Integrating multi-threading and accelerators into DUNE-ISTL

Dirk Ribbrock[1] Steffen Müthing[3] Markus Geveler[1]
Dominik Göddeke[2] Stefan Turek[1] Peter Bastian[3]

[1]Faculty of Mathematics, TU Dortmund

[2]Faculty of Mathematics and Physics, Stuttgart University

[3]Interdisciplinary Center for Scientific Computing, Heidelberg University

Sparse Solvers for Exascale
Greifswald, March 24, 2015

technische universität dortmund

Motivation

DUNE: Framework approach to software development

- integrated toolbox of simulation components
- exisiting body of complex applications
- good performance + scalability for traditional MPI model

Challenges

- utilise current hardware (MT CPU / Xeon Phi / GPU)
- provide ‘reasonable’ upgrade path for existing applications

Solution

- use existing ISTL interface for seamless integration
- exploit Dune’s various wrapper interfaces
- application only needs to change a few typedefs

Vectorisation and Parallelisation Approaches

General

- kernel templates for different SIMD sizes, data types and alignments
- Xeon Phi cards and GPU cards as 'independent' MPI nodes

CPU / Xeon Phi

- rely on auto-vectoriser in modern compilers
  - write easily vectorisable code
    (loop structure, alignment hints, fixed sizes)
  - verify compiler output
- Intel TBB for thread parallelism
  - fine-grained control over task scheduler
  - good C++ integration
  - more 'natural' than pragma-based OpenMP

GPU

- CUBLAS for BLAS-1 operations
- own CUDA kernels for more complex operations

Containers

Matrix format crucial for overall performance

- iterative solvers mostly consist of SpMV, preconditioners
- wide range of problems + AMG
  $\Rightarrow$ general purpose format required

Established formats

- CPU: (block) CSR, special-purpose (bands, symmetric, ...)
- GPU: ELL and variants (chunking, slicing, hybrid)
- Xeon Phi: (block) CSR, ELL

Problem

Matrix conversions expensive (time + memory)
Each format needs own assembly / SpMV / preconditioner kernels

SELL-C-σ as Cross-Platform Matrix Format

Kreutzer, Hager, Wellein, Fehske, Bishop 2013

'A unified sparse matrix data format for efficient general sparse matrix-vector multiply on modern processors with wide SIMD units'

Block-sorted and chunked ELL (SELL-C-σ) offers competitive performance across modern architectures (CPU, GPU, Xeon Phi)

- adopt SELL-C-σ as shared matrix format in DUNE (including assembly phase)
- ignore sorting σ for now (FEM matrices sufficiently regular)
- differentiate by memory domain (Host, GPU, Xeon Phi)
- memory domain and SIMD block size C via extended C++ allocator interface
- containers encapsulate device dependencies

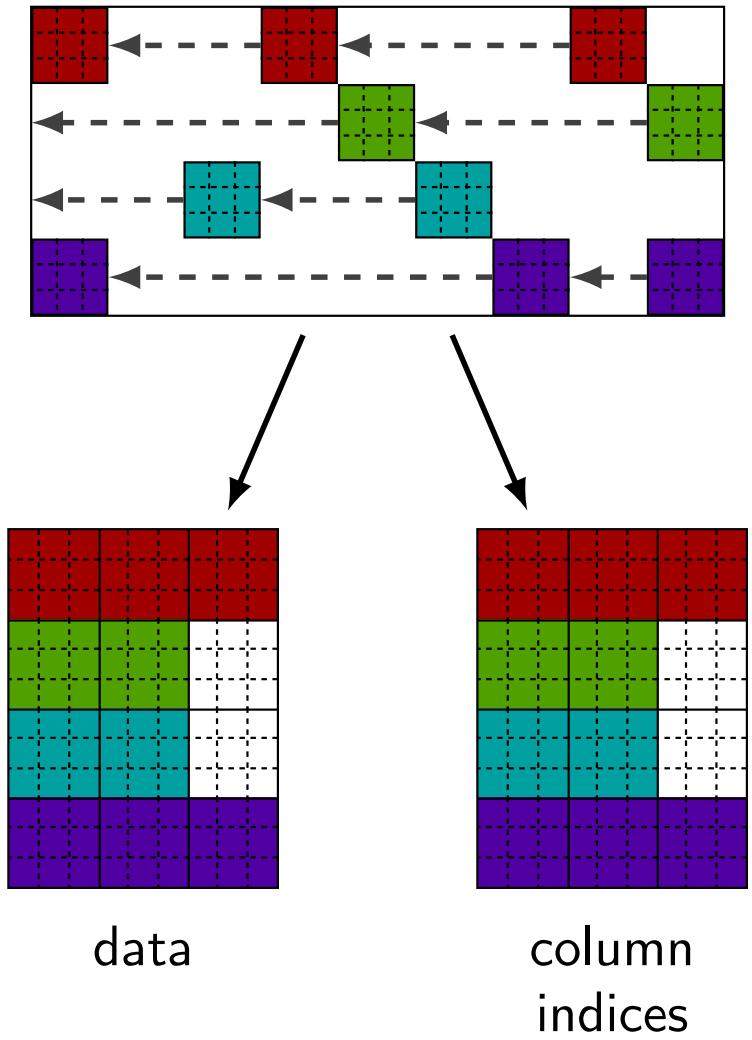
Discontinuous Galerkin and Block Matrices

Discontinuous Galerkin

- good for modern architectures
- well-suited for EXA-DUNE applications
- natural block structure

Implementation Wishlist

- store block pattern only
  $\Rightarrow$ Reduce memory footprint
- stay SIMD friendly
- block-structured algorithms (preconditioners etc.)



Discontinuous Galerkin and Block Matrices

Discontinuous Galerkin

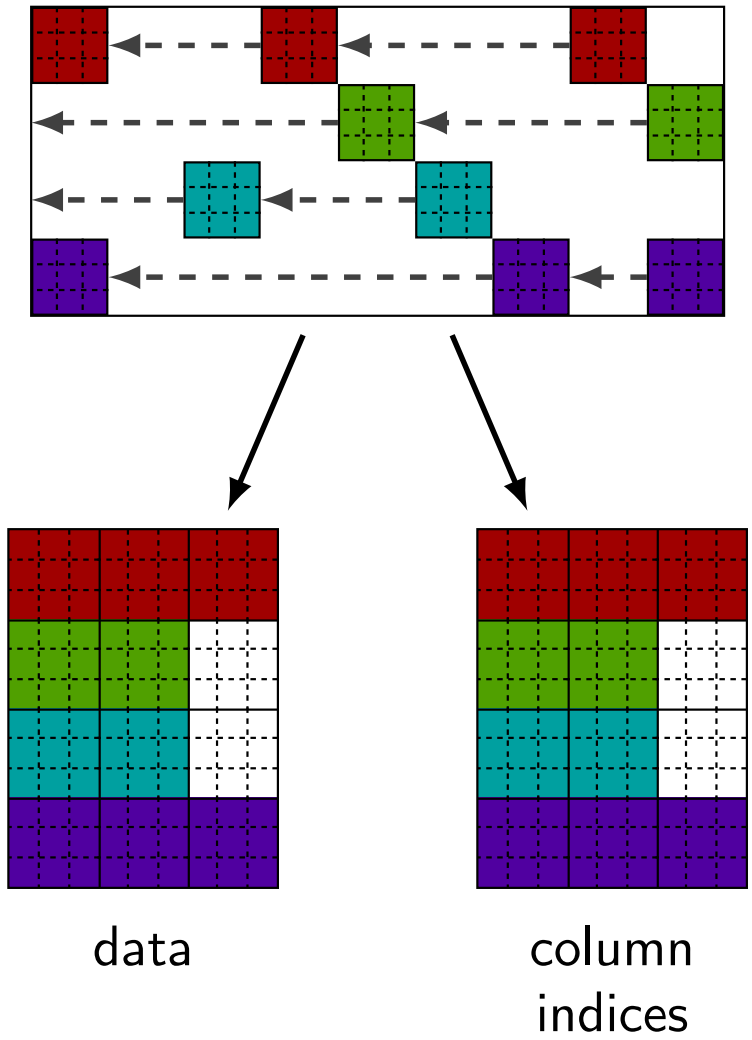
- good for modern architectures
- well-suited for EXA-DUNE applications
- natural block structure

