Exceptional service in the national interest

Sandia National Laboratories

Performance Portable Sparse Matrix-Matrix Multiplication for Modern Many-core Architectures

Mehmet Deveci,

Erik Boman, Siva Rajamanickam

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

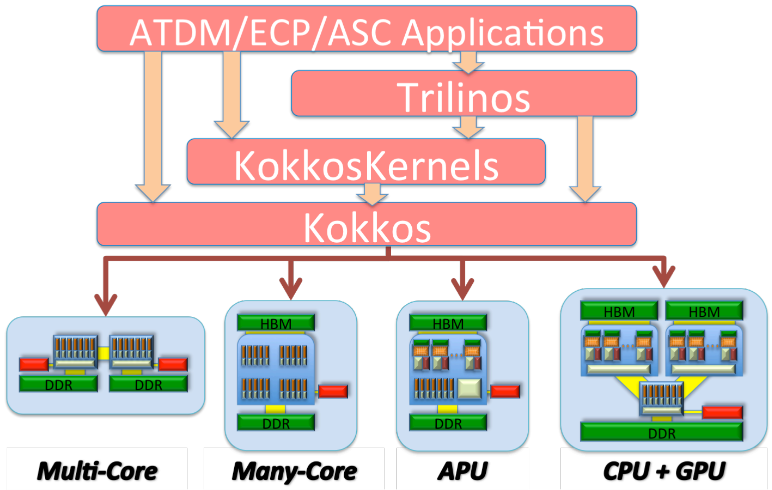
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Performance Portability

- Portability: Being able to run same code across various architectures
  - CPU, GPU, KNL
  - Performance portability
- Shift in architectures:
  - Threaded multi-core architectures
  - Many-core machines
  - GPUs: threads cannot be replaced by MPI ranks
  - More heterogeneity & diversity
    - Problem: $$$ spent on re-writing existing application codes
- Portable programming models
  - OpenCL, OpenACC, Kokkos
  - eliminate/separate the concerns of future architectures

Performance Portability

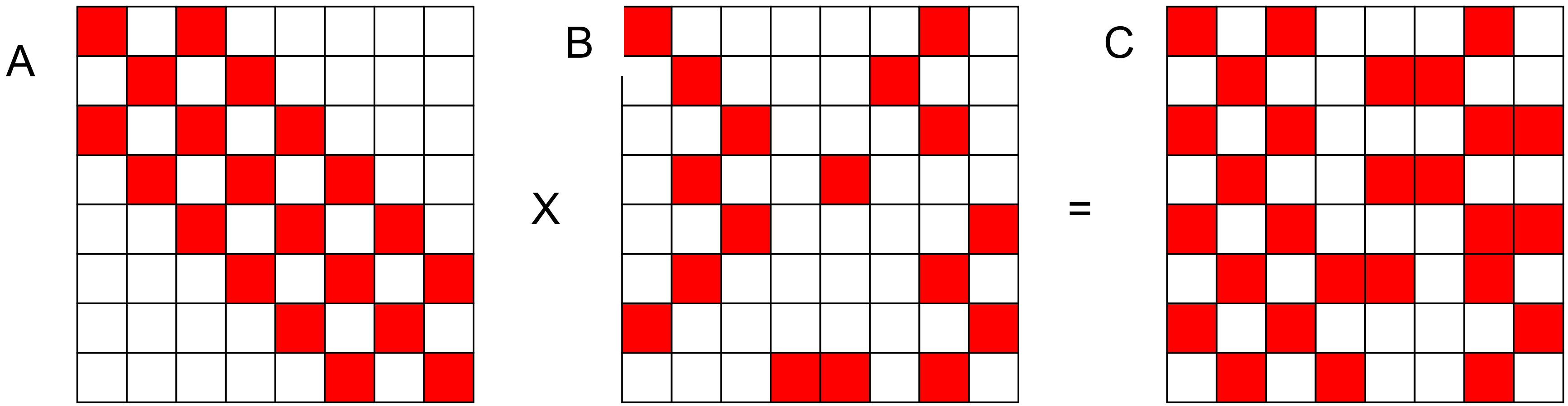
- Kokkos:
  - Layered collection of template C++ libraries
  - Manages data access patterns
  - Execution spaces, Memory spaces
- Kokkos provides tools for portability
  - Performance portability does not come for free.
  - Not trivial for sparse matrix and graph algorithms



- KokkosKernels:
  - Layer of performance-portable kernels
- We study design decisions for achieving portability for sparse matrix algorithms
  - In this work our application problem: SPGEMM

