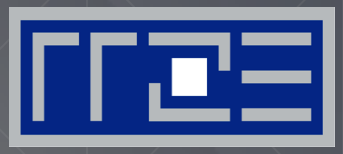
ERLANGEN REGIONAL COMPUTING CENTER



Sparse Matrix-Vector Multiplication with Wide SIMD Units: Performance Models and a Unified Storage Format

Moritz Kreutzer, Georg Hager, Gerhard Wellein

SIAM PP14 MS53
Feb 20, 2014

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Agenda

- Related work
- Motivation for a SIMD-compatible sparse matrix format
- Deficiencies of CRS
- Constructing SELL-C-σ
- Roofline model for SELL-C-σ
- Performance results for
  - Intel Sandy Bridge
  - Intel Xeon Phi
  - Nvidia K20
- Conclusions & outlook

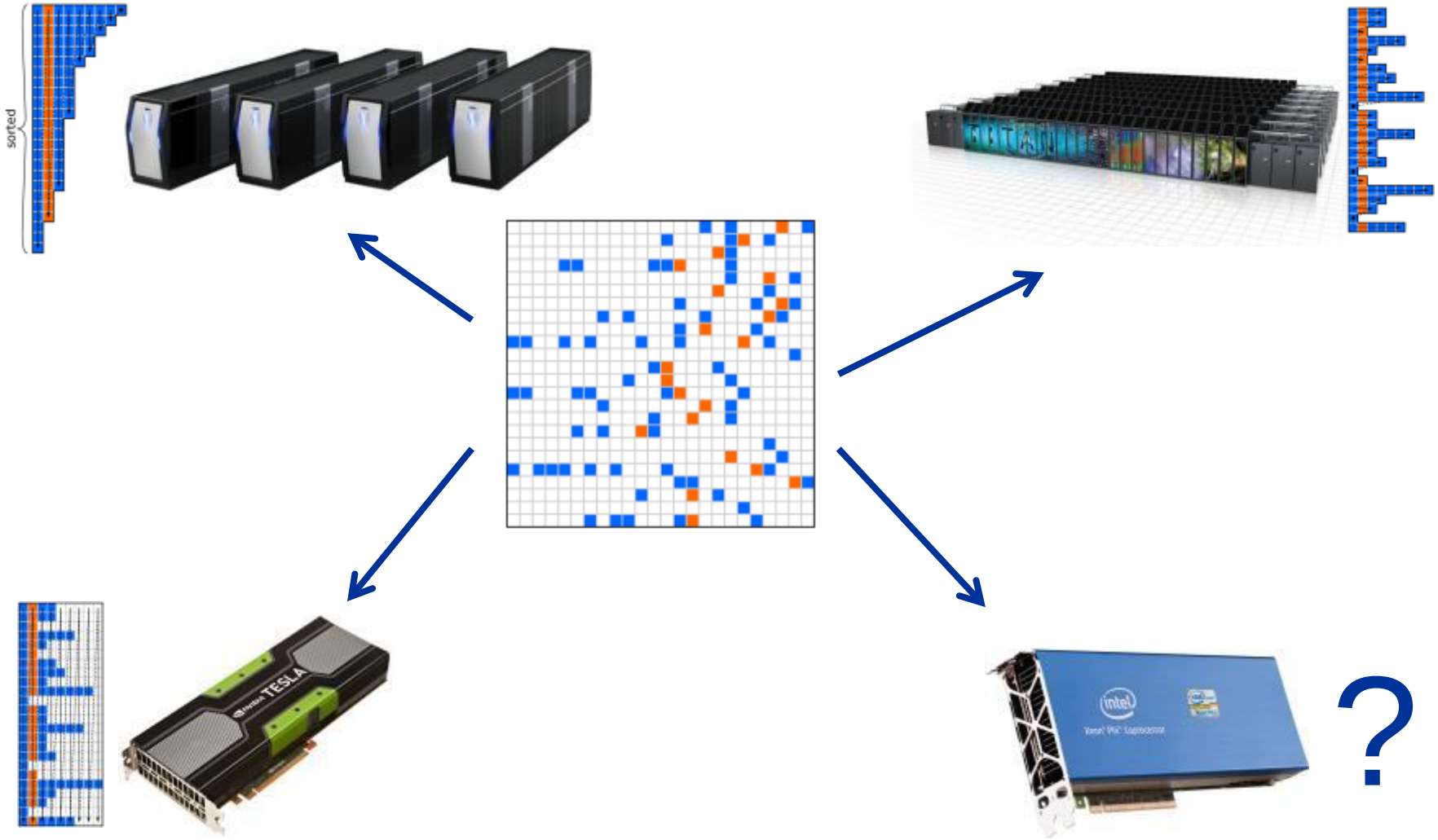
M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for modern processors with wide SIMD units*. Submitted. Preprint: arXiv:1307.6209