Implementation Wishlist

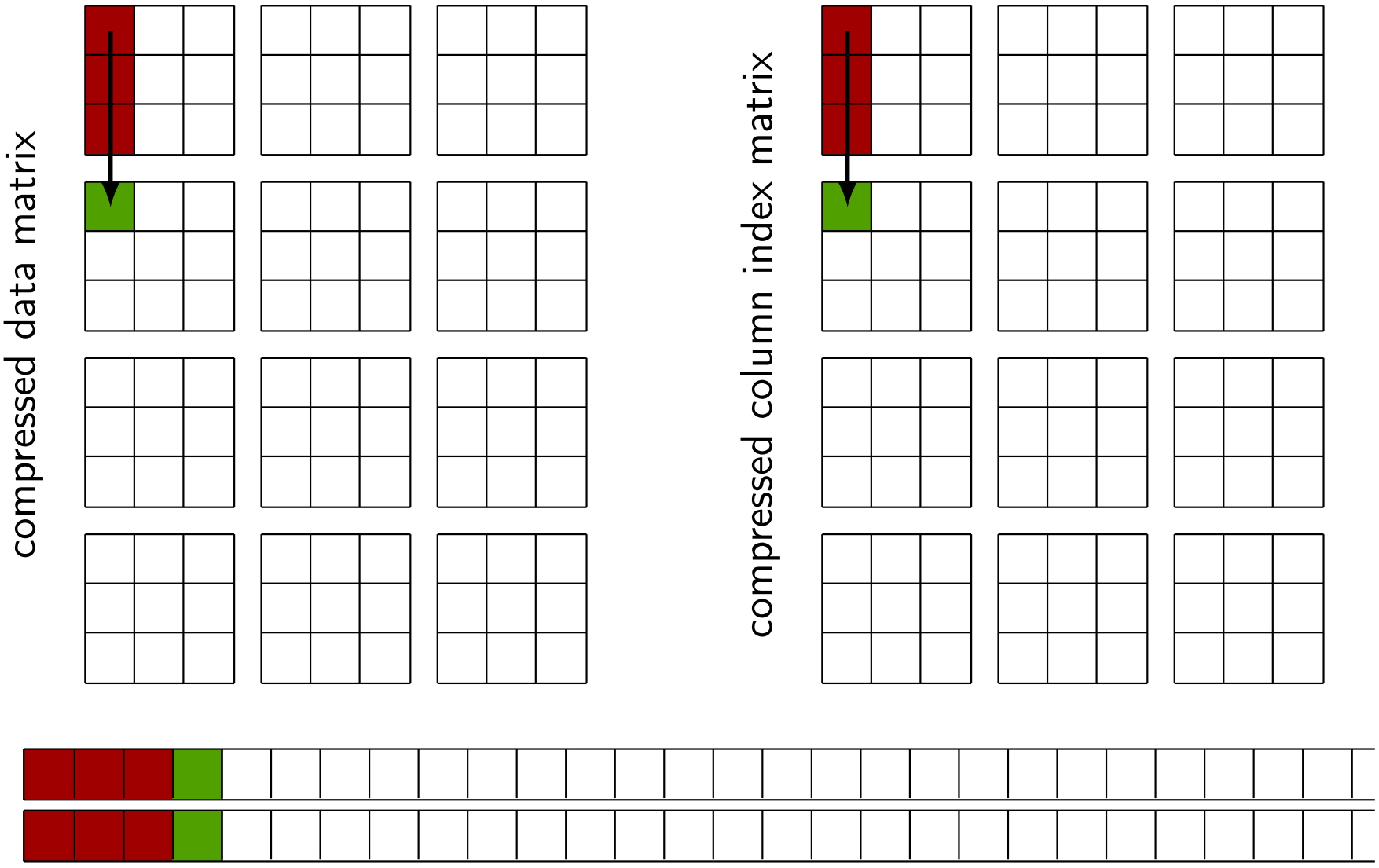
- store block pattern only
  $\Rightarrow$ Reduce memory footprint
- stay SIMD friendly
- block-structured algorithms (preconditioners etc.)

