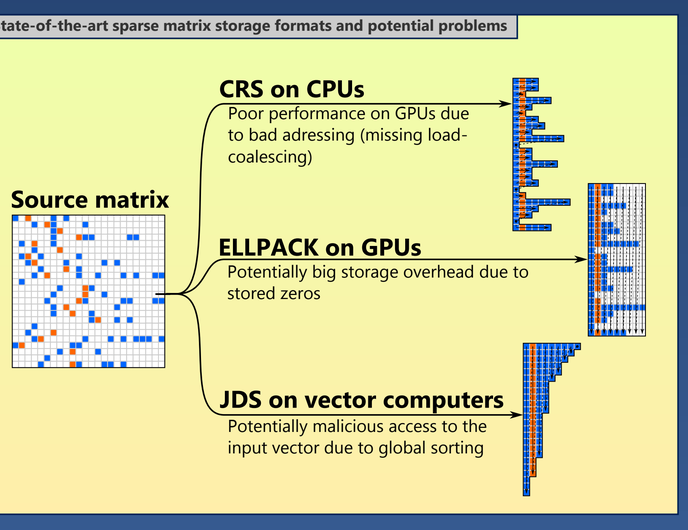
A Unified Sparse Matrix Storage Format for Heterogeneous Systems

Moritz Kreutzer, Georg Hager, and Gerhard Wellein
{moritz.kreutzer,georg.hager,gerhard.wellein}@fau.de
Erlangen Regional Computing Center, Friedrich-Alexander University of Erlangen-Nuremberg, Germany

Introduction

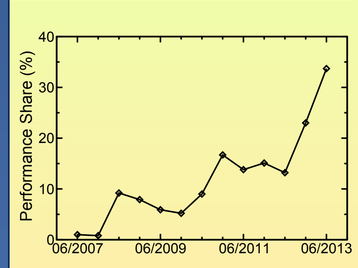
Sparse Matrix Vector Multiplication (SpMVM, y=Ax+y) is a very bandwidth-demanding operation and one of the most time-consuming routines in many scientifical applications.
Sparse matrices (number of non-zeros N_{nz}=O(N)) are usually stored in a way which omits (most of) the zero entries in order to minimize the storage overhead.
The SpMVM performance relies on the optimal storage format which is closely related to the compute architecture in use.
The following image illustrates the state-of-the-art storage formats for three different architectures and potential problems that may arise for them.

State-of-the-art sparse matrix storage formats and potential problems



Emerging co-processors/accelerators like the Intel Xeon Phi or Nvidia Tesla GPUs are of special interest for the spMVM because of their large memory bandwidth together with a high level of on-chip parallelism.
Upcoming cluster systems will probably be more and more of heterogeneous nature, i.e., consisting of standard CPUs together with accelerators. Sustainable and modern high-performance software should be able to utilize all parts of a heterogeneous cluster. Hence, establishing a single storage format for sparse matrices that yields good performance on all architectures is of broad interest.

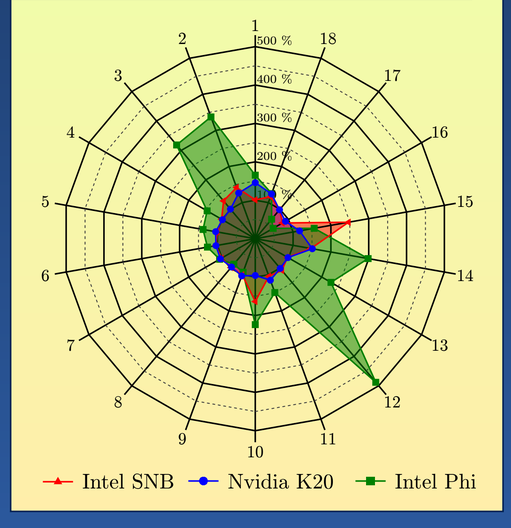
Accelerator/Co-Processor Performance Share of TOP500 Systems



Contribution

This work demonstrates the feasibility of a unified storage format for sparse matrices, namely **SELL-C-σ**. It builds on Sliced ELLPACK, which has only been used on GPUs up to now, and provides competitive performance on a wide variety of processor designs. It has two parameters, C and σ, for which good values can be predicted in advance. A performance model allows us to study the interplay of C, σ and further matrix properties such as the "chunk occupancy" and the number of non-zeros per row.

Relative SELL-32-σ (σ matrix-dependent) performance for three architectures and 18 test matrices where 1..4 are corner cases and 5..18 are the Williams matrices from the University of Florida Sparse Matrix Collection
Intel Xeon Sandy Bridge (SNB) and Xeon Phi w/ baseline CRS, Nvidia Tesla K20c w/ baseline ELLPACK-R



SELL-C-σ construction

Non-zero matrix entries and column indices are stored column-wise in chunks of height C (C=1: CRS). All rows of a chunk are padded with zeros to have the same length. Storage overhead is minimal and performance is usually best if the chunk structure is chosen in accordance with the width of a SIMD (Single Instruction Multiple Data) register on x86 and Intel MIC.

