FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Performance Engineering of the Kernel Polynomial Method on Large-Scale CPU-GPU Systems

Moritz Kreutzer*, Georg Hager*, Gerhard Wellein*,
Andreas Pieper#, Andreas Alvermann#, Holger Fehske#

*: Erlangen Regional Computing Center, Friedrich-Alexander University of Erlangen-Nuremberg
#: Institute of Physics, Ernst Moritz Arndt University of Greifswald

29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)
May 26, 2015
Hyderabad, India

This work was supported (in part) by the German Research Foundation (DFG) through the Priority Programs 1648 “Software for Exascale Computing” under project ESSEX and 1459 “Graphene”.

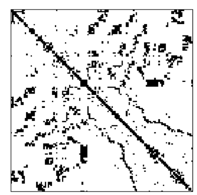
Prologue

What is this about?

The Kernel Polynomial Method (KPM)

Approximate the complete eigenvalue spectrum of a large sparse matrix.

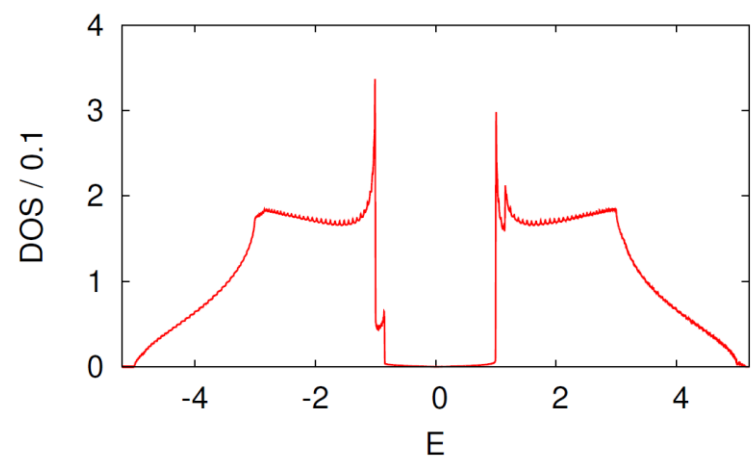
Large, Sparse



$$\boldsymbol{H}\,\boldsymbol{x} = \lambda\,\boldsymbol{x}$$

$$\underbrace{\{\lambda_1, \lambda_2, \dots, \lambda_k, \dots, \lambda_{n-1}, \lambda_n\}}$$

Good approximation to full spectrum
(e.g. Density of States)



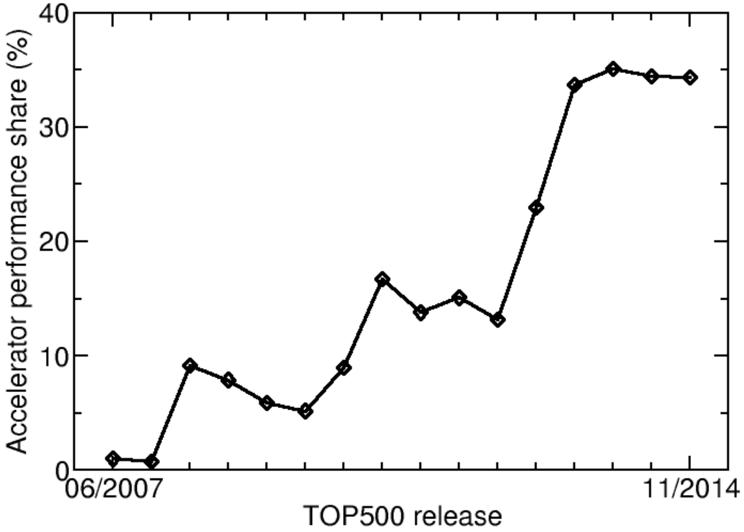
A. Weiße, G. Wellein, A. Alvermann, H. Fehske: **"The kernel polynomial method"**, Rev. Mod. Phys., vol. 78, p. 275, 2006.
E. di Napoli, E. Polizzi, Y. Saad: **"Efficient estimation of eigenvalue counts in an interval"**, Preprint. http://arxiv.org/abs/1308.4275

Why optimize for heterogeneous systems?

