Holistic Performance Engineering for Sparse Iterative Solvers

Jonas Thies*  Dominik Ernst†  Melven Röhrig-Zöllner*

* Institute of Simulation and Software Technology
  German Aerospace Center (DLR)
† Erlangen Regional Computing Center (RRZE)
  University of Erlangen-Nuremberg

project ESSEX
Sparse Eigenvalue Problems

Formulation: Find some Eigenpairs \((\lambda_j, v_j)\) of a large and sparse matrix (pair) in a target region of the spectrum

\[ A v_j = \lambda_j B v_j \]

- \( A \) Hermitian or general, real or complex
- \( B \) may be identity matrix (or not)
- ‘some’ may mean ‘quite a few’, 100-1000 or so

Applications

Quantum and Fluid Mechanics

Graphene, Anderson localization, Hubbard model, Driven cavity, Rayleigh-Benard convection

DLR applications
Overview of some Sparse Eigensolvers

**Computationally related algorithms**

- CG $\iff$ Lanczos
- GMRES $\iff$ Arnoldi
- Block GMRES $\iff$ Block Krylov-Schur
- Jacobi-Davidson $\iff$ inexact Newton
  
*etc.*

**Typical algorithmic pattern:**

- obtain new search direction(s) $v_j$ (e.g. by using a matvec or solving a system),
- orthogonalize against previous vectors $V = [v_0 \ldots v_{j-1}$ and expand $V \leftarrow [V, v_j]$,
- solve projected problem for $V^T A V$ directly.
Performance of Sparse Matrix Algorithms

Typical operations are memory-bounded:

- ‘spMVM’ $y \leftarrow A \cdot x$,
- vector operations, $s \leftarrow x^T y$, $x \leftarrow \alpha x + \beta y$

... unless the data sets are small:

- CPU/KNL: OpenMP overhead $\approx 25\mu s$
- GPU: launch latency $\approx 35\mu s$
Why Performance Engineering?

**simple(?) operation:** $C = V^T V$, $V \in \mathbb{R}^{1M \times 4}$ on an Intel Haswell CPU
Why Performance Engineering?

simple(?) operation: \( C = V^T V, V \in \mathbb{R}^{1M \times 4} \) on an Intel Haswell CPU
Why Performance Engineering?

**simple(?) operation:** $C = V^T V$, $V \in \mathbb{R}^{1M \times 4}$ on an Intel Haswell CPU
Why Performance Engineering?

**simple(?) operation:**  $C = V^T V$, $V \in \mathbb{R}^{1M \times 4}$ on an Intel Haswell CPU
Test Hardware

- “Skylake”: Intel Xeon Scalable, 4 × 14 cores @2.6GHz, 384 GB DDR4 RAM
- “KNL”: Intel Xeon Phi, 64 cores @1.4GHz, 16 GB HBM (cache mode)
- “Volta”: NVidia Tesla V100-SXM2 GPU, 16 GB HBM2 (+UVM)
Test Hardware

- “Skylake”: Intel Xeon Scalable, $4 \times 14$ cores @2.6GHz, **384 GB DDR4** RAM
- “KNL”: Intel Xeon Phi, 64 cores @1.4GHz, 16 GB HBM (cache mode)
- “Volta”: NVidia Tesla V100-SXM2 GPU, **16 GB HBM2** (+UVM)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Skylake</th>
<th>KNL</th>
<th>Volta</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>360</td>
<td>338</td>
<td>812</td>
</tr>
<tr>
<td>store</td>
<td>200</td>
<td>167</td>
<td>883</td>
</tr>
<tr>
<td>triad</td>
<td>260</td>
<td>315</td>
<td>843</td>
</tr>
</tbody>
</table>

Measured streaming memory bandwidth [GB/s]
Test Hardware

- **“Skylake”:** Intel Xeon Scalable, 4 × 14 cores @2.6GHz, **384 GB DDR4** RAM
- **“KNL”:** Intel Xeon Phi, 64 cores @1.4GHz, 16 GB HBM (cache mode)
- **“Volta”:** NVidia Tesla V100-SXM2 GPU, **16 GB HBM2** (+UVM)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Skylake</th>
<th>KNL</th>
<th>Volta</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>360</td>
<td>338</td>
<td>812</td>
</tr>
<tr>
<td>store</td>
<td>200</td>
<td>167</td>
<td>883</td>
</tr>
<tr>
<td>triad</td>
<td>260</td>
<td>315</td>
<td>843</td>
</tr>
</tbody>
</table>