Problem

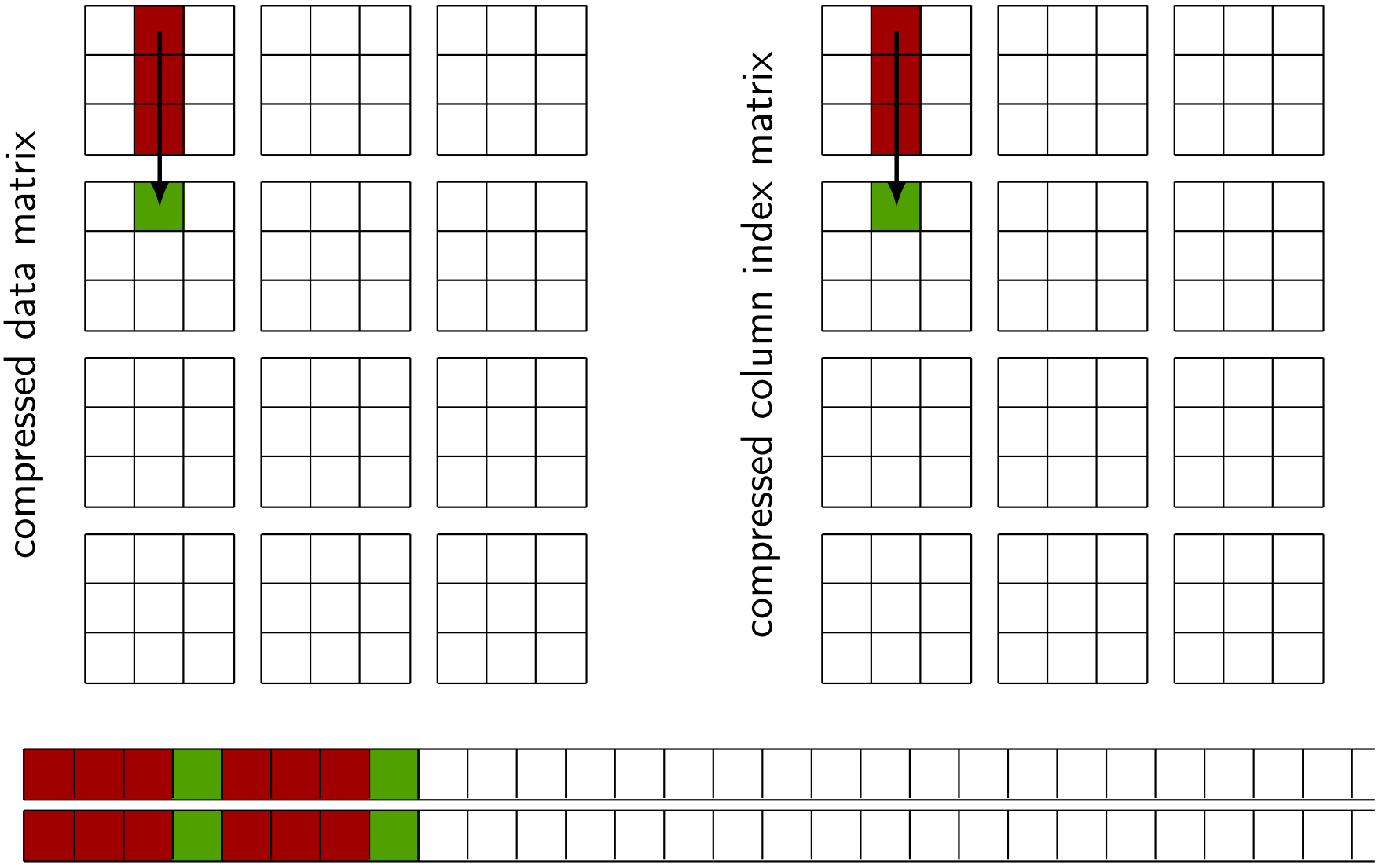
DG block size vs. SIMD chunk size



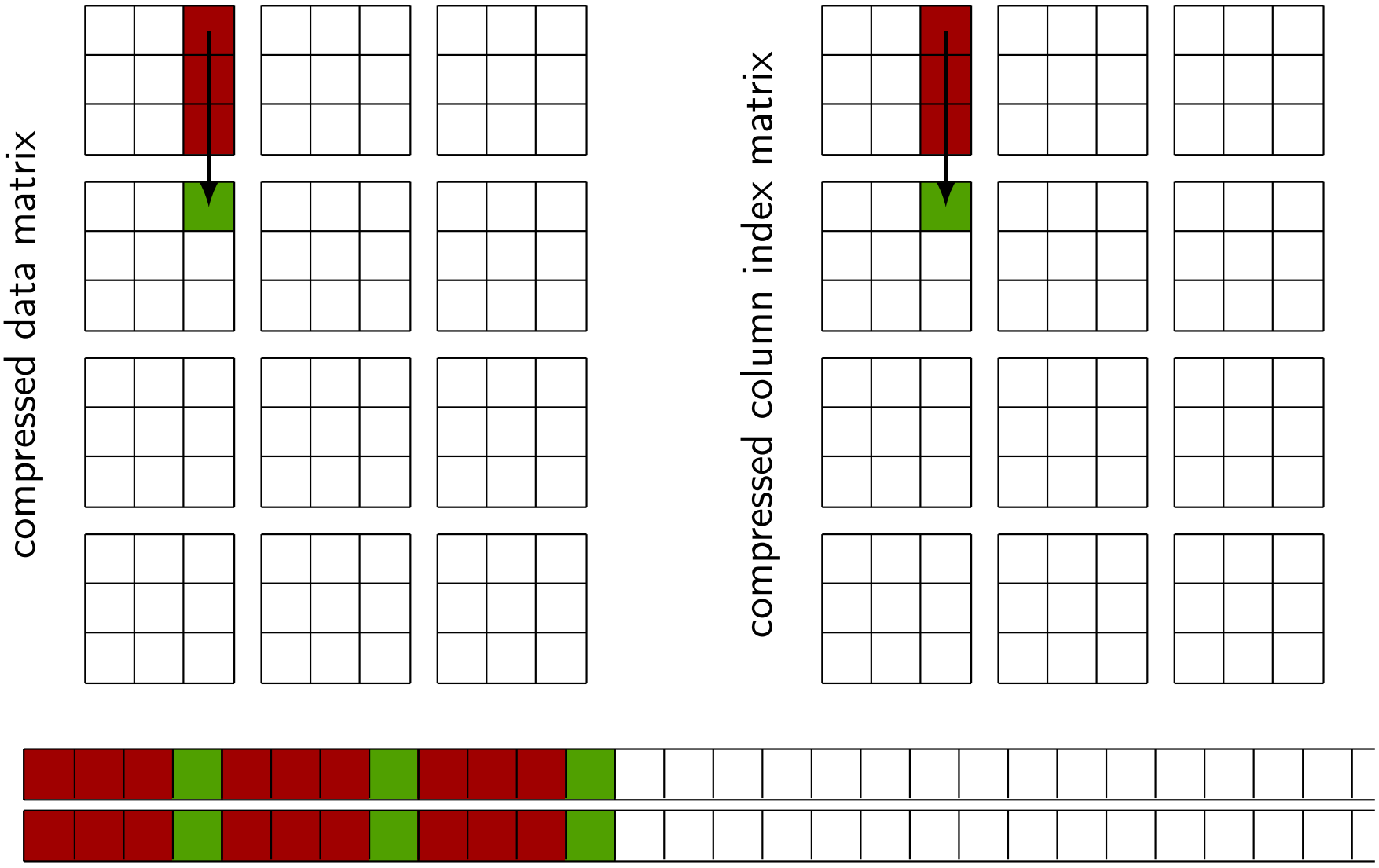
Standard SELL-C matrix storage ($n_b = 3$, $C = 4$)



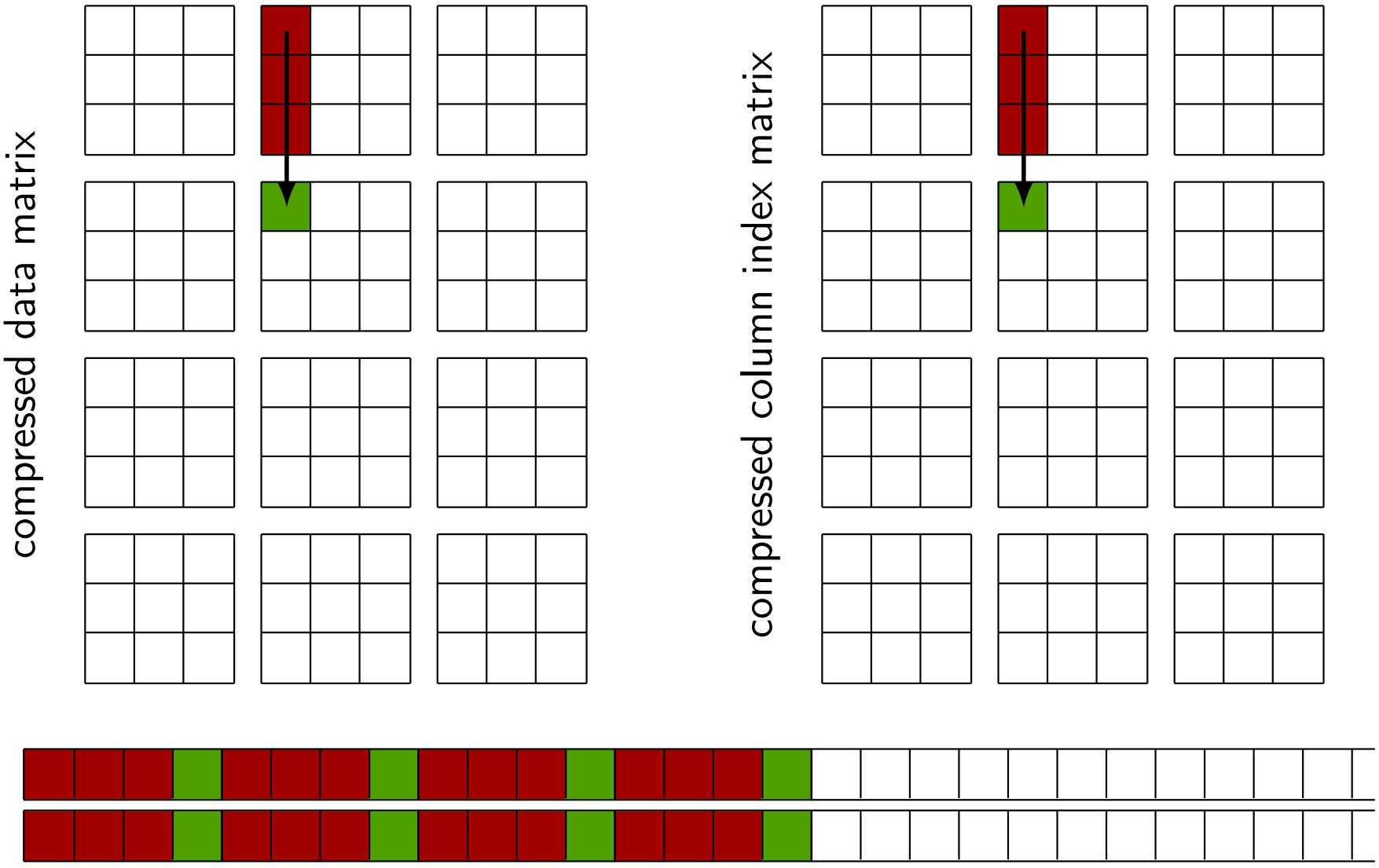
Standard SELL-C matrix storage ($n_b = 3$, $C = 4$)



