Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation

Moritz Kreutzer*, Georg Hager*, Gerhard Wellein*, Holger Fehske†, Achim Basermann‡ and Alan R. Bishop§
*Erlangen Regional Computing Center, Erlangen, Germany
†Ernst-Moritz-Arndt University of Greifswald, Greifswald, Germany
‡German Aerospace Center (DLR), Simulation and Software Technology, Cologne, Germany
§Theory, Simulation and Computation Directorate, Los Alamos National Laboratory, Los Alamos, NM, USA

***Abstract*—Sparse matrix-vector multiplication (spMVM) is the dominant operation in many sparse solvers. We investigate performance properties of spMVM with matrices of various sparsity patterns on the nVidia "Fermi" class of GPGPUs. A new "padded jagged diagonals storage" (pJDS) format is proposed which may substantially reduce the memory overhead intrinsic to the widespread ELLPACK-R scheme while making no assumptions about the matrix structure. In our test scenarios the pJDS format cuts the overall spMVM memory footprint on the GPGPU by up to 70%, and achieves 91% to 130% of the ELLPACK-R performance. Using a suitable performance model we identify performance bottlenecks on the node level that invalidate some types of matrix structures for efficient multi-GPGPU parallelization. For appropriate sparsity patterns we extend previous work on distributed-memory parallel spMVM to demonstrate a scalable hybrid MPI-GPGPU code, achieving efficient overlap of communication and computation.**

***Keywords*-GPGPU; Sparse matrices; CUDA**

I. Introduction and related work

A. Sparse matrix-vector multiplication on GPGPUs

The solution of large eigenvalue problems or extremely sparse systems of linear equations is a central part of many numerical algorithms in science and engineering. Sparse matrix-vector multiplication (spMVM) is often the dominating component in such solvers, and may easily consume most of the total runtime. General-purpose computation on graphics processing units (GPGPUs) is an attractive option for this operation due to the large memory bandwidth available to high-end graphics chips and their inherent massive parallelism. Implementations of spMVM on GPGPUs have been a field of active research in recent years [1, 2, 3], and several storage formats have been proposed. Out of those, the ELLPACK-R format [3] has gained widespread acceptance. However, although there is a long history of distributed-memory parallel spMVM codes (see [4] and references therein), there is to our knowledge no efficiency or feasibility analysis of multi-GPU spMVM.

This work has two goals: It provides an alternative sparse MVM storage format that has a significantly smaller memory footprint than ELLPACK(-R) but provides better performance in most cases on modern nVidia GPGPUs. Furthermore, it extends previous work on distributed-memory spMVM for general matrices to multiple GPGPUs.

B. Testbed

The nVidia "Fermi" class of GPGPU-based accelerators (Tesla C/M20X0) used for the benchmarks implement the "GF100" architecture and comprise 14 *streaming multiprocessors* (MPs), each with 32 in-order arithmetic logic units (ALUs). One ALU can execute one single-precision (SP) multiplication and one addition per cycle, which leads to an overall peak performance of 896 flops per cycle on the whole chip at clock frequencies above 1 GHz. At double precision (DP) the theoretical peak performance is halved. The boards are currently available with device memory sizes of 3 (C2050) or 6 GB (C2070), and feature deactivatable ECC protection. In streaming benchmarks the device memory delivers about 91 GB/s sustained with ECC enabled (120 GB/s w/o ECC) [5]. All MPs share a 768 kB L2 cache, whose detailed specifications are undisclosed.

The ALUs in an MP are driven in a single instruction multiple data (SIMD) manner (also termed SIMT model, where "T" stands for "threads"). All threads running on an MP are controlled by a simple instruction scheduler that can switch quickly between chunks of threads called *warps*, in order to hide latencies. A warp (or a subset of it) is the actual SIMD unit on this device, and it is essential that consecutive threads in a warp access consecutive memory locations (this is called *coalescing*). Although still important, coalescing constraints have been somewhat relaxed with the GF100 architecture due to its L2 cache, which was not present on earlier models.

The parallel runs have been conducted on the *Dirac*[1] GPGPU cluster at the National Energy Research Scientific Computing Center (NERSC) in Berkeley. This cluster features 50 GPU nodes, of which 44 contain one nVidia Tesla C2050 card with 3 GB of device memory.