Measured streaming memory bandwidth [GB/s]
Test Hardware

- “Skylake”: Intel Xeon Scalable, 4 × 14 cores @2.6GHz, **384 GB DDR4** RAM
- “KNL”: Intel Xeon Phi, 64 cores @1.4GHz, 16 GB HBM (cache mode)
- “Volta”: NVidia Tesla V100-SXM2 GPU, **16 GB HBM2** (+UVM)

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Skylake</th>
<th>KNL</th>
<th>Volta</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>360</td>
<td>338</td>
<td>812</td>
</tr>
<tr>
<td>store</td>
<td>200</td>
<td>167</td>
<td>883</td>
</tr>
<tr>
<td>triad</td>
<td>260</td>
<td>315</td>
<td>843</td>
</tr>
</tbody>
</table>

Measured streaming memory bandwidth [GB/s]

<table>
<thead>
<tr>
<th>( n_b )</th>
<th>1M</th>
<th>2M</th>
<th>4M</th>
<th>8M</th>
<th>16M</th>
<th>32M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>23</td>
<td>37</td>
<td>58</td>
<td>78</td>
<td>83</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>35</td>
<td>53</td>
<td>68</td>
<td>81</td>
<td>88</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>53</td>
<td>66</td>
<td>83</td>
<td>88</td>
<td>95</td>
</tr>
<tr>
<td>8</td>
<td>51</td>
<td>70</td>
<td>85</td>
<td>87</td>
<td>99</td>
<td>100</td>
</tr>
</tbody>
</table>

“% roofline” of \( X^T Y, X, Y \in \mathbb{R}^{N \times n_b} \) on Volta.
Example: Conjugate Gradients (CG)

- 1000 CG iterations
- 3D 7-point Laplacian

**Matrix Size:**

<table>
<thead>
<tr>
<th>grid</th>
<th>N</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$256^3$</td>
<td>16.7M</td>
<td>2.2 GB</td>
</tr>
<tr>
<td>$512^3$</td>
<td>132M</td>
<td>18 GB</td>
</tr>
</tbody>
</table>

**Performance (GFlop/s):**

- **Skylake:**
  - BLAS1: 32.5
  - CG $256^3$: 33
  - CG $512^3$: 10

- **KNL:**
  - BLAS1: 39.4
  - CG $256^3$: 24
  - CG $512^3$: 21

- **Volta:**
  - BLAS1: 105
  - CG $256^3$: 82
  - CG $512^3$: 63

**Hardware Platforms:**

- Skylake
- KNL
- Volta
Increasing the Flop Intensity

**Aim:** push operations to the right

**Block solvers** (block size $n_b$)
- inner product $\implies$ factor $n_b^2$ more flops
- vector updates remain BLAS1 ($X \leftarrow X + \alpha Y$)
- **Caveat:** may increase number of iterations
- **Example:** block GMRES for multiple RHS

**Kernel fusion**
- example: compute $Y \leftarrow AX$ and simultaneously $C = X^T Y$ "for free"
- requires specialized kernels
- no deterioration of numerics

**Mixed Precision** (work in progress)
- store (block) vectors in single precision
- compute in double to maintain numerical stability
- also allows larger problems on GPU
Increasing the Flop Intensity

**Aim:** push operations to the right

**Block solvers** (block size $n_b$)
- inner product $\implies$ factor $n_b^2$ more flops
- vector updates remain BLAS1 ($X \leftarrow X + \alpha Y$)
- **Caveat:** may increase number of iterations
- **Example:** block GMRES for multiple RHS

**Kernel fusion**
- example: compute $Y \leftarrow AX$ and simultaneously $C = X^T Y$ ‘for free’
- requires specialized kernels
- no deterioration of numerics

---

**Chart 8 > PASC’18 > J. Thies et al.**

Sparse Solvers > PASC’18

---

Mixed Precision (work in progress)
- store (block) vectors in single precision
- compute in double to maintain numerical stability
- also allows larger problems on GPU
Increasing the Flop Intensity