Related (recent) work

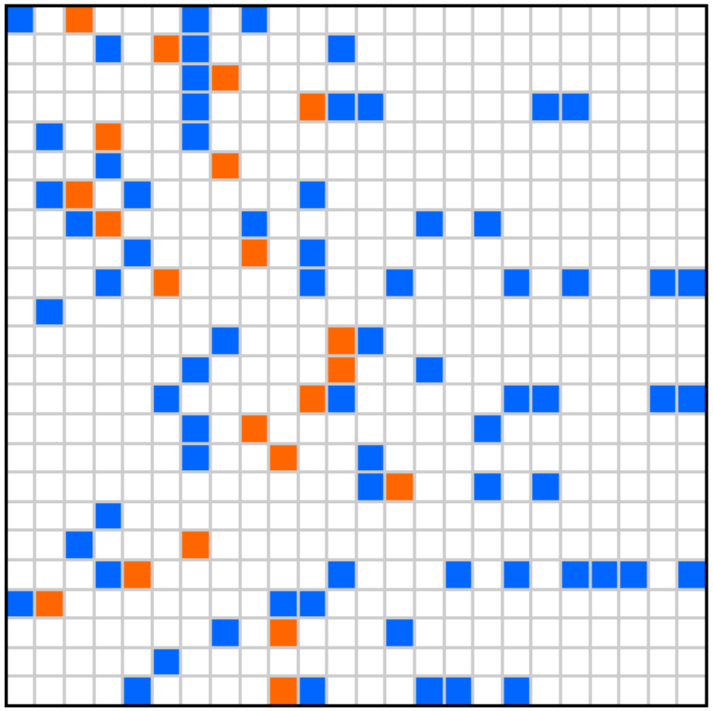
- X. Liu, M. Smelyanskiy, E. Chow and P. Dubey. *Efficient sparse matrix-vector multiplication on x86-based many-core processors*. In: Proc. ICS '13 (ACM, New York, NY, USA) 273-282. DOI: 10.1145/2464996.2465013

- E. Saule, K. Kaya and U. V. Çatalyürek. *Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi*. Proc. PPAM 2013, arXiv:1302.1078
- A. Monakov, A. Lokhmotov and A. Avetisyan. *Automatically tuning sparse matrix-vector multiplication for GPU architectures*. In: Y. Patt et al. (eds.), LNCS vol. 5952 111-125. DOI: 10.1007/978-3-642-11515-8_10
- A. L. A. Dziekonski and M. Mrozowski. *A memory ecient and fast sparse matrix vector product on a GPU*. Progress In Electromagnetics Research 116, (2011) 49-63. DOI: 10.2528/PIER11031607

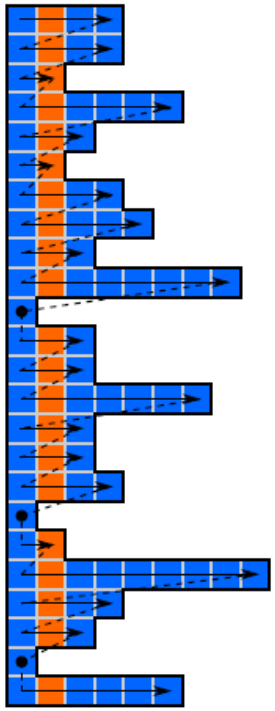
Variety of Sparse Matrix Storage Formats



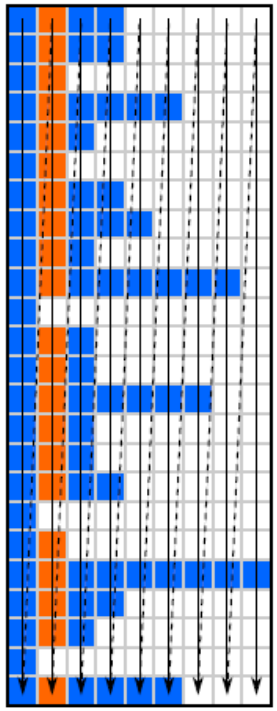
Standard sparse matrix storage formats



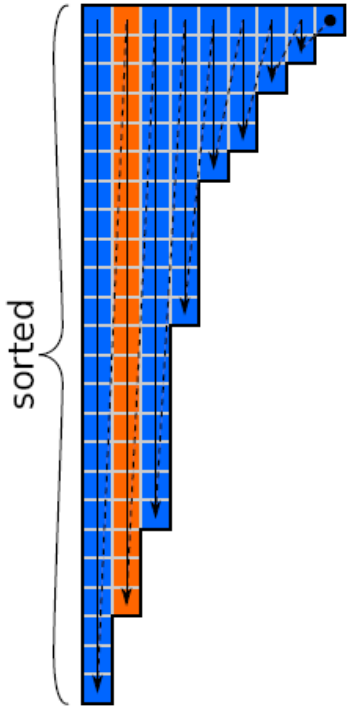
original matrix



CRS



ELLPACK



JDS

CRS on wide SIMD units

- When number of nonzeros per row is small
  - Significant remainder loop overhead (partial/no vectorization)
  - Large impact of row reduction
  - Alignment constraints require additional peeling
  - Small loop count
- GPGPUs
  - One warp per row: similar problems
  - Outer loop parallelization: Loss of coalescing