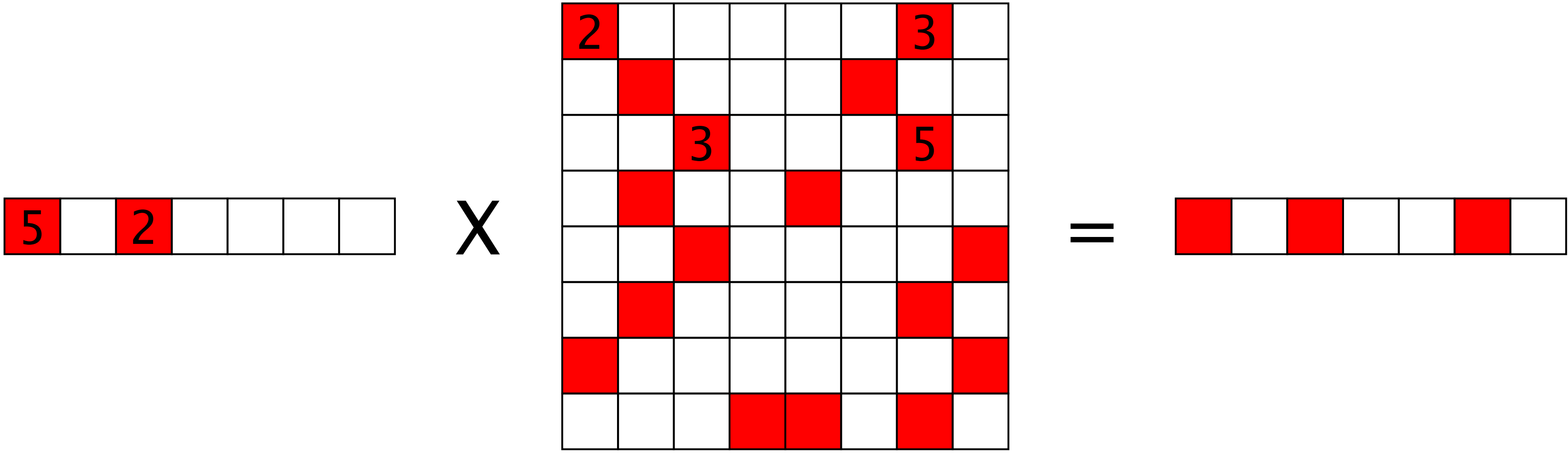
Sparse Matrix Matrix Multiplication

Sandia National Laboratories



- SPGEMM: fundamental block for
  - Algebraic multigrid $R \times A_{fine} \times P = A_{coarse}$
  - Various graph analytics problems: clustering, betweenness centrality…
- More complex than most of the other sparse BLAS and graph problems:
  - Extra irregularity: nnz of C is unknown beforehand.
  - Requires thread private data structures

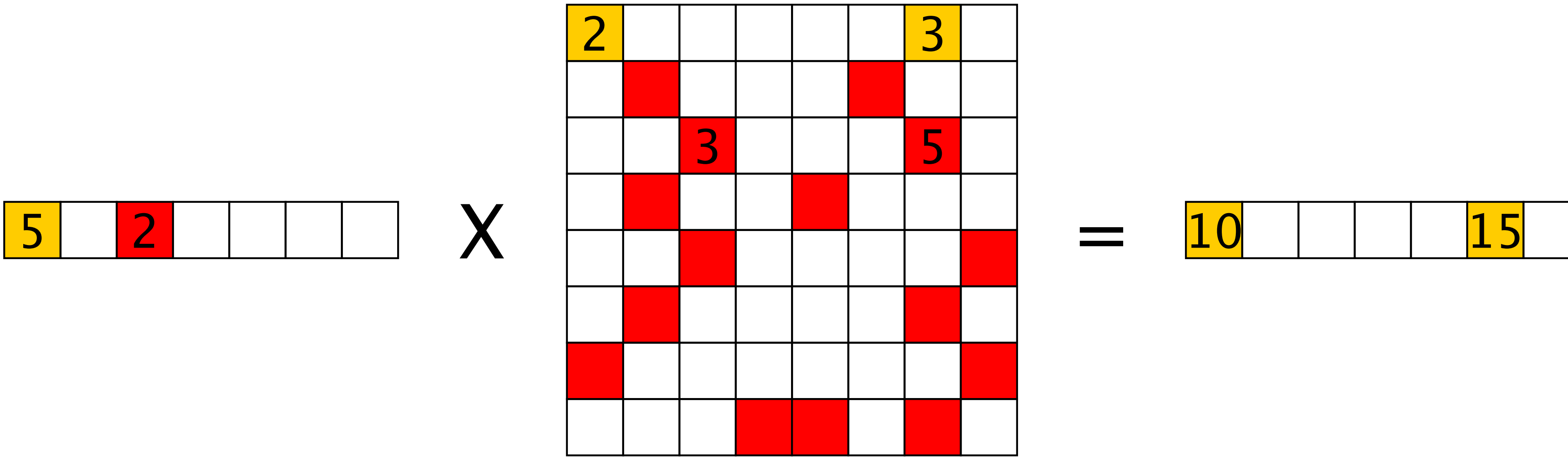
Background



- Sequential Algorithms: 1D [Gustavson 78]