One third of TOP500 performance stems from accelerators.

But: Few <u>truly</u> heterogeneous software.

(Using both CPUs and accelerators.)



The Kernel Polynomial Method

Algorithmic Analysis

The Kernel Polynomial Method

Compute Chebyshev polynomials and moments.

Basic algorithm and algorithmic optimizations:

Exploit knowledge from all software layers!

for $r = 0$ to $R-1$ **do** — Application: Loop over random initial states

$\quad |v\rangle \leftarrow |\mathrm{rand}()\rangle$

$\quad$ Initialization steps and computation of η_0, η_1

$\quad$ **for** $m = 1$ to $M/2$ **do** — Algorithm: Loop over moments

$\quad\quad \mathrm{swap}(|w\rangle, |v\rangle)$

$\quad\quad |u\rangle \leftarrow H|v\rangle$

$\quad\quad |u\rangle \leftarrow |u\rangle - b|v\rangle$

$\quad\quad |w\rangle \leftarrow -|w\rangle$

$\quad\quad |w\rangle \leftarrow |w\rangle + 2a|u\rangle$

$\quad\quad \eta_{2m} \leftarrow \langle v|v\rangle$

$\quad\quad \eta_{2m+1} \leftarrow \langle w|v\rangle$

$\quad$ **end for**

end for

Building blocks: (Sparse) linear algebra library

▷ spmv()
▷ axpy()
▷ scal()
▷ axpy()
▷ nrm2()
▷ dot() — Dot Product

-1 **do**

steps and computation of η_0, η_1

$M/2$ **do**

$, |v\rangle)$

$(H - b\mathbb{1})|v\rangle - |w\rangle$ &

$= \langle v|v\rangle$ &

$= \langle w|v\rangle$ ▷ aug_spmv()

$= \langle w|v\rangle$ ▷ aug_spmmv()

Augmented Sparse Matrix Vector Multiply

Augmented Sparse Matrix Multiple Vector Multiply

Analysis of the Algorithmic Optimization

- **Minimum code balance of vanilla algorithm:**
  complex double precision values, 32-bit indices, 13 non-zeros per row, application: topological insulators

$$B_{vanilla} = 3.39\ Bytes/Flop$$ (B = inverse computational intensity)

- **Identified bottleneck: Memory bandwidth**
  ➔ Decrease memory transfers to alleviate bottleneck