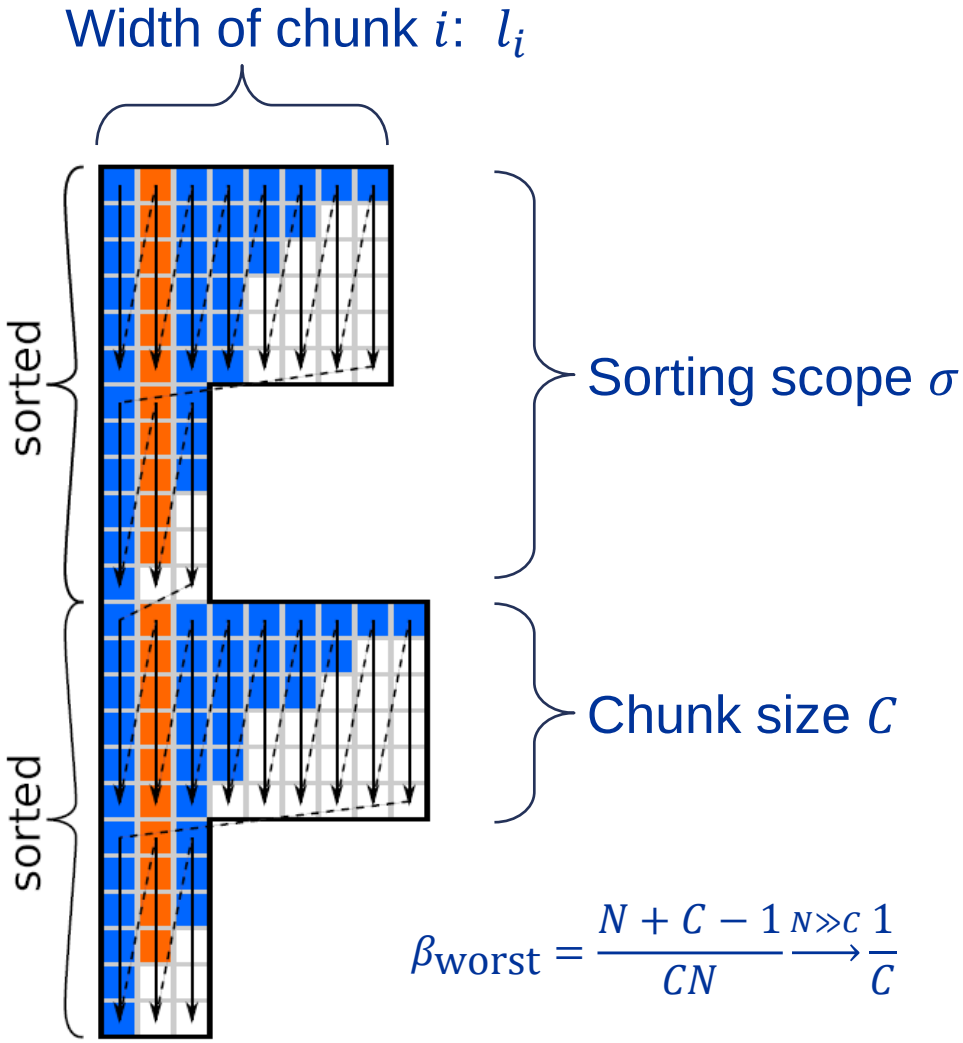
```
for(i = 0; i < N; ++i)
{
  tmp0 = tmp1 = tmp2 = tmp3 = 0.;
  for(j = rpt[i]; j < rpt[i+1]; j+=4)
  {
    tmp0 += val[j+0] * x[col[j+0]];
    tmp1 += val[j+1] * x[col[j+1]];
    tmp2 += val[j+2] * x[col[j+2]];
    tmp3 += val[j+3] * x[col[j+3]];
  }
  y[i] += tmp0+tmp1+tmp2+tmp3;
  // remainder loop
  for(j = j-4; j < rpt[i+1]; j++)
    y[i] += val[j] * x[col[j]];
}
```

Constructing SELL-C-σ

1. Pick chunk size C (guided by SIMD/T widths)
2. Pick sorting scope σ
3. Sort rows by length within each sorting scope
4. Pad chunks with zeros to make them rectangular
5. Store matrix data in “chunk column major order”

“Chunk occupancy”: fraction of “useful” matrix entries

$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}$$



Matrix characterization

"Corner case" matrices from "Williams Group":

Test case	N	N_{nz}	N_{nzr}	density	$\beta_{\sigma=1}^{C=16}$	$\beta_{\sigma=256}^{C=16}$
RM07R	381,689	37,464,962	98.16	2.57e-04	0.63	0.93
kkt_power	2,063,494	14,612,663	7.08	3.43e-06	0.54	0.92
Hamrle3	1,447,360	5,514,242	3.81	2.63e-06	1.00	1.00
ML_Geer	1,504,002	110,879,972	73.72	4.90e-05	1.00	1.00

Remaining matrices:

pwtk	217,918	11,634,424	53.39	2.45e-04	0.99	1.00
shipsec1	140,874	7,813,404	55.46	3.94e-04	0.89	0.98
consph	83,334	6,010,480	72.13	8.65e-04	0.94	0.97
pdb1HYS	36,417	4,344,765	119.31	3.28e-03	0.84	0.97
cant	62,451	4,007,383	64.17	1.03e-03	0.90	0.98

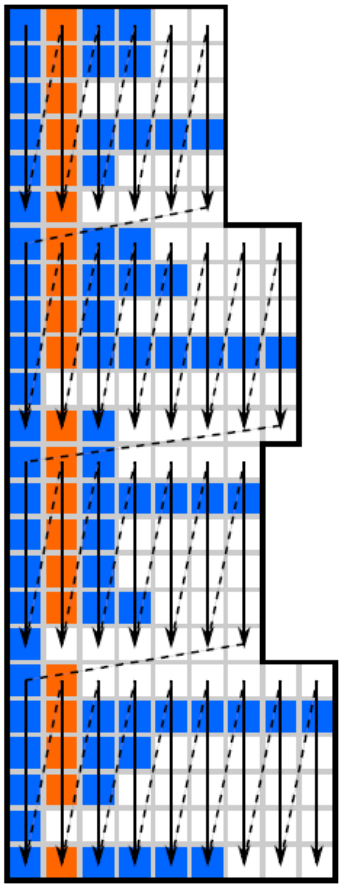
...