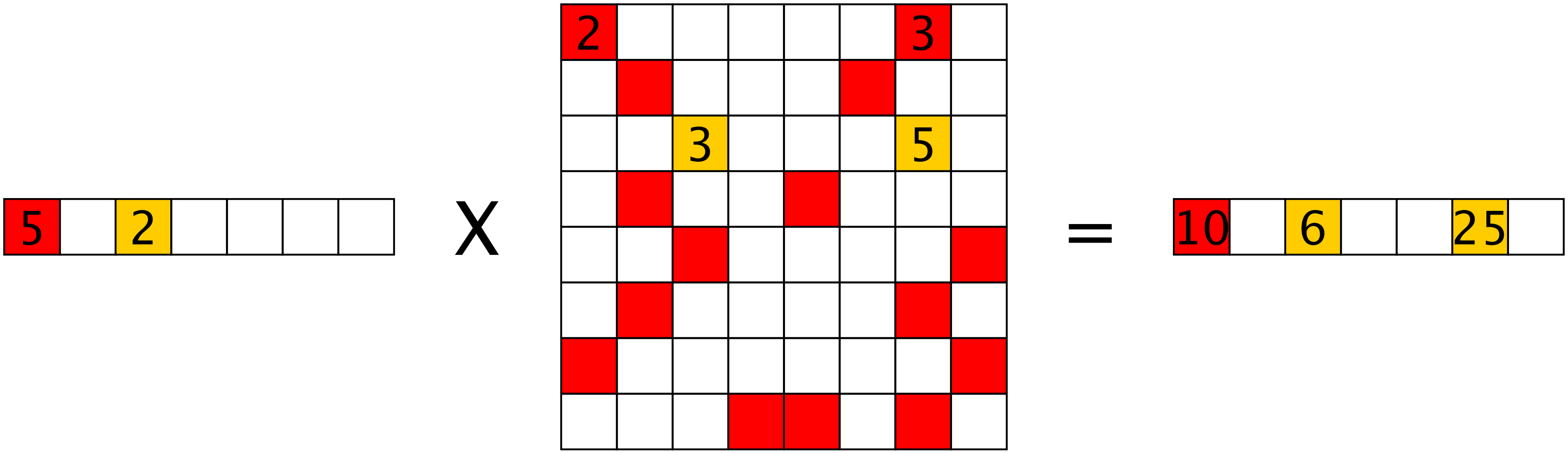
$$A(i,*) \times B = C(i,*)$$

Background



- Sequential Algorithms: 1D [Gustavson 78]

Background



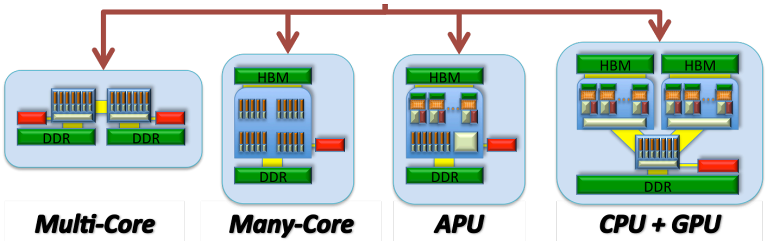
Background

- Distributed memory algorithms:
  - 1D Trilinos, 2D CombBLAS, 3D [Azad 15], Hypergraph-based: [Akbudak 14], [Ballard 16]
- Shared memory algorithms: based on 1D Gustavson algorithm
  - Differ in the data structure they use for accumulation
- Multi-threaded algorithms:
  - Dense Accumulator [Patwary 15]
  - Sparse Heap accumulators: ViennaCL, CommBlass
  - Sparse accumulators: MKL
- GPUs:
  - CUSP: Expand – Sort – Collapse
  - AmgX, cuSPARSE, bhSparse [Liu 14]