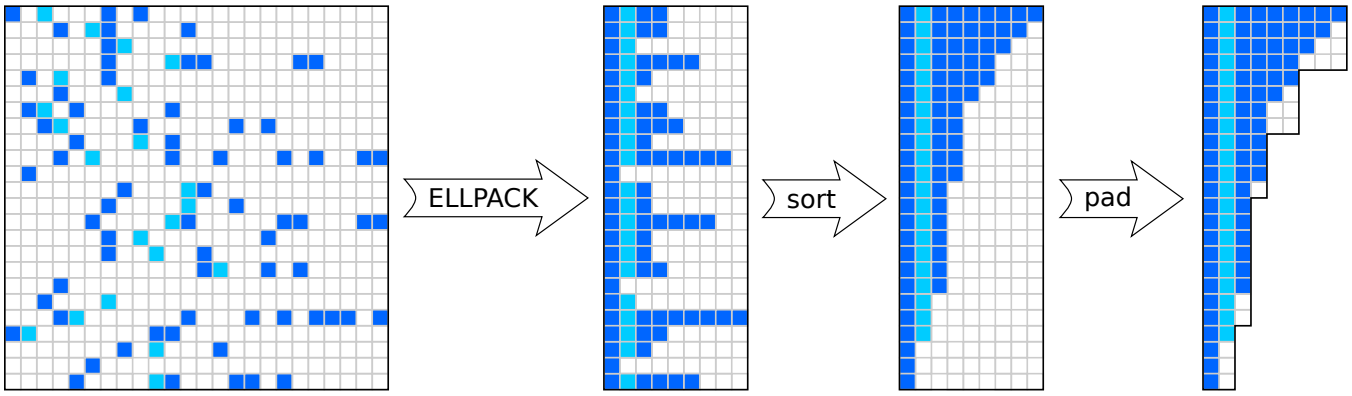


Figure 1: Derivation of the pJDS format from a sparse matrix. In the pJDS format a blocking size of $b_r = 4$ is used.

C. Test matrices

HMEp: This matrix originates from the quantum-mechanical description (using the so-called Holstein-Hubbard model) of a one-dimensional solid with six lattice sites, populated with six electrons coupled to 15 phonons (quantized lattice vibrations). The resulting matrix of dimension 6.2×10^6 is very sparse, with approximately 15 non-zeros per row. It also contains contiguous off-diagonals of length 15,000.

sAMG: This matrix was generated by the adaptive multigrid code sAMG (see [6, 7], and references therein) for the irregular discretization of a Poisson problem on a car geometry. Its matrix dimension is 3.4×10^6 with an average of $N_{\text{nzr}} \approx 7$ entries per row.

DLR1: This matrix comes from an adjoint problem computation (turbulent transonic flow over a wing) with the TAU CFD system of the German Aerospace Center (DLR). TAU performs complex flow simulations on unstructured hybrid grids. The associated grid had 46,417 points (6 unknowns in each point), and the resulting matrix is nonsymmetric with a dimension of 2.8×10^5 and an average of $N_{\text{nzr}} \approx 144$ entries per row.

DLR2: This matrix stems from a linear problem for an aerodynamic gradients calculation. A transonic inviscid flow over a wing was simulated with TAU. The associated grid had 108,396 points, and the matrix is nonsymmetric with a dimension of 5.4×10^5 and an average of $N_{\text{nzr}} \approx 315$ entries per row. It consists entirely of dense 5×5 subblocks.

UHBR: The last matrix originates from aeroelastic stability investigations of an ultra-high bypass ratio (UHBR) turbine fan with a linearized Navier-Stokes solver [9]. This solver is part of the parallel simulation system TRACE (Turbo-Machinery Research Aerodynamic Computational Environment) which was developed by DLR's Institute for Propulsion Technology. Its matrix dimension is 4.5×10^6 with an average of $N_{\text{nzr}} \approx 123$ entries per row.

[1] http://www.nersc.gov/users/computational-systems/dirac/

II. GPGPU SPMVM

A. Introducing the padded JDS formats

ELLPACK-R [3] is a variant of the original ELLPACK storage format [1, 10] and sets today the standard for implementing spMVM operations on GPGPUs. ELLPACK(-R) should be used if no regular substructures such as off-diagonals or dense blocks can be exploited. The idea is to compress the rows by shifting all non-zero entries to the left (first step in Fig. 1) and storing the resulting $N \times N_{\text{nzr}}^{\max}$ rectangular matrix[2] column-by-column consecutively in main memory, where $N_{\text{nzr}}^{\max}$ is the maximum number of non-zeros per row. Thus, in contrast to CPU storage formats, ELLPACK contains zero entries (white boxes in Fig. 1). Thread parallelization of the spMVM is row-wise by assigning consecutive rows to the threads of a block (i.e., outer loop iterations in Listing 1 are mapped to threads in a round-robin way).