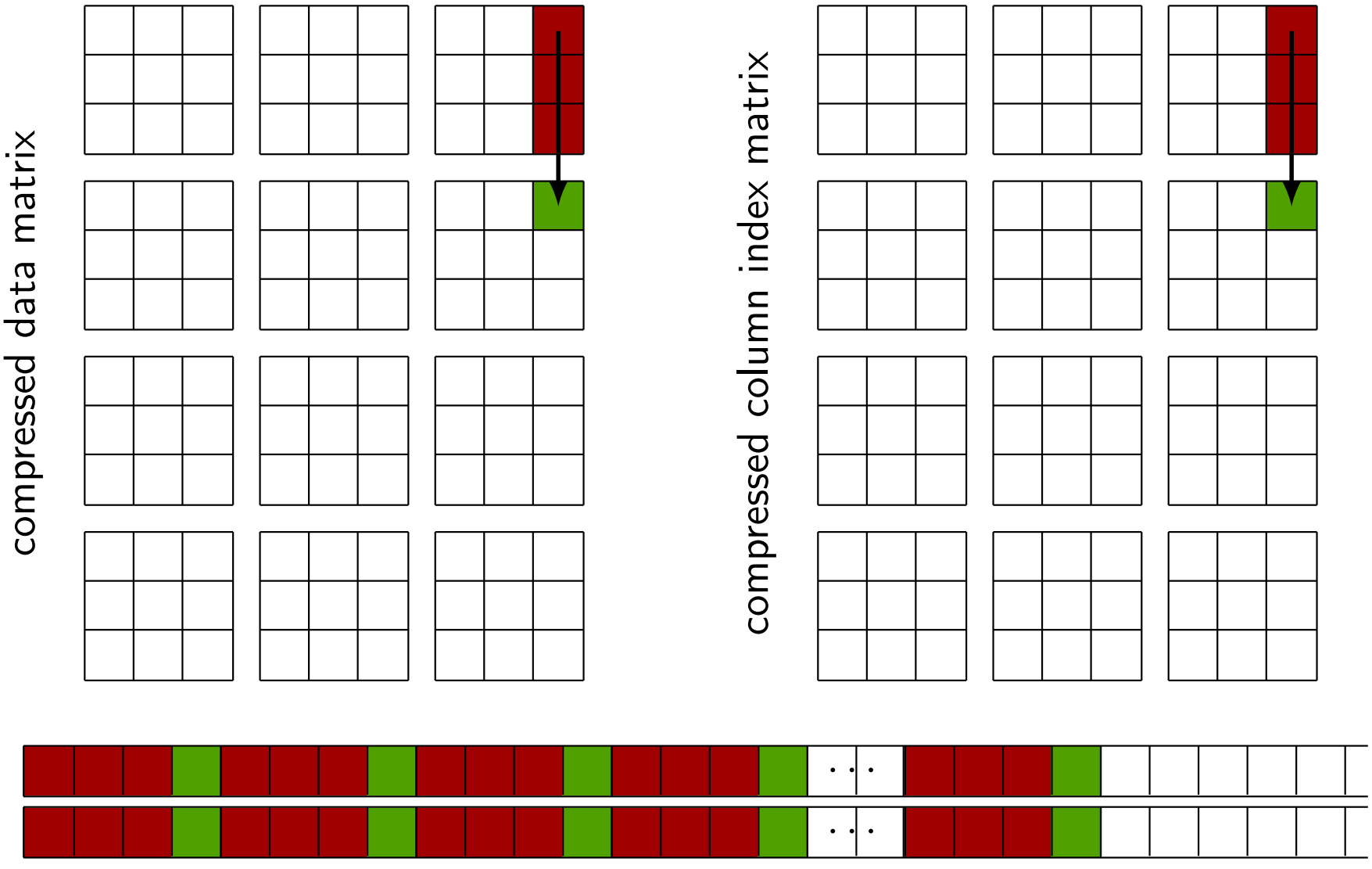
Standard SELL-C matrix storage ($n_b = 3$, $C = 4$)



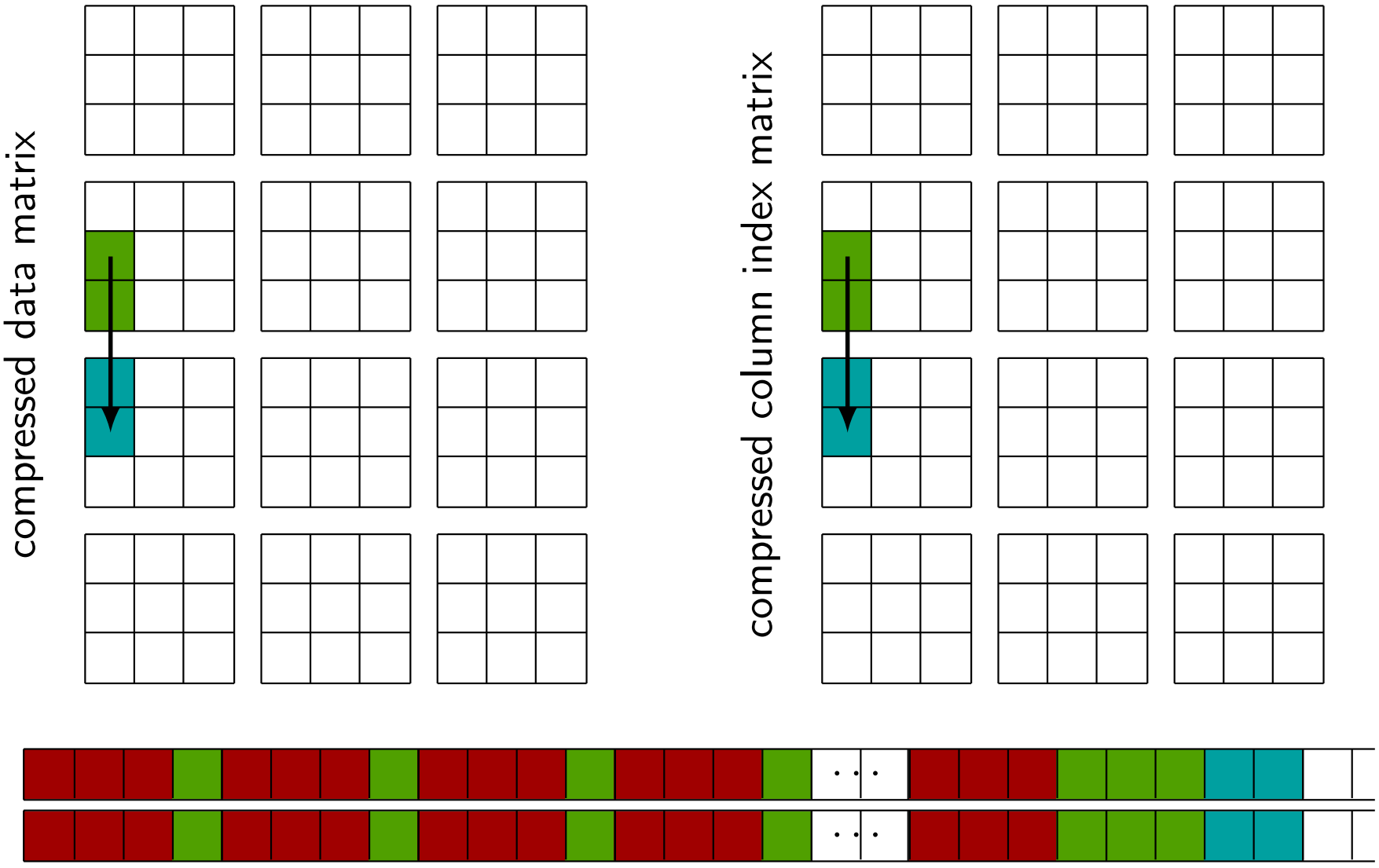
Standard SELL-C matrix storage ($n_b = 3$, $C = 4$)