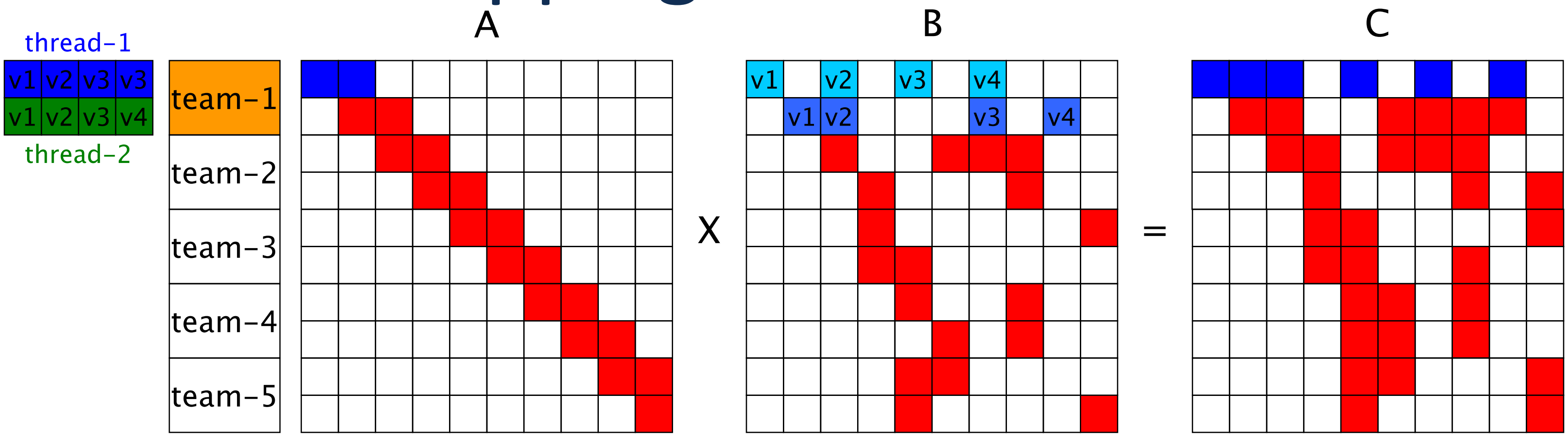
Portable SPGEMM: KKMEM

Sandia National Laboratories

- Variety in architectures
  - Tens/Hundreds/thousands of threads
  - CPUs/lightweight-cores/streaming multiprocessors (SPMD/SIMD)
  - Shared / high bandwidth / DDR memory
- Native multi-threaded algorithms
  - Fewer threads, more memory & more work per thread
- GPU algorithms
  - Thousands of threads, less memory & less work per thread
- Design decisions
  - Work distribution to threads
  - Scalable data structures
  - Limitations of specific architectures



Thread Mapping



- Each team works on a bunch of rows of C (or A)
  - Team: Thread block (GPU) group of hyper-threads in a core (CPU)
- Each worker in team works on consecutive rows of C
  - Worker: Warp (GPUs), hyperthread (CPU)
  - More coalesced access on GPUs,
  - Better L1-cache usage on CPUs.
- Each vectorlane in a worker works on a different multiplications within a row:
  - Vectorlane: Threads in a Warp (GPUs), vector units (CPU)

Data Structures

- Two-level Hashmap Accumulator:
  - 1st level accumulator: GPUs shared memory or a small memory that will fit in L1 cache
  - 2nd level goes to global memory
- Memory Pool: Only some of the workers need 2nd level hash map. They request memory from memory pool.
  - Allows threads scalable dynamic allocation on GPUs
  - Fixed size, fixed alignment

```
#pragma omp parallel
{
  data_type *my_data = new data_type[m];
  //initialize my_data ---> O(m)
  //once O(m) per thread
#pragma omp for
  for (i = 1…n){
    //work on my_data ---> O(k) and k << m
    //re-initialize my_data ----> O(k)
  }
}
```

Architecture Limitations

- Size and structure of rows are unknown at the beginning
  - over-allocation: expensive
  - dynamically increase: not suitable to GPUs
  - Estimation methods: not cheaper than calculating the actual size in practice
- Two-phase:
  - symbolic - calculate #nnz
  - then numeric - actual flops