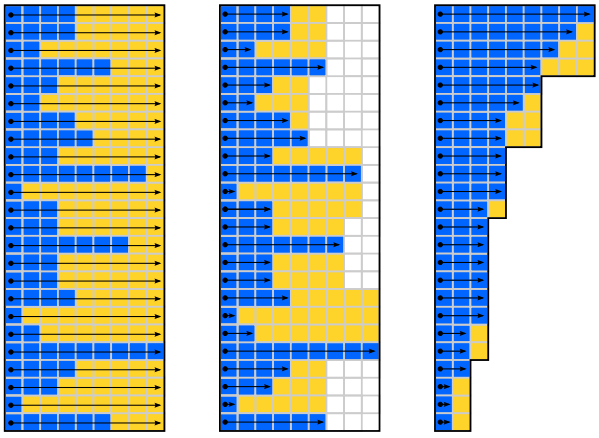
Listing 1: The standard ELLPACK-R spMVM kernel

```
1 for(i=0; i < N; ++i)
2     for(j=0; j < rowmax[i]; ++j)
3         c[i] += val[j*N + i] *
4                 rhs[col_idx[j*N + i]];
```

The increased memory footprint of the ELLPACK format ensures load coalescing within thread warps for access to the matrix entries (`val[]`) and the index array (`col_idx[]`), which points to the right hand side (RHS) vector elements (`rhs[]`). While data alignment became of minor importance with the latest nVidia GPGPU generations, load coalescing is still a must for attaining reasonable data transfer rates. In the original ELLPACK scheme the threads were still loading and operating on the zero matrix entries, wasting memory bandwidth and compute resources.

The ELLPACK-R scheme uses the same storage format, but threads only execute non-zero contributions (the number

[2] Typically the matrix dimension N must also be padded to a multiple of the warp size.



(a) ELLPACK (b) ELLPACK-R (c) pJDS

Figure 2: Scheduling patterns and required storage size for the different matrix formats assuming a four thread warp. Arrows indicate computation and data accesses executed by the threads. White boxes show redundant data storage, and light boxes indicate redundant data storage and useless hardware reservation.

of non-zeros per row is stored in `rowmax[]`), avoiding redundant data transfers. However, all threads of a warp occupy on-chip resources until the thread executing the longest row has finished. Figures 2a and b compare the overhead of the ELLPACK(-R) schemes assuming a warp size of four threads. ELLPACK-R reduces computation and data accesses to the possible minimum (arrows in Fig. 2b). However, the imbalanced row lengths impose reservation of unused hardware units (light boxes). Moreover the redundant storage (indicated by white and light boxes) stays the same.

A simple idea, derived from the Jagged Diagonals Storage (JDS) format used for vector computers, can drive the matrix format towards better utilization of compute resources and storage. First the rows of the ELLPACK scheme are sorted according to the number of non-zeros, starting with the longest row ("sort" step in Fig. 1). Then, blocks of b_r consecutive rows (where b_r should be the warp size) are padded to the longest row within the block ("pad" step in Fig. 1). We call the result "padded Jagged Diagonals Storage" (pJDS). This maintains load coalescing while most of the zero entries can be eliminated. Since the columns typically have different lengths, a (small) array `col_start[]` of size ($N_{\mathrm{nzr}}^{\max} \times 4$ byte) is required to store the starting offset of each column. The pJDS kernel is shown in listing 2.

Listing 2: The spMVM kernel of the pJDS format

```
1 for(i=0; i < N; ++i)
2     for(j=0; j < rowmax[i]; ++j){
3         col_offset = col_start[j];
4         c[i] += val[col_offset + i] *
5                 rhs[ col_idx[col_offset + i] ];
6     }
```

It maintains the structure and simplicity of the ELLPACK-R kernel but provides potential for (substantial) data reduction and better hardware utilization. The main drawback of the format is that the spMVM operation needs to be performed in a permuted basis. However, for most iterative spMVM algorithms such as Krylov subspace methods, permutation of the indices needs to be done only before the start and after the end of the algorithm, while the complete iterative scheme works on the permuted elements. On the downside, the permutation of the matrix rows can destroy matrix structures such as off-diagonals or local dense blocks, leading to a loss of load coalescing or cache reuse on the RHS vector.

Compared to other formats such as, e.g., BELLPACK [2] or ELLR-T [3], the pJDS format is suited for general unstructured matrices and does not use any a priori knowledge about the matrix structure. There are also no matrix-dependent tuning parameters.