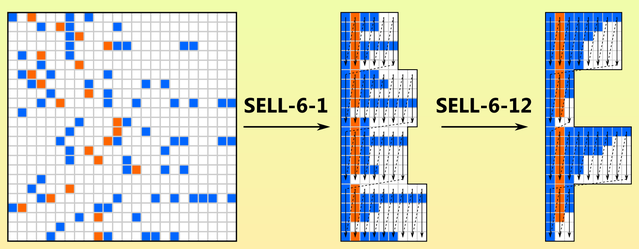
On GPUs, SIMD≙SIMT (Single Instruction Multiple Threads) with width = warp size (32 threads on Nvidia Kepler). In a heterogeneous setting: Choose C in accordance with the largest available SIMD width.

SIMD width of selected architectures

	SIMD width
Intel Sandy Bridge	256 bits
Intel Xeon Phi	512 bits
Nvidia Kepler	2048 bits

Sorting rows by the number of non-zero entries within a limited scope σ of rows reduces the overhead as equally large rows get close to each other and can increase performance. However, if σ is too large (extremal case: global sorting) performance may drop (potentially malicious access pattern to x).

Construction of SELL-C-σ from a sparse matrix



SELL-C-σ data (besides C and σ):

`double val[]`	matrix values
`double col[]`	column indices
`int chunkWidth[]`	length of the chunk's longest row
`int chunkStart[]`	each chunk's starting offset in val/col
`int nRows`	number of rows (multiple of C)

The SELL-C-σ spMVM kernel can be easily vectorized by the compiler and requires neither a remainder loop nor a vector reduction, in contrast to the vectorized CRS kernel.

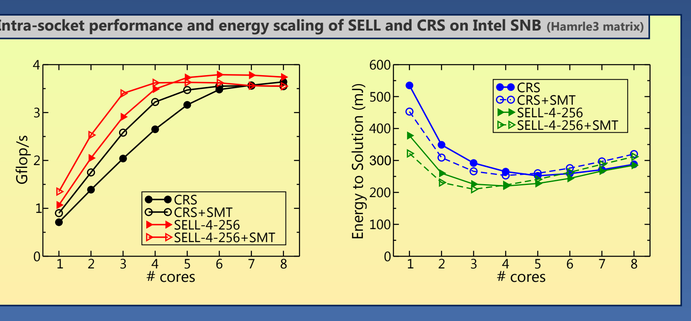
SELL-4-σ kernel in C

```
for(i = 0; i < nRows/4; ++i)
{ // loop over chunks
  for(j = 0; j < chunkWidth[i]; ++j)
  { // loop inside chunk
    y[i*4+0] += val[chunkStart[i]+j*4+0] *
                x[col[chunkStart[i]+j*4+0]];
    y[i*4+1] += val[chunkStart[i]+j*4+1] *
                x[col[chunkStart[i]+j*4+1]];
    y[i*4+2] += val[chunkStart[i]+j*4+2] *
                x[col[chunkStart[i]+j*4+2]];
    y[i*4+3] += val[chunkStart[i]+j*4+3] *
                x[col[chunkStart[i]+j*4+3]];}}
```

Scaling and Energy

SpMVM is a bandwidth-bound kernel for large data sets and obviously, SELL-C-σ's SIMD capabilities cannot increase the bandwidth and saturated spMVM performance. However, due to faster in-socket scaling a considerable amount of energy can be saved in SELL-C-σ compared to CRS. Using Simultaneous Multi-Threading (SMT) can further improve the energy efficiency.

Intra-socket performance and energy scaling of SELL and CRS on Intel SNB (Hamrle3 matrix)



Performance Model

The "chunk occupancy" β quantifies the overhead by the zero padding. It is the ratio between N_{nz} and the elements stored in the SELL-C-σ format (N_C denotes the number of chunks):

$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot \texttt{chunkWidth[i]}}$$

$$\beta_{worst} = \frac{\sum_{i=0}^{N_c}(N+C-1)}{\sum_{i=0}^{N_c} CN} = \frac{N+C-1}{CN} \overset{N \gg C}{\longrightarrow} \frac{1}{C}$$

matrix with one line of length N and C-1 lines of length 1 per chunk

$$\beta_{best} = \frac{\sum_{i=0}^{N_c} C \cdot \texttt{chunkWidth[i]}}{\sum_{i=0}^{N_c} C \cdot \texttt{chunkWidth[i]}} = 1$$

matrix with all lines of equal length in each chunk

The code balance of the SELL spMVM kernel for large data sets can be given as follows (8-byte values, 4-byte indices, N_{nzr} denoting the number of non-zeros per row):