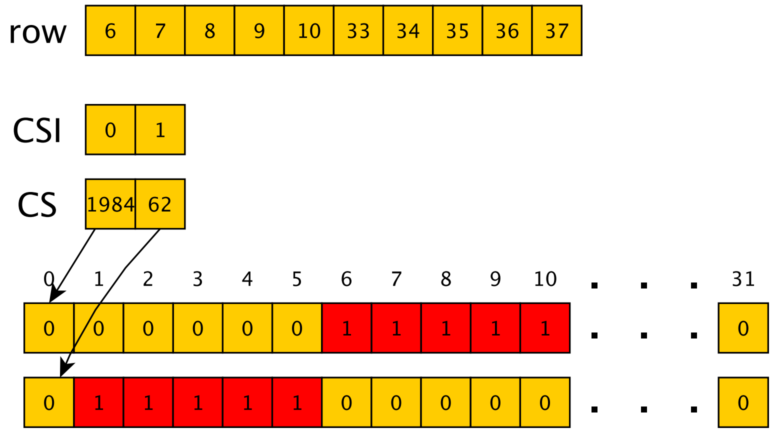
Require: A representing the input mesh, b right handside vector

```
//time step
for timestep ∈ [0, n] do
    X_0 ← initial guess
    //nonlinear solve
    for k ∈ [0, ...] until X_0 converges do
        A^k ← assemble_matrix (A, X_k) //linear matrix
        //calculate residual
        r_k ← b − A^k × X_k
        //solve problem - using multigrid
        Δ_{X_k} ← solve(A^k, r_k)
        //update the solution
        X_{k+1} ← X_k + Δ_{X_k}
```

- Repetitive multiplications for different numeric values with same symbolic structure

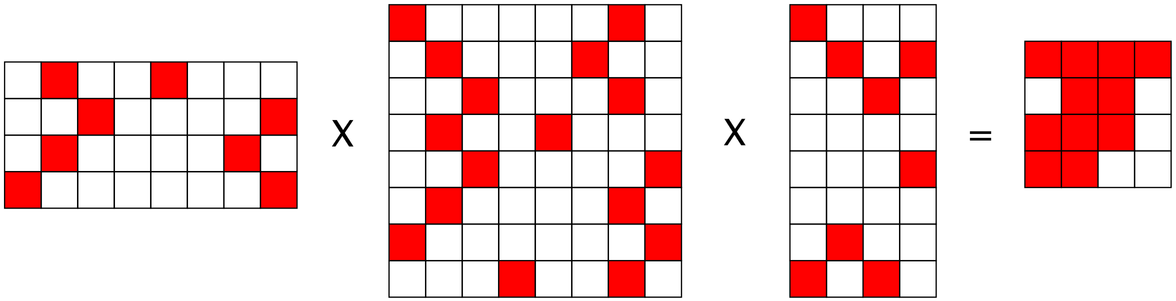
Two-Phase SpGEMM

- Doubles the amount of work performed
- Symbolic phase: works on the symbolic structure – no floating values
  - performs unions on rows to find the structure/size of the output row
  - compression method to speedup first phase and reduce its memory requirements
- Compression: Compress the rows of B: O(nnz(B)) using 2 integers.
  - Column Set Index (CSI): represents column set index
  - Column Set (CS): the bits represent the existence of a column
- Advantages:
  - Symbolic complexity: O(FLOPS) -> on average ~O(avgdeg(A)x nnz(B))
  - How much memory we need is unknown and locally-overestimated as max row flops