- **Algorithmic optimizations reduce code balance:**

$$B_{aug\_spmv} = 2.23\ B/F$$ kernel fusion

$$B_{aug\_spmmv}(R) = 1.88/R + 0.35\ B/F$$ put *R* vectors in block

Consequences of Algorithmic Optimization

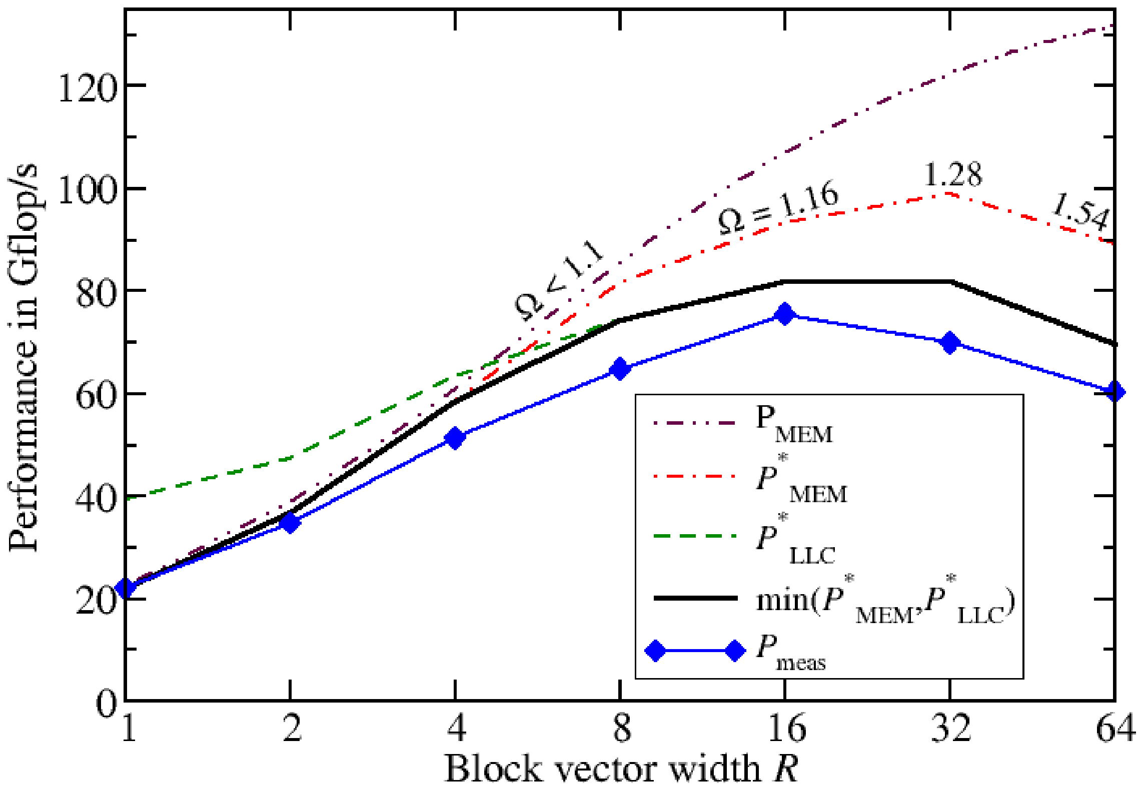
- **Mitigation of the relevant bottleneck**
  ➔ Expected speedup

- **Other bottlenecks become relevant**
  ➔ Achieved speedup may not be $B_{vanilla}/B_{aug\_spmmv}$

- **Block vectors are best stored interleaved**
  ➔ May impose larger changes to the codebase

- `aug_spmmv()` **no part of standard libraries**
  ➔ Implementation by hand is necessary

CPU roofline performance model



$P = \frac{b}{B}$ **Gflop/s**

→ Performance limit for bandwidth-bound code

b = max. bandwidth = 50 GB/s
B = code balance

$\Omega = \frac{\text{Actual data transfers}}{\text{Minimum data transfers}}$

S. Williams, A. Waterman, D. Patterson: **"Roofline: An insightful visual performance model for multicore architectures"**, *Commun. ACM*, vol. 52, p. 65, 2009.

Implementation

How to harness a heterogeneous machine in an efficient way?

Implementation

Algorithmic optimizations lead to a *potential* speedup.

➔ We “merely” need an efficient implementation!

Data or task parallelism?

- **MAGMA: task parallelism between devices**
  `http://icl.cs.utk.edu/magma/`
- **Kernel fusion Task parallelism**

➔ Data-parallel approach suits our needs

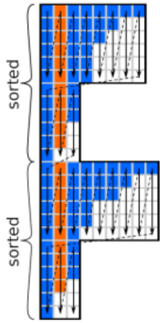
Implementation

Data-parallel heterogeneous work distribution

- **Static work-distribution by matrix rows/entries**
- **Device workload ←→ device performance**

SELL-C-σ sparse matrix storage format

- **Unified format for all relevant devices**
- **Currently no runtime-exchange of matrix data (dynamic load balancing, future work)**