**Aim:** push operations to the right

**Block solvers** (block size \( n_b \))
- inner product \( \Rightarrow \) factor \( n_b^2 \) more flops
- vector updates remain BLAS1 (\( X \leftarrow X + \alpha Y \))
- **Caveat:** may increase number of iterations
- **Example:** block GMRES for multiple RHS

**Kernel fusion**
- example: compute \( Y \leftarrow AX \) and simultaneously \( C = X^T Y \) ‘for free’
- requires specialized kernels
- no deterioration of numerics

**Mixed Precision** (work in progress)
- store (block) vectors in single precision
- compute in double to maintain numerical stability
- also allows larger problems on GPU
Example: Block Orthogonalization

Problem definition

- Given orthogonal vectors \((w_1, \ldots, w_k) = W\)
- For \(X \in \mathbb{R}^{n \times n_b}\) find orthogonal \(Y \in \mathbb{R}^{n \times \tilde{n}_b}\) with

\[
YR_1 = X - WR_2, \quad \text{and} \quad W^T Y = 0
\]

Two phase algorithms

Phase 1 Project: \(\tilde{X} \leftarrow (I - WW^T)X\)

Phase 2 Orthogonalize: \(Y \leftarrow f(\tilde{X})\)

- suitable \(f\):
  - SVQB (Stathopoulos and Wu, SISC 2002)
  - TSQR (Demmel et al., SISC 2012)

- Each phase messes with the accuracy of the other. \(\rightarrow\) iterate
Block Orthogonalization with Kernel Fusion

**Rearrange and fuse operations** to reduce memory traffic:

- **Phase 2** \( \tilde{X} \leftarrow X \tilde{M} \), \( N \leftarrow W^T \tilde{X} \)
- **Phase 1** \( \tilde{X} \leftarrow X - WN \), \( M \leftarrow \tilde{X}^T \tilde{X} \)
- **Phase 3** \( \tilde{X} \leftarrow X \tilde{M} \), \( M \leftarrow \tilde{X}^T \tilde{X} \)

⇒ use SVQB or Cholesky-QR

**Increased precision**

**Idea** Calculate value and error of each arithmetic operation

- Store intermediate results as **double-double** (DD) numbers
- Based on arithmetic building blocks (2Sum, 2Mult)
  Muller et al.: Handbook of Floating-Point Arithmetic, Springer 2010
- Exploit FMA operations (AVX2)
Block Orthog: runtime to convergence

Orthog. $n_b$ vectors against a block of $n_{proj}$, 12-core Haswell CPU
Software I: our Kernel Library

**GHOST**

General, Hybrid-parallel and Optimized Sparse Toolkit

- provides memory-bounded kernels for sparse solvers
- data structures:
  - row- or col-major block vectors
  - SELL-C \( - \sigma \) for sparse matrices
- written (mostly) in C
- ‘MPI+X’ with X OpenMP, CUDA and SIMD intrinsics
- runs on Peta-scale systems (Piz Daint, Oakforest-PACS)
- can use heterogeneous systems (e.g. including CPUs, MIC and GPUs)

https://bitbucket.org/essex/ghost
Software II: Algorithms and Integration Framework

**PHIST** Pipelined, Hybrid-parallel Iterative Solver Toolkit

- Interfaces: C, C++, Fortran, Python
- testing and benchmarking tools
- includes performance models
- various linear and eigensolvers

Select ‘backend’ at compile time:

- **GHOLST**, builtin (Fortran)
- **Trilinos**
- PETSc

https://bitbucket.org/essex/phist
Example: Anasazi Block Krylov-Schur on Skylake CPU

**Matrix:** non-symmetric 7-point stencil, $N = 128^3$
(var. coeff. reaction/convection/diffusion)

- Anasazi’s kernel interface is mostly a subset of PHIST’s
  \[\Rightarrow\] extends PHIST by e.g. BKS and LOBPCG
- not optimized for block vectors in row-major storage
Example: Anasazi Block Krylov-Schur on Volta GPU