Experiments

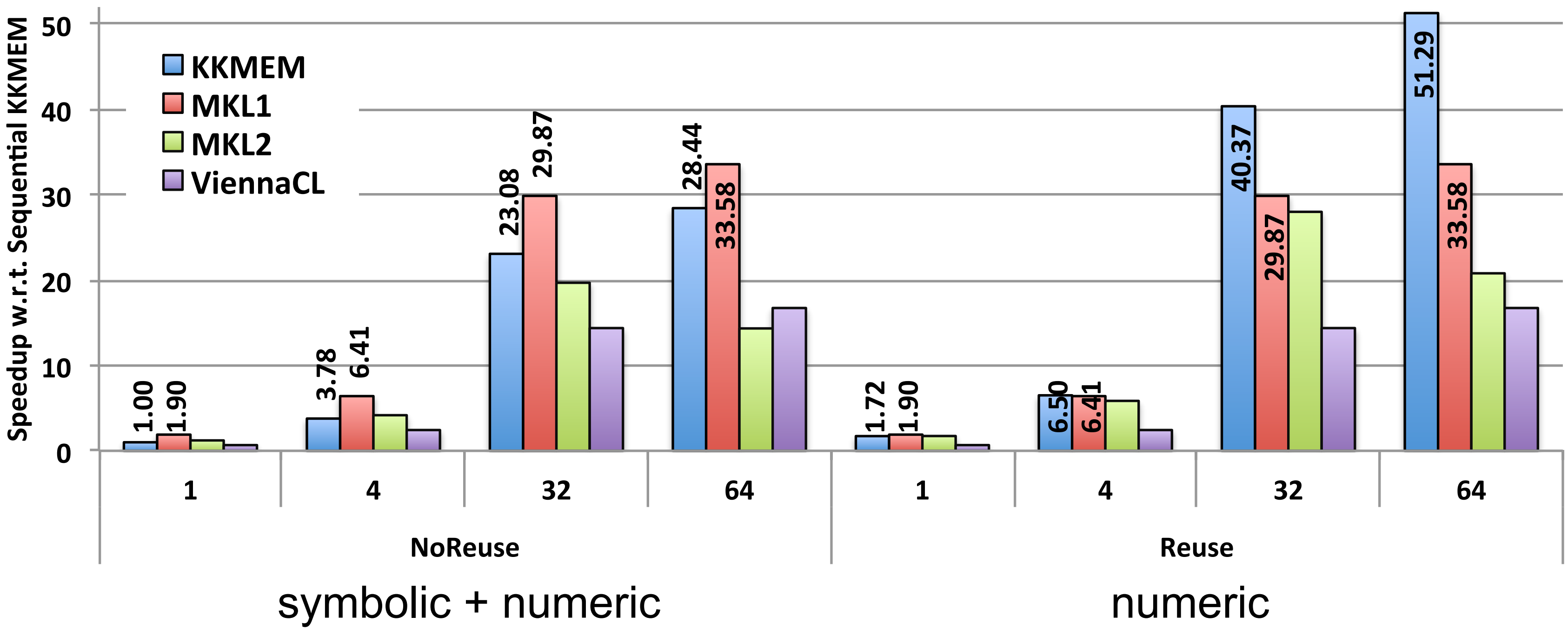
- Experiments on Haswell CPUs, KNLs, GPUs
  - Haswell: 2 sockets x 16 cores x 2 hyperthreads - 2:30 GHz
  - KNL: 68 cores x 4 hyperthreads - 1.40 GHz
    - 16 Gb HBW MCDRAM (476.2 GB/s)
    - 96 GB DDR4 (84.3 GB/s)
  - GPUs: Pascal P100 CC 6.0
- Multigrid multiplications → $A_{coarse} = R_{restriction} \times A_{fine} \times P_{prolongation}$