The sparsity pattern determines the data reduction potential of pJDS. If the matrix has a constant row length (`rowmax[]`=$N_{\mathrm{nzr}}^{\max}$), ELLPACK and pJDS both have no storage overhead ($N \times N_{\mathrm{nzr}}^{\max}$ non-zeros). On the other hand, if there is one fully populated row and a single entry in all others, the plain ELLPACK format would store the full matrix, i.e., $N \times N$ elements. In pJDS it is sufficient to hold $b_r \times N + N - b_r = (b_r + 1) \times N - b_r$ entries. At a typical value of $b_r = 32$ one can expect a substantial reduction of the memory footprint for matrices with a wide variation in row lengths.

The row length histograms (Fig. 3) for DLR1/2, sAMG, and HMEp show that there is plenty of data reduction potential for those matrices. DLR1 should benefit least from the pJDS format since it has the lowest relative width (`max(rowmax[])/min(rowmax[])` ≈ 2) and most of the weight is clustered close to the maximum row length, i.e., 80% of the rows have a length of $0.8 \times N_{\mathrm{nzr}}^{\max}$. In contrast, the longest row of sAMG is more than four times larger than the smallest one, and short rows account for most of the weight. The data reduction rates finally achieved by using pJDS instead of ELLPACK follow this qualitative discussion and are shown in Table I. Considering the limited amount and high cost of device memory on GPGPUs, pJDS delivers a useful shrink of the memory footprint for spMVM on GPGPUs; e.g., the DLR2 matrix fits (in double precision) on an nVidia Fermi C2050 GPGPU only when using the pJDS format. The overhead of pJDS compared to a minimum implementation (storing the non-zeros only) is less than 0.01% for the matrices considered here (choosing $t_b = 32$).

The improved hardware utilization by pJDS (compare Figs. 2b and c) is also reflected in the performance numbers. In most scenarios gains of up to 30% can be achieved with pJDS, while the largest penalty is limited to 5%. Since

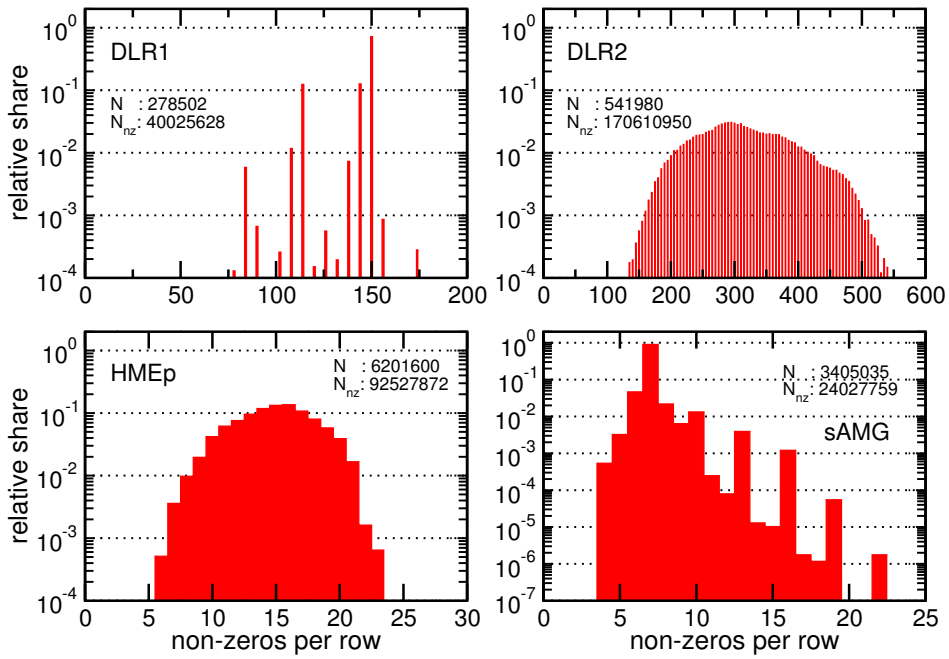


Figure 3: Row length distribution histograms of the matrices described in Sect. I-C. The bin size is 1 for all cases.