SELL-C-σ kernel

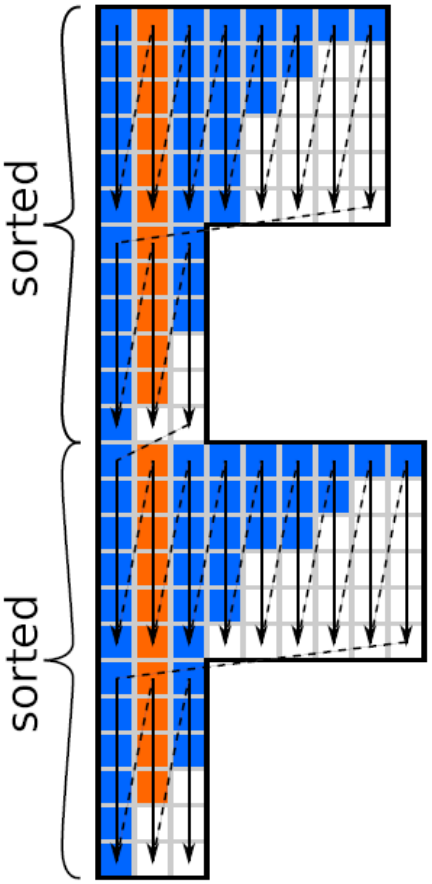
```
for(i = 0; i < N/4; ++i)
{

  for(j = 0; j < cl[i]; ++j)
  {
    y[i*4+0] += val[cs[i]+j*4+0] *
                x[col[cs[i]+j*4+0]];
    y[i*4+1] += val[cs[i]+j*4+1] *
                x[col[cs[i]+j*4+1]];
    y[i*4+2] += val[cs[i]+j*4+2] *
                x[col[cs[i]+j*4+2]];
    y[i*4+3] += val[cs[i]+j*4+3] *
                x[col[cs[i]+j*4+3]];
  }
}
```