Standard SELL-C matrix storage ($n_b = 3$, $C = 4$)



Standard SELL-C matrix storage ($n_b = 3$, $C = 4$)



Blocked ELL Formats

Blocked ELL format development driven by computer science

- typically as optimisation for externally provided data
- often additional preprocessing on matrix (reordering etc.)
- matrix parameters often coincide with hardware parameters
  e.g. block size typically multiple of SIMD size
- store blocks as dense matrices

$\Rightarrow$ Not usable for us, need to support arbitrary block sizes

Blocked ELL Formats

Blocked ELL format development driven by computer science

- typically as optimisation for externally provided data
- often additional preprocessing on matrix (reordering etc.)
- matrix parameters often coincide with hardware parameters
  e.g. block size typically multiple of SIMD size
- store blocks as dense matrices

$\Rightarrow$ Not usable for us, need to support arbitrary block sizes

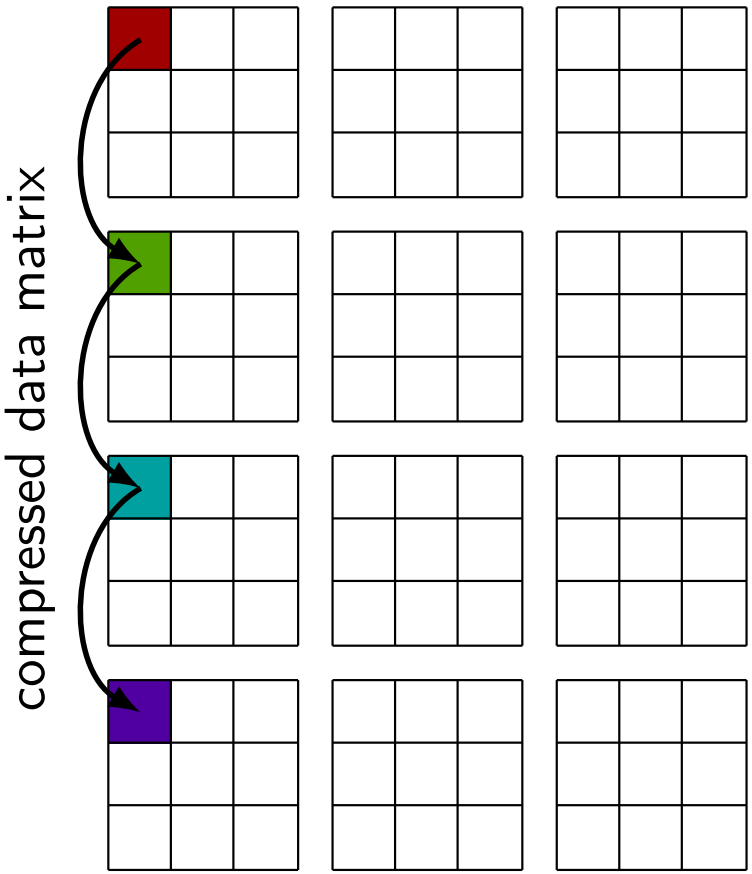
Choi, Singh, Vuduc '10

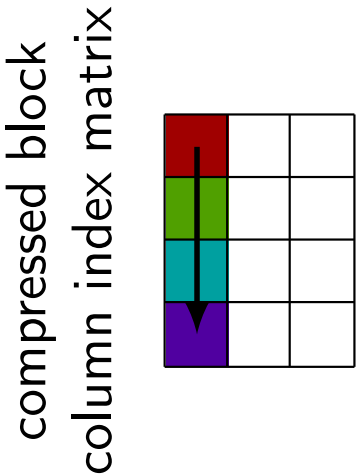
'Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs'

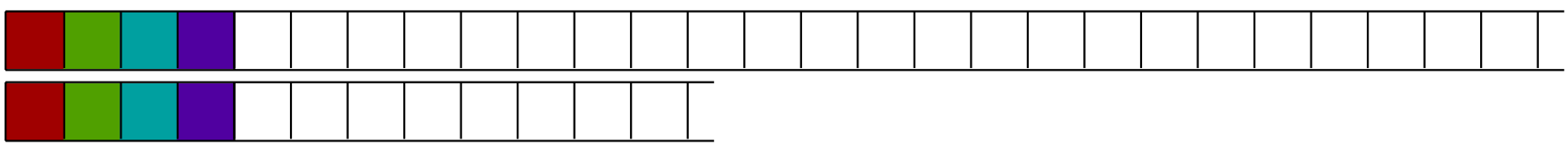
Interleaved storage of blocks from C block rows

- move SIMD from scalar level to block level
- vectorise algorithms by operating on *multiple independent blocks*

Blocked SELL-C matrix storage ($n_b = 3$, $C = 4$)