		DLR1	DLR2	HMEp	sAMG
data reduction [%]		17.5	48.0	36.0	68.4
SP ECC=0	ELLPACK-R	22.1	15.2	15.8	14.6
	pJDS	**27.6**	**18.7**	**18.9**	**19.5**
SP ECC=1	ELLPACK-R	18.0	**13.2**	**12.1**	11.6
	pJDS	**19.1**	12.1	11.6	**12.6**
DP ECC=0	ELLPACK-R	**18.7**	11.7	**12.3**	11.1
	pJDS	18.3	**14.6**	12.2	**13.0**
DP ECC=1	ELLPACK-R	**12.9**	**9.6**	**7.9**	7.8
	pJDS	**12.9**	9.5	7.5	**8.5**
Westmere EP	CRS (DP)	5.7	5.8	3.9	4.1

Table I: Data reduction of pJDS in comparison to the ELLPACK format and performance (in GF/s, excluding data transfers) of the different storage formats on an nVidia Fermi C2070 GPGPU. The best performance for each matrix and execution environment (DP/SP with ECC on/off) is highlighted. The last row shows the performance of a dual-socket (12 core) Intel Westmere node using the compressed row storage (CRS) format. See [4] for implementation and hardware details.

the row permutation may destroy regular substructures, the DLR2 and HMEp matrices do show some performance drop or only moderate speed-ups due to reduced cache reuse and load coalescing for the RHS vector. This problem is more severe on older GPGPU generations without L2 cache, such as the Tesla C1060. Here it is also necessary to map the array holding the column starting offsets (`col_start[]`) to the texture cache.

Although we did not encounter any failures during our benchmarks, activating ECC on the GPU is appropriate for all HPC applications where reliability is crucial. Thus we present performance results with ECC enabled as well as disabled. Note that there is no simple model to quantify the impact of ECC on the performance (apart from a general reduction of achievable memory bandwidth), since the details of the ECC mechanism are undisclosed.

In summary, the pJDS format presents a very attractive alternative to the ELLPACK-R scheme on modern GPGPUs both in terms of performance and memory footprint. On standard multicore CPUs, however, no blocked or plain JDS variant is able to outperform the standard CRS format [8].

B. GPGPU performance model and PCIe transfer impact

Due to the small (or non-existent) data cache on GPGPUs, the expected speedup compared to a multicore socket is usually smaller than the ratio of memory bandwidths. The worst-case code balance of the ELLPACK and pJDS kernels for double precision is

$$B_{\mathrm{W}}^{\mathrm{DP}} = \left(\frac{8+4+8\alpha+16/N_{\mathrm{nzr}}^{\mathrm{max}}}{2}\right)\frac{\text{bytes}}{\text{flop}} = \left(6+4\alpha+\frac{8}{N_{\mathrm{nzr}}^{\mathrm{max}}}\right)\frac{\text{bytes}}{\text{flop}}\,. \quad (1)$$

The parameter $1/N_{\mathrm{nzr}}^{\mathrm{max}} \leq \alpha \leq 1$ quantifies the possible re-use of RHS data from cache: If there is no cache, i.e., if each load to the RHS vector goes to memory, we have $\alpha = 1$. Hence, cache is able to reduce the balance by some amount. In the ideal case $\alpha = 1/N_{\mathrm{nzr}}^{\mathrm{max}}$ each RHS element has to be

loaded only once. This corresponds to the $\kappa = 0$ case in [4]. Note that B_W^{DP} may change from block to block due to different values of N_{nzr}^{max}, and that the `col_start[]` array is assumed to always come from cache. In the following we assume an average value N_{nzr} for the number of non-zeros per row.

The bandwidth model (1) is only valid for the kernel execution on the device, and does not include the data transfers required to bring the RHS vector to the GPU and the result back to the host. However, one can extend the model to incorporate those overheads. Since two distinct bandwidths are involved we now look at the expected wallclock times for the pure spMVM (T_{MVM}) and the required data transfer of the RHS and LHS vectors over the PCIe bus (T_{PCI}):

$$T_{MVM} = \frac{8N}{B_{GPU}} \left[N_{nzr} \left(\alpha + \frac{3}{2} \right) + 2 \right] \quad \text{and} \qquad T_{PCI} = \frac{16N}{B_{PCI}} \ . \tag{2}$$

This shows that a low PCIe bandwidth has less impact on the overall execution time if N_{nzr} is large, hence we can estimate the range of favorable N_{nzr} values: Setting $T_{MVM} \leq T_{PCI}$, i.e., assuming more than 50% penalty from the PCIe transfers, we arrive at

$$N_{nzr} \leq \frac{2(B_{GPU}/B_{PCI} - 1)}{\alpha + 3/2} \ . \tag{3}$$

In the worst case, $\alpha = 1/N_{nzr}$ and $B_{GPU} \gtrsim 20 B_{PCI}$ lead to $N_{nzr} \leq 25$. On the other hand, if $\alpha = 1$ and $B_{GPU} \approx 10 B_{PCI}$ we have $N_{nzr} \leq 7$. Thus we do not expect a significant benefit from GPGPU acceleration for the HMeP and sAMG matrices described above, since those have $N_{nzr} \approx 15$ and 7, respectively.