- Some matrices used in the literature for AxA

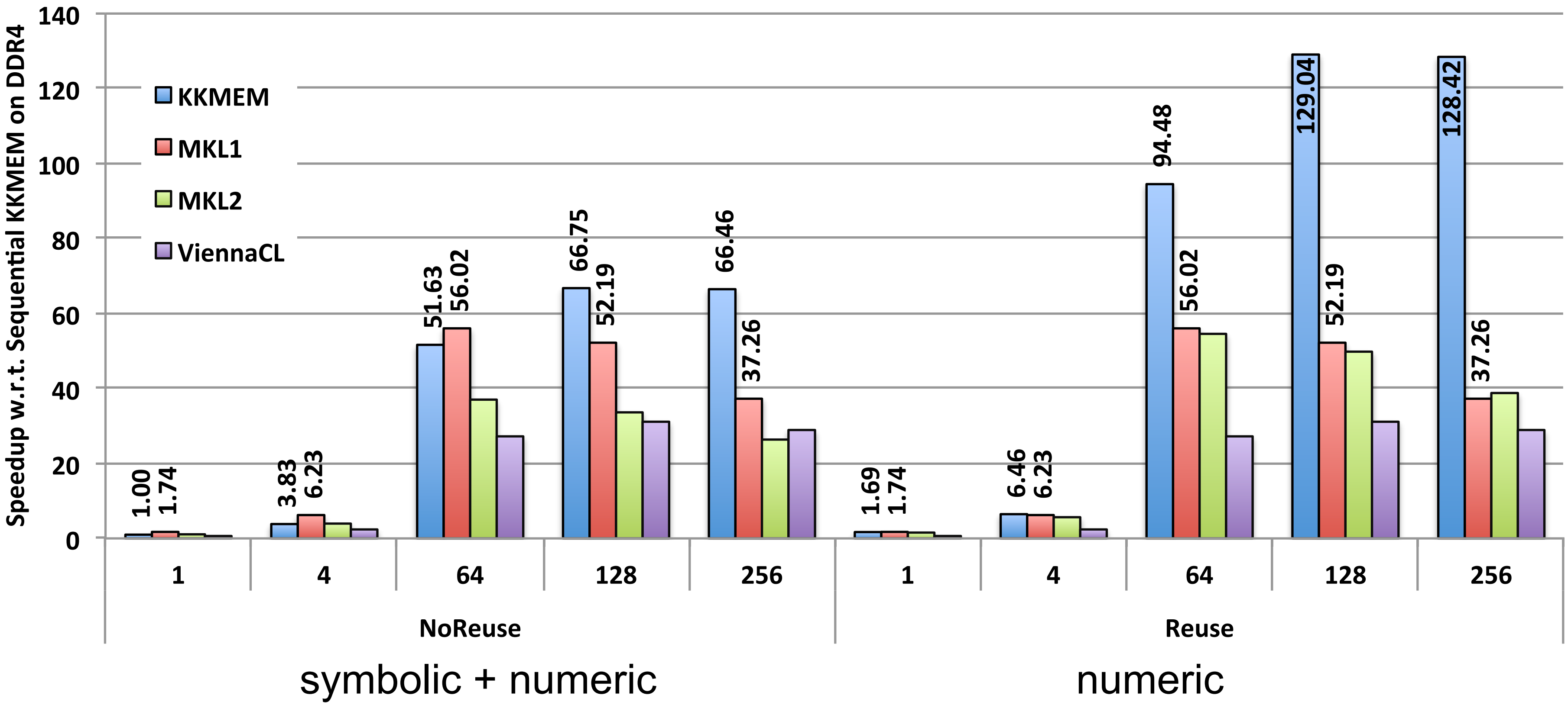
Haswell

- Geometric mean of 20 multiplications: 8 AxA, 12 multigrid
- Compared against 2 OpenMP methods in MKL and 1 in ViennaCL



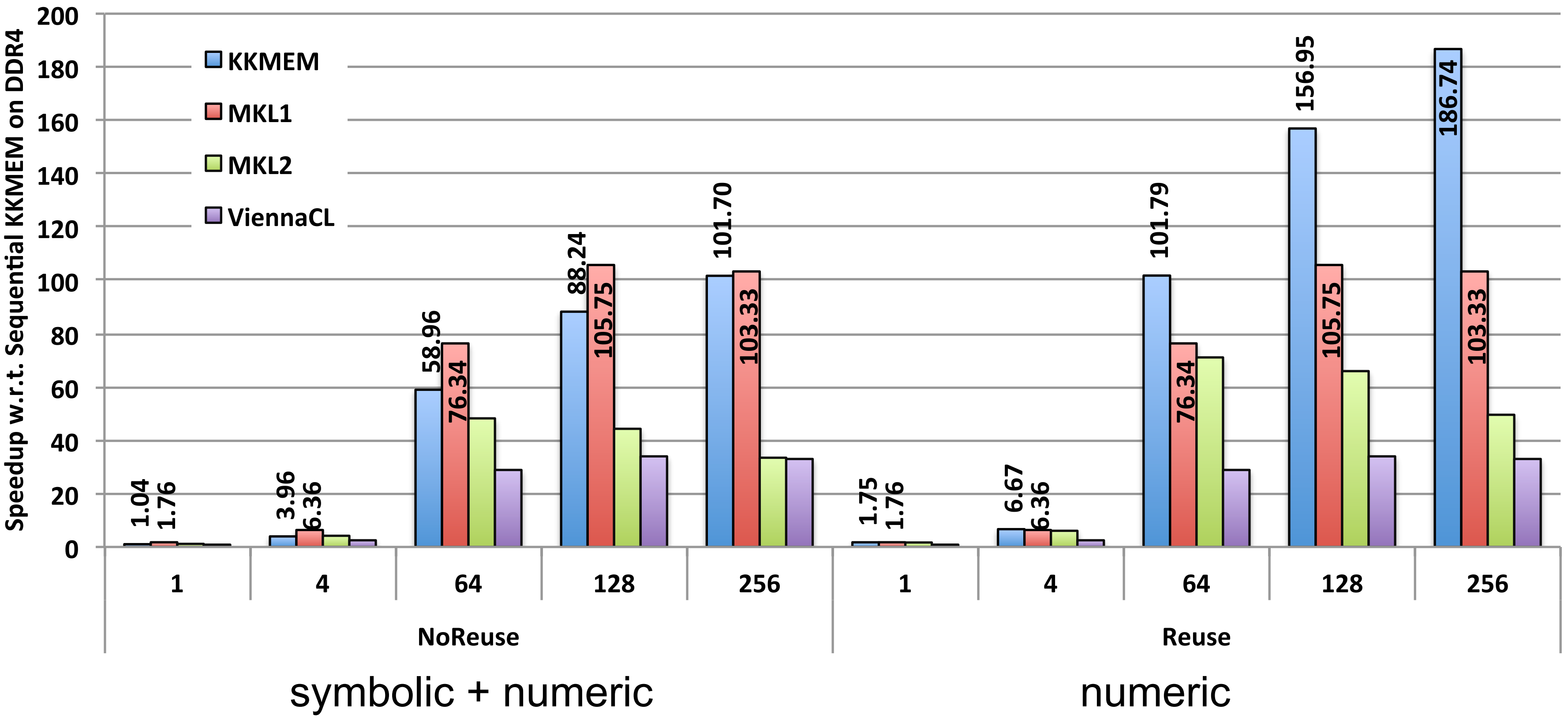
KKMEM uses less memory → data is more localized → less likely to have memory bandwidth problems → better thread scalability.