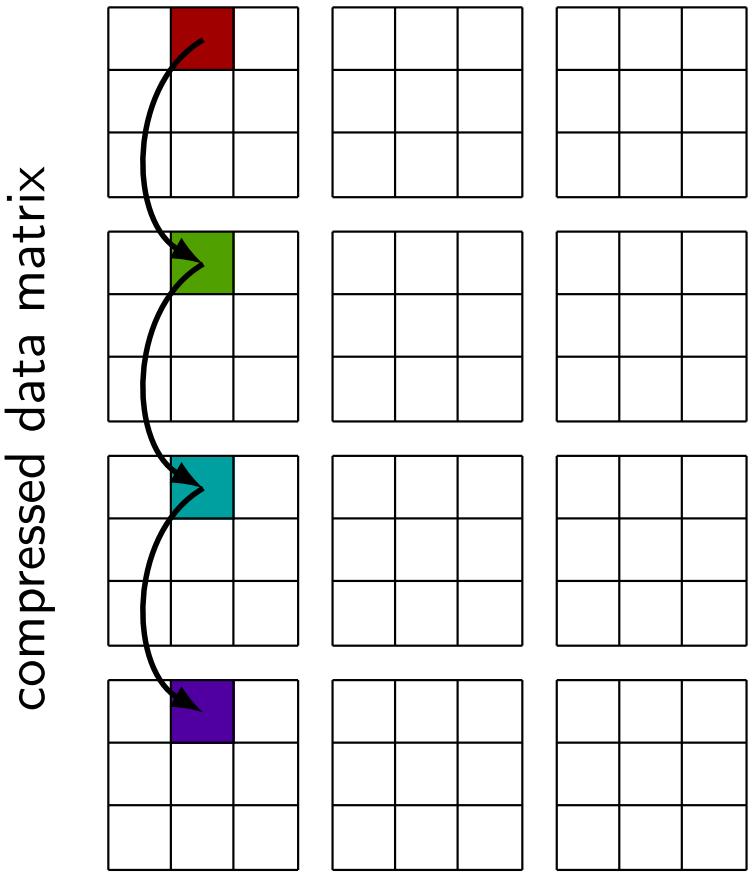


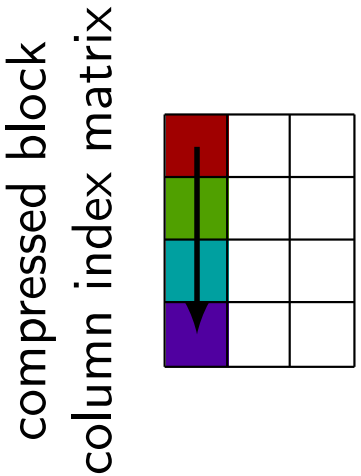


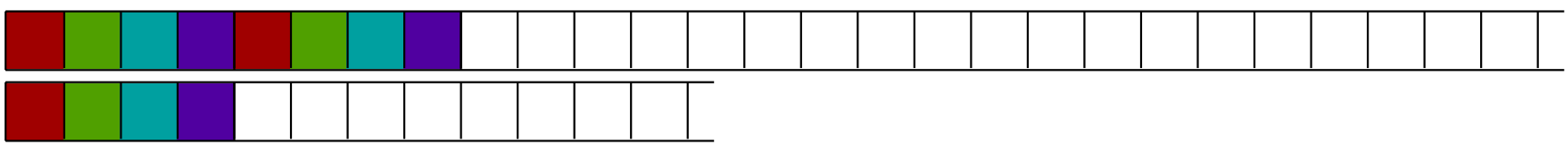


memory layout of data and column index arrays

Blocked SELL-C matrix storage ($n_b = 3$, $C = 4$)

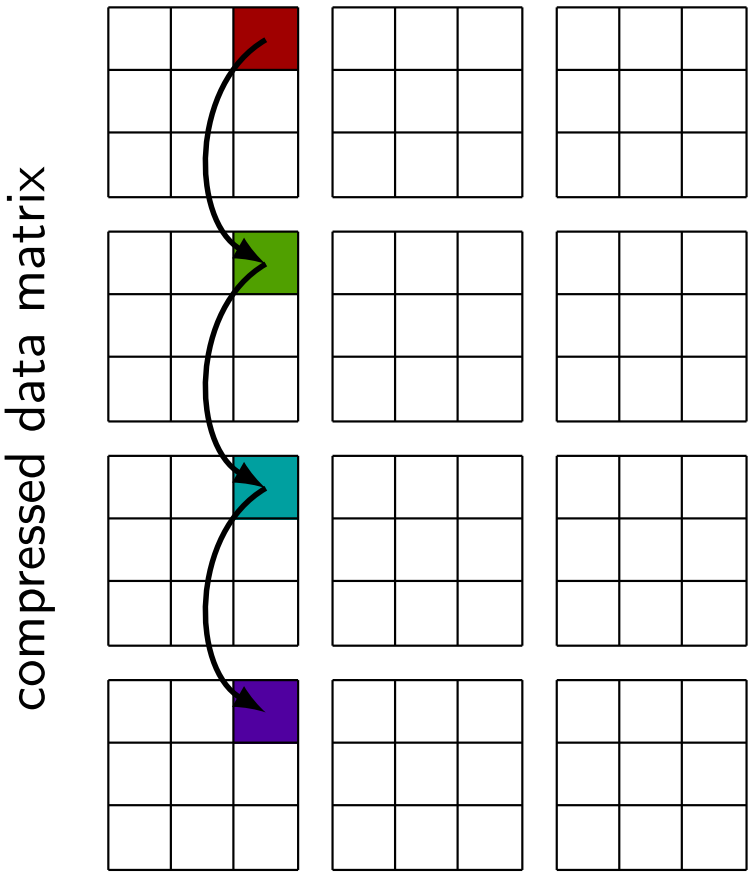


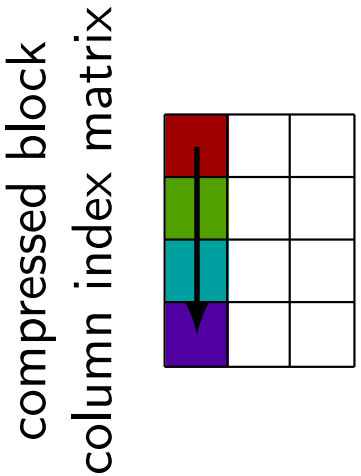


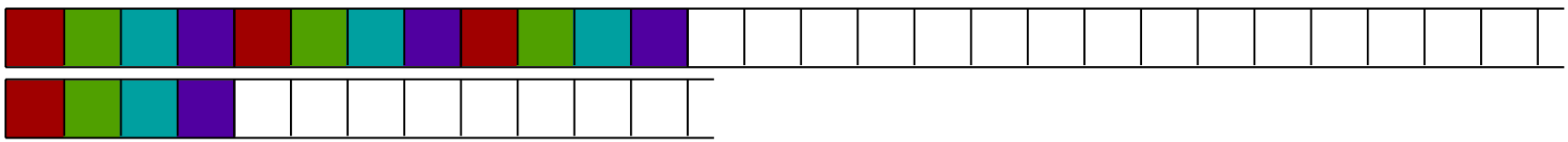


memory layout of data and column index arrays

Blocked SELL-C matrix storage ($n_b = 3$, $C = 4$)