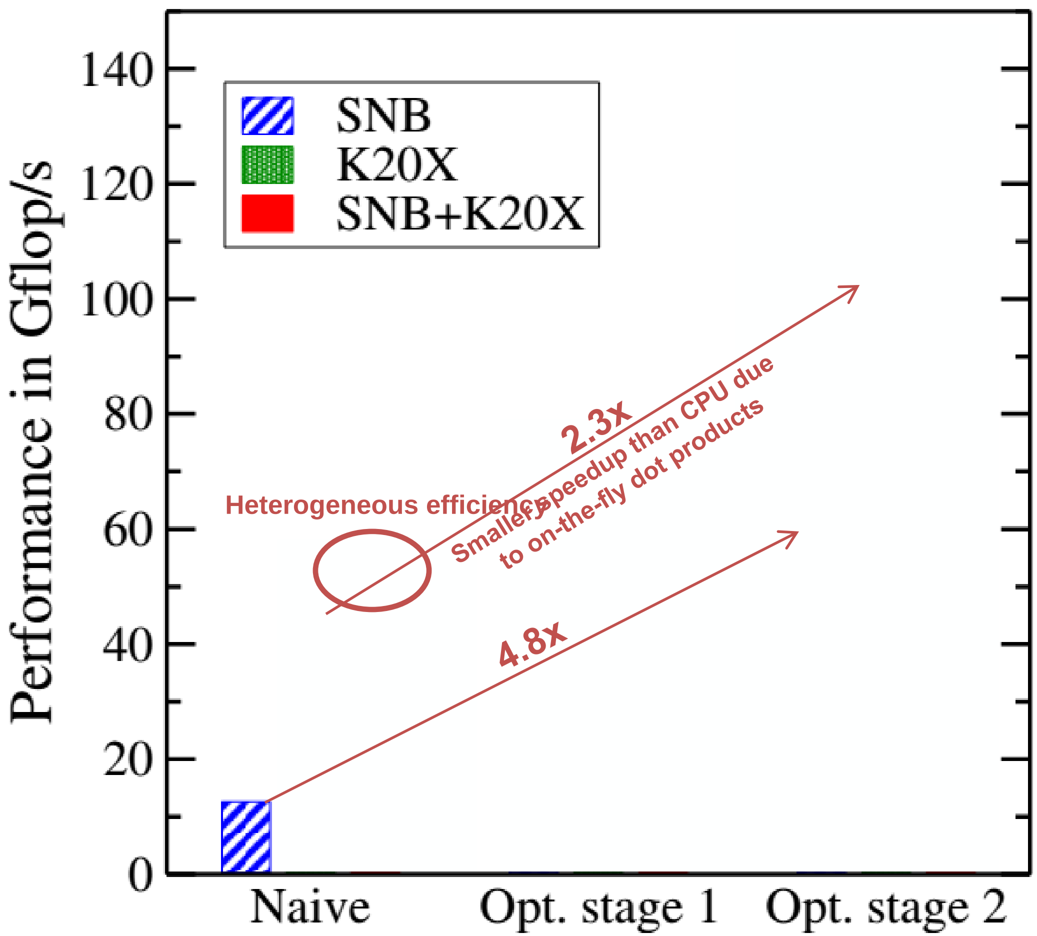


M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. R. Bishop, **"A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units"**, *SIAM J. Sci. Comput.*, vol. 36, p. C401, 2014

