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

Friday Lunch Talk @ Innovative Computing Laboratory
November 06, 2015
The University of Tennessee, Knoxville, TN, USA

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.

$$H \mathbf{x} = \lambda \mathbf{x}$$

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

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

Why optimize for heterogeneous systems?

One third of TOP500 performance stems from accelerators.

But: Few truly 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  
    |v⟩ ← |rand()⟩  
    Initialization steps and computation of η₀, η₁ 
    for m = 1 to M/2 do  
        swap(|w⟩, |v⟩)  
        |u⟩ ← H|v⟩  
        |u⟩ ← |u⟩ - b|v⟩  
        |w⟩ ← -|w⟩  
        |w⟩ ← |w⟩ + 2a|u⟩  
        η₂m ← ⟨v|v⟩  
        η₂m+1 ← ⟨w|v⟩  
    Algorithm: Loop over moments
```

```
Application: Loop over random initial states
```

Building blocks:
(Sparse) linear algebra library

- `spmv()`: Sparse matrix vector multiply
- `axpy()`: Scaled vector addition
- `scal()`: Vector scale
- `axpy()`: Scaled vector addition
- `nrm2()`: Vector norm
- `dot()`: Dot Product
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
  $|v\rangle \leftarrow |\text{rand()}\rangle$
  Initialization steps and computation of $\eta_0, \eta_1$
  for $m = 1$ to $M/2$ do
    swap($|w\rangle, |v\rangle$)
    $|u\rangle \leftarrow H|v\rangle$
    $|u\rangle \leftarrow |u\rangle - b|v\rangle$
    $|w\rangle \leftarrow -|w\rangle$
    $|w\rangle \leftarrow |w\rangle + 2a|u\rangle$
    $\eta_{2m} \leftarrow \langle v|v\rangle$
    $\eta_{2m+1} \leftarrow \langle w|v\rangle$
  end for
end for

for $r = 0$ to $R - 1$ do
  $|v\rangle \leftarrow |\text{rand()}\rangle$
  Initialization steps and computation of $\eta_0, \eta_1$
  for $m = 1$ to $M/2$ do
    swap($|w\rangle, |v\rangle$)
    $|w\rangle = 2a(H - b1)|v\rangle - |w\rangle$ &
    $\eta_{2m} = \langle v|v\rangle$ &
    $\eta_{2m+1} = \langle w|v\rangle$
  end for
end for

Augmented Sparse Matrix Vector Multiply
The Kernel Polynomial Method

Compute Chebyshev polynomials and moments.

Basic algorithm and algorithmic optimizations:
Exploit knowledge from all software layers!
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

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

• **Identified bottleneck: Memory bandwidth**
  ➞ Decrease memory transfers to alleviate bottleneck

• **Algorithmic optimizations reduce code balance:**
  \[ B_{\text{aug\_spmv}} = 2.23 \frac{B}{F} \]  
  kernel fusion
  \[ B_{\text{aug\_spmmv}}(R) = 1.88/R + 0.35 \frac{B}{F} \]  
  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 $B_{\text{vanilla}} / B_{\text{aug \_ spmmv}}$ 😞

- 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

\[ P = \frac{b}{B} \text{ Gflop/s} \]

→ Performance limit for bandwidth-bound code

\[ b = \text{max. bandwidth} = 50 \text{ GB/s} \]
\[ B = \text{code balance} \]

\[ \Omega = \frac{\text{Actual data transfers}}{\text{Minimum data transfers}} \]

Intel Xeon E5-2660v2 “Ivy Bridge”

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
  ➔ 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 $\leftrightarrow$ device performance

SELL-C-$\sigma$ sparse matrix storage format

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

Performance results

Does all this really pay off?
Single-node Heterogeneous Performance

SNB: Intel Xeon E5-2670 “Sandy Bridge”, K20X: Nvidia Tesla K20X, Complex double precision data (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](http://tiny.cc/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...