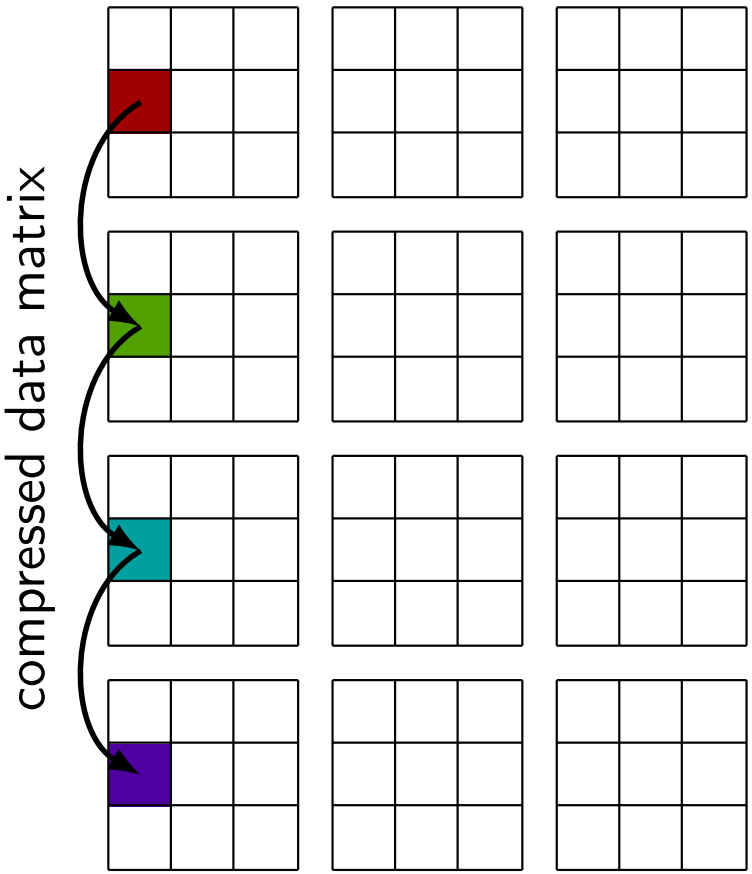


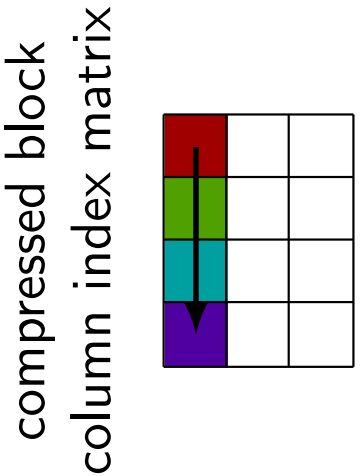




memory layout of data and column index arrays

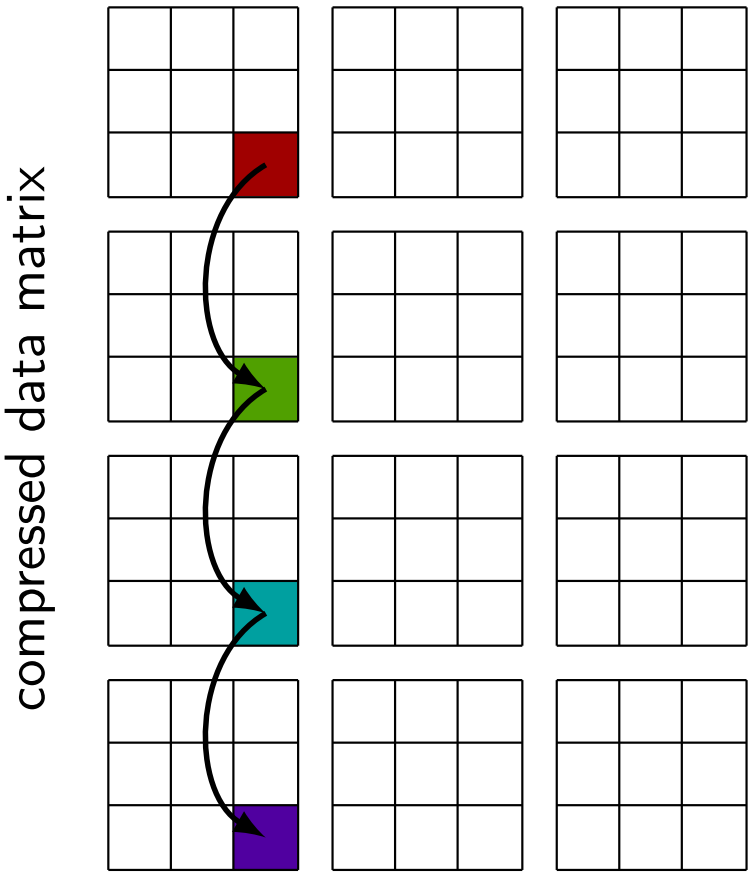
Blocked SELL-C matrix storage ($n_b = 3$, $C = 4$)

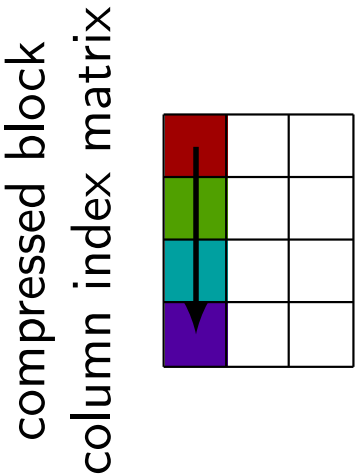




memory layout of data and column index arrays

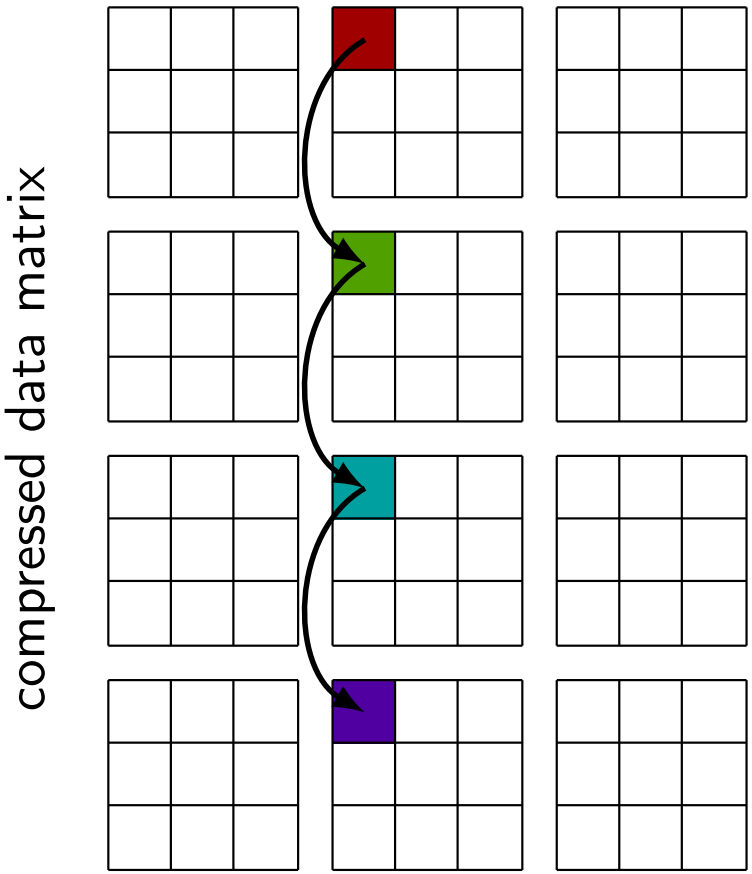
Blocked SELL-C matrix storage ($n_b = 3$, $C = 4$)

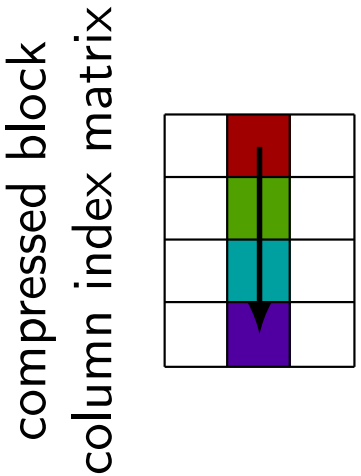




memory layout of data and column index arrays

Blocked SELL-C matrix storage ($n_b = 3$, $C = 4$)





memory layout of data and column index arrays

Blocked SELL Performance Gains - SpMV

Example: 8×8 dense DG Block Matrices

- SpMV is memory bound
- scalar effort:
  3 loads per matrix entry
- blocked SELL needs only $1/64$ of column indices
  $\Rightarrow$ Column index access negligible
- blocked effort:
  $\sim$ 2 loads per matrix entry