If we want less than 10% penalty from PCIe transfers ($T_{MVM} \geq 10 T_{PCI}$) we get

$$N_{nzr} \geq \frac{20 B_{GPU}/B_{PCI} - 2}{\alpha + 3/2} \ , \tag{4}$$

so at $B_{GPU} \approx 10 B_{PCI}$ and $\alpha = 1$ a value of $N_{nzr} \gtrsim 80$ is sufficient. This is certainly satisfied for all DLR matrices. In the worst case, i.e., at $B_{GPU} \approx 20 B_{PCI}$ and $\alpha = 1/N_{nzr}$ one arrives at $N_{nzr} \gtrsim 266$; in this case we can expect a measurable impact of PCIe transfer overhead for all matrices considered here.

III. Distributed-memory spMVM parallelization

As described in Sect. II-B, matrices with small N_{nzr} are no good candidates for GPGPU acceleration since the required PCIe transfers for the RHS and LHS vectors dominate the runtime. Although this penalty is somewhat mitigated by the fact that in some real applications parts of those vectors may be kept on the device, all data that has to be communicated using MPI must also cross the PCIe bus to the GPGPU. For the HMEp (sAMG) matrix we arrive at a single-GPU performance level of 3.7 (2.3) GF/s, which is already below the capability of a typical dual-socket server node (see Table I). Hence, we restrict the discussion in this section to the DLR1 and UHBR matrices. Although they also suffer from PCIe transfers to some extent (10.9 GF/s vs. 12.9 GF/s for DLR1), there is still a substantial advantage over the CPU version.

All runs were performed in double precision and with active ECC on the NERSC *Dirac* cluster. The ELLPACK-R format was used throughout, since the matrix storage format is of minor importance for the double precision case (see Table I) and for the concepts we want to demonstrate here. An implementation of the multi-GPGPU code with the pJDS format and an analysis of its performance implications is ongoing work and will follow the strategy described in [11].

A. Multi-GPGPU spMVM

The basic design patterns and choices described in [4] for distributed-memory parallel sparse MVM also apply for the multi-GPGPU case. We distinguish between *vector mode*, which resembles the programming style on vector-parallel machines, and *task mode*, which dedicates host resources (threads) to different tasks, i.e., communication and computation. In this work we consider three alternatives:

- *Vector mode* without overlap of communication and computation. The required communication to distribute the nonlocal RHS elements among the processes is separate from the actual spMVM communication, which is performed in a single step.
- *Vector mode* using naive overlap of communication and computation by nonblocking MPI. The spMVM must be split into a local and a nonlocal part, and the former may be overlapped with MPI. Since the result vector must be written twice, there is a slight increase in memory traffic, which adds another $8/N_{nzr}$ bytes/flop to the code balance (1). Due to the rather large N_{nzr} of the DLR1 and UHBR matrices we expect a performance penalty of below 10%, though. Since most MPI libraries do not support asynchronous nonblocking point-to-point communication, we do not expect this variant to have any advantage even over vector mode without overlap.
- *Task mode* using a dedicated thread for MPI in order to implement reliably asynchronous communication. Figure 4 shows an event timeline that visualizes the different tasks executed on two host threads (or more if there are multiple GPGPUs in a node) and the GPGPU. Depending on the ratio of communication to computation time, the possible performance benefit can be at most a factor of two. At strong scaling we expect task mode and vector mode performance to converge.