Performance results

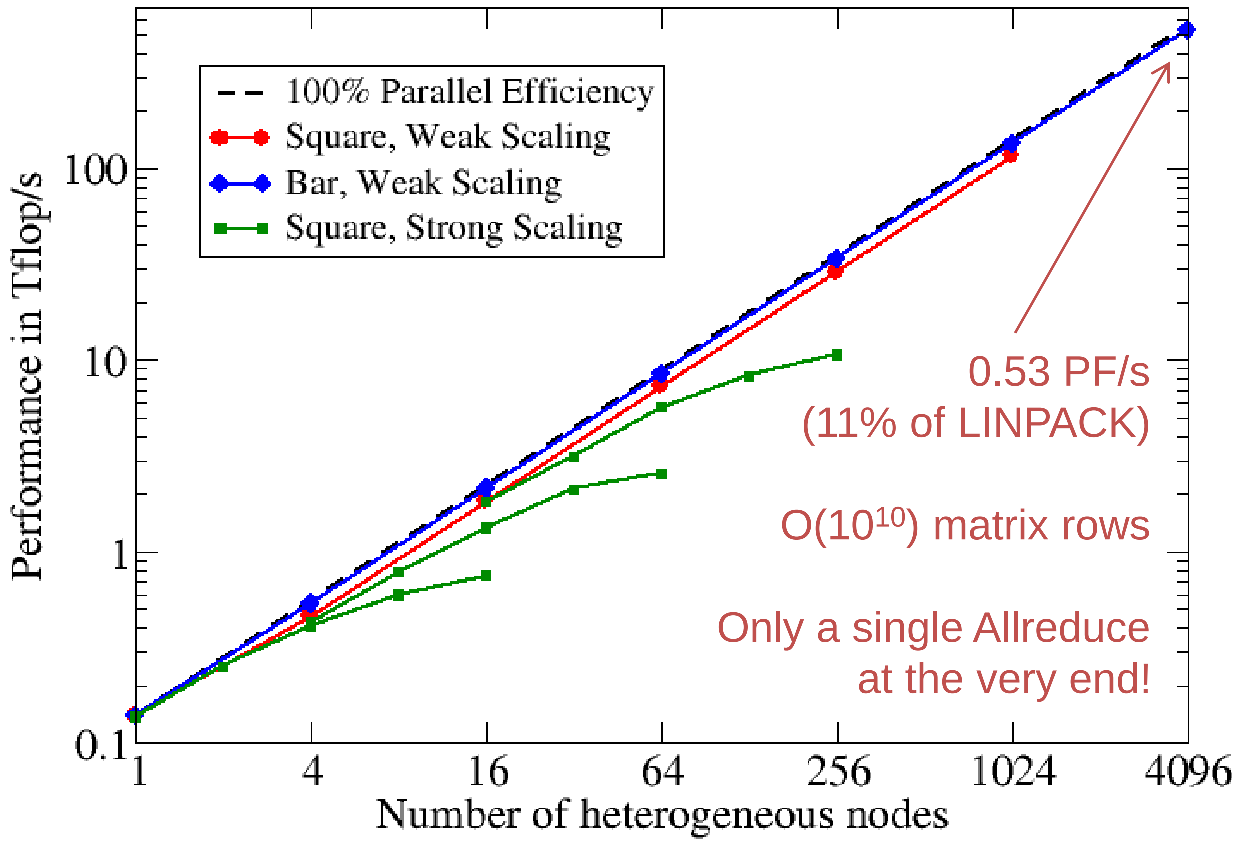
Does all this really pay off?

Single-node Heterogeneous Performance



SNB: Intel Xeon Sandy Bridge, K20X: Nvidia Tesla K20X, Complex double precision matrix/vectors (topological insulator)

Large-scale Heterogeneous Performance



CRAY XC30 – Piz Daint*

- 5272 nodes, each w/
  - 1 octacore Intel Sandy Bridge
  - 1 Nvidia Kepler K20x
- Peak: 7.8 Pflop/s
- LINPACK: 6.3 Pflop/s
- Largest system in Europe

*Thanks to CSCS/O. Schenk/T. Schulthess for granting access and compute time

Epilogue

Try it out! (If you want...)

Download our building block library and KPM application: `http://tiny.cc/ghost`

GHOST *General, Hybrid, and Optimized Sparse Toolkit*

- **MPI + OpenMP + SIMD + CUDA**
- **Transparent data-parallel heterogeneous execution**
- **Affinity-aware task parallelism (checkpointing, comm. hiding, etc.)**
- **Support for block vectors**
  - Automatic code generation for common block vector sizes
  - Hand-implemented tall skinny dense matrix kernels
- **Fused kernels (arbitrarily "augmented SpMMV")**
- **SELL-C-σ heterogeneous sparse matrix format**
- **Various sparse eigensolvers implemented and downloadable**

. . .

Backup Slides

Only in the unlikely case I was too fast...

Conclusions

- Model-guided performance engineering of KPM on CPU and GPU
- Decoupling from main memory bandwidth
- Optimized node-level performance
- Embedding into massively-parallel application code
- Fully heterogeneous peta-scale performance for a sparse solver

Outlook

- Applications besides KPM
- Automatic (& dynamic) load balancing
- Optimized GPU-CPU-MPI communication
- Further optimization techniques (cache blocking, ...)
- Performance engineering for Xeon Phi (already supported