**Matrix:** non-symmetric 7-point stencil, $N = 128^3$
(var. coeff. reaction/convection/diffusion)

- Anasazi’s kernel interface is mostly a subset of PHIST’s
  $\longrightarrow$ extends PHIST by e.g. BKS and LOBPCG
- not optimized for block vectors in row-major storage
Example: Anasazi Block Krylov-Schur on Volta GPU

**Matrix:** non-symmetric 7-point stencil, \( N = 128^3 \)
(var. coeff. reaction/convection/diffusion)

- Anasazi’s kernel interface is mostly a subset of PHIST’s
- \( \rightarrow \) extends PHIST by e.g. BKS and LOBPCG
- not optimized for block vectors in row-major storage
Example: Anasazi Block Krylov-Schur on Volta GPU

**Matrix:** non-symmetric 7-point stencil, $N = 128^3$
(var. coeff. reaction/convection/diffusion)

- Anasazi’s kernel interface is mostly a subset of PHIST’s
  $\implies$ extends PHIST by e.g. BKS and LOBPCG
- not optimized for block vectors in row-major storage
Example: Anasazi Block Krylov-Schur on Volta GPU

**Matrix:** non-symmetric 7-point stencil, $N = 128^3$
(var. coeff. reaction/convection/diffusion)

- Anasazi’s kernel interface is mostly a subset of PHIST’s
  $\implies$ extends PHIST by e.g. BKS and LOBPCG
- not optimized for block vectors in row-major storage
Can we do better?

**PHIST PerfCheck**: replace timing output by simple performance model

- Anasazi BKS with $n_b = 4$ gives lines like this:

<table>
<thead>
<tr>
<th>function(dim) / (formula)</th>
<th>total time</th>
<th>%roofline</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,*iflag=0)</td>
<td>6.156e+00</td>
<td>11.7</td>
<td>174</td>
</tr>
<tr>
<td>STREAM_TRIAD((nV+nW)<em>n</em>sizeof(<em>ST</em>))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Can we do better?

**PHIST PerfCheck:** replace timing output by simple performance model

- Anasazi BKS with $n_b = 4$ gives lines like this:

<table>
<thead>
<tr>
<th>function(dim) / (formula)</th>
<th>total time</th>
<th>%roofline</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,*iflag=0)</td>
<td>6.156e+00</td>
<td>11.7</td>
<td>174</td>
</tr>
<tr>
<td>STREAM_TRIAD((nV+nW)<em>n</em>sizeof(<em>ST</em>))</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**‘realistic’ option:** report strided data accesses due to ‘views’ and adjust perf. model

<table>
<thead>
<tr>
<th>function(dim) / (formula)</th>
<th>total time</th>
<th>%roofline</th>
</tr>
</thead>
<tbody>
<tr>
<td>phist_Dmvec_times_sdMat_inplace(nV=4,nW=4,ldV=85,*iflag=0)</td>
<td>6.013e+00</td>
<td>23.8</td>
</tr>
<tr>
<td>STREAM_TRIAD((nV+nW_)<em>n</em>sizeof(<em>ST</em>))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>streaming_strided_indices(nV=4,nW=4,ldV=85)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Block Jacobi-Davidson QR

- Use *inexact Newton* rather than Krylov sequence
- *JDQR*: subspace acceleration, locking and restart (Fokkema’99)

Block Jacobi-Davidson correction equation

- $n_b$ current approximations: $A\tilde{v}_i - \tilde{\lambda}_i \tilde{v}_i = r_i$, $i = 1, \ldots, n_b$
- previously converged Schur vectors $(q_1, \ldots, q_k) = Q$
- solve approximately (with $\tilde{Q} = (Q \quad \tilde{v}_1 \quad \ldots \quad \tilde{v}_{n_b})$):

\[(I - \tilde{Q}\tilde{Q}^T)(A - \tilde{\lambda}_i I)(I - \tilde{Q}\tilde{Q}^T)x_i = -r_i \quad i = 1, \ldots, n_b\]

- use some steps of a block(ed) iterative solver
- orthogonalize new directions $x_1, \ldots, x_{n_b}$ (outer subspace iteration)
BJDQR: ‘Numerical Overhead’

With larger block size...

- number of (outer) iterations decreases
- total number of operations increases
- tested here for various matrices
BJDQR: ‘Numerical Overhead’

With larger block size...