# DOF	scalar [s]	blocked [s]	speedup
884736	0.0260	0.0176	1.32
1404928	0.0414	0.0281	1.32
2097152	0.0621	0.0421	1.32
2985984	0.0882	0.0601	1.32
4096000	0.1204	0.0827	1.31
5451776	0.1610	0.1098	1.32
6229504	0.1836	0.1259	1.31

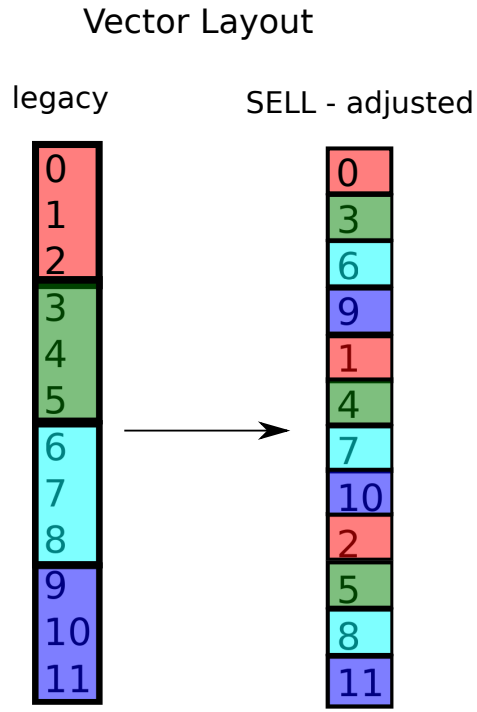
CPU: Intel Core i7 X 980 @ 3.33 GHz, 6 cores

What about the GPU ?

Memory access not optimal anymore

- Cuda thread layout: one thread per row
- recall Blocked SELL memory layout: consecutive threads work on different DG blocks
  $\Rightarrow$ Threads in on warp work on different DG blocks
- column index reuse complicated, ongoing effort
- reorder vector memory layout for coalesced memory access

$\Rightarrow$ alltogher no great performance gain over scalar SELL yet



Precompute diagonal block inversion

- cublasDgetrfBatched, cublasDgetriBatched are well suited for this
- preconditioner application is just a small SpMV

Preliminary GPU Results

- development version
- 8×8 dense DG block matrices
- does only work when DG / SELL / CUDA block sizes matching
- processing of rows exceeding warp/CUDA block size still buggy
- nevertheless promising results

# DOF	scalar [s]	blocked [s]	speedup
884736	0.00728148	0.00408948	1.44
1404928	0.011638	0.006572	1.44
2097152	0.0174008	0.00960892	1.45
2985984	0.0247102	0.0138912	1.44
4096000	0.0340097	0.0187993	1.45
5451776	0.0451971	0.0252429	1.44
6229504	0.0517102	0.0290875	1.44

GPU: Nvidia Tesla C2070, Fermi

ISTL: Solvers and Preconditioners

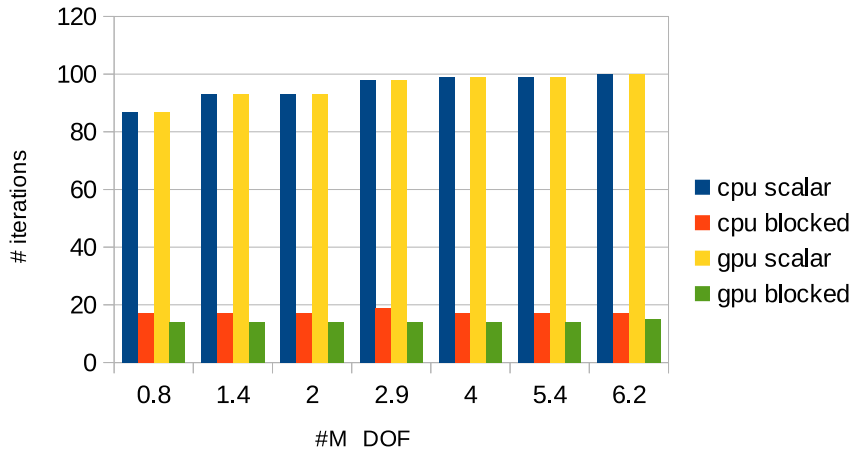
Leverage strengths of exisiting ISTL library

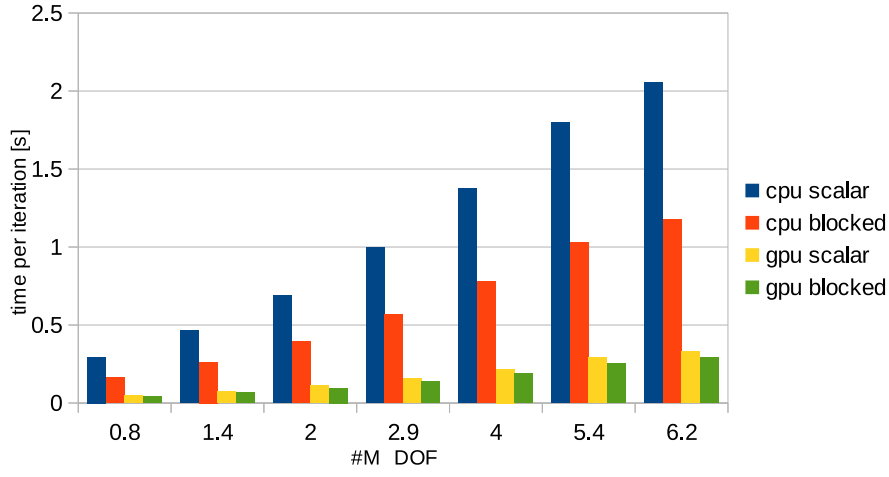
- strong separation of data structures and algorithms
- solvers defined in terms of abstract linear operators and generic containers

Implementation

- new vectors / matrices encapsulate data layout, kernel execution, parallelism
- new template specializations of data-aware components:
  - preconditioners
  - parallel components requiring MPI communication
- higher level components (Krylov solvers, Newton, ...) not modified
- mostly transparent to the user
  idea: change some typedefs to solve on GPU

PCG Solver Benchmark





- CG solver with point / block Jacobi
- 8×8 dense DG blocks
- CPU: Intel Core i7 X 980 @ 3.33 GHz, 6 cores
- GPU: Nvidia Tesla C2070, Fermi

Hybrid Parallelism – Optimum MPI node size

Comparison of different partitionings between MPI and multi-threading on a 48-core machine

h^{-1}	$t_{48/1}[s]$	$t_{8/6}[s]$	$\frac{t_{48/1}}{t_{8/6}}$	$t_{4/12}[s]$	$\frac{t_{48/1}}{t_{4/12}}$	$t_{1/48}[s]$	$\frac{t_{48/1}}{t_{1/48}}$
192	262.8	259.5	1.01	622.6	0.42	1695.0	0.16
256	645.1	600.2	1.07	1483.3	0.43	2491.7	0.26

- $t_{p/n}$: execution time with p MPI ranks and n threads each
- 4-socket AMD Opteron 6172, 8×6 core NUMA domains
- process / thread pinning to avoid task migration jitter
- massive performance breakdown when crossing the UMA domain size
- use multithreading within UMA domain and MPI across UMA domains

Summary

Lessons learned

- possible to (mostly) shield users from hardware complexity
- shared matrix format at framework level feasible / competitive
- nevertheless different optimisation techniques on different architectures necessary
- exploitation of DG block structure pays off
- UMA domain good criterion for MPI / threading decision

Still to do

- better preconditioners (AMG)
- efficient MPI-CUDA implementation

This work was funded in part by the German Research Foundation (DFG) under the priority programme 1648 "Software for Exascale Computing - SPPEXA"

Normalized pcg execution time