$C = 4$

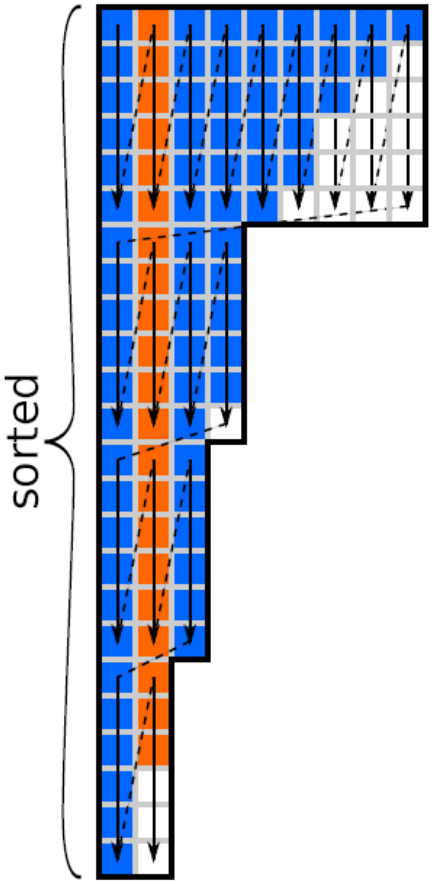
Variants of SELL-C-σ



SELL-6-1
β=0.51



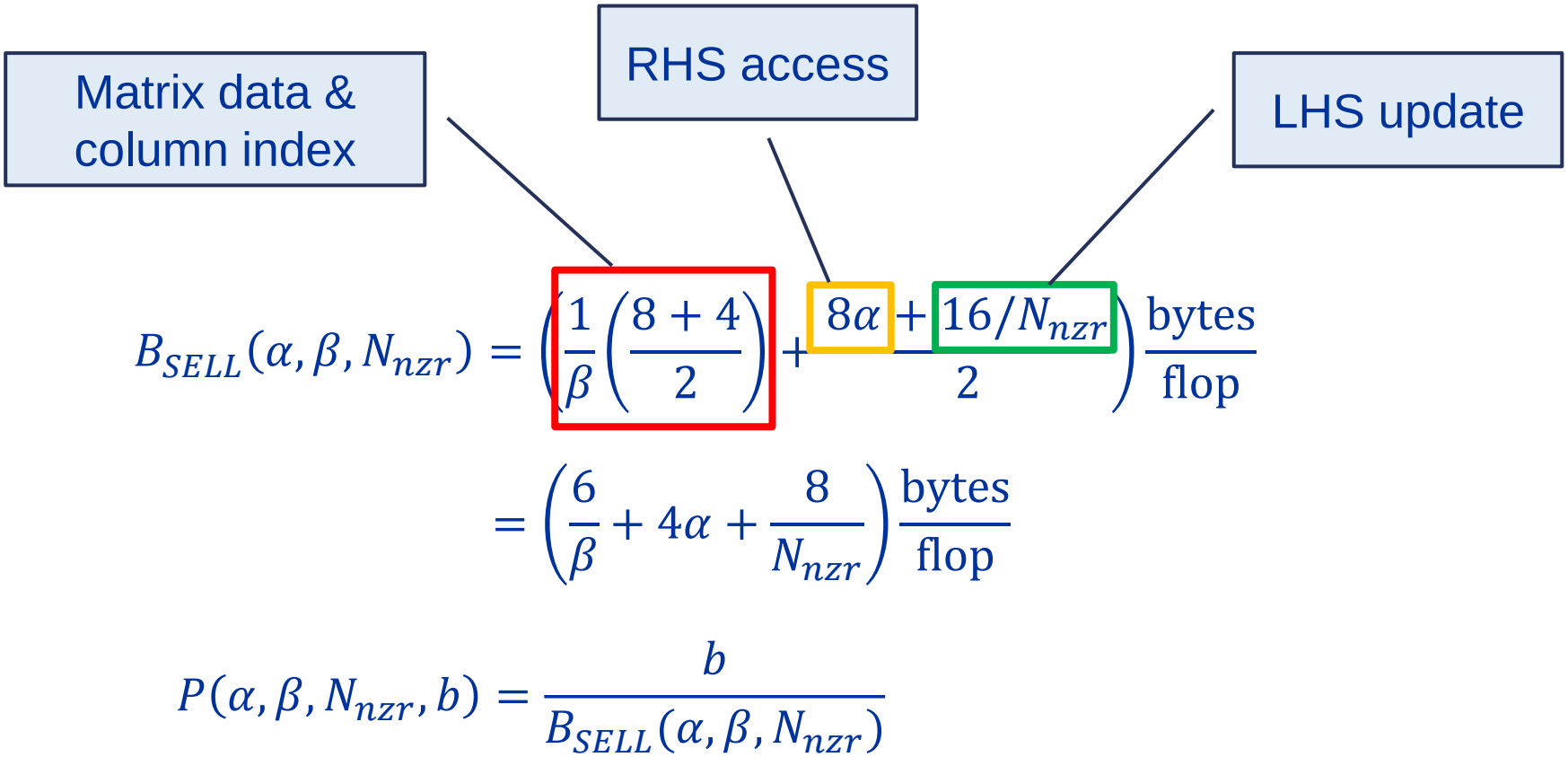
SELL-6-12
β=0.66



SELL-6-24
β=0.84

Roofline performance model for SELL-C-σ

Code balance (double precision FP, 4-byte index):



The α parameter

Corner case scenarios:

1. $\alpha = 0$ → RHS in cache
2. $\alpha = \frac{1}{N_{nzc}}$ → Load RHS vector exactly once

If $N_{nzc} \gg 1$, RHS traffic is insignificant: $\bar{P} = \frac{b\beta}{6\,\text{bytes/flop}}$

3. $\alpha \approx 1$ → Each RHS load goes to memory

4. $\alpha > 1$ → Houston, we've got a problem ☺

Determine α by measuring actual spMVM memory traffic (HPM)

Determine RHS traffic

$\boldsymbol{V_{meas}}$ is the measured overall memory data traffic (using, e.g., likwid-perfctr)

Determine $\boldsymbol{\alpha}$:

$$\alpha = \frac{1}{4}\left(\frac{V_{meas}}{N_{nz} \cdot 2\ \text{bytes}} - 6 - \frac{8}{N_{nzr}}\right)$$

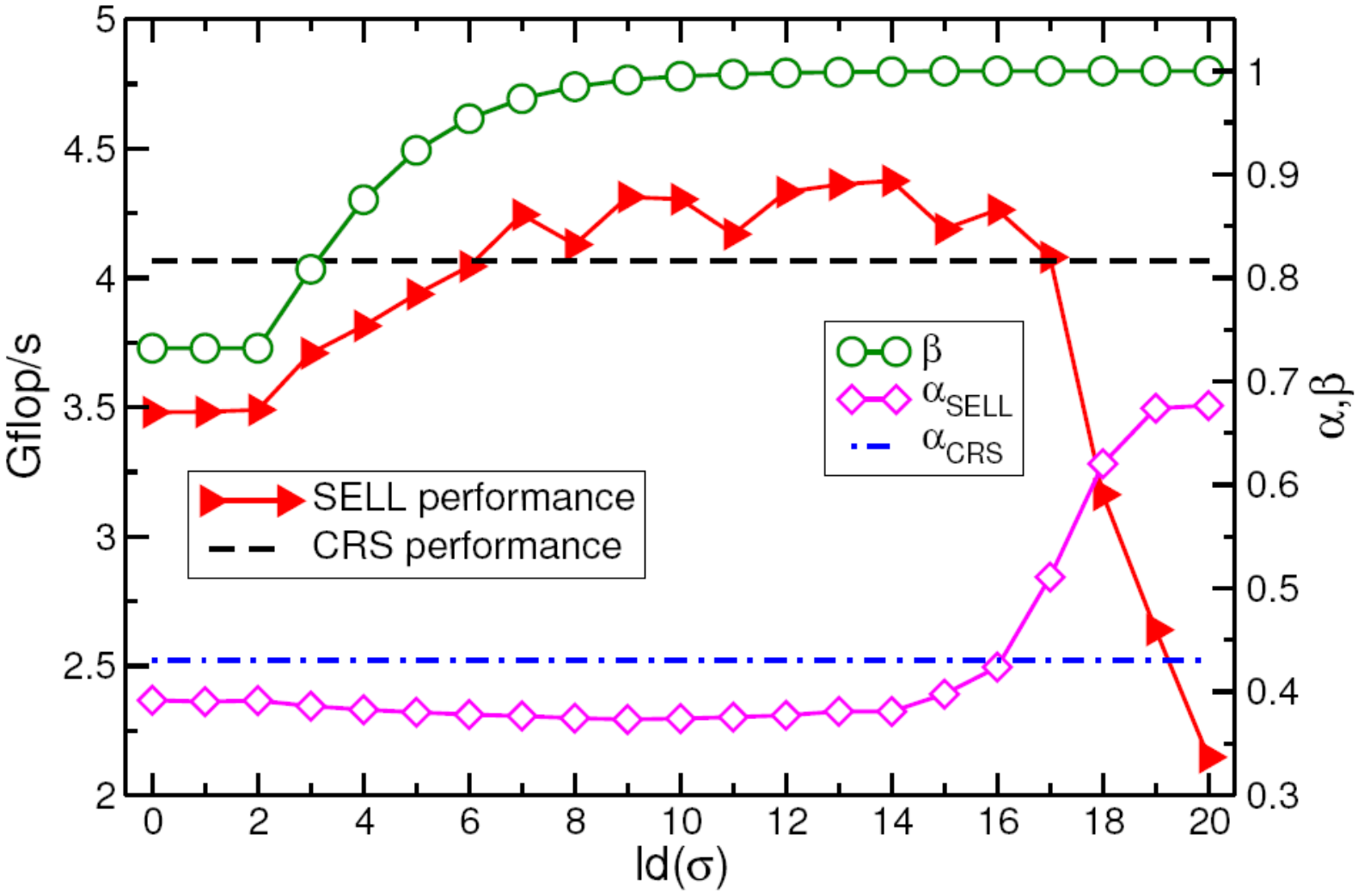
Example: kkt_power matrix on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6$, $N_{nzr} = 7.1$
- $V_{meas} \approx 258\ \text{MB}$
- → $\alpha = 0.43$, $\alpha N_{nzr} = 3.1$
- → RHS is loaded 3.1 times from memory
- and:

$$\frac{B_{CRS}^{DP}(\alpha)}{B_{CRS}^{DP}(1/N_{nzc})} = 1.15$$

15% extra traffic
→ optimization potential!

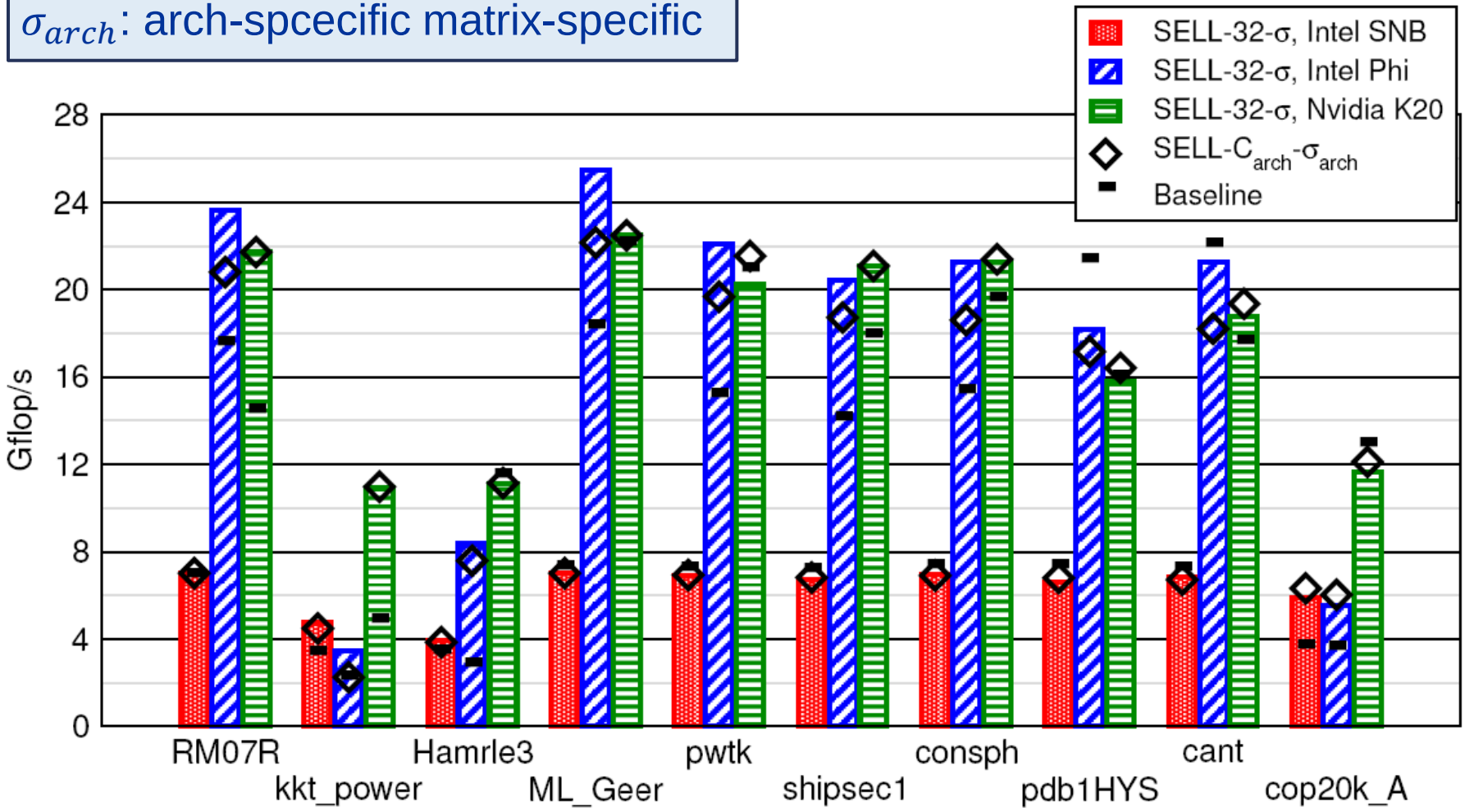
Impact of σ on α for kkt_power on SNB



Results for memory-bound matrices

σ: cross-arch matrix-specific

σ_{arch}: arch-spcecific matrix-specific

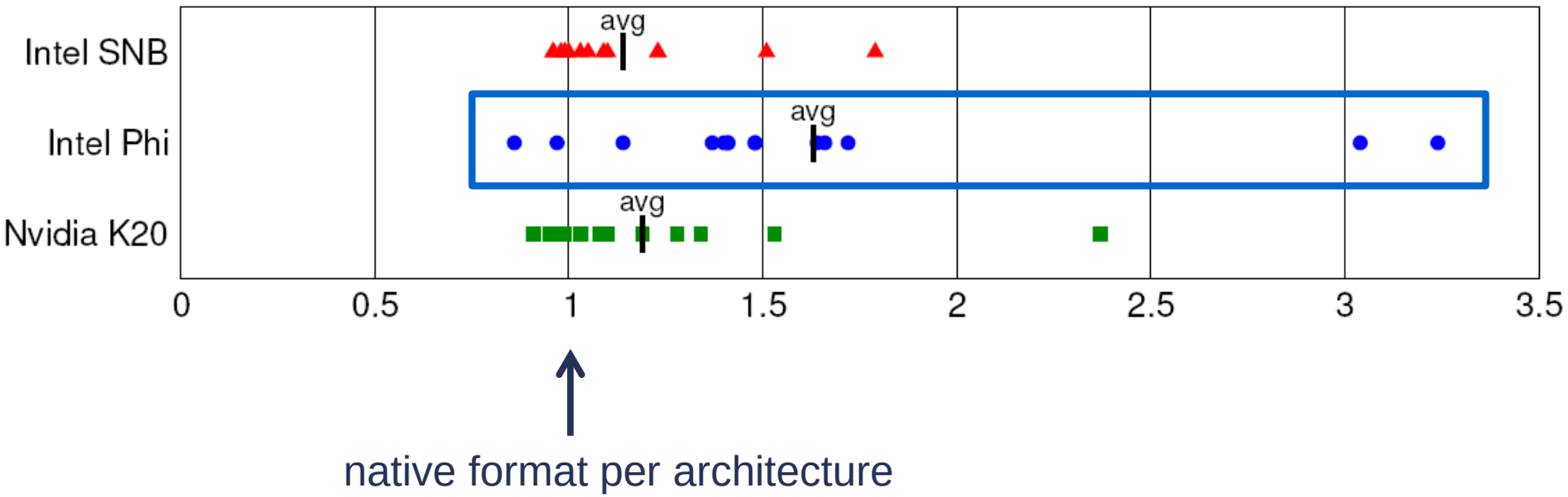


SELL-C-σ: Relative Performance Benefit

(16 square matrices)

Generic SELL-32-σ format performance vs. baselines:

- CRS (from MKL) for SNB and Xeon Phi
- HYB (CUSPARSE) for K20



Conclusion

- Wide SIMD/SIMT architectures pose challenges for spMVM
  - Short loops (CRS)
  - Fill-in (ELLPACK)