- number of (outer) iterations decreases
- total number of operations increases
- tested here for various matrices
BJDQR: ‘Numerical Overhead’

With larger block size...

- number of (outer) iterations decreases
- total number of operations increases
- tested here for various matrices
BJDQR on Different Hardware (here: Laplace problem)

- Skylake ($512^3$): 32.5 GFlop/s
- KNL ($256^3$): 39.4 GFlop/s
- Volta ($128^3$): 105 GFlop/s
BJDQR on Different Hardware (here: Laplace problem)

- **Skylake** (512$^3$): 32.5 GFlop/s
- **KNL** (256$^3$): 39.4 GFlop/s
- **Volta** (128$^3$): 105 GFlop/s

**CG GFlop/s**

- **Skylake** (512$^3$): 32 GFlop/s
- **KNL** (256$^3$): 24 GFlop/s
- **Volta** (128$^3$): 76 GFlop/s
BJDQR on Different Hardware (here: Laplace problem)

![Chart showing performance of BJDQR on different hardware](chart.png)
BJDQR on Different Hardware (here: Laplace problem)

![Graph showing performance comparison between Skylake, KNL, and Volta for different tasks and hardware configurations. The Y-axis represents GFlop/s (GigaFLOP/s), and the X-axis represents the hardware types: Skylake (512^3), KNL (256^3), Volta (128^3). The graph includes bars for BLAS1 GFlop/s, CG GFlop/s, BJDQR-1 GFlop/s, and BJDQR-2 GFlop/s for each hardware type.](image-url)
 BJ DQR on Different Hardware (here: Laplace problem)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>BLAS1 GFlop/s</th>
<th>CG GFlop/s</th>
<th>BJDQR-1 GFlop/s</th>
<th>BJDQR-2 GFlop/s</th>
<th>BJDQR-4 GFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skylake (512³)</td>
<td>32.5</td>
<td>49.4</td>
<td>73</td>
<td>51</td>
<td>103</td>
</tr>
<tr>
<td>KNL (256³)</td>
<td>24</td>
<td>35</td>
<td>47</td>
<td>55</td>
<td>76</td>
</tr>
<tr>
<td>Volta (128³)</td>
<td>105</td>
<td>132</td>
<td>147</td>
<td>221</td>
<td></td>
</tr>
</tbody>
</table>
Preconditioning with ML

- non-symm. PDE as before
- use AMG preconditioner ML from Trilinos ("non-symmetric smoothed aggregation")

<table>
<thead>
<tr>
<th>problem size</th>
<th>preconditioner</th>
<th>iterations</th>
<th>spMVMs</th>
<th>$t_{tot}$</th>
<th>$t_{gmres}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128^3$</td>
<td>GMRES</td>
<td>447</td>
<td>9875</td>
<td>52.6s</td>
<td>25.0s</td>
</tr>
<tr>
<td></td>
<td>GMRES+ML</td>
<td>26</td>
<td>612</td>
<td>21.2s</td>
<td>11.4s</td>
</tr>
<tr>
<td>$256^3$</td>
<td>GMRES</td>
<td>781</td>
<td>17223</td>
<td>1300s</td>
<td>571s</td>
</tr>
<tr>
<td></td>
<td>GMRES+ML</td>
<td>40</td>
<td>922</td>
<td>346s</td>
<td>183s</td>
</tr>
<tr>
<td>$512^3$</td>
<td>GMRES</td>
<td>&gt;1k</td>
<td>&gt;22k</td>
<td>&gt;1h</td>
<td>&gt;1h</td>
</tr>
<tr>
<td></td>
<td>GMRES+ML</td>
<td>32</td>
<td>746</td>
<td>624s</td>
<td>320s</td>
</tr>
</tbody>
</table>
Summary

- Two libs for high-performance sparse solvers: GHOST & PHIST
- support algorithm developer by
  - kernel interface inspired by MPI
  - kernels & algorithmic core operations
  - built-in performance models
- Holistic performance engineering needed for sparse block solvers
  - tight interplay of data structures, kernels, core and algorithm
  - Example: Block Krylov-Schur with different kernels and orthogonalization routines

Next steps:
- model that takes fast and slow memory segments into account
- demonstrate strong scaling advantage of block methods

https://bitbucket.org/essex/[ghost|phist]