KNL – DDR4



MKL has issues with bandwitdh/latency earlier. KKMEM becomes faster after 64 threads.

KNL - MCDRAM



More bandwidth improves MKL's performance, but still hits bandwidth bound on 128 threads. KKMEM scales there.

Pascal P100 GPUs

- Compared against CUSP, bhSPARSE, ViennaCL, cuSPARSE
- Best performance on 17 matrices
- CUSP, bhSPARSE, ViennaCL runs out of memory 19, 8, and 4 matrices.

	CUSP	bhSPARSE	ViennaCL	cuSPARSE
2cubes_sphere	4.54	1.20	1.06	3.62
cage12	3.13	0.75	1.22	2.74
webbase	0.66	0.54	5.18	2.30
offshore	5.25	1.33	1.21	7.08
filter3D	5.78	0.83	1.47	4.30
hugebubbles20_0	4.99	4.81	1.94	12.14
Europe	3.41	5.57	2.57	2.50
cant	12.83	1.05	1.42	0.77
hood	14.22	0.97	1.77	1.72
pwtk	17.88	1.13	2.06	1.53
Empire_R_AP		0.89	0.65	0.88
Empire_RA_P		1.03	0.41	0.68
Laplace_R_A		0.68	0.73	2.71
Laplace_A_P		2.57	1.00	11.65
Laplace_R_AP		2.36	1.24	5.24
Laplace_RA_P		1.67	0.65	3.32
Brick_R_A		1.16	1.82	4.91
Empire_R_A		1.09	1.06	1.11
Empire_A_P		3.60	1.05	1.48
Brick_RA_P		1.26	0.43	1.14
ldoor		1.09	1.88	1.76
delaunay_n24			1.74	1.12
Brick_R_AP			0.76	1.91
channel			1.51	3.10
Brick_AP			0.95	4.54
cage15				4.86
Bump				1.58
audi				1.54
dielFilterV3real				1.85
Geomean:	5.25	1.36	1.22	2.43

Compression & Overall Results

- Memory required by accumulators
  - Average: by 53 %
  - Max : by 96 %
- # Insertions
  - Average: by 59 %
  - Max : by 91 %
- Overall geometric mean of the execution times

	KNL-DDR4	KNL-MCDRAM	Haswell	Pascal
Best Method	0.790	0.477	0.362	0.342
KKMEM	0.676	0.480	0.455	0.328
	1.17x	99%	80%	1.04x

Conclusions & Future Work

- How much performance will be sacrificed for portability?
  - We do not sacrifice much in terms of performance on highly-threaded architectures
- The key to performance portability:
  - thread scalable data structures
  - efficient memory (locality) use
  - correct thread hierarchy mapping
- The data structures and compression: major in performance and robustness
- Designing for application use cases such as the reuse
  - significantly better performance than past methods

For more information

- KokkosKernels:
  - Download through Trilinos: http://trilinos.org
  - Public git repository: http://github.com/trilinos
  - Public git repository: http://github.com/kokkos
- For more information:
  - mndevec@sandia.gov
- Thanks to:
  - NNSA ASC program
  - DOE ASCR SciDAC FASTMath Institute
  - ATDM

References

- F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition," ACM Transactions on Mathematical Software (TOMS), vol. 4, no. 3, pp. 250{269, 1978.
- Buluç, Aydin, and John R. Gilbert. "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments." *SIAM Journal on Scientific Computing* 34.4 (2012): C170-C191.
- Azad, Ariful, et al. "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication." *arXiv preprint arXiv:1510.00844* (2015).
- Akbudak, Kadir, and Cevdet Aykanat. "Simultaneous Input and Output Matrix Partitioning for Outer-Product--Parallel Sparse Matrix-Matrix Multiplication." *SIAM Journal on Scientific Computing* 36.5 (2014): C568-C590.
- Ballard, Grey, et al. "Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication." *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 2015.
- Patwary, Md Mostofa Ali, et al. "Parallel efficient sparse matrix-matrix multiplication on multicore platforms." *International Conference on High Performance Computing*. Springer International Publishing, 2015.
- Liu, Weifeng, and Brian Vinter. "An efficient GPU general sparse matrix-matrix multiplication for irregular data." *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014.