  - Reduction overhead (CRS)
  - Low vectorization ratio (CRS)
- SELL-C-σ alleviates or eliminates most of these problems
- SELL-C-σ provides good spMVM performance for a large range of sparse matrices compared to native formats
  - Intel SNB: Competitive with CRS
  - Intel Xeon Phi: Outperforms CRS by far on average
  - K20: Competitive with ELLPACK/HYB
- Architecture-specific parameter settings have small impact vs. generic values

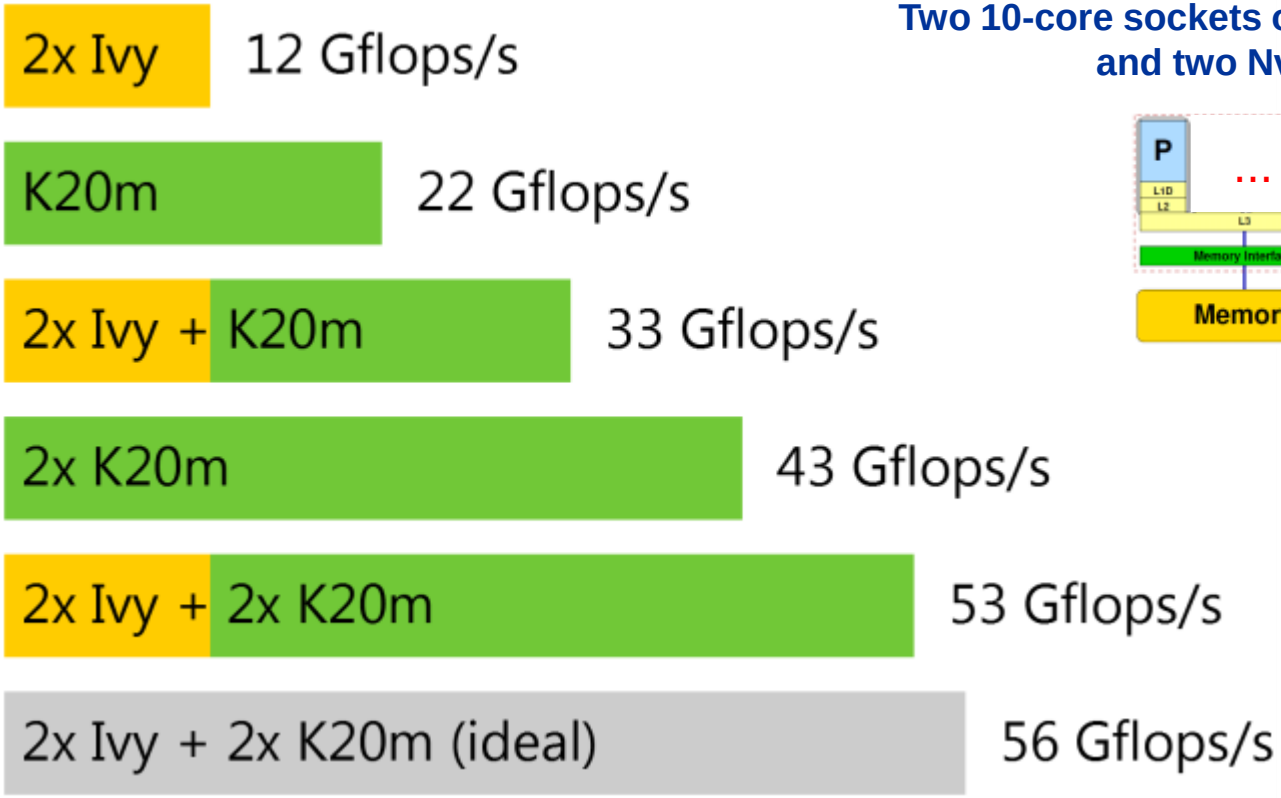
Outlook

- TODOs
  - More elaborate load balancing (better heuristics)
  - Explicit prefetching
- GATech variant to be in future MKL editions
- Upcoming (initial release for 2014):