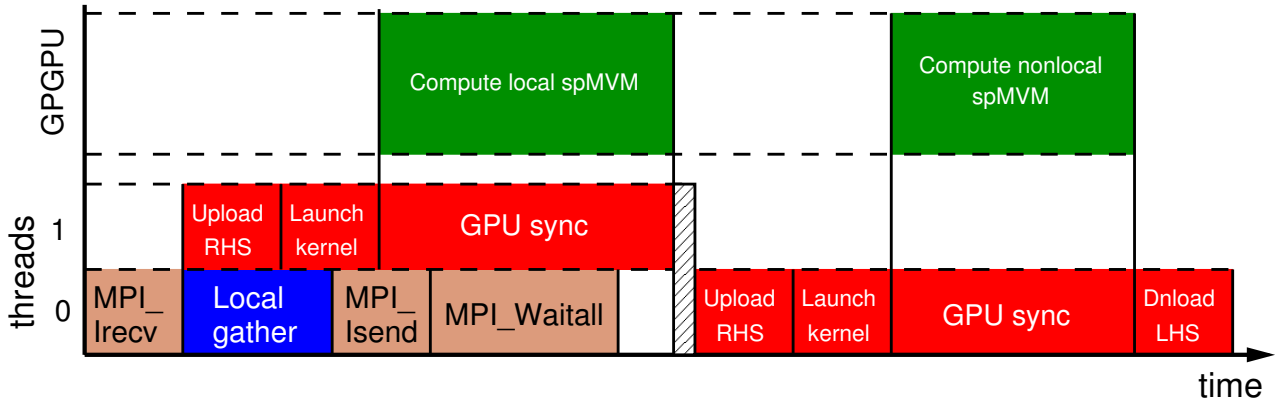


Figure 4: Timeline for GPGPU-based spMVM kernel including host data transfers with a dedicated host thread for asynchronous MPI communication (thread 0). The "local gather" is the collection of data to be sent to other processes into a contiguous buffer.

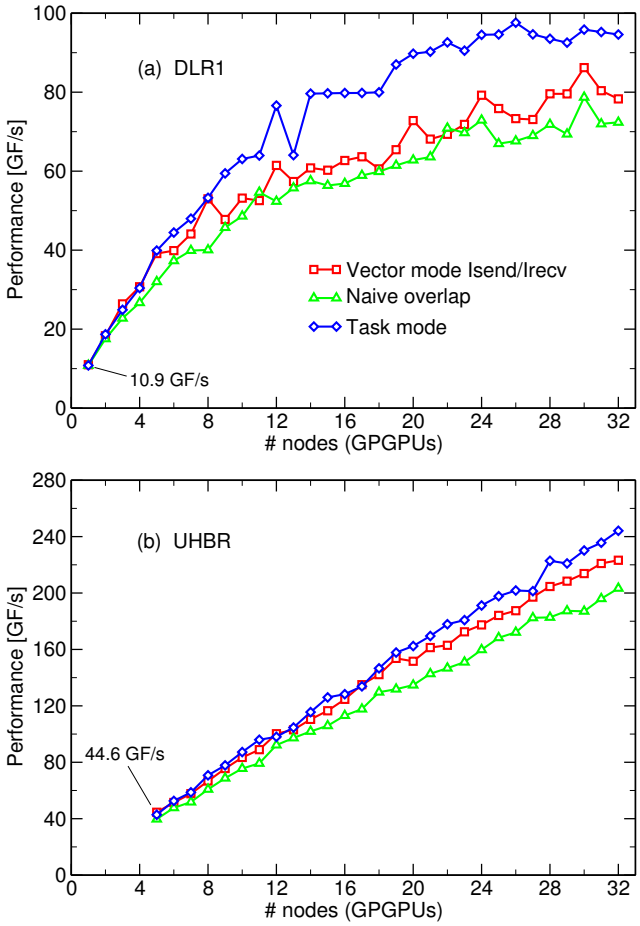


Figure 5: Strong scaling results for DLR1 (a) and UHBR (b) on the Dirac cluster. Due to memory restrictions on the C2050 cards it was not possible to run the UHBR case on fewer than five nodes.

B. *Performance results*

Figures 5a and 5b show strong scaling results for the DLR1 and UHBR matrices, respectively, on the Dirac cluster. Task mode achieves better performance than any of the vector modes in both cases; however, the details differ considerably:

DLR1 has a rather small dimension of 2.8×10^5, so that only 8750 rows (about 1.3×10^6 non-zeros) are left per GPGPU at 32 nodes. The smallness of the per-GPGPU subproblem leads to a substantial performance drop, which mainly originates from the nonlocal part in the naive vector and task mode versions. It can, however, be partially compensated by asynchronous communication. At larger node counts the performance of all variants starts to converge, as expected.

UHBR has a much larger number of non-zeros at a similar N_{nzr} as DLR1, and thus does not show an analogous per-GPGPU performance breakdown when scaling up the node count. Scalability is very good in task mode with a parallel efficiency of 84% at 32 nodes (about 70% with naive overlap vector mode). Since the communication requirements are weaker than for DLR1, we do not see a similarly large benefit from overlapping communication at the node counts accessible on the cluster used.

IV. Conclusions and Outlook

We have introduced a new "padded JDS" sparse matrix format, which is suitable for sparse matrix-vector multiplication on GPGPUs at similar or better performance levels than the popular ELLPACK-R format, with a potential for significant memory savings.

