior to Intel's MKL. As the SELL-C/SELL-C-σ formats, which the SELL-P is based on, control the fill-in via the blocksize, and, in the case of SELL-C-σ, via row sorting, the overhead due to padding rows with zeros is reduced. Although the SELL-P introduces some zeros for the rowlength being divisible by the thread number t (in our experiments we chose $t = 8$), the SELL-P kernel on the GPU is superior to the ELLPACK kernel for all test cases except for the very small and very sparse TREFETHEN_20000. This test matrix is, like CAGE10, very small and sparse, and although we included it for completeness, we should not put too much emphasis on the respective results. For the other cases, switching from the ELLPACK to the SELL-P kernel results in speedup factors between 1.1 and 9. Compared to the CSR implementations, the SELL-P is superior to the MKL-CSR using the CPU, and outperforms the straight-forward CSR on GPU by almost an order of magnitude. As we refrained from applying any sorting of the rows and any matrix-specific optimization of the kernel configuration, this already shows the suitability of the SELL-P storage format for GPU computing. The obviously more challenging part is to match the performance of the in-house developed sparse matrix vector kernels from NVIDIA, as these are not only highly tuned, but also apply some dynamic adaptation to the specific test matrix. We observe that the SELL-P SpMV is able to match the CUSPARSE-CSR and CUSPARSE-HYB performance for all test cases, in particular, the performance of SELL-P is often inferior to one, and superior to the other CUSPARSE routine. For about 40% of the test cases, SELL-P even outperforms both CUSPARSE routines, e.g., AUDIKW_1, CANT, CRANKSEG_2, F1, INLINE_1, LDOOR and M_T1. Only for the very structured matrices like BONE010 or BONES10 (see Figure 4d), where the plain ELLPACK already provides very good performance, the CUSPARSE-HYB is always superior. Being able to keep up with the highly tuned sparse matrix vector products featured by NVIDIA's CUSPARSE library revels the suitability of the SELL-C/SELL-C-σ - based SELL-P storage format for GPU computing and the high efficiency of the developed kernel. While already applying problem-specific optimizations like choosing t and/or b would improve the performance even further, techniques like reordering rows to minimize the fill-in in a preprocessing step have shown to be very useful to reducing the runtime of the SpMV [12]. We refrain from applying these steps not only because we aim for a generic matrix vector kernel, but also because it might be difficult to account for the overhead of an optimization phase in a complex algorithm.

V. SUMMARY AND FUTURE WORK

We have accepted the challenge of deriving an efficient implementation of a sparse matrix vector product for the SELL-C/SELL-C-σ format on NVIDIA GPUs. For this purpose we modified the input format to the padded sliced ELLPACK format (SELL-P) that allows to assign multiple threads to each matrix row. Following a detailed description, we have shown in runtime experiments that the developed kernel is highly competitive to the CUSPARSE library and outperforms straight-forward implementations of matrix vector kernels for

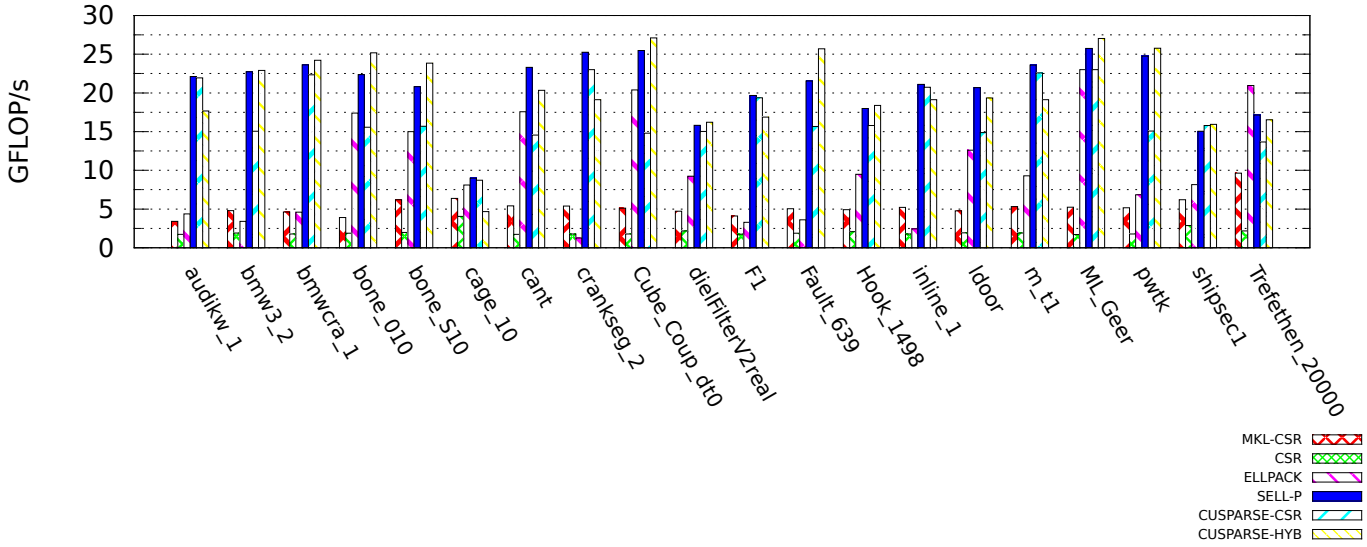


Fig. 5: Performance of the different sparse matrix vector kernels using either the CPU (MKL-CSR) or the GPU.

other formats. While this underpins the suitability of the SELL-C/SELL-C-σ storage format for GPU architectures, future research should focus on how to integrate efficient row-sorting into the SELL-P SpMV kernel. Furthermore, using multiple GPUs poses the question of how to distribute the matrix to multiple accelerators for a balanced workload.

ACKNOWLEDGMENTS

This research was supported in part by DOE grant #DE-SC0010042, NVIDIA, and the National Science Foundation.

REFERENCES

[1] Intel® Math Kernel Library for Linux* OS. Document Number: 314774-005US, October 2007. Intel Corporation.

[2] J. Aliaga, H. Anzt, M. Castillo, J. Fernández, G. León, J. Pérez, and Qu. Performance and energy analysis of the iterative solution of sparse linear systems on multicore and manycore architectures. In *Lecture Notes in Computer Science, 10th Int. Conf. on Parallel Processing and Applied Mathematics – PPAM 2013*, (accepted).

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[4] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA, December 2008.

[5] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[6] Aydın Buluç, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, pages 721–733, 2011.

[7] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[8] NVIDIA Corp. *NVIDIA CUDA TOOLKIT V5.5*, July 2013.

[9] Eduardo F. D'Azevedo, Mark R. Fahey, and Richard T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *Proceedings of the 5th International Conference on Computational Science - Volume Part I*, ICCS'05, pages 99–106, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] Michael Garland. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 2–6, New York, NY, USA, 2008. ACM.

[11] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.

[12] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. A unified sparse matrix data format for modern processors with wide simd units. *CoRR*, abs/1307.6209, 2013.

[13] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 111–125, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] NV. *CUSPARSE LIBRARY*, July 2013.

[15] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 edition, August 2009.

[16] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[17] F. Vaázquez, G. Ortega, J.J. Fernández, and E.M. Garzón. Improving the performance of the sparse matrix vector product with gpus. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1146–1151, June 2010.

[18] Sam Williams, Nathan Bell, Jee Choi, Michael Garland, Leonid Oliker, and Richard Vuduc. Sparse matrix vector multiplication on multicore and accelerator systems. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore Processors and Accelerators*. CRC Press, 2010.