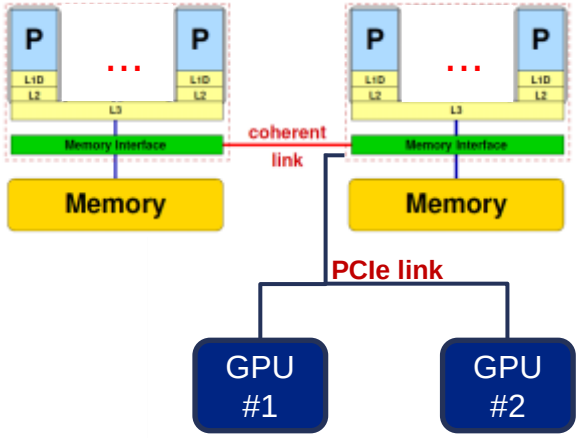
General
Hybrid
Optimized
Sparse
Toolkit

- Sparse building blocks
- Various matrix formats incl. SELL-C-σ
- Hybrid/heterogeneous MPI+X parallelism
- Communication hiding
- Built-in threading & tasking model

SpMVM Performance in a Heterogeneous System



Two 10-core sockets of Intel Xeon Ivy Bridge and two Nvidia Tesla K20m GPUs



(SELL-32-1, ML_Geer matrix, 64-bit values, 32-bit indices, ECC=1)

THANK YOU.

SELL-C-σ with IMCI intrinsics

```
int c, j, offs;
__m512d tmp1, tmp2, val, rhs;
__m512i idx;

#pragma omp parallel for schedule(runtime) private(j,offs,tmp1,tmp2,val,rhs,idx)
for (c=0; c<nRowsPadded>>4; c++)
{ // loop over chunks
  tmp1 = _mm512_load_pd(&_lhs[c<<4]  );     // load 8 LHS values
  tmp2 = _mm512_load_pd(&_lhs[c<<4+8]);     // load next 8 LHS values
  offs = cs[c]; // the initial offset is the start of this chunk

  for (j=0; j<cl[c]; j++)
  { // loop inside chunk from 0 to the length of the chunk
    val = _mm512_load_pd(&_val[offs]);       // load 8 matrix values
    idx = _mm512_load_epi32(&_col[offs]);    // load 16 indices
    rhs = _mm512_i32logather_pd(idx,_rhs,8); // gather RHS using lower 8 indices
    tmp1= _mm512_add_pd(tmp1,_mm512_mul_pd(val,rhs)); // multiply & accumulate
    offs+= 8;

    val = _mm512_load_pd(&_val[offs]);       // load next 8 matrix values
    idx = _mm512_permute4f128_epi32(idx,_MM_PERM_BADC); // lo <-> hi idx
    rhs = _mm512_i32logather_pd(idx,_rhs,8);// gather rhs lower 8 indices
    tmp2= _mm512_add_pd(tmp2,_mm512_mul_pd(val,rhs)); // multiply & accumulate
    offs += 8;
  }

  _mm512_store_pd(&_lhs[c<<4]  , tmp1);     // store 8 LHS values
  _mm512_store_pd(&_lhs[c<<4+8], tmp2);     // store next 8 LHS values
}
```

SELL-C-σ with AVX intrinsics

```
int c, j, offs;
__m256d tmp, val, rhs;
__m128d rhstmp;

#pragma omp parallel for schedule(runtime) private(j,offs,tmp,val,rhstmp)
for (c=0; c<nRowsPadded>>2; c++)
{ // loop over chunks
  tmp = _mm256_load_pd(&_lhs[c<<2]); // load 4 LHS values
  offs = cs[c]; // the initial offset is the start of this chunk

  for (j=0; j<cl[c]; j++)
  { // loop inside chunk from 0 to the length of the chunk
    val    = _mm256_load_pd(&_val[offs]);                // load 4 matrix values
    rhstmp = _mm_loadl_pd(rhstmp,&_rhs[_col[offs++]]);   // load 1st RHS value
    rhstmp = _mm_loadh_pd(rhstmp,&_rhs[_col[offs++]]);   // load 2nd RHS value
    rhs    = _mm256_insertf128_pd(rhs,rhstmp,0);         // insert lo part of RHS
    rhstmp = _mm_loadl_pd(rhstmp,&_rhs[_col[offs++]]);   // load 3rd RHS value
    rhstmp = _mm_loadh_pd(rhstmp,&_rhs[_col[offs++]]);   // load 4th RHS value
    rhs    = _mm256_insertf128_pd(rhs,rhstmp,1);         // insert hi part of RHS
    tmp    = _mm256_add_pd(tmp,_mm256_mul_pd(val,rhs)); // accumulate
  }

  _mm256_store_pd(&_lhs[c<<2], tmp); // store 4 LHS values
}
```

CRS spMVM performance

Best case B_{CRS} = 6 Byte/FLOP