Via suitable performance models we have derived a condition for the average number of non-zeros per matrix row that guarantees a useful performance benefit of GPGPU-based spMVM in comparison to standard server nodes, the main

parameter being the ratio between PCI express bandwidth and GPGPU memory bandwidth.

Finally we have extended previous work on efficient distributed-memory hybrid (MPI+OpenMP) spMVM parallelization to the multi-GPGPU case. Using dedicated host threads for explicitly asynchronous MPI communication we were able to improve significantly over naive "vector-like" approaches and show the potentials and limitations of this solution.

Future work will cover more extensive scaling studies on larger GPGPU clusters, an implementation of the pJDS format in the multi-GPGPU code, a thorough investigation of the performance degradation with strong scaling, and the application of our results to a production-grade eigensolver.

During the preparation of the manuscript it came to our attention that other research groups have devised sparse matrix formats that share some features with pJDS, most notably the "sliced ELLPACK" and "sliced ELLR-T" formats [12, 13]. A thorough comparison of pJDS with those alternative approaches is work in progress.

ACKNOWLEDGMENTS

Discussions with Jan Treibig and Thomas Zeiser are gratefully acknowledged. We are indebted to Matthias Griessinger for initial implementations of the GPGPU spMVM kernels, to Gerald Schubert for providing CPU comparisons, and to K. Stüben and H. J. Plum for providing and supporting the sAMG test case. This research used the *Dirac* GPGPU cluster of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Part of this work was supported by the competence network for scientific high performance computing in Bavaria (KONWIHR) via the project HQS@HPC-II.

REFERENCES

[1] N. Bell and M. Garland: *Implementing sparse matrix-vector multiplication on throughput-oriented processors.* Proc. SC'09. DOI:10.1145/1654059.1654078

[2] J. W. Choi, A. Singh, and R. W. Vuduc: *Model-driven autotuning of sparse matrix-vector multiply on GPUs.* Proc. PPoPP'10. DOI:10.1145/1693453.1693471

[3] V. Vázquez, J. J. Fernández, and E. M. Garzón: *A new approach for sparse matrix vector product on NVIDIA GPUs.* Concurrency and Computation: Practice and Experience **23**(8), 815–826 (2011). DOI:10.1002/cpe.1658

[4] G. Schubert, G. Hager, H. Fehske, and G. Wellein: *Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems.* Parallel Processing Letters **21**(3), 339–358 (2011). DOI:10.1142/S0129626411000254

[5] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein: *Performance engineering for the Lattice Boltzmann method on GPGPUs: Architectural requirements and performance results.* Accepted for publication in Computers & Fluids. Preprint: http://arxiv.org/abs/1112.0850

[6] K. Stüben: *An Introduction to Algebraic Multigrid.* In: U. Trottenberg et al. (Eds.): Multigrid: Basics, Parallelism and Adaptivity, Academic Press (2000).

[7] http://www.scai.fraunhofer.de/en/business-research-areas/numerical-software/products/samg.html

[8] G. Schubert, G. Hager and H. Fehske: *Performance limitations for sparse matrix-vector multiplications on current multicore environments.* In: S. Wagner et al., High Performance Computing in Science and Engineering, Garching/Munich 2009. Springer, ISBN 978-3642138713 (2010), 13–26. DOI:10.1007/978-3-642-13872-0_2

[9] A. Basermann et al.: *HICFD - Highly Efficient Implementation of CFD Codes for HPC Many-Core Architectures.* In: Proceedings of CiHPC, Springer 2011 [in print]

[10] R. Grimes, D. Kincaid, and D. Young. ITPACK User's Guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Aug. 1979. http://rene.ma.utexas.edu/CNA/ITPACK/

[11] G. Wellein, G. Hager, A. Basermann, and H. Fehske: *Fast sparse matrix-vector multiplication for TFlop/s computers.* In: J. Palma, J. Dongarra (Ed.): High Performance Computing for Computational Science — VECPAR2002, LNCS 2565, Springer Berlin (2003). DOI:10.1007/3-540-36569-9_18

[12] A. Monakov, A. Lokhmotov, A. Avetisyan: *Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures.* In: Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, X. Martorell (Eds.): Lecture Notes in Computer Science, Springer, ISBN 978-3-642-11514-1 (2010), 111–125. DOI:10.1007/978-3-642-11515-8_10

[13] A. Dziekonski, A. Lamecki, M. Mrozowski: *A Memory Efficient and Fast Sparse Matrix Vector Product on a GPU.* Progress In Electromagnetics Research **116**, 49–63 (2011). DOI:10.2528/PIER11031607