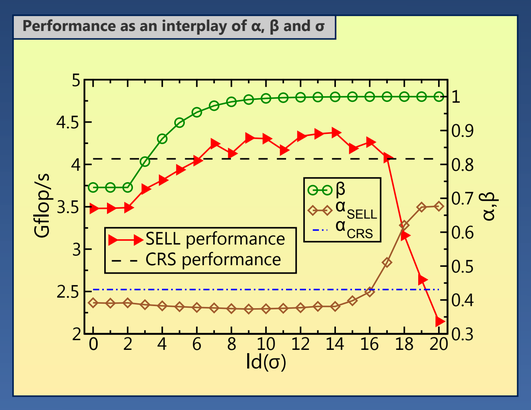
$$B_{SELL}^{DP} = \left(\frac{1}{\beta} \left(\frac{\overbrace{8}^{\texttt{val}}+\overbrace{4}^{\texttt{col}}}{2} \right) + \frac{\overbrace{8\alpha}^{\texttt{x}}+\overbrace{16/N_{nzr}}^{\texttt{y}}}{2} \right) \frac{\text{bytes}}{\text{flop}}$$

This knowledge can, e.g., be used for performance predictions based on a roofline model.

The factor α quantifies the efficiency of the access to x.
1. Each load of x goes to main memory: α=1
   (or even larger if the cache is organized in cache lines)
2. Access to x is perfectly cached: $\alpha=1/N_{nzr}$.

Increasing the sorting scope σ results in a perfect β=1. However, if σ gets too large there is a high chance that the access pattern to x worsens (α increases) → performance drops.

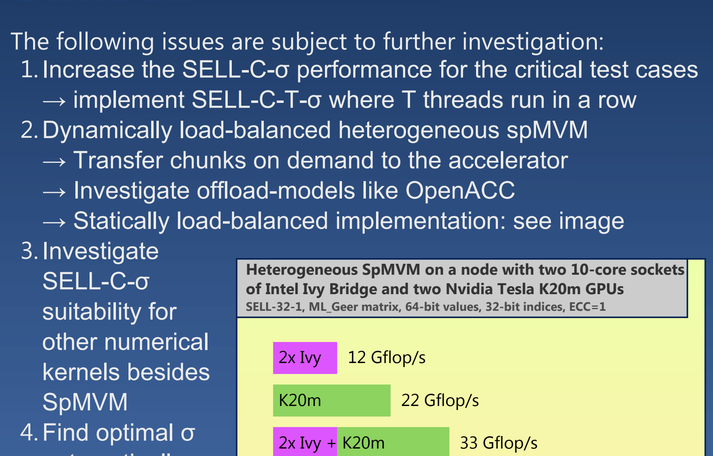
Performance as an interplay of α, β and σ



Unified Layout Performance

When using SELL-C-σ as a unified layout for the three architectures, we set C=32 (warp size on Nvidia Kepler). σ is matrix-dependent.
The following image shows the SELL-32-σ performance for 18 test cases relative to the
1. baseline (same data as in Sec. Contribution) and
2. best measured performance (best format, C, and σ)

Performance of the unified data layout



The performance of the unified data layout barely falls behind the best performance significantly. For some cases with poor performance, better tuning parameters or a change to SELL-C-T-σ (see Sec. Outlook) may improve the results.

Conclusions

We have presented SELL-C-σ as in ideal candidate for a unified sparse matrix data layout. Compared to device-specific formats for three different architectures, it yields comparable or better performance on a wide range of sparse matrices. Especially on the Xeon Phi it is far superior to the standard CRS format.

Outlook

The following issues are subject to further investigation:
1. Increase the SELL-C-σ performance for the critical test cases
   → implement SELL-C-T-σ where T threads run in a row
2. Dynamically load-balanced heterogeneous spMVM
   → Transfer chunks on demand to the accelerator
   → Investigate offload-models like OpenACC
   → Statically load-balanced implementation: see image
3. Investigate SELL-C-σ suitability for other numerical kernels besides SpMVM
4. Find optimal σ automatically when reading in the matrix
5. Exploit special matrix structures

Heterogeneous SpMVM on a node with two 10-core sockets of Intel Ivy Bridge and two Nvidia Tesla K20m GPUs
SELL-32-1, ML_Geer matrix, 64-bit values, 32-bit indices, ECC=1

Acknowledgments

This work was supported (in part) by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) under project ESSEX.

M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop. *A unified sparse matrix data format for modern processors with wide SIMD units.* CoRR abs/1307.6209

D. R. Kincaid, T. C. Oppe and D. M. Young. *ITPACKV 2D user's guide.* Report CNA-232, The University of Texas at Austin, May 1989

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

A. Monakov, A. Lokhmotov, and A. Avetisyan. *Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures.* In: Y. Patt et al. (eds.), vol. 5952 of LNCS, DOI 10.1007/978-3-642-11515-8_10

E. Saule, K. Kaya and Ü. V. Çatalyürek. *Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi.* CoRR abs/